Patofil: An MGLAIR Agent for a Virtual Reality Drama

SNeRG Technical Note 38

A report submitted in partial fulfillment
of the requirements of the degree for
Master of Science in Computer Science
at the State University of New York at Buffalo
by

Trupti Devdas Nayak
Department of Computer Science and Engineering
td23@cse.buffalo.edu

Advisor: Dr. Stuart C. Shapiro

May 10, 2005

Abstract

This paper discusses the design issues and implementation details for building an interactive, intelligent agent using the GLAIR architecture. The agent (also known as *actor-agent*) is designed for an interactive virtual reality drama titled *The Trial The Trail*. Agent research for the interactive drama is discussed with focus on one actor-agent named Patofil who is one of the main characters in *The Trial The Trail*. The virtual drama is currently under production and acts constituting the drama are being developed and tested. Among other acts, Patofil appears in Act 3, Scene 1 of the drama, which is the focus of this paper. Patofil's knowledge layer is implemented in SNePS (a knowledge representation and reasoning system). Lower layers of GLAIR involve the implementation of Patofil's embodiment and the virtual world, both of which are rendered using Ygdrasil, a virtual reality environment authoring tool. The SNePS knowledge layer includes Patofil's beliefs, acts and policies which this paper discusses in detail. An actor-agent possesses modalities like speech, hearing, vision, navigation, mood and a sense of time. Modalities can be defined as software/hardware acting resources used by the agent for communicating with other agents, participants and objects in the world. The implementation of modalities for Patofil and future plans for MGLAIR (Modal-GLAIR) are also discussed.

Acknowledgements

I would like to thank my advisor Dr. Stuart C. Shapiro, for the invaluable advice and direction he has provided to me. Dr. Shapiro has always been encouraging in my pursuits and has taught me to think *outside the box*. He has always been there to help with issues/problems during the course of my Master's studies and has provided many insights which have enabled me to look at issues realistically and actively seek their solutions. Most importantly, he has taught me to question rather than accept and to strive towards perfection in all endeavors. I hope to apply all that I learned under his guidance in my present and future academic and professional endeavors.

I would like to thank Prof. Josephine Anstey for providing me with the opportunity to work on the production of the drama *The Trial The Trail*. It has been a great learning experience and I have been tremendously influenced by Prof. Anstey's enthusiasm for her work. She has taught me to pursue whatever one believes in. From her, I learned about creativity and art. I now see things from an artist's viewpoint and am aware of the numerous issues involved in producing an interactive drama for the general public.

I wish to thank Orkan Telhan for his work on implementing the *embodiment and virtual world* using Ygdrasil and also for making the long hours spent on testing and debugging Patofil's acts and scenes more fun.

Contents

List of Figures

0.1 Introduction

This project report provides a comprehensive discussion of the design and implementation of an actor-agent named Patofil who plays the role of a guide in an interactive drama titled *The Trial The Trail* [6]. Patofil is implemented in GLAIR and is endowed with modalities. A modality is a software/hardware acting resource the agent makes use of to communicate with other agents, participants and the world. Patofil's knowledge layer contains beliefs, her acts and policies about when to perform these acts. This knowledge layer is implemented in SNePS [2] which is a knowledge representation and reasoning system.

Section 2 discusses prior and current research on an actor-agent for an interactive virtual reality drama titled *The Trial The Trail*. The features desired in an interactive, believable, intelligent actor-agent are also discussed. Section 3 provides details of the tools and systems used for fleshing out an actor-agent. An overview of the GLAIR architecture is provided and the concept of modalities is explained. An actor-agent follows a script. Section 4 provides an overview of Act 3, Scene 1 which is the script that Patofil follows. Snares which form an important emotion-manipulation component of scenes in the drama are explained. Section 5 discusses major design issues faced, like handling triggers/cues from the world, contingency handling and prioritizing between acts. Section 6 provides details of Patofil's implementation, including her beliefs, the structure of primitive actions and complex plans at the knowledge layer. Modalities which use socket connections for providing a means of communication between the lower layers of the PML (implemented as processes running on different machines) are discussed. Instructions for loading and demonstrating Patofil using a Java-based graphical simulator are listed in Section 7. Section 8 summarizes the paper and Section 9 discusses future plans which include completing production of *The Trial The Trail* and extending the GLAIR architecture to MGLAIR (Modal-GLAIR), by implementing a modality package (using CLOS, The Common Lisp Object System) which can be integrated with GLAIR; thus providing the agent-designer with tools for easily and efficiently endowing an agent with modalities.

0.2 Agents in a Virtual Reality Interactive Drama

0.2.1 Prior Research

Previous GLAIR agents [3] specifically developed for interactive virtual reality dramas include Princess Cassie [8] who was designed for an earlier version of *The Trial The Trail*. Princess Cassie is one of the first GLAIR-based agents implemented for a scene in a virtual reality drama. During the design and development of Princess Cassie, the idea of modalities and developing a modality-based architecture was born. It was hoped that in the future, GLAIR would be extended to include functionality that would allow the design and specification of each modality as part of a generalized "suite of modalities" [8]. The idea involved extending GLAIR by including a modality package, giving rise to Modal-GLAIR

or the MGLAIR architecture. The current modality suite implemented for GLAIR is primitive and is being used for implementing actor-agents like Patofil in the virtual reality drama titled *The Trial The Trail*. Future plans involve extending the GLAIR architecture to include modalities which will be intrinsic to the MGLAIR architecture. In subsequent sections, MGLAIR refers to the GLAIR plus modalities architecture being currently used for Patofil.

0.2.2 Current Research

Current research aims towards creating a generic agent architecture which will enable an agent designer to implement intelligent, interactive, believable and cognitive agents and endow them with modalities for communicating with other agents, participants and the world. As an endeavor towards this goal, we have been working on designing actor-agents for a virtual reality drama. The actor-agent discussed in this report is a guide named Patofil who interacts with human participants and guides them through snares in the drama. Subsequent discussions include agents, agent intelligence, extent of interactivity that can be pre-programmed and the effectiveness achieved through the implementation of cognitive agents for an experiential system used by human participants.

Building Agents for virtual reality

An agent in the context of an interactive drama is one who has the cognitive and perceptual capability to interact intelligently and believably with participants. In future agent designs it is envisioned that the agent will be capable of eliciting the participant's emotions, and its perceptual sensors will be capable of detecting the participant's mental attitudes and emotional states. The agent uses perceptual inputs as cues for reasoning before choosing a particular sequence of activities. An interactive drama involves both actors and audiences. The interactions between the actors and the audience play an important role in determining the impact of the drama on the psyche of the audience. A richly scripted drama with no interactive scope might not be as effective as one which focuses on these very interactions and audience choices to determine a dynamically selected path diverging from the main story arc.

Agents designed for playing the roles of actors have to possess a broad set of perceptual sensory input/output channels similar to human beings. A multitude of modal expressions are required to express the agent actor's emotions and motivations. Similar to a human actor in a play/drama, an actor agent is given a script, which outlines the main story to be followed, providing leeway for decisions and different actions which can be taken by the participant and which might potentially affect the main story arc. The extent to which the story deviates from the main storyline depends on the context of the contingency situations. A contingency situation can be described as an event that may occur but that is not likely or intended.

The Trial The Trail is an interactive drama where the actor-agents are given pre-written scripts. The participant is an audience member who interacts with the actor-agents but does not have a script. The participant does as he wishes, and the actor-agents interact and react believably, while following the main story line, but improvising as needed [1]. The extent of improvisation which can be developed while the actor-agent is being designed *(off-line)* or while the actor-agent is performing *(on-line)* is limited. There is considerable effort involved in providing a unique experience for each participant. The script explained in Section 0.4 is rich and diverse and allows participants to perform a variety of actions while interacting with the actor-agents.

Actor-agents require a structured execution plan(s). Scripts therefore play an important role in determining the extent of interactivity possessed by an agent. An English script which a human actor can comprehend, cannot be directly represented by the agent. The script is instead first formulated as a set of plans. But agents cannot be given instructions for every trivial situation in the script. It has always been a difficult design decision where a choice between agent autonomy is weighed against the pre-scripted plans that the actor-agent follows. Sometimes, there is a conflict between the two but a balanced compromise is ideal. The agent should be autonomous to an extent wherein it can react intuitively to its surroundings and the perceptions from the virtual environment. But it should also adhere to a script, thus providing a purpose for its autonomous actions. We try to maintain this balance by providing the agent with high level plans that represent the main story arc and a set of plans called contingency plans (for handling participant actions) to choose from for execution based on the context of the script.

0.3 Introduction to tools/systems used for implementing Patofil: An actor-agent for *The Trial The Trail*

0.3.1 An Overview

The agent architecture is a three-tier architecture titled Grounded Layered Architecture with Integrated Reasoning, namely GLAIR [9] shown in Figure 1. The lower layers of GLAIR can be integrated with the virtual reality software package namely Ygdrasil [7]. Ygdrasil implements the embodiment of the agent in PMLc and SAL of GLAIR and communicates with PMLb of GLAIR through TCP/IP sockets. The mind of the agent resides at the top layer which is the Knowledge Layer of the GLAIR architecture.

[1] Conceived by Prof. Shapiro while drawing up a comparison between theatrical plays with audiences and interactive VR dramas with participants

0.3.2 Layers of GLAIR

The GLAIR architecture is shown in Figure 1 and consists of the layers:

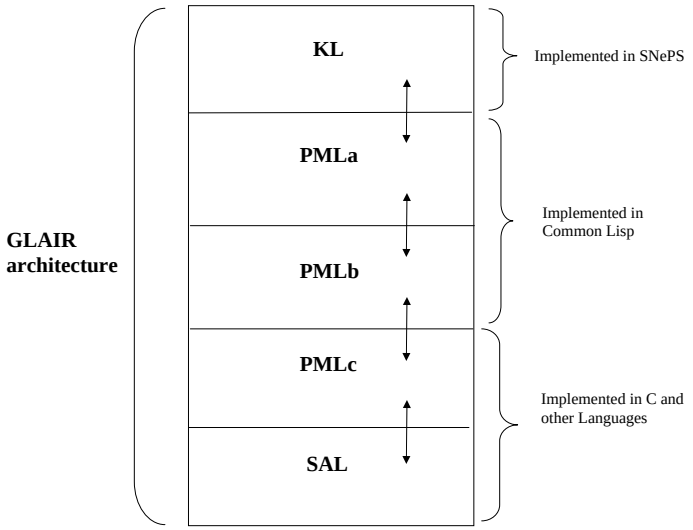

Figure 1: The GLAIR Architecture

- The Knowledge Layer or the KL is the layer where the consciousness of the agent resides. The KL communicates with PMLa of MGLAIR. The knowledge layer of GLAIR agents is implemented in SNePS [1], [2]. The KL contains the agent's beliefs, plans and policies for executing these plans. The plans executed by the GLAIR agent are scripted with SNeRE constructs. SNeRE is the acting component of SNePS, also known as the SNePS Rational Engine, which allows acting to be incorporated in an agent. The design and implementation details of the agent Patofil using SNePS and SNeRE are discussed in Section 0.6).

- The PML mainly contains routines for primitive acts which are utilized by the KL. There are three sub-layers in the PML, namely PMLa, PMLb and PMLc.
  - PMLa consists of Knowledge Layer primactions (primitive actions) and functions written in Lisp and SNePS. The PMLa primactions are attached to corresponding primitive actions which are used in the KL.
  - PMLb consists of Lisp functions which provide an interface between the PMLa and lower layers of GLAIR (PMLc and SAL). In the GLAIR agent, the PMLb consists largely of Lisp functions for modalities.
  - The PMLc consists of low level functions which access the SAL and thus control the agents motors, sensors and other bodily functions. The PMLc is generally implemented in a different language, sometimes even using different tools. In the case of Patofil, the PMLc is implemented in Ygdrasil which is used for implementing Patofil's embodiment and the virtual reality environment; but is still a layer of the GLAIR architecture.
- The SAL is the Sensori Actuator layer where the cognitive agent's motors and sensors are implemented. SAL is implemented in C and other languages (Java), depending on the particular implementation in the hardware or software-simulated robot. In the case of Patofil, the SAL is implemented in Ygdrasil (discussed in Section 0.3.2).

Lower Layers of GLAIR and Ygdrasil

The virtual environment and agent embodiment for *The Trial The Trail* is rendered using Ygdrasil [7], which is a VR authoring software package. Ygdrasil is a framework developed as a tool for creating networked virtual environments. It is focused on building the behaviors of virtual objects from re-usable components, and on sharing the state of an environment through a distributed scene graph mechanism. Ygdrasil is built in C++, around SGI's IRIS Performer visual simulation toolkit and the CAVERNsoft G2 library [7]. The integration between GLAIR and Ygdrasil is implemented at the lower layers of the GLAIR architecture, namely PMLc and SAL. The higher layers of GLAIR are not aware of the details of this implementation. In Patofil's implementation the higher layers of GLAIR are on a different machine than the lower layers of GLAIR. Hence, communication between the PMLb and PMLc across the network is through TCP/IP sockets.

0.3.3 GLAIR Architecture with Modalities

Human beings have sensory organs to perceive changes and events in the world and this information is communicated to their brain which processes the input and in turn commands their body to perform a variety of actions. Similarly an agent's embodiment possesses a suite of modalities or sensory capabilities which enable it to perceive and these perceptions are sent to its brain/mind. Modalities

like vision, hearing, speech, animation, navigation, emotion and time are integrated with the GLAIR architecture. Thus the MGLAIR (M for modality) architecture shown in Figure 2 is an extension of the GLAIR architecture, wherein a generic suite of modalities are provided and this suite can be modified as per the requirements of a particular agent design.

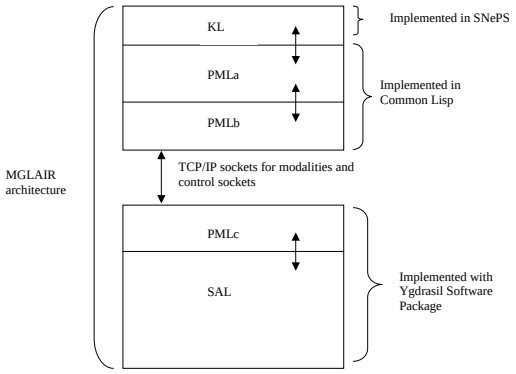

Figure 2: The Modal-GLAIR Architecture

In the drama *The Trial The Trail*, Patofil is one of the main characters. An MGLAIR actor-agent plays Patofil's role in the interactive drama. The agent possesses a set of modalities which correspond to a human's sensory and effector organs. Patofil has 7 modalities which are:

1. Speech - Natural language is the main mode of communication for the agent with the participant. Speech lines are in the form of strings which are communicated to the agent's embodiment.

2. Animation - The actor-agent has a head and a body with wing shaped arms and feet. Body language is an effective way of communicating with the participant, and animation aims to add visual communication in addition to speaking natural language.

3. Navigation - Movement of the agent from one point to another is determined by the navigation commands which are sent across the socket from PMLb to PMLc. Agent navigation is an immensely rich area of research by itself. Obstacle avoidance and path planning are just a few of the numerous issues which arise during the implementation of navigation capabilities for an agent.

4. Mask - The actor-agents wear a mask similar to the facial masks worn by theatrical actors who perform on stage. The agent does not have a visible face, instead the mask morphs between expressions conveying the agent's mood. The masks show happiness, sadness, anger, fear as demanded by Patofil's acts.

5. Perception - Perceptions from the world are required for two purposes. Firstly, from these perceptions, the agent is made aware of the participant's actions. Secondly, changes and events in the world are communicated through the perception socket to the agent. Perceptions can be visual or auditory.

    - Vision - Visual cues are communicated to the agent through the vision modality. Visual cues include information about the position of the participant, movement of objects in the world, the agent's own movements and position relative to other agents, participants etc.
    - Hearing - Lines of speech delivered by the agents in the world are input to the hearing modality to enable the agent to know when its speech act is completed by its embodiment in the world. This is shown in Figure 3 where the output of the speech socket is piped as input to the hearing socket. This modality was initially modeled as "self-perception" which is equated to the way human beings pace their speech while talking. When humans get feedback (by hearing their own speech) they continue talking; similarly the actor-agents pace their speech delivery by waiting for feedback which is through hearing their own voice.

6. Timer - A new aspect explored in the implementation of the agents like actor-agent Patofil is the issue of time. Princess Cassie [8] didn't have a sense of time. The timer modality provides Patofil with a sense of timing in the sense that Patofil can *time* her acts. Some acts have to be performed for a specified duration of time, unless interrupted by the participant or other events occurring in the world. In order to keep track of the time elapsed, Patofil uses the timer modality and starts a timer for the act. When the timer elapses, a belief that the timer has expired is added to her knowledge base and this *timer elapsed* event is a cue which triggers subsequent acts. Timers are not communicated across the mind to the embodiment. Instead they form a mental entity and remain in the agent's mind. Timer commands include *starting, pausing* and *terminating* a specified timer. Each timer is associated with a *name* which is assigned when the timer is created and initialized with a *duration*. Functions for controlling the timer are defined and explained in Section 0.6.6.

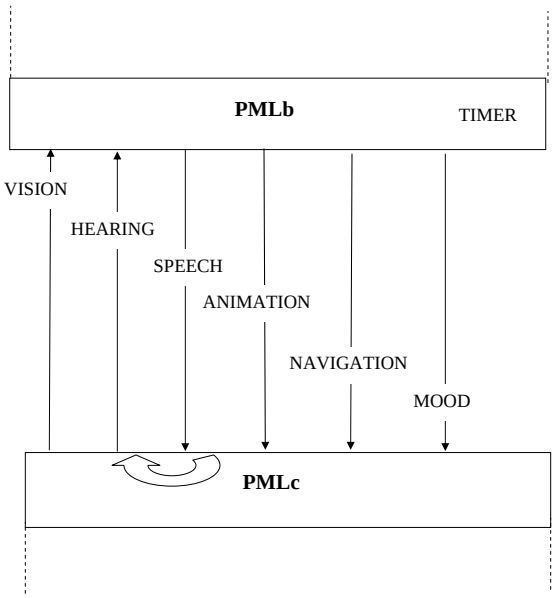

Figure 3: Modality implementation with socket connections for communication between the PMLb and PMLc

Communication between the agent(s) and participant is in natural language. This medium of communication is superior and intuitive when compared to a text based interface. It is superior because communicating through speech enables the participant to make use of his/her hearing while absorbing elements of virtual reality environment through his/her eyes. We believe this provides a more immersive experience than reading from a text based interface. It is intuitive because as human beings, we *speak* to other humans for communicating our ideas. Presently, a natural language interface is being developed which will enable the participant to respond to the agent through natural language. In the future, the participant will communicate with the agent through speech and the sentences the participant utters will be *heard* and *understood* by the agent.

0.4 Overview of Act 3, Scene 1

As of April 2005, Act 3 of the drama *The Trial The Trail* is under production [12], [19]. Mid-project participant tests were planned and executed in October 2004. Act 3, Scene 1 involves the main character agent Patofil and the participant. During the course of Scene 1, actor-agent Patofil conducts a vigil by meditating on a grassy mound and encourages the participant to follow her directions. The VR environment for this scene consists of a green grass mound shown in Figure 4 and Figure 5. The other main character in the drama is Filopat who is visible in Figure 4 but makes an appearance only at the end of Act 3, Scene 1.

Figure 4: A view of the mound, ruins, Patofil and Filopat in *The Trial The Trail*

There are ruins of a chapel on the mound. Trees and grass surround the ruins. In the distance, Will-O-the-Wisps can be seen floating around. Patofil and the participant can bat at the whisps and watch them float around. The edge of the mound is surrounded by reeds through which the participant can navigate.

Act 3 consists of three Scenes [13]:

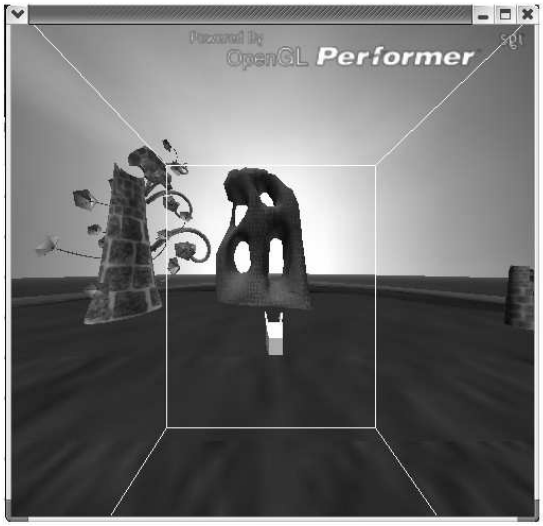

Figure 5: Participant view of mound and ruins after entry of Act 3, Scene 1, **Snare 1** in *The Trial The Trail*

- Scene 1: Patofil and the participant start the vigil on the mound as shown in Figure 5 by keeping silent and not fidgeting. Depending on the participant's reactions, Patofil reacts interactively. For example, if the participant leaves the mound without performing the vigil, Patofil admonishes the participant and entreats him/her to come back. On the other hand, after passage of some time, Patofil herself loses interest in the vigil and encourages the participant to go off the mound. The scene ends with Patofil running off into the reeds and the participant hears a far away, spine-tingling scream.

- Scene 2: Begins with Patofil being chased by bad guys. Three bad guys surround the participant

and taunt him/her. Two other bad guys drag Patofil into the faraway reeds and are out of sight.

- Scene 3: At dawn Filopat (another actor-agent) comes to find the participant and Patofil, but discovers only the participant and no Patofil in sight.

The main plan is a skeletal structure representing the highest level of goals which Patofil endeavors to achieve. The contingencies are deviations from the main story line and the agent should be able to steer the narrative back to the main thread seamlessly. In this paper, we discuss Act 3, Scene 1 of the *The Trial The Trail*.

0.4.1 What are snares?

Definition of **snare**: *Something (often something deceptively attractive) that catches you unawares.* From this viewpoint, we have divided the script for an actor-agent into modular sections, each of which can be effectively called a **snare**. Each snare is designed to collect information about what the user is doing, and interpret it in the context of the snare. This information is used for determining the sequence of acts that the actor-agent will follow in the next act. A snare can be envisioned as an act, but with unobtrusive emotion-gauging or emotion-manipulating sections, which manipulate the emotions of the user. The user can react/act in any manner he chooses during the drama. Each snare has contingency plans for dealing with the user's actions within the snare. Transitions between snares are determined by what the participant is doing in order to provide a smooth flow with plans always being performed in context. By designing the script for an agent as a set of snares, modularization and easier comprehension has been achieved. See [6] for more information on the concept of snares and their development.

0.4.2 Knowledge about the script and contingencies

As Patofil proceeds through the set of plans and contingencies, she requires knowledge about the acts she has executed or is currently executing. There are 5 snares for Act 3, Scene 1 of *The Trial The Trail* and many contingency situations. Contingencies are dependent on context of the script and occur because of participant actions or reactions. Patofil remembers each plan she is currently executing by maintaining a working memory (WM) and a short-term memory (STM). For more details of working memory and short-term memory, see Section 0.6.6. Patofil's main plan has the following **snares** in Act 3, Scene 1:

1. Snare 1: The Vigil
    - Act 3, Scene 1 starts with Patofil meditating
    - She encourages the participant to meditate along with her
2. Snare 2: Play with Whisps

- She gets bored with meditating and plays with whisps
- She encourages the participant to join her and play with the whisps

3. Snare 3: Whisps leave

- She gets bored with playing with whisps
- The whisps leave the mound, trying to lure the participant off the mound

4. Snare 4: Jump around the rocks

- She wanders about the mound looking at ruins
- She encourages the participant to look around the ruins on the mound

5. Snare 5: Leave mound

- She suggests leaving the mound
- She leaves the mound
- She encourages the participant to leave the mound with her
- She goes towards the reeds
- She enters the reeds, gets out of sight of the participant and screams and Act 3, Scene 1 ends.

Contingencies are taken into account depending on the context of the script which is currently executing and whether the contingency affects the main story arc or not. Hence the contingency triggers/cues are classified as *active* in certain Snares and are inactive in others. The **contingencies** that assume higher priority when compared to the above snares are:

1. Active in Snare 1

- Participant speaking (U_Speaking)
- Participant fidgeting (U_Fidgeting)
- Participant moving towards edge of mound (U_At_Edge)
- Participant leaving mound (U_Off_Mound)
- Gong sounds (Gong_Sounds)

2. Active in Snare 2

- Participant moving towards edge of mound (U_At_Edge)
- Participant leaving mound (U_Off_Mound)

- Participant hitting/playing with Whisps(U_Hit_Whisp)
- Gong Sounds (Gong_Sounds)

3. Active in Snares 3,4 and 5

- Participant near edge of mound (U_At_Edge)
- Participant leaving mound (U_Off_Mound)
- Gong sounds (Gong_Sounds)

The aforementioned snares can be visualized as a sequence of activities which Patofil pursues. These activities are successful if predetermined goal(s) are achieved upon executing the activity. Activities are represented as SNeRE complex acts.

0.5 Design Issues

0.5.1 Triggers/Cues

Patofil's mind is conscious at the knowledge layer. Her embodiment is implemented in Ygdrasil. Communication between the mind and the body is necessary for Patofil to be aware of her actions and their completion in the world. This is achieved through triggers or cues. The concept of triggers can be understood by comparing actor-agents to actual actors in a drama. Apart from using their own knowledge for sequencing acts, human actors use other cues or certain lines of speech to guide their performances. These cues can be from other actors, stage props, people behind the scenes etc. Similarly, an agent actor receives cues/triggers from the world and these help her to continue with her acts. The Figure 6 lists all cues received by Patofil through the course of Act 3, Scene 1.

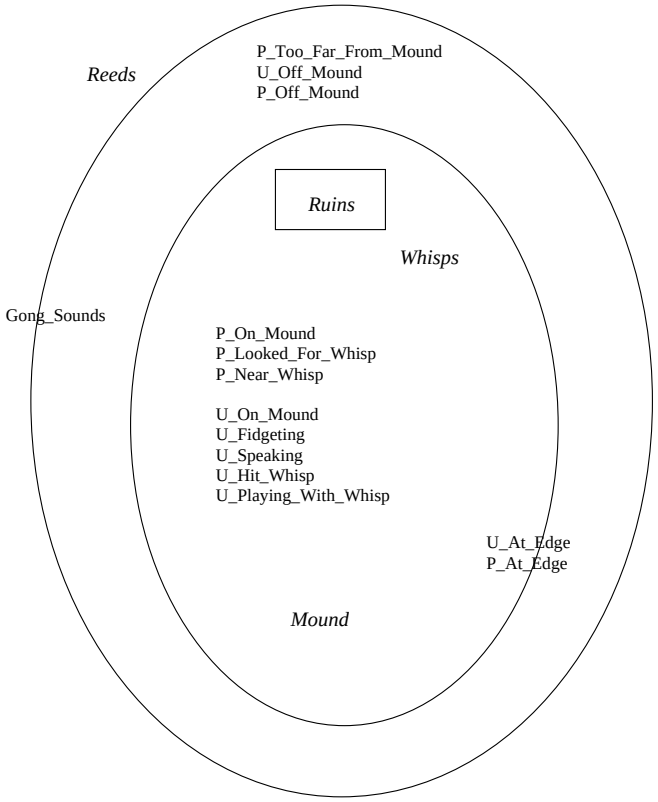

Figure 6: Triggers/Cues received by Patofil in Act 3, Scene 1 of *The Trial The Trail*

Triggers/Cues are communicated through the agent's modalities. The modalities are listed in Section 0.3.3. Triggers/cues are classified based on the modalities through which they are communicated. They are:

1. Speech Cues: Lines spoken by Patofil through the speech modality are cued back *(feedback)* through the hearing modality. This enables Patofil to know when her utterance has been completed by her embodiment in the world. This cue is similar to the way human beings sequence their speech. Humans know they have completed saying a line when they get feedback by hearing it.

Each speech line that Patofil utters through the speech modality is echoed back to her through the hearing modality. The speech lines map onto *.wav* files on the Ygdrasil side and these *.wav* files are played to provide the effect of Patofil saying her speech lines. Act 3, Scene 1, Snare 1 contains many speech lines which are pre-recorded by a human voice (to impart a human-like quality to her speech rather than a string of words generated by a mechanized voice). It is not possible to provide all of these speech lines in the paper. Instead, a few examples are provided to give an idea about these speech strings/commands which Patofil utters. For example, in the first sub-plan of Snare 1, Patofil starts the act by commanding the participant to keep quiet and meditative on the mound. She says the following in sequence:

- NoTalking
- StandStraight
- ContemplateTheInfinite
- DontLeaveTheHill

2. Navigation Cues: When Patofil commands her embodiment to move, the actual movement in the virtual world takes longer when compared to the time taken to issue the command. When Patofil's embodiment completes the navigation, then the change in the situation in the world is communicated to Patofil through her vision modality so she can see her present position and know that she has moved. Patofil receives knowledge about her (embodiment) position and that of the participant. These messages take the following form:

   - P_On_Mound: Cue that Patofil is now on the mound
   - P_At_Edge: Cue that Patofil is at the edge of the inner circle of the mound
   - P_Off_Mound: Cue that Patofil is off the mound
   - P_Arrived_ruinPoint1: Cue that Patofil has arrived at a pre-determined point near the ruin
   - P_Arrived_ruinPoint2: Cue that Patofil has arrived at a pre-determined point near the ruin
   - P_Too_Far_From_Mound: Cue that Patofil has wandered too far away from the safety of the mound
   - P_Near_Whisp: Cue that Patofil has navigated next to a whisp
   - U_On_Mound: Cue that the user/participant is now on the mound
   - U_Off_Mound: Cue that user/participant is off the mound (outermost circle of the mound)
   - U_At_Edge: Cue that user/participant is at the edge of the mound

3. Animation Cues: Patofil commands her embodiment to gesture and these gestures take longer than the issuance of the command, similar to navigation commands. When the animation act is

completed, cues are sent through the vision modality to sequence Patofil's actions. However, there are also some animations which are translated to visible gestures in the world instantaneously and Patofil does not require notification about their completion. The animations which do require notification of completion in the world are:

- P_Reverent2_Over: Patofil has finished performing the animation of being reverent. Reverent2 is different from Reverent, where Reverent is being ordinarily reverent and takes 2 seconds for performing. On the other hand, Reverent2 is an animation that takes up to 10 seconds.
- P_Peeking_Over: Patofil has finished performing the act of peeking.
- P_Look_Around_Over: Patofil has finished performing the act of looking around

4. Events in World Cues: Some events in the world occur independent of Patofil's actions. These are communicated to Patofil through her vision and hearing modalities. These events are triggered by the participant's actions or because of the actions of other agents or objects in the world. These event cues are:

   - Triggers/Cues communicated through vision modality
     (a) U_Fidgeting: Trigger that the user/participant is moving and fidgeting
     (b) U_Hit_Whisp: Trigger that the user/participant has successfully hit a whisp
     (c) U_Playing_With_Whisp: Trigger that the user/participant is playing with whisps
     (d) Whisps_Coming: Trigger/Cue that the Whisps have started coming onto the mound in the world.
     (e) P_Bat_Whisp: Trigger that Patofil successfully batted a whisp.
   - Triggers/Cues communicated through hearing modality
     (a) U_Speaking: Trigger that the user/participant is speaking (through microphone input)
     (b) Gong_Sounds: Trigger that the gongs have sounded in the world. Communicated through the hearing modality

These triggers/cues allow Patofil to reason and use her knowledge for performing acts interactively and believably with the participant. Patofil has a set of high-level plans which represent the flow of the main story line. She also has a set of simpler plans which are used by the high-level, complex plans.

0.5.2 Handling Contingencies

What is a contingency?

Contingencies can be compared to interrupts. In a programming environment, an interrupt is something (It can be considered an error which is attributed to anomalous execution) which might stop the

normal execution of a program depending on the priorities of the main program and the interrupt in that particular context. Similarly, when the main plan is being executed by an agent (in this case, actor-agent Patofil), there are triggers (Listed in Section 0.5.1) which can be sent by the world needing immediate attention. Some of these triggers activate contingency plans. One can think of the main plan as having the lowest priority amongst all of Patofil's plans. When a trigger activates a contingency plan, the contingency assumes higher priority than the main plan depending on the context (snare) of the script. There are triggers/cues which do not activate any contingency plans. They represent beliefs which Patofil adds to her knowledge layer for future use during the drama.

Suppose Patofil is on the mound in Act 3, Scene 1 of *The Trial The Trail* and she is encouraging the participant to do a silent vigil. If the participant tries to speak or fidget, Patofil interrupts her main plan and handles the contingency triggers. Whether the interrupted plan is completed depends on the situation at the end of handling the contingency. When the agent is handling a contingency, she is said to be in a state of contingency. A state of contingency can be at the beginning, middle or end of a complex act, or while performing a primitive act. As of now, we consider the former case when a contingency needs to be handled while the agent is executing a plan. When a contingency situation occurs, the agent has to interrupt the main plan, handle the contingency and return to the point in the plan where it had temporarily stopped.

Handling Single Level Contingencies

Single level contingencies occur when a contingency interrupts an active plan. It can be handled thus:

```
Check if currently active plan is a valid context for present contingency
If Yes
        - Memorize present state in the currently active plan
        - Pause timers currently running in active plan
        - Handle contingency by invoking a contingency handler plan
        - Recall remembered state to restart previously active plan
        - Restart paused timers
```

If the contingency is not a valid situation for the present context of the script then the agent is not required to handle the contingency and it is ignored. The agent continues with the rest of her plans. Patofil's contingency handling plans are in SNePSLOG and they are presented and explained in Section 0.6.3.

0.5.3 Priorities

As Ismail observed *Cassie should be capable of talking about her own actions, her intentions, and the relative priorities of her different acts.* [14] Similarly with Patofil, Filopat and all the other agents

who will be actors in the interactive drama *The Trial The Trail*. The basic recurring theme is that interrupt handling involves the definition of priorities among acts (or goals) [14]. The main plans are of the lowest priority when they are executed by the agent. When a contingency occurs in the world, the contingency plan is at a higher priority with reference to the main, high-level plan. There are cases where the contingency will have to be ignored because it does not have higher priority than the main plan(s) in that context.

For example, in the first scene, when Patofil is doing a vigil and trying to persuade the participant to follow her instructions, if the participant talks or fidgets (both being contingencies) Patofil's highest priority will be to admonish the participant and tell him to keep quiet/stop fidgeting as is appropriate. But as the drama progresses and snare 1 is completed, the participant can speak and move around without his actions being considered contingencies by Patofil. Thus, priorities of certain plans (contingency or otherwise) change over time according to their context.

In the present implementation, Patofil does *not* have an explicit representation of relative priorities between acts. Instead, she checks for the context of each wordly trigger/cue and performs an act only if the context is valid. This method is not efficient in the sense that a lot of checking is done at every level of her performance to ensure that acts are being executed in context and she is handling the recent most trigger/cue that has come in. For example: In the Snare 2, Playing with whisps, Patofil has a number of conditions to check for each trigger that comes in (validating the context). The *User hit whisp*, *User hit whisp timer elapsed* and *Remind of Danger timer elapsed* events can occur at the same time. In such a case, it would be useful for Patofil to have knowledge about priorities and which trigger to ignore and which cue to pay attention to.

0.6 Implementing Patofil

0.6.1 The Knowledge Layer

The knowledge layer is the top-most layer of the GLAIR architecture. This layer contains Patofil's goals, plans, beliefs, policies and other entities which enable her to make decisions and execute actions. The Knowledge Layer (KL) contains the SNePSLOG representation of the script. SNePSLOG is a logic interface to SNePS and is easier to write, debug and comprehend than SNePSUL. At the KL, two levels of acts are represented. They are the primitive acts and complex acts, which are represented in SNeRE (SNePS Rational Engine). Primitive acts are ones that are performed by the agent "subconsciously"; it can perform them, but cannot reason about how it does. Complex acts are composed of primitive acts or other complex acts. The subsequent sections discuss Patofil's primitive and complex acts which she requires to have knowledge of for Act 3, Scene 1 of the *The Trial The Trail*.

Primitive Acts for Patofil

Primitive acts are executed by Patofil subconsciously. These acts are used by Patofil at her knowledge layer, and are defined at lower layers of GLAIR, in the PML. These primitive acts are:

- *store*: store the name of the currently executing plan in working memory.
- *memorize*: store the contents of the working memory in short term memory.
- *recall*: retrieve the contents of the short term memory.
- *say*: communicate speech string through speech modality (through socket in this case) for Patofil's embodiment to utter the speech string.
- *animate*: communicate animation command through animation modality
- *navigate*: communicate navigation command through navigation modality
- *mood*: communicate mood command through mood modality
- *start-timer*: start a specific timer with specified duration
- *pause-timer*: pause the specified timer
- *restart-timer*: restart the specified timer
- *stop-timer*: stop the specified timer

Simple and Complex plans for Patofil

Plans can be classified as *simple* plans and *complex* plans. Simple plans use the ActPlan structure to specify a plan for performing an act. The simple plans specified for Patofil use **performAct**. **performAct** is a unit of acting which uses an agent's modalities, mostly all of the modalities, sometimes a subset of them. These acting modalities are speaking (speech modality), gesturing (animation modality), navigating (movement modality), changing facial expressions (mood modality) and setting timers (timer modality). Patofil has a wide repertoire of simple plans which she combines to form complex plans. There are a set of simple and complex plans defined for every snare of the script. The complex plans use built-in constructs like:

- snsequence: A built-in construct that enables sequencing of acts
- do-one: A built-in construct that takes a set of acts and does one of them randomly
- do-all: A built-in construct that takes a set of acts and performs all of them, in random sequence

There are also constructs which have been defined by the agent designer like csequence, snsequence_5 etc. which make use of the built-in constructs in SNePS.

0.6.2 Snare 1: The Vigil

The implementation of Patofil's knowledge layer is explained by outlining two snares (The Vigil and Play with Whisps) and the corresponding representation in SNePSLOG. Patofil stores knowledge about her current acts, position of the participant and herself etc.

In SNePSLOG, frames are defined which are used while translating SNePSLOG representation to internal SNePSUL. The reason we use SNePSLOG is because it's a logical interface to SNePS and is easier to program and comprehend. Apart from the built-in frames, the following are defined using *define-frame* and used by Patofil:

```
;;; Frames for enabling snsequencing of 2 or more acts
define-frame snsequence_2 (action object1 object2)
define-frame snsequence_3 (action object1 object2 object3)
define-frame snsequence_4 (action object1 object2 object3 object4)
define-frame snsequence_5 (action object1 object2 object3 object4 object5)
define-frame snsequence_6 (action object1 object2 object3 object4 object5 object6)
define-frame snsequence_7 (action object1 object2 object3 object4 object5 object6 object7)
define-frame snsequence_8 (action object1 object2 object3 object4 object5 object6 object7
                                  object8)

;;; Frames for primitive actions
;;; say(x): speech modality communicates x to agent embodiment
define-frame say (action object1)

;;; navigate(x): navigation modality communicates x to agent embodiment
define-frame navigate (action object1)

;;; mood(x): mood modality communicates x to agent embodiment
define-frame mood (action object1)

;;; animate(x): animate modality communicates x to agent embodiment
define-frame animate (action object1)

;;; vision(x): vision modality communicates x to agent embodiment
define-frame vision (action object1)

;;; hearing(x): hearing modality communicates x to agent embodiment
```

```
define-frame hearing (action object1)

;;; InformSM(x): a stage-director modality which communicates
;;; x to the world (eventually will be a connection to the stage-manager)
define-frame InformSM (action object1)

;;; store(x): store the contents of x in working memory
define-frame store (action object1)

;;; memorize(x): memorize the contents x by transferring it to short-term memory
define-frame memorize (action mesg1)

;;; recall(x): recall contents of short-term memory
define-frame recall  (action mesg2)

;;;; Frames for timer actions
;;; start_timer(x,y): start timer with name x and for duration y
define-frame start_timer (action name duration)

;;; restart_timer(x): reset the timer with name x
define-frame restart_timer (action name)

;;; stop_timer(x): stop the timer with name x
define-frame stop_timer (action name)

;;; problem(x,y,z): In x, more of y is needed for act z
define-frame problem (nil situation needMore act)

;;; Frames for complex acts
;;; performAct(a,b,c,d): say speech a, do animation b, move with
;;; navigation c and have a mood d
define-frame performAct (action speech anim navig mood)

;;; csequence(a,b): csequence takes an act a, and a set of beliefs b,
;;; first it believes the arguments of b (which are whendo's),
;;; and then performs the act a.
define-frame csequence (action object2 object1)
```

```
;;; snare(x): x is the snare Patofil is currently in
define-frame snare (nil snareName)

;;; timerState(x, y): Timer x is in state y
define-frame timerState (nil member state)

;;; eventInWorld(x, y): An event y has occurred in the world involving agent x
define-frame eventInWorld (nil agent event)

;;; Location(a, x): Agent/Object a is in location x.
define-frame Location (nil agent location)

;;; subLocation(a, x): agent a is at location x
;;; (different from the Location frame because these positions do not
;;; need to be mutually exclusive from the other Locations)
define-frame subLocation (nil agent sublocation)

;;; said(x,y): Agent x completed saying y
define-frame said(nil agent speech)

;;; lineToSay(x,y) : Agent Patofil believes that line x is to be said for the act y
define-frame lineToSay (nil speechLine actName)

;;; alert(x, y, z): alerts agent-designer that x has happened with y in the act z
define-frame alert (action object1 object2 object3)

;;; animated(x) : Agent Patofil completed performing animation x
define-frame animated(nil agent animation)

;;; ParticipantAction(a, x): Agent a has done the act x
define-frame ParticipantAction(nil agent act)

;;; Frame for a dummy do act
;;; do(x) will perform the action x
define-frame do (nil action)
```

```
;;; Frame for adding knowledge
;;; addToKL(x): add to knowledge layer the statement x
define-frame addToKL (action object1)
```

These frames are attached to primactions/functions which are defined in the PML. The actions are attached using *attach-primaction*.

```
^^
;;; attach-primactions to function definitions
(attach-primaction
        believe believe
        disbelieve disbelieve
        withall withall
        withsome withsome
        achieve achieve
        snsequence snsequence
        sniterate sniterate
        do-all do-all
        do-one do-one
        snif snif
        snsequence_2 snsequence
        snsequence_3 snsequence
        snsequence_4 snsequence
        snsequence_5 snsequence
        snsequence_6 snsequence
        csequence snsequence
        say sayPrimFun
        navigate navigatePrimFun
        vision visionPrimFun
        hearing hearingPrimFun
        mood moodPrimfun
        animate animatePrimFun
        InformSM SMfun
        memorize memorizefun
        recall recallfun
        store storefun
        start_timer start-timer
        restart_timer restart-timer
```

```
        stop_timer stop-timer
        performAct performActAll
        alert alert
        addToKL addToKL)

^^
```

Patofil's beliefs, representation for acts and policies are discussed in detail in subsequent sections. One amongst Patofil's beliefs is that she is in any **one** snare at any given point of time. This knowledge helps her in determining the relative priority between different cues coming in from the world and she uses the context of the script (the snare being performed) for sequencing her acts and interacting with the participant. The Snare 1 is **TheVigil** which has 2 snare sub-plans: **vigil1**, **vigil2**. The **play with whisps** and **jumpOnRocks** are two other Snares which Patofil can transition to, from **TheVigil** snare.

```
andor(1,1){snare(vigil1),
           snare(vigil2),
           snare(jumpOnRocks),
           snare(whisps)}.
```

Patofil keeps track of the Participant's position on the mound through visual cues she receives from the world. She/participant can be at only **one** position at a given point of time. Patofil herself keeps track of her positions because she can be at any one position at any point of time.

```
;;; Location of User/Participant and Patofil can be one of the following:
;;;   On mound
;;;   At Edge of Mound
;;;   Off Mound

;;; User represents the Participant/User
andor(1,1){Location(User, On_Mound),
           Location(User, Off_Mound),
           Location(User, At_Edge)}.

;;; I represents Patofil
andor(1,1){Location(I, On_Mound),
           Location(I, Off_Mound),
           Location(I, At_Edge)}.
```

Snare 1 consists of smaller acts, Vigil 1 and Vigil 2.

- Vigil 1: The first section of Snare 1 consists of Vigil 1. In the first stage, Patofil is on the mound along with the participant. The plan is executed in sequence where Patofil tells the participant not to talk and join her in doing a silent vigil. After a few moments she gets bored and breaks out in a giggle showing lack of seriousness. This act-plan can be interrupted by a number of other triggers which come in from the world. The participant can speak/fidget. He can move towards the edge of the mound or go off it. Depending on the particular scenario, Patofil reacts accordingly to the participant. If he fidgets during the silent vigil, she admonishes him.

  Whenever Patofil (I) is on mound, believe that whenever User is on mound, do all : start the vigil timer for 70 seconds, store in working memory the act that she is performing, believe the part of the act (snare sub-plan) she is currently in, inform the stage manager about the snare currently being performed and perform the act vigilOnePlan

```
;;; Vigil 1
;;; When I (Patofil) is on the mound
;;;    and when the Participant is on the mound
;;;         do the following:
;;;             1. Start the vigil timer for 70 seconds
;;;             2. Store the snare name
;;;             3. Believe that currently executing snare is vigil1
;;;             3. Inform Stage Manager (SM) about entry into snare_1
;;;             2. Proceed with first plan
whendo(Location(I, On_Mound),
            believe(whendo(Location(User, On_Mound),
                    do-all({start_timer(vigil_Timer, 70),
                            store(Meditating),
                            believe(snare(vigil1)),
                            InformSM(Snare_1),
                            do(vigilOnePlan)}))).
```

  For performing act vigilOnePlan, do meditation acts in sequence. csequence is a construct defined by the agent designer and takes 2 arguments. The first is an act. The second is a believe on a set of policies (whendo's) which sequence Patofil's acts. Other constructs can be used but this provides a compact way of representing Patofil's plans. After the first line of speech, Patofil waits for notification about her position in the world. When she is in front of the user, she continues with her meditative acts.

```
;;; vigilOnePlan consists of
```

```
;;;     1. Be silent and reverent
;;;     2. Stay with Participant while telling him the importance of being
;;;        meditative and quiet.
ActPlan(do(vigilOnePlan),
        csequence(do(MeditateAct1),
                    believe({whendo(subLocation(I, In_Front_Of_User),
                                do(MeditateAct2)),
                             whendo(said(I,P1_StandStraight),
                                do(MeditateAct3)),
                             whendo(said(I,P1_ContemplateTheInf),
                                do(MeditateAct4))}))).

;;; Each performAct takes a set of arguments, which represent a command for
;;; each modality. performAct( Speech, Animation, Navigation, Mood)
ActPlan(do(MeditateAct1),
          performAct(P1_NoTalking, Reverent, In_Front_Of_User, Happy)).

ActPlan(do(MeditateAct2),
          performAct(P1_StandStraight, Reverent, In_Front_Of_User, Happy)).

ActPlan(do(MeditateAct3),
          performAct(P1_ContemplateTheInf, Reverent, Stay_With_User, Eyes_Closed)).

ActPlan(do(MeditateAct4),
          performAct(P1_DontLeaveTheHill, Reverent, Stand_Still, Eyes_Closed)).
```

- Vigil 2: Follows the Vigil 1 act. Consists mainly of the Giggle-Peek-Meditate cycle which provides the user with opportunity to meditate with Patofil, only to realize that she is giggling through the meditation and this displays her frivolous nature. Contingencies can occur during this act and the cycle is restarted after the contingencies are handled. Contingencies are listed in 0.6.3.

  From Vigil1 (being meditative), Patofil's next act is Vigil2, where she stops being quiet and meditative. This act is cued by the last speech Line of Vigil1 being uttered. Patofil checks if the vigil timer has elapsed, in which case she should not start her giggle-peek-meditate cycle. In the event the timer has not elapsed, Patofil starts her cycle of giggling-peeking-meditating by doing the act meditating. This cycle aims to show her lack of seriousness for the vigil. In this act, during the meditating part of the cycle, Patofil checks if the User is fidgeting. If so, she admonishes him. The User_fidgeting check is made using the vision modality.

```
;;; Vigil 2

;;; Continuing from Vigil 1 to Vigil 2
;;; Cue is hearing the last line of Vigil 1: P1_DontLeaveTheHill
;;; Patofil follows a Giggle-Peek-Meditate cycle that is very important
;;; to this part of the snare. She follows this cycle repeatedly, unless
;;; a contingency has to be handled, or the vigil timer expires.

;;; A check for User_Fidgeting is made by explicitly sending a Check_U_Fidget command
;;; across the vision socket. This is equivalent to Patofil actively seeking visual
;;; cues to check if the User is fidgeting.

whendo(said(I,P1_DontLeaveTheHill),
        snsequence(believe(snare(vigil2)),
                   snif({if(timerState(vigil_Timer,elapsed),
                                    do(nothing)),
                         else(snsequence_3(do(Meditating),
                                           store(GigglePeekMeditate),
                                           vision(Check_U_Fidget)))})))).
```

Meditating and Peeking actPlans use Silence speech strings which do not play any pre recorded voice.

```
;;; ActPlan to be meditative
ActPlan(do(Meditating),
        performAct(Silence1, Reverent2, Stand_Still, Eyes_Closed)).

;;; ActPlan to do peeking
ActPlan(do(Peek),
        performAct(Silence3, Peeking, Face_User, Happy)).

;;; A do-nothing, no-op
ActPlan(do(nothing),
        say(Nothing)).
```

Whenever Patofil (I) have finished being reverent, believe that whenever user is still on mound, check if vigil timer has elapsed. If yes, do nothing, else store the act being performed and continue the giggle-peek-meditate cycle by peeking

```
;;; After meditating, do a peeking action
wheneverdo(animated(I, Reverent2_Over),
        believe(wheneverdo(Location(User, On_Mound),
                        snif({if(timerState(vigil_Timer, elapsed),
                                                do(nothing)),
                                else(snsequence_3(do(Peek),
                                                store(GigglePeekMeditate),
                                                disbelieve(animated(I, Reverent2_Over))
                                    ))}))))).
```

After I have finished peeking, check vigil timer. If elapsed, do nothing, else store act name and continue giggle-peek-meditate cycle by giggling

```
;;; After peeking, giggle to illustrate that Patofil considers the vigil task
;;; frivolous
wheneverdo(animated(I, Peeking_Over),
        believe(wheneverdo(Location(User, On_Mound),
                        snif({if(timerState(vigil_Timer, elapsed),
                                                do(nothing)),
                                else(snsequence_3(do(Giggling),
                                                store(GigglePeekMeditate),
                                                disbelieve(animated(I,
                                                        Peeking_Over))))}))))).

;;; Giggling involves doing one of many alternative speechLines and corresponding
;;; animation, navigation and mood. The do-all, do-one combination provides a
;;; powerful operator for randomizing the choice every time the act is to be performed,
;;; thus increasing the number of unique acts which can be performed by Patofil.
ActPlan(do(Giggling),
        do-all({do-one({say(P1_Giggle1),
                        say(P1_Giggle2),
                        say(P1_Giggle3)}),
                animate(Laugh),
                mood(Happy),
                navigate(Face_User)})).

;;; Since saying a giggle is randomized, irrespective of the giggle chosen,
;;; it is believed that P1_Giggled.
```

```
{said(I, P1_Giggle1), said(I, P1_Giggle2), said(I, P1_Giggle3)} => {said(I, P1_Giggled)}.
```

Continue the cycle. Also do a check if User is fidgeting since meditating act is going to be performed

```
;;; After giggling, continue checking for user fidgeting and continue meditating
;;; (restart the cycle)
wheneverdo(said(I,P1_Giggled),
        believe(wheneverdo(Location(User, On_Mound),
                        snsequence_4(do-all({disbelieve(said(I, P1_Giggle1)),
                                             disbelieve(said(I, P1_Giggle2)),
                                             disbelieve(said(I, P1_Giggle3)),
                                             disbelieve(said(I,P1_Giggled))}),
                                     vision(Check_U_Fidget),
                                     store(GigglePeekMeditate),
                                     do(Meditating))))).

;;; If a contingency occurs, memorize and recall ensure that the present/active
;;; plan's context is saved and retrieved. The recall function retrieves
;;; the message that Patofil said last (In this case, the ``GigglePeekMeditate''
;;; which she stored is retrieved, and she believes she ``said'' this. This belief
;;; triggers the following on return from contingency:
wheneverdo(said(I,GigglePeekMeditate),
        believe(wheneverdo(Location(User, On_Mound),
                        snif({if(timerState(vigil_Timer, elapsed),
                                    do(nothing)),
                              else(snsequence(disbelieve(said(I,GigglePeekMeditate)),
                                              animate(Reverent2)))}))))).
```

0.6.3 Contingency Plans and *Co-designing* aspects for Patofil

While performing Vigil 1 and Vigil 2 acts (constituting Snare 1), a number of contingency situations can occur. The User can fidget, speak, move around, go off the mound etc. Based on the participant's actions, Patofil reacts and responds in various ways. These contingencies can occur at any point of execution of the act. Patofil memorizes her present context of the drama, and handles the contingency. She also stops/pauses any running timers based on the context of the contingency situation. The timers are restarted when she has handled the contingency and is moving on to the rest of her plans. She recalls the state she was interrupted in, to enable her to proceed with the rest of her acts. The following contingency triggers/cues can come in through the world when Patofil is performing Snare 1 (The Vigil):

- User Fidgeting: User starts moving around and Patofil admonishes him for fidgeting.
- User Speaking: User speaks, detected by Patofil (through hearing modality) and she admonishes user.
- User At Edge of Mound: User navigates to edge of mound and Patofil entreats them to come back. This contingency is more complicated than others, because there are three transitions which can occur. A transition is a context-based move to other plans in order to continue with the story-line. After the User is at edge of the mound, there are three events that can occur:
  1. User returns On Mound: Patofil makes a transition to a plan which praises the user for coming back and continues with her Giggle-Peek-Meditate cycle.
  2. User ignores Patofil and goes off mound: Patofil keeps calling the user back, but eventually follows him off the mound.
  3. Vigil timer expires: Patofil decides that the user is at edge and bored and starts playing with the Whisps to interest the user.

Patofil has to handle these contingency situations intelligently and believably. Co-designing aspects have been implemented for Patofil which also helps her handle her contingency situations better. In subsequent sections, Patofil's plans for handling User Speaking, User Fidgeting and User At Edge are given in SNePSLOG. Before proceeding, let us understand what co-designing means.

A co-designing agent is an intelligent agent who is capable of participating in its own design. This section provides a summary of our discussions during the course of the seminar on co-designing agents, which aim at describing a co-designing agent, discussing the features and capabilities such an agent is endowed with and the feasibility of developing and implementing a co-designing agent with available systems and tools. Before getting into what makes Patofil a co-designing agent, let us have a look at the capabilities of co-designing agents in general. A co-designing agent can report and answer questions during three stages:

- Before being on the job: The agent answers questions about likely situations that might be encountered and knows how to handle these situations, or notifies the agent designer in case of a lack of knowledge or a contradiction in its knowledge base.
- While on the job: The agent answers questions about its reasoning and actions, why it is doing what it is doing, why it took a particular decision etc.
- After the job: The agent is capable of reporting on all its previous actions and the motivations behind these decisions. The agent reports on problems encountered and why it did or did not perform a certain action.

The goal of the seminar group was to discuss these aspects of a co-designing agent and endeavor to come up with a prototype which implements some or all of the above features in an agent. This section discusses the implementation of co-designing aspects for Patofil while she is **on the job** and **after the job**. When Patofil is playing the role of an *actor-agent* in an interactive drama, situations arise where she determines a lack of knowledge in handling a known situation or discovers a situation not known apriori. In such cases, the agent designer is notified with appropriate information regarding the situation and the designer can subsequently *update* Patofil's knowledge base with new information or *provide access* to a library or file with functions/plans. A co-designing agent is an agent capable of *learning* by retaining new information. Design iterations involve gaining new knowledge which is stored and re-used in future situations.

There is some relevant literature which discusses designing agents which are capable of explaining their own actions [17] or agents which modify their internal processes and use an *Agent factory* [18]. These papers discuss the design of agents that are capable of remembering their actions and answer questions posed by the agent designer after the job. In [18], A self-modifying agent has been described as an agent which is capable of modifying its internal states, and tasks it has been assigned. The paper describes this process as a *re-design* which is carried out by the agent either through an internal self-modification process (reflective) or through an external self-modification process (agent factory). The agent factory can be likened to an agent-designer who has access to the agent's knowledge layer and can modify or update the contents. Patofil provides such a tool for the agent designer who can add new information to Patofil's knowledge base and she retains this information for her subsequent acts.

[17] discusses Debrief: an explanation capability designed for SOAR agents determines the motivation for decisions by recalling the context in which they were made and determining factors for these decisions. Debrief learns to recognize similar situations where the same decision will be made for the same reasons. A co-designing agent has to incorporate a system with similar features as Debrief. [17] provided background knowledge for endowing agents with the capability to explain their actions/decisions to their agent designer. The SOAR agents implement event memory which is used by Debrief for reconstructing past sequence of actions so that they can be examined. Similarly, actor-agent Patofil stores knowledge about the acts she is currently performing and she uses this knowledge to determine the context of her acts. She also assigns context to the incoming triggers/cues from the world, some of which are active only in certain sections/contexts of the script. Patofil retains new knowledge given by the agent designer and this enables her to use this new knowledge when similar situations occur in the future.

Making a start

How does one go about developing an interactive, intelligent agent who is aware of itself and its capabilities to such an extent that it also participates in its own design? During discussions on building

co-designing agents, one common thought was that the agent should possess the capability of interacting and communicating with the agent designer: either in natural language or in the language of the underlying implementation. This interaction can take different forms. The interaction can be one-way or two-way. For example, *producing reports* or *signaling alerts* is a one-way interaction from the agent to its designer and *question-answering* is a two-way interaction between the agent and its designer. Both are explored in subsequent sections.

Implementing Co-designing features for Patofil

Patofil follows a pre-written script which is represented as beliefs, act-plans and policies in her knowledge layer. While performing an act, Patofil communicates with the world and the participant through *modalities*. The speech modality enables Patofil to say her speech lines in the world. Each act consists of lines which Patofil utters, along with corresponding animation, navigation and mood commands. The lines assigned to Patofil for each act are based on the pre-written script she has to perform. While performing her acts, Patofil might discover that she does not have enough lines to say for a particular act. In such a case, the agent designer has to be notified.

A co-designing aspect implemented for Patofil gives her the ability to **alert** the agent-designer when she has run out of lines to say for a particular act. Consider the act she performs when the trigger/cue User_Fidgeting comes in from the world. Patofil has to react to this *contingency* situation (a situation that might or might not occur and depends on the participant's actions or independent events in the world) and has a store of lines which she uses to reprimand the user for moving during the silent vigil. Each of these speech lines are associated with the act they are meant for. Patofil also has a belief about whether her speech lines have already been uttered or not.

The SNePSLOG representation for frames and assertions/beliefs for implementing this co-designing aspect is:

```
;;; lineToSay(x,y) : line x is for the act y
define-frame lineToSay (nil speechLine actName)

;;; Lines for the act Fidgeting
lineToSay(P1_StopFidgeting, Fidgeting).
lineToSay(P1_YoureMeantToStayS, Fidgeting).
lineToSay(P1_Concentrate, Fidgeting).
lineToSay(P1_Meditate, Fidgeting).

;;; Lines for the act Speaking
```

```
lineToSay(P1_Shhh, Speaking).
lineToSay(P1_ItsASilentVigil, Speaking).
lineToSay(P1_BeQuiet, Speaking).

;;; Lines for the act of praising the user when he returns on mound
lineToSay(P1_LetsMeditate, PraiseUser).
lineToSay(P1_OhGood, PraiseUser).
lineToSay(P1_ThatsBetter, PraiseUser).
lineToSay(P1_WiseChoice, PraiseUser).
```

Patofil has to have knowledge about the lines she has finished uttering and lines she has yet to utter. This belief of having finished uttering speech lines is constructed and added to her knowledge base when she has *heard* (through the hearing modality) her speech lines being uttered by her embodiment in the world. It is necessary for her to know which lines she has not yet uttered. This can be done by asserting that she has *not said* the lines for *Fidgeting* or other acts, at the beginning of the drama. The reason Patofil is designed to not repeat her lines is because she randomly chooses lines to say for a particular act and if she repeats herself, she loses credibility as a believable agent. The easiest way to seem mechanical and machine-like is by repeating yourself. Therefore, Patofil is aware of lines she has said previously and she does not repeat them. The randomization is still implemented to allow a larger number of unique acts to be generated.

```
;;; said(x,y): Agent x completed saying y
define-frame said(nil agent speech)

;;; Assert the lines not yet uttered/said
;;; For user fidgeting act
~said(I, P1_StopFidgeting).
~said(I, P1_YoureMeantToStayS).
~said(I, P1_Concentrate).
~said(I, P1_Meditate).

;;; For user speaking act
~said(I, P1_Shhh).
~said(I, P1_ItsASilentVigil).
~said(I, P1_BeQuiet).

;;; For praise the user act
~said(I, P1_LetsMeditate).
```

```
~said(I, P1_OhGood).
~said(I, P1_ThatsBetter).
~said(I, P1_WiseChoice).
```

The act plan for reprimanding the user for fidgeting uses the **withsome** construct to choose an unsaid line to utter. The **withsome** binds a variable **?lines** to those lines which satisfy the condition(s) in the **andor**. If this variable are bound successfully, then the value is used by the **say** primaction. The **say** is used by Patofil to utter lines in the world. There are two condition(s) in the **andor**. Firstly, the line has to belong to the particular act: namely *Fidgeting* or *Speaking* or *PraiseUser* and secondly, the line has not yet been uttered: represented by the *said* belief. If there is no such value which binds to *?lines* and satisfies the **andor**, then the agent designer is alerted.

PMLa: Definition of the **say** primaction

```
;;; Individual say
(define-primaction sayPrimFun (object1)
  (writeToSocket object1 speechsocket))
```

SNePSLOG representation of ActPlans and policies for this co-designing aspect:

```
;;; Trigger coming in through vision modality is User_Fidgeting
;;; Do following:
;;; 1. Remember the main snare-subplan (Giggle-Peek-Meditate cycle) being executed
;;; 2. Invoke actplan to handle contingency
;;; 3. Recall main snare-subplan to restart Giggle-Peek-Meditate cycle
wheneverdo(ParticipantAction(User, Fidgeting),
                snif({if(snare(vigil1),
                      snsequence_6(do(memorize),
                                   stop_timer(vigil_Timer),
                                   do(ReprimandForFidgeting),
                                   do(recall),
                                   disbelieve(ParticipantAction(User, Fidgeting)),
                                   restart_timer(vigil_Timer))),
                      else(do(nothing))})).

ActPlan(do(ReprimandForFidgeting),
        do-all({withsome(?lines, andor(2,2){lineToSay(?lines, Fidgeting),
                                            ~said(I, ?lines)},
                        say(?lines),
```

```
                    alert(ranOutOf, SpeechLines, UserFidgeting)),
            do-one({mood(Happy), mood(Eyes_Closed)}),
            do-one({animate(Reverent), animate(Happy)}),
            do-one({navigate(Stay_With_User), navigate(In_Front_Of_User)})})).
```

Patofil should not repeat the same speech redundantly, hence a random choice (implemented as a do-one) is made between a set of speech lines which are pre-recorded. Patofil's reactions to the participants' actions should seem natural and believable. Using a do-one for each modality (apart from speech, a random choice can be made for the mood, the animation and the navigation) produces more than one way of reprimanding a participant. From the above example, there are 4*2*2*2 = 32 different ways in which Patofil can reprimand the participant without sounding repetitious. This structure is used in other actplans to increase the number of unique acts which can be generated.

The contingency plans for handling user fidgeting, user speaking, praise user use a similar structure. When the trigger/cue comes in from the world, Patofil memorizes her present act and handles the contingency. She also stops/pauses all timers active for the main plan. She recalls her previous act after handling the contingency and restarts the previously paused timers. Each ActPlan represents a proposition that doing a specified act involves performing the specified plan. The plans use a do-all and do-one combination which produces unique act sequences which are randomly chosen by Patofil.

```
;;; Handle User Speaking
;;; Trigger coming in through hearing modality is User_Speaking
wheneverdo(ParticipantAction(User, Speaking),
        snif({if(snare(vigil1),
                snsequence_6(do(memorize),
                             stop_timer(vigil_Timer),
                             do(ReprimandForSpeaking),
                             do(recall),
                             disbelieve(ParticipantAction(User, Speaking)),
                             restart_timer(vigil_Timer))),
             else(do(nothing))})).

;;; This actplan produces 3*2*2*2 = 24 unique acts
ActPlan(do(ReprimandForSpeaking),
        do-all({withsome(?lines, andor(2,2){lineToSay(?lines, Speaking),
                                            ~said(I, ?lines)},
                         say(?lines),
                         alert(ranOutOf, SpeechLines, UserSpeaking)),
                do-one({mood(Happy), mood(Eyes_Closed)}),
```

```
                do-one({animate(Reverent), animate(Happy)}),
                do-one({navigate(Stay_With_User), navigate(In_Front_Of_User)})})).
```

When the participant is at the edge of the mound, Patofil reprimands him for going to the edge. She also reprimands the participant when he goes off the mound and praises him if he comes back on the mound.

```
;;; Handle User going off mound or to edge
wheneverdo(Location(User, At_Edge),
        snif({if(snare(vigil2), snsequence_5(store(UserAtEdge),
                                             do(memorize),
                                             stop_timer(vigil_Timer),
                                             do(ReprimandForGoingToEdge),
                                             restart_timer(vigil_Timer))),
                   else(do(nothing))})).

ActPlan(do(ReprimandForGoingToEdge),
        csequence(do(Reprimand1),
                  believe({whendo(said(I,P1_WhereAreYouGoing),
                                  snif({if(Location(User, At_Edge),
                                        do(Reprimand2)),
                                        else
                                        (do-all({do(recall),
                                                 restart_timer(vigil_Timer)}))})),
                           whendo(said(I,P1_DontGoOffTheMound),
                                  snif({if(Location(User, At_Edge),
                                        do(Reprimand3)),
                                        else
                                        (do-all({do(recall),
                                                 restart_timer(vigil_Timer)}))})),
                           whendo(said(I,P1_ItsNotSafeOutTher),
                                  snif({if(Location(User, At_Edge),
                                        do(Reprimand4)),
                                        else
                                        (do-all({do(recall),
                                                 restart_timer(vigil_Timer)}))})),
                           whendo(said(I,P1_WereMeantToStayOn),
                                   snif({if(Location(User, At_Edge),
```

```
                                        do(Reprimand5)),
                                       else
                                       (do-all({do(recall),
                                                restart_timer(vigil_Timer)}))})),
                            whendo(said(I,P1_FilopatWillBeSoMa),
                                   believe(subLocation(User,Still_At_Edge)))})))).

ActPlan(do(Reprimand1),
        performAct(P1_WhereAreYouGoing, Agitated, Near_Edge_By_User, Neutral)).

ActPlan(do(Reprimand2),
        performAct(P1_DontGoOffTheMound, Agitated, Stand_Still, Neutral)).

ActPlan(do(Reprimand3),
        performAct(P1_ItsNotSafeOutTher, Agitated, Stand_Still, Mad)).

ActPlan(do(Reprimand4),
        performAct(P1_WereMeantToStayOn, Agitated, Stand_Still, Mad)).

ActPlan(do(Reprimand5),
        performAct(P1_FilopatWillBeSoMa, Agitated, Stand_Still, Mad)).

ActPlan(do(Reprimand6),
        performAct(P1_WhereAreYouGoing, Agitated, Stand_Still, Neutral)).

{said(I, P1_WhereAreYouGoing), said(I, P1_DontGoOffTheMound),
 said(I, P1_ItsNotSafeOutTher), said(I, P1_WereMeantToStayOn),
 said(I, P1_FilopatWillBeSoMa)} => {said(I, LinesForCallingBack)}.

;;; When user comes back on mound, praise the user
lineToSay(P1_LetsMeditate, PraiseUser).
lineToSay(P1_OhGood, PraiseUser).
lineToSay(P1_ThatsBetter, PraiseUser).
lineToSay(P1_WiseChoice, PraiseUser).

~said(I, P1_LetsMeditate).
~said(I, P1_OhGood).
```

```
~said(I, P1_ThatsBetter).
~said(I, P1_WiseChoice).

ActPlan(do(PraiseForComingBack),
        do-all({withsome(?lines, andor(2,2){lineToSay(?lines, PraiseUser),
                                            ~said(I, ?lines)},
                         say(?lines),
                         alert(ranOutOf,
                               SpeechLines,
                               PraiseUserForComingBack)),
                do-one({animate(Reverent),
                        animate(Agitated)}),
                mood(Happy),
                navigate(Mound_Center)})).

{said(I, P1_LetsMeditate), said(I, P1_OhGood),
 said(I, P1_ThatsBetter),  said(I, P1_WiseChoice)} => {said(I, GigglePeekMeditate)}.

wheneverdo(said(I, LinesForCallingBack),
        believe(wheneverdo(Location(User, On_Mound),
                         do(PraiseForComingBack)))).

;;; To get back to admonishing if User is still at edge
wheneverdo(subLocation(User,Still_At_Edge),
        snsequence_3(disbelieve(subLocation(User,Still_At_Edge)),
                     do(ReprimandAgain),
                     believe(said(I,ReprimandedAgain)))).

ActPlan(do(ReprimandAgain),
        do-one({do(Reprimand6),
                do(Reprimand2),
                do(Reprimand3)})).

wheneverdo(Location(User, On_Mound),
        believe(wheneverdo(said(I,ReprimandedAgain),
                           do(recall)))).
```

The above plans provide Patofil with the knowledge to handle contingency situations for Snare 1

: The Vigil. Another co-designing feature implemented is the **alert** function which is a primaction defined at the PMLa layer. The alert provides Patofil with a tool to inform the agent designer about a problematic situation that has occurred. The primaction takes three arguments.

- The type of event that occurred. (ranOutOf)
- The object the event involved. (speechLines, animations, navigation commands, moods).
- The act involved. (UserFidgeting, UserSpeaking, PraiseUser).

```
;;; Primaction for alert defined at the PMLa
(define-primaction alert (object1 object2 object3)
        (print (format nil "I'm sorry! I ~a ~a for the act ~a" object1 object2 object3))
        #!((perform (build action believe
                    object1 (build situation ~object1 needMore ~object2 act ~object3)))))
```

Apart from printing out on an alert message on standard output, the alert primaction also constructs a belief that allows the agent-designer to access Patofil's knowledge layer *after the job* to list the problems she faced.

```
;;; problem(x,y,z): In x, more of y is needed for act z
define-frame problem (nil situation needMore act)
```

The agent designer can add new knowledge to Patofil's knowledge layer while she is *on the job*. When the agent designer observes that Patofil has run out of speechLines for a particular act, he/she can add new speechLines and the necessary accompanying knowledge to enable Patofil to continue with her act. Consider the act *User Fidgeting*. Initially, Patofil has a store of 4 speechLines to choose from. She reprimands the participant for fidgeting until she runs out of lines to say. At this point, the agent designer can add more lines for the act *Fidgeting* and she will be able to continue reprimanding the participant if he/she continues to fidget.

Patofil prints out the messages she wants to communicate with the agent designer on standard output. The agent designer can interact with Patofil when she is reasoning and acting at the *snepslog* prompt, which is represented by ":". The agent designer can use the primaction **addToKL** which writes the new knowledge (in SNePSLOG) to a file on disk. The file is
/projects/robot/Ygdrasil/Agents/Patofil/SNePSLOG/Ver2/codesigning.snepslog.
This file is loaded by Patofil every time she starts her role as the actor-agent. Patofil remembers the information input by the agent designer by storing the new knowledge in the form or acts, beliefs or policies.

```
;;; Frame for adding knowledge
define-frame addToKL (action object1)
```

At the PMLa layer: *addToKL opens a file stream for writing and appends the contents of the existing file with knowledge that the agent designer has entered.* ***tell*** *is used to tell Patofil the new knowledge that the agent designer has entered.*

```
;;; Primaction to write snepslog code entered by agent-designer into a file
;;; Patofil loads this file in the main demo file
(define-primaction addToKL (object1)
  (let ((Kstring (format nil "~a" (first object1))))
     ;;; open stream for file to be written to
    (with-open-file (output-stream "/projects/robot/Ygdrasil/Agents/Patofil
                                    /SNePSLOG/Ver2/codesigning.snepslog"
                                   :direction :io
                                   :if-exists :append)
      ;;; tell Patofil the new knowledge
      (tell Kstring)
      ;;; write the new knowledge into file
      (write-line Kstring output-stream))))
```

In the following example, Patofil notifies the agent designer that she has run out of speechLines for the act *UserFidgeting*. The agent designer in turn gives Patofil a new line *P1_DontMoveAround* to say for this act. The agent designer also gives Patofil the knowledge that she has not yet uttered this line. When the *User_Fidgeting* trigger comes in again from the world, either in this act or in future acts, Patofil responds with the new speech line *P1_DontMoveAround*, and continues with the rest of her acts.

```
...
"I'm sorry! I (ranOutOf) (SpeechLines) for the act (UserFidgeting)"
...

: perform addToKL("lineToSay(P1_DontMoveAround, Fidgeting)")

 CPU time : 0.00

: perform  addToKL("~said(I, P1_DontMoveAround)")

 CPU time : 0.00
```

The **tell** interface used by the primaction *addToKL* adds the knowledge input by the agent-designer to Patofil's current knowledge base. However the file containing added knowledge has to be demo-ed every time Patofil's knowledge layer is loaded.

```
: demo "codesigning.snepslog"

File /projects/robot/Ygdrasil/Agents/Patofil/SNePSLOG/Ver2/codesigning.snepslog
is now the source of input.


 CPU time : 0.00

: lineToSay(P1_DontMoveAround, Fidgeting)

  lineToSay(P1_DontMoveAround,Fidgeting)

 CPU time : 0.00

: ~said(I, P1_DontMoveAround)

  ~said(I,P1_DontMoveAround)

 CPU time : 0.00

:
End of /projects/robot/Ygdrasil/Agents/Patofil/SNePSLOG/Ver2/codesigning.snepslog
demonstration.

 CPU time : 0.01
```

This knowledge is retained in her knowledge layer because it is contained in the file codesigning.snepslog which is loaded (demo-ed) every time before Patofil starts her role as an actor-agent. The next time the trigger/cue comes in from the world, Patofil uses her new knowledge: she reprimands the participant with the new speech Line. In the following example:

- *Messages/Cues from the world are received by Patofil and printed out on standard output. These messages are NOT enclosed in parantheses. The* ***User_Fidgeting*** *message is received from the world and printed as shown.*

- *Messages from Patofil to her embodiment/world are send across the modality sockets, and also printed out on standard output. These messages are enclosed in parantheses, as shown by* ***P1_DontMoveAround****,* ***Eyes_Closed****,* ***Reverent*** *and* ***In_Front_Of_User***

- *Other informational messages are printed out on standard output. They indicate memorizing, storing, recalling etc.*

```
:
User_Fidgeting
MEMORIZING: (GigglePeekMeditate)   ((GigglePeekMeditate))
(P1_DontMoveAround)
(Eyes_Closed)
(Reverent)
(In_Front_Of_User)
...
```

Apart from interacting with the actor-agent when she is *on the job*, the agent-designer can access her knowledge layer *after the job*. All the problems faced by Patofil can be listed out by asking her: **problem(?x, ?y, ?z)** at the snepslog prompt. This representation in Patofil's knowledge base about problems she encountered during her acts are useful for debugging and re-scripting purposes *after the job*. They also serve as a report-producing tool that the agent designer can use.

```
: problem(?x, ?y, ?z)?

  problem(ranOutOf,SpeechLines,UserFidgeting)
  problem(ranOutOf,SpeechLines,UserSpeaking)

 CPU time : 0.01
```

Improving the drama with User Tests

This co-designing aspect provides agent-designers and the script-writers with an agent who directly or indirectly contributes towards writing the script and accommodating participant requirements. Based on the whether Patofil runs out of lines for particular acts or other situations which occur, changes can be made to the script to accommodate a larger cross section of participants. In the case of Fidgeting, there might be very few users who fidget a lot and thus Patofil runs out of lines to reprimand them with. But with sufficient number of user tests, we will gain feedback on what changes to incorporate in the script, whether Patofil needs more lines for a particular act etc.

Other co-designing aspects

Now that an idea of what a co-designing agent constitutes has been discussed along with a simple prototype implementation of co-designing aspects for an actor-agent, future plans include the design and implementation of co-designing agents who are more powerful and interactive. The discussions in this section provide an idea of how to go about developing interactive co-designing agents who communicate with the agent designer. A co-designing agent will display self-awareness, intelligence and provide the agent designer with tools which help in its design. Consider Patofil, who can be endowed with more co-designing capabilities, like:

- Report on every act that she has performed and keep track of all contingency situations which interrupted higher level plans.
- Inform the agent designer if she is required to perform acts which conflict with each other.
- Enumerate situations which can occur and determine if she has plans for performing in these situations, if not notify the agent designer.

The seminar provided a useful platform to review present literature relevant to co-designing agents and to discuss, outline and understand the idea of a co-designing agent; how to go about building such an agent and why such an agent might be required. Apart from reading and discussing literature on agents, exercises in implementing co-designing aspects for Patofil: An MGLAIR agent in a virtual reality drama provided a good learning experience. This in turn has given rise to many more ideas which can be implemented in the future and which will endow Patofil and other actor-agents with the capability of interacting successfully with the agent designer, thus improving their level of believability and interactivity.

0.6.4 Transitions between Snare subplans

A transition is a passage from one form, state, style, or place to another. Transitions provide a smooth flow between the different acts in our drama. Initially, there were no clear-cut transitions defined between acts and Patofil flitted from one to another. But we have introduced context-based transition acts between the snares, which enable a participant to understand and sometimes anticipate the move to future acts in the drama. A transition is implemented as a short mid-act, or a sequence of one/more speech lines which prepare the participant for what is coming next. For example, the transitions in Snare 1 are:

- If user fidgets too much or is not meditating, Patofil makes a transition and starts playing with whisps

```
;;; Plans for Transition 1
```

```
;;; If user fidgets too much or is not meditating, Patofil starts playing with whisps

;;; Transition to Play With Whisps (Snare 2)
;;; Have a counter for checking when user has fidgeted beyond a threshold
;;; and perform Vigil1Transition1 plan
ActPlan(do(Vigil1Transition1),
        csequence(do(V1T1Plan1),
                  believe({whendo(said(I,P1_ImNotGoingToMedit),
                                  do(V1T1Plan2)),
                           whendo(said(I,P1_ButWhatCanWeDo),
                                  do(V1T1Plan3))}))).

ActPlan(do(V1T1Plan1),
        performAct(P1_ImNotGoingToMedit, Neutral, In_Front_Of_User, Neutral)).

ActPlan(do(V1T1Plan2),
        performAct(P1_ButWhatCanWeDo, Neutral, Stand_Still, Neutral)).

ActPlan(do(V1T1Plan3),
        performAct(P1_IKnowLetsPlayWith, Neutral, In_Front_Of_User, Neutral)).
```

- Patofil gets bored after vigil timer expires, and encourages participant to play with whisps

  When the vigil timer expires, the participant has not taken any actions and Patofil continues with her sequence of plans. She aims to create a playful mood and starts playing with the whisps which are floating around on the mound. She makes this transition by saying lines of speech which set the scene for the next act without seeming disconnected or out of context.

```
;;; Plans for Transition 2
;;; Patofil gets bored after vigil timer expires, and encourages participant
;;; to play with whisps

;;; When vigil_timer elapses, go to Transition 2
wheneverdo(timerState(vigil_Timer, elapsed),
        snsequence(believe(snare(whisps)),
                   do(Vigil1Transition2))).

ActPlan(do(Vigil1Transition2),
        csequence(do(V1T2Plan1),
```

```
believe({whendo(said(I,P1_ThisIsSooooBoring),
                do(V1T2Plan2)),
         whendo(said(I,P1_IAmNotGoingToMedit),
                do(V1T2Plan3)),
         whendo(said(I,P1_ButWhatCanWeDo),
                do(V1T2Plan4))})).
```

Each act is associated with a plan which Patofil performs. These actPlans are used to form complex plans which Patofil performs based on her policies. In this case, whenever the vigil timer expires, Patofil performs this transition act.

```
ActPlan(do(V1T2Plan1),
          performAct(P1_ThisIsSooooBoring, Agitated, In_Front_Of_User, Mad)).

ActPlan(do(V1T2Plan2),
        performAct(P1_IAmNotGoingToMedit, Neutral, Stay_Still, Happy)).

ActPlan(do(V1T2Plan3),
        performAct(P1_ButWhatCanWeDo, Neutral, Stay_Still, Neutral)).

ActPlan(do(V1T2Plan4),
        performAct(P1_IKnowLetsPlayWith, Whisp_Dance, In_Front_Of_User, Happy)).
```

- User has spontaneously started to play with whisps and Patofil follows users cue

  In the event that the User has noticed the whisps on the mound and started playing with them, Patofil follows the user and starts playing with the whisps. The trigger/cue for this transition act is the notification that participant has Hit_whisp. Patofil agrees with the participant that playing with whisps looks like fun and carries on to the playing with whisps act, which is Snare 2

```
;;; Plans for Transition 3
;;; User has spontaneously started to play with whisps
;;; Patofil follows users cue

whendo(ParticipantAction(User, Hit_Whisp),
       snif({if(snare(whisps), do(nothing)),
             else(do(V1T3Plan1))})).

ActPlan(do(V1T3Plan1),
```

```
        performAct(P1_OhThatLooksLikeFun, Happy, In_Front_Of_User, Neutral)).

whendo(said(I,P1_OhThatLooksLikeFun),
        do(PlayWithWhisps)).
```

- User has left the mound, whisps follow user, Patofil encourages user to play with whisps off the mound

  When participant is at edge, Patofil believes that when participant goes off the mound, she make a transition to the next snare, that is playing with whisps and she starts a timer called remindDanger which she uses as a cue to remind the participant of danger every 5 or so seconds

```
;;; Plans for Transition 4
;;; User has left the mound, whisps follow user, Patofil encourages user to play
;;; with whisps off the mound

;;; Vigil1Transition4 is triggered by this
wheneverdo(Location(User, At_Edge),
        snsequence(believe(whendo(Location(User, Off_Mound),
                                  do(Vigil1Transition4))),
                   start_timer(RemindDanger, 5))).

ActPlan(do(Vigil1Transition4),
        csequence(do(V1T4Plan1),
                  believe({whendo(subLocation(I, Followed_User),
                                  do(V1T4Plan2)),
                           whendo(said(I,P1_ButWhatCanWeDo),
                               do(V1T4Plan3))}))).

ActPlan(do(V1T4Plan1),
        performAct(P1_ImComingToo, Walking, Follow_User, Neutral)).

ActPlan(do(V1T4Plan2),
        performAct(P1_ButWhatCanWeDo, Walking, Face_User, Happy)).

ActPlan(do(V1T4Plan3),
        performAct(P1_IKnowLetsPlayWith, Walking, In_Front_Of_User, Happy)).
```

0.6.5 Snare 2: Playing with Whisps

Section 0.6.2 discussed Patofil's beliefs, acts and policies for the first snare: The Vigil. Section 0.6.4 provided the transitions which Patofil performs while moving from snare **Vigil** to snare **Play with Whisps**. This section provides SNePSLOG representation of the knowledge required for the snare **Play with Whisps**.

Once Patofil has started the second snare, which is *playing with the whisps*, there is a different, playful mood to the drama. After a silent, serious vigil, the whisps are supposed to make the participant happy and excited. The whisps are interactive objects in the world and Patofil is notified of the participant's actions with the whisps. From the Triggers/Cues given in Figure 6, the following occur in Snare 2: Play with whisps.

1. P_Near_Whisp: Patofil has arrived near a whisp
2. U_Hit_Whisp: User has successfully hit/batted at a whisp
3. P_Looked_For_Whisp: Patofil has found a whisp
4. U_Playing_With_Whisp: User is playing/attempting to hit whisps

These triggers/cues come in from the world and Patofil's uses this information to sequence her plans. In Snare 2: Play with Whisps, Patofil follows a repetitious cycle of playing with the whisps. She sets a timer for the duration of the snare and it is around 120 seconds. During this time, she encourages the participant to play with her, and if he is ignoring her and trying to get off the mound, she suitably warns him and entices him to come back. A brief outline of Patofil's performance is provided:

The Main Whisp-Playing Cycle

Patofil plays with whisps is a simple repetitious cycle. Choose a whisp, hit the whisp, wait while the whisp floats away, repeat cycle. She and the user can either be off the mound or on the mound as she plays. The whisps follow them wherever they go.

Contingencies which are active during this snare

1. User hits a whisp: Patofil makes comments and continues playing with the whisps.
2. User is not hitting whisps: Patofil encourages participant and tries to get him/her to play.
3. User is off the mound: Patofil periodically remind her of dangers
4. User is off the mound and comes back onto the mound, go to transition 3
5. User goes to the edge of the mound: Patofil stresses the danger of leaving the mound and tells her to come back. At this point, the User can do any of the following:

- User comes back: go to Main cycle of playing with whisps
- User goes off the mound: go to transition 2
- X seconds elapse: go to transition 1

6. After X seconds elapse and User is on the mound go to transition 1
7. After X seconds and User is off the mound go to transition 2

Transitions for the Snare: Play with whisps

1. Go to Snare 3: Whisps_Start_Leaving
2. Go to Snare 5: Leave_Mound
3. If the user was off the mound, and has returned to mound, Patofil and the whisps also return, Patofil is pleased and proceeds to Snare 4: Jump_On_Rocks

This SNePSLOG representation might not be complete and is only a subset of the knowledge that Patofil requires to perform her acts for Snare 2. These beliefs, acts and policies are currently being tested and code changes will be updated in the KL snepslog file.

Upon transitioning from Snare 1, Patofil continues to Snare 2. She starts her main Whisps Plan which is a repetitious cycle where she plays with whisps and while playing with them, she encourages the user. At the beginning of the snare, Patofil starts a Play_With_Whisps timer for 120 seconds. Once the timer is up, she continues onto the other snares.

```
;;; Snare 2 : Playing with Whisps
;;; Transition from Snare 1, Transition 1,2 or 4
wheneverdo(said(I,P1_IKnowLetsPlayWith),
        snsequence(do(WhispsMainPlan),
                   start_timer(Play_With_Whisps, 120))).

;;; The main act plan for playing with whisps
;;; start a whisp_play timer for 120 seconds
;;; start a user-hit-whisp timer for 4 seconds, Patofil
;;; waits for this time to give the user a chance to hit a whisp
;;; also check if user is hitting a whisp
ActPlan(do(WhispsMainPlan),
        snsequence_3(do(PlayWithWhisps),
                     start_timer(UHitWhisp, 4),
                     vision(Check_U_Hit_Whisp))).
```

```
ActPlan(do(PlayWithWhisps),
        performAct(Silence5, Look_Around, Stand_Still, Neutral)).
```

Patofil completes looking around and continues to navigate to whisp happily

```
;;; Animation cue
wheneverdo(animated(I, Look_Around_Over),
        performAct(Silence6, Walking, Move_To_Whisp, Happy)).
```

The Move_To_Whisp navigation returns notification when Patofil has moved near a whisp. The cue received is Patofil is Near_Whisp. Patofil can be near whisp while performing any snare, because the whisps are active right from the beginning of the drama and float around nearby. A check if the snare currently being performed is play_with_whisps is made. If yes, Patofil performs a whisp play act and bats at the whisp

```
;;; Whenever Patofil is near a whisp, bat it happily
wheneverdo(subLocation(I, Near_Whisp),
        snif({if(snare(whisps), do(WhispPlay)),
                      else(do(nothing))})).


;;; ActPlan to play one of WhispPlay acts
ActPlan(do(WhispPlay),
        do-all({do-one({say(P1_WhispPlay1),
                        say(P1_WhispPlay2),
                        say(P1_WhispPlay3),
                        say(P1_WhispPlay4)}),
                animate(Bat),
                navigate(Face_Whisp),
                mood(Happy)})).
```

The participant/user can hit whisps and when he successfully hits a whisp. Upon receiving this cue, Patofil is pleased and encourages the participant/user to continue hitting the whisps and play with her.

```
wheneverdo(ParticipantAction(User, Hit_Whisp),
                snsequence_3(do(Encourage),
                             vision(Check_U_Hit_Whisp),
                             stop_timer(UHitWhisp))).


;;; When User Hits Whisp
```

```
ActPlan(do(Encourage),
        do-all({do-one({say(P1_DidYouGetIt),
                        say(P1_HowFarCanYouHitThe),
                        say(P1_HowManyHaveYouHit)}),
                mood(Happy),
                animate(Walking),
                navigate(Move_To_Whisp)})).

{said(I, P1_DidYouGetIt), said(I, P1_HowFarCanYouHitThe),
 said(I, P1_HowManyHaveYouHit)} => {said(I, EncouragingLines)}
```

After encouraging the participant, Patofil continues with her repetitious main whisp playing cycle

```
wheneverdo(said(I, EncouragingLines),
        snsequence(disbelieve(said(I, EncouragingLines)),
                   do(WhispsMainPlan))).
```

Patofil starts a user_hit_timer for 4 seconds during the performance of her main whisps plan. This timer allows the user some time to try hitting the whisps. If the user has not hit the whisps and the timer elapses, Patofil encourages the user to hit a whisp and returns to her main play_with_whisps plan in which she also restarts the user_hit_whisp timer. If the user hits a whisp, the user_hit_whisp cue comes in from the world and Patofil performs other acts.

```
;;; U Hit Timer expired
wheneverdo(timerState(UHitWhisp, elapsed),
        snsequence(do(EncourageToHit),
                   disbelieve(timerState(UHitWhisp, elapsed)))).

ActPlan(do(EncourageToHit),
        do-all({do-one({say(P1_TryHittingThe),
                        say(P1_YouCanUseEitherHa),
                        say(P1_GetCloseToThem),
                        say(P1_CanYouGetThem)}),
                mood(Happy),
                animate(Walking),
                navigate(Move_To_Whisp)})).

{said(I, P1_TryHittingThe), said(I, P1_YouCanUseEitherHa),
 said(I, P1_GetCloseToThem), said(I, P1_CanYouGetThem)}
```

```
 => {said(I, EncourageToHitLines)}.

wheneverdo(said(I, EncourageToHitLines),
        snsequence(do(WhispsMainPlan),
                   disbelieve(said(I, EncourageToHitLines)))).
```

The user is free to move as they please. If they go off the mound, Patofil starts a remind of danger timer which allows the user to come back within this time. If the user is still off the mound when the timer elapses, Patofil reminds the user of all the dangers that are out there and pleads with them to return back on the mound. She also restarts her remind of danger timer. She then returns to her main plan and is interrupted by the expiry of the remind of danger timer in which case if the user is still off the mound, she reminds them of the dangers around.

```
;;; Remind of Danger
wheneverdo(timerState(RemindDanger, elapsed),
        believe(wheneverdo(Location(User, Off_Mound),
                        snsequence(do(RemindOfDanger),
                                   restart_timer(RemindDanger))))).

ActPlan(do(RemindOfDanger),
        do-all({do-one({say(P1_IHopeItsSafeOutHe),
                        say(P1_IThoughtIHeardSome),
                        say(P1_IThinkSomethingsW),
                        say(P1_DontYouThinkWeShou)}),
                do-one({mood(Neutral), mood(Mad)}),
                do-one({animate(Reverent), animate(Neutral), animate(Look_Around)}),
                do-one({navigate(Stand_Still), navigate(Stand_Still)})})).

{said(I,P1_IHopeItsSafeOutHe), said(I, P1_IThoughtIHeardSome),
 said(I, IThinkSomethingsW), said(I, P1_DontYouThinkWeShou)}
 => {said(I, RemindingOfDangerLines)}.

wheneverdo(said(I, RemindingOfDangerLines),
        snsequence(do(WhispsMainPlan),
                   disbelieve(said(I, RemindingOfDangerLines)))).
```

The user being at edge of the mound is a contingency and is handled as such. Patofil calls the user back repeatedly.

```
;;; When User is At Edge in snare 2
```

```
whenever do(Location(User, At_Edge),
        snif({if(snare(whisps), do(CallUserBack)), else(do(nothing))})).

ActPlan(do(CallUserBack),
        snsequence_3(start_timer(Stay_On_Mound, 15),
                     do(CallUserBack1),
                     believe({whendo(said(I, P1_DontGoOffTheMound),
                                     do(CallUserBack2)),
                              whendo(said(I, P1_FilopatSaysItsDangerous),
                                     do(CallUserBack3)),
                              whendo(said(I, P1_YoureNotProtected),
                                     do(CallUserBack4))}))).

ActPlan(do(animationCallUserBack),
        do-all({mood(Neutral), animate(Agitated), navigate(Near_Edge_By_User)})).

ActPlan(do(CallUserBack1),
        do-all({say(P1_DontGoOffTheMound), do(animationCallUserBack)})).

ActPlan(do(CallUserBack2),
        do-all({say(P1_FilopatSaysItsDangerous), do(animationCallUserBack)})).

ActPlan(do(CallUserBack3),
        do-all({say(P1_YoureNotProtected), do(animationCallUserBack)})).

ActPlan(do(CallUserBack4),
        do-all({say(P1_ThingsLurkOutThere), do(animationCallUserBack)})).
```

0.6.6 Inside the PML

The PMLa and PMLb are implemented in Lisp and SNePS while the PMLc is implemented in Ygdrasil. Communication between PMLb and PMLc is through sockets as discussed in 0.6.7. This section provides a look at the contents of Patofil's PMLa and PMLb sub-layers.

PMLa sub-layer

The PMLa contains definitions of functions, primactions, variables which are used by the knowledge layer. In case of actor-agent Patofil, the PMLa contains storage variables for working memory, short-

term memory. Primactions defined in the PMLa map to acts in the KL. These functions, primactions are defined in Lisp.

- Load required files: The pmlb layer and timer functions are loaded

```
;;; Load the pmlb file
(load "/projects/robot/Ygdrasil/Agents/Patofil/SNePSLOG/Ver2/pmlb_SNLOG_Ver2.cl")

;;; Load Michael Kandefer's file "timer.cl"
(load "/projects/robot/Ygdrasil/Agents/Patofil/SNePSLOG/Ver2/timer.cl")
```

- Variables: Storage variables used by acts in the knowledge layer. Patofil stores knowledge of her currently executing acts in *WM* (working memory). She stores contents of *WM* in *STM* when she has to memorize an act. She can also recall (retrieve contents of *STM*) in order to know which act she was previously performing.

```
(defparameter *TIMER-LIST* nil "Contains the list of all timers the agent is
                                currently using.")
;;; Define variable for Working Memory
(defvar *WM* nil)

;;; Define variable for Short Term Memory
(defvar *STM* nil)
```

- Primactions:

```
;;; performAct primaction for communicating commands to all
;;; modalities (speech, animation and navigation)

(define-primaction performActAll (speech anim navig mood)
  (writeToSocket speech speechsocket)
  (writeToSocket anim animatesocket)
  (writeToSocket navig navigatesocket)
  (writeToSocket mood moodsocket))

;;; These functions are for cases of communicating commands
;;; to individual modalities

;;; Individual say
```

```
(define-primaction sayPrimFun (object1)
  (writeToSocket object1 speechsocket))

;;; Individual mood
(define-primaction moodPrimfun (object1)
  (writeToSocket object1 moodsocket))

(define-primaction hearingPrimFun (object)
  (writeToSocket object1 hearingsocket))

(define-primaction visionPrimFun (object1)
  (writeToSocket object1 visionsocket))

;;; Individual animate
(define-primaction animatePrimFun (object1)
  (writeToSocket object1 animatesocket))

;;; Individual navigate
(define-primaction navigatePrimFun (object1)
  (writeToSocket object1 navigatesocket))

;;; Control socket communication
(define-primaction SMfun (object1)
  (writeToSocket object1 SMsocket))

;;; Memorize in *STM* (Short Term Memory)
(define-primaction memorizefun ()
  (cl-user::push *WM* *STM*)
  (format t "MEMORIZING: ~a   ~b~%" *WM* *STM*))

;;; Recall from *STM* (Short Term Memory)
(define-primaction recallfun ()
  ;; Debug statement
  (let ((mesg (lisp:pop *STM*)))
    (add-selfpercept mesg)
    (format t "RECALLING: ~a~%" mesg)))
```

```
;;; Store in *WM* (Working Memory)
(define-primaction storefun (object1)
  (setf *WM* object1))

;;; Stop all SNeRE acts
(define-primaction goOfffun (object1)
  (stopSnere))

;;; Do nothing
(define-primaction nullfun ()
  (print "Doing nothing"))
```

- Functions: Timer functions are in this layer. The timer functions have been defined by Michael Kandefer. The main timer control functions are in lisp and in the file */projects/robot/Ygdrasil/Agents/BadGuys/timer.cl*. The primactions defined in the PMLa use functions defined in the *timer.cl* file. The primactions for timer control are:

  1. start-timer (timerName, duration): Start a timer with specified name for specified duration
  2. stop-timer (timerName): Stop the specified timer
  3. restart-timer (timerName): Restart the specified timer for the previously specified duration

  At the PMLa, the following function is defined which adds the belief that a specific timer has expired/elapsed when the duration of that timer is up. Based on this belief, Patofil triggers/cues her other acts.

```
;;; When timer elapses, believe that timer has elapsed
(defun add-times-up (timer)
  ;;; Debug statements
  (print "ADDING")
  (print timer)
  "Add a node with a unique timer name and elapsed state"
  #!((add member ~timer state elapsed)))
```

- Function to start up socket connections with initial broker socket (See Section 0.6.7 for more details)

```
(defun startSockets()
  ;;Set brokersocket to initial connection
```

```
  (setf brokersocket (OpenConnection))

;;; Function SocketConnProcess (defined in PMLb level)
;;; automatically establishes socket connections for
;;; all modalities
  (SocketConnProcess brokersocket)

;;; Thread(s) started for Broker, Hearing, Vision and Timer sockets
;;; which monitor input and output
  (mp:process-run-function "Broker Thread" #'SocketConnProcess brokersocket)
  (mp:process-run-function "Hearing thread" #'ProcessInput1 hearingsocket)
  (mp:process-run-function "Vision Thread" #'ProcessInput2 visionsocket)
  (mp:process-run-function "Timer thread" #'ProcessInput3 timersocket))
```

- Function call (Function definition in PMLb) which waits for user input if socket connections are desired or not.

```
;;; Function (defined in PMLb) for user to input
;;; if socket connections are desired or not
(socketOption)
```

PMLb sub-layer

The PMLb contains functions for establishing socket connections to the PMLc and Lisp functions that deal with modalities and low-level communication with the embodiment of the agent, namely the PMLc and SAL layers which are implemented in Ygdrasil. The PMLb also communicates with the KL and is thus the middle layer between the high-level knowledge and the low-level implementation of bodily functions.

- Variables:

```
;;; Establishing socket connections between PMLb and PMLc
;;; Open socket connections to Ygdrasil
(defvar *vrMachine* "mediastudy11.fal.buffalo.edu"
  "Name of the machine where the VR software is running.")

;;; Default broker port number
(defvar *brokerPort* 4515
  "The broker socket number on Yg side where PMLb connects")
```

- Functions which require user input.

```
(defun promptedRead (prompt)
  "Prompts user for input"
  (format t "~&~a:  " prompt)
  (read))

(defun socketOption ()
  "User chooses option whether to start up sockets or just test run the demo file"
  (format t "Do you want to start up sockets?[y/n] : ")
  (let ((opt (read-line)))
    (if (string= opt "y")
        (startSockets))))

(defun promptedReadWithDefault (prompt default)
  "Prompts user for input, if no input takes default value"
  (format t "~&~a [~a]:  " prompt default)
  (let ((inp (read-line)))
    (if (string= inp "")
        default
      (read-from-string inp))))
```

- Functions for establishing socket connections.

```
(defun OpenConnection ()
‘‘Opens a connection to the pre-defined *brokerPort*’’
  (setf *vrMachine*
    (promptedReadWithDefault "What machine is running the VR software?"
                             "mediastudy12.fal.buffalo.edu"))
  (socket:make-socket :remote-host *vrMachine* :remote-port *brokerPort*))

(defun OpenConnection2 (port)
  "Opens a connection to the specified port number"
  (socket:make-socket :remote-host *vrMachine* :remote-port port))
```

- Functions for write to socket.

```
;;; General function to write to socket
;;; Speech, animation, navigation, emotion, control and timer strings
```

```
;;; optArg is 1 when writing to timer socket because formatting
;;; is not needed

(defun writeToSocket (mesg socketType &optional optArg)
  (format t "~&~a~%" mesg)
  (sleep 0.2)
  (case optArg
    (1 (write-line mesg socketType))
    (t (write-line (format nil "~&~A" (sneps:choose.ns mesg)) socketType)))
  (force-output socketType))

;;; General function for timer action
(defun timerAction (arg1 socketType arg2)
  (case arg2
    (1 (write-line (format nil "pause('~A')" (sneps:choose.ns arg1)) socketType))
    (2 (write-line (format nil "resume('~A')" (sneps:choose.ns arg1)) socketType))
    (3 (write-line (format nil "abort('~A')" (sneps:choose.ns arg1)) socketType)))
  (force-output socketType))
```

0.6.7 Socket Connections between PML and SAL

Getting Started

Sockets provide the main means of communication for machines on a network. The mind of the agent (SNePS) is connected to the embodiment (implemented in Ygdrasil) through sockets. This is because Patofil's mind is implemented on a different machine than the embodiment. The modality sockets are for vision, hearing, speech, navigation, animation, emotion and time. There is also a broker socket connection which is made between PMLb and Ygdrasil. Through this socket connection, the rest of the port numbers (for the modalities) are communicated to SNePS. On obtaining the port numbers, the socket connections are implicitly handled without human intervention. The process which is started up for handling the broker socket and for connecting to the rest of the sockets implicitly is given below.

There is a stage-manager socket for communicating high-level knowledge or command strings between SNePS and Ygdrasil. Certain events are required to be triggered by SNePS and this is achieved by sending the command string to activate the event to Ygdrasil. In the future, an actual stage-manager agent is envisioned who will oversee all the other actor-agents and the other actor-agents will report to the stage-manager. Each actor-agent will be a separate agent implementation, but they will be capable of communication and taking orders from the stage-manager.

Automated Socket connections

A broker socket handles the socket connections for the modalities and control commands. The broker socket is initialized by Ygdrasil, and when PMLb connects to this socket, the port numbers for the other socket connections (perception, self perception, speech, animation, navigation, emotion, timer, control-command) are piped across the broker socket connection. The PMLb side receives these port numbers and automatically connects to them by invoking the appropriate function. For every agent, one socket number is sufficient to set the socket connections between the mind and the embodiment implicitly. The broker socket is closed after all the other socket connections are implicitly made. SocketConnProcess is called when the main file is loaded and it starts up a process at the broker socket which waits for input from Ygdrasil. Ygdrasil sends socket numbers which are connected to automatically by the SocketConnProcess function.

```
;;; Function to initialize broker socket
(defun startSockets()
;;; Set brokersocket to initial connection
  (setf brokersocket (OpenConnection))

;;; Function SocketConnProcess (in PMLb layer) implicitly
;;; connects all modality and control sockets
  (SocketConnProcess brokersocket)

;;; Thread(s) started for Broker, Perception, Self-Perception
;;; and Timer sockets which monitor input and output
  (mp:process-run-function "Broker Thread" #'SocketConnProcess brokersocket)
  (mp:process-run-function "Hearing thread" #'ProcessInput1 hearingsocket)
  (mp:process-run-function "Vision Thread" #'ProcessInput2 visionsocket)
  (mp:process-run-function "Timer thread" #'ProcessInput3 timersocket))
```

0.6.8 The SAL

The lower sub-layers of the PML (PMLc) and the SAL are implemented largely in Ygdrasil. The higher consciousness of the agent resides in the Knowledge Layer (implemented in SNePS) and the lower layers execute bodily functions and provide perceptual feedback. The Sensori-Actuator Layer is implemented in Ygdrasil as functions which manipulate Patofil's embodiment. When Patofil's mind commands her body to navigate to a point, the Ygdrasil functions execute the actual commands required to move Patofil's virtually simulated body to the desired position. In a virtual reality environment, the issues faced with hardware agents do not arise, but GLAIR-based agents have previously been developed for hardware robots like the Magellan Pro irobot [16]. This project aims towards larger goal(s) of developing

a general agent architecture for designing intelligent, cognitive agents capable of communicating in natural-language.

0.6.9 A Graphical Simulator

TCP/IP socket connections are between PMLb of the MGLAIR and Ygdrasil. Perception triggers/cues like Patofil's position, the participant's reactions/position in the world, actions of objects in the world etc are sent across the socket by Ygdrasil, and the SNePS-based Knowledge Layer reasons about these according to the script. Actions are taken by Patofil and communicated from her mind across the socket connections from SNePS (her mind) to Ygdrasil (her embodiment). For the purposes of testing and debugging during the development of code, it is imperative that testing and changing code be quickly and cleanly accomplished. Hence a simulator program was written in Java which simulates Ygdrasil or the embodiment of the agent. The perceptions are now simulated and sent across the sockets by the programmer operating on a click-button GUI to generate the required triggers for a particular scene. SNePS (the mind of the agent) reacts to these triggers as if they were being sent by Ygdrasil (the embodiment of the agent) itself.

The simulator package consists of one JAVA program namely **Server.java**. This program starts up the vision, hearing, speech, animation, navigation, mood and timer sockets (simulating the corresponding sockets on Ygdrasil). The output of the speech socket is piped as input to the hearing socket. This enables the agent to hear what she speaks. The animation, navigation, speech and mask output are printed to standard output. This java program uses *java.awt* to build an interface for the agent designer to simulate triggers/cues which are communicated to Patofil's mind (Knowledge Layer). See Section 0.10 for the *Server.java* code.

0.7 Instructions for starting up a demonstration

Instructions for starting the GUI and connecting to PMLa of GLAIR are:

- **Start the Java GUI and set up socket ports waiting for connections.**

  1. If Java program Server.java is not already compiled, compile it.

     ```
     pollux {/projects/robot/Ygdrasil/Agents/GUI} > javac Server.java
     ```

  2. Run Server

     ```
     pollux {/projects/robot/Ygdrasil/Agents/GUI} > java Server
     ```

- **Start up Lisp, load SNePS and load the snepslog file containing Patofil's knowledge base.**

1. Start up ACL Lisp.

   ```
   pollux {/projects/robot/Ygdrasil/Agents/Patofil/SNePSLOG/Ver2} > acl
   ```

2. Start up sneps by loading /projects/shapiro/Sneps/sneps262 at the Lisp prompt

   ```
   cl-user(1): :ld /projects/shapiro/Sneps/sneps262
   ```

3. Enter snepslog mode

   ```
   cl-user(2): (snepslog)
   ```

4. Load the latest version (snepslog file) for Patofil Knowledge Layer

   ```
   : demo ‘‘/projects/robot/Ygdrasil/Agents/Patofil
            /SNePSLOG/Ver2/Patofil_Ver2.snepslog’’
   ```

5. The following user input is desired:

   ```
   Do you want to start up sockets?[y/n] :
   ```

   Say **y** or **n** as desired. **n** is chosen if no socket connections are to be made and only the snepslog file is to be loaded (to check for snepsul errors and unrecognized syntax).

6. If socket connections are desired, (by entering **y** for the previous question) the machine name to connect to is requested from the user. The default machine is *mediastudy12.fal.buffalo.edu* but if the socket connections are to be made to the Java GUI program running on *pollux.cse.buffalo.edu*, then the user has to enter *pollux.cse.buffalo.edu* as the machine name.

   ```
   What machine is running the VR software? [mediastudy12.fal.buffalo.edu]:
   ```

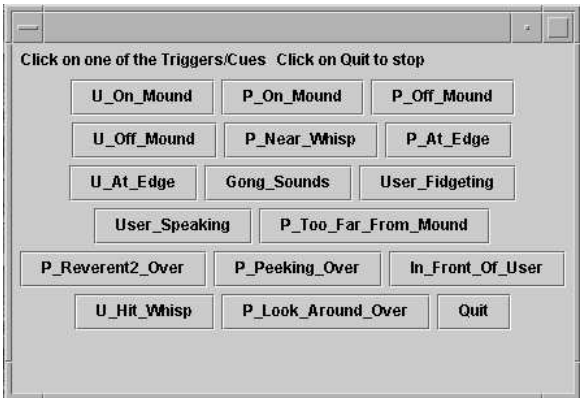

Figure 7: Graphical Simulator

Connections are made from SNePS to the corresponding sockets and the agent designer can simulate Ygdrasil by clicking the buttons representing various triggers shown in Figure 7. On receiving these triggers/cues, Patofil s mind will *reason* and respond. Her responses are printed on standard output so the agent designer can see her line of reasoning and the sequence of acts she is performing. The simulator proves to be useful for debugging purposes. Because of the ASCII implementation, it is also suitable for demonstrating agent behavior without the explicit need for connecting to Ygdrasil.

0.8 Conclusions

This report provides a comprehensive discussion of an actor-agent implemented in GLAIR, whose Knowledge Layer is implemented in SNePS. Each layer of the GLAIR architecture is explained in detail and the corresponding implementation is provided along with documentation. The design and implementation of the actor-agent for the virtual reality drama *The Trial The Trail* has undergone many iterations and re-designs. New ideas have been implemented and new issues have been encountered and handled. There are still many more issues which have to be handled efficiently. The co-designing aspects implemented for Patofil is a small step towards implementing many more such co-designing capabilities for actor-agents and intelligent agents in general. This project has provided a good learning experience for fleshing out a GLAIR-based co-designing actor-agent who is intelligent, interactive and believable. One of the goal(s) of this project is to contribute towards research in agent design and development and we believe our work contributes towards these larger goal(s).

0.9 Future Work

0.9.1 Complete Production of *The Trial The Trail*

The design and implementation of Patofil for the Act 3, Scene 1 of *The Trial The Trail* was a starting step towards building an intelligent *actor-agent* for interactive, virtual reality dramas. Future work involves completing the rest of the scenes in the drama. Each scene explores new avenues in agent-design and believability. Multi-agent development, implementing mechanisms to explore the participant's mental attitude and use that information in other scenes etc. are the focus of on-going research. Patofil's design which can thus be extended to other similar actor-agents who are in other scenes of the drama.

0.9.2 The Motivation for extension of GLAIR to MGLAIR

In previous implementations of GLAIR agents [10], [11]. due to lack of a modular package for modalities, similar functions and variables were repeatedly declared and redundant functions were defined at the PMLa and PMLb for each GLAIR agent. The addition of more functions resulted in an ad-hoc structure of the PML rendering it unreadable and difficult to test and debug. The present design for Patofil includes implementation of modalities, but they are defined and implemented across the lower layers of GLAIR.

CLOS is the Common Lisp Object System used for implementing classes and methods in Lisp, which enable the GLAIR agent designer to use the modality package for implementing modalities by instantiating modalities for each agent. This modality package will implement classes and methods, with focus on an object-oriented design. Each modality can be treated as an object with a set of functional capabilities. The agent designer is required to instantiate only the modalities as per the design specifications for an agent. Each modality object can be considered as a "black box" whose

internal design need not be known to the agent-designer. The integration of the modality package and GLAIR results in an improved object-oriented agent architecture, namely MGLAIR (Modal-GLAIR).

Advantages of MGLAIR with a CLOS modality package

1. The modular structure facilitates easier programming and better comprehension.
2. Object oriented programming enables the use of powerful features of CLOS.
3. The agent designer is not required to know elaborate internal details of a modality. A modality can be treated as a "black box" and used by the designer.
4. The implementation of modalities will thus be modularized. Each modality can be used by the agent irrespective of the functioning of the other modalities. A modality is comparable to a body resource. A modality or body resource cannot be used by the agent to perform conflicting acts. For example, consider an agent who is planning to talk to person A and follow person B at the same time. The action talking uses the vision modality to direct the agent's attention to the person being addressed. The action following uses the vision modality to seek the person/thing being followed. Either of the two actions can be executed at any given time. Hence, a modality should be aware of which act is using it and if another act requests the same modality when it is being used, an error occurs and the conflict should be resolved. These are some of the design issues involved in extending GLAIR to MGLAIR.

Bibliography

[1] S. C. Shapiro. SNePS: A logic for natural language understanding and commonsense reasoning. In Iwanska and S. C. Shapiro, editors, Natural Language Processing and Knowledge Representation: Language for Knowledge and Knowledge for Language, pages 175-195. AAAI Press/ MIT Press, Menlo Park, CA, 2000.

[2] S. C. Shapiro and The SNePS Implementation Group. SNePS 2.6.1 User Manual Department of Computer Science, University at Buffalo
URL: http://www.cse.buffalo.edu/sneps/Manuals/manual261.pdf

[3] Agents (Cognitive Robotics/Embodied Agents) From SNeRG (SNePS Research Group) Website. Last Accessed: April 10th, 2005 URL:http://www.cse.buffalo.edu/sneps/Projects/agents.html

[4] Michael Mateas Andrew Stern, Architecture,, Authorial Idioms and Early Observations of the Interactive Drama Faade, December 5, 2002 CMU-CS-02-198

[5] Mateas, M. and Stern, A., A Behavior Language for Story-Based Believable Agents.2002. In Ken Forbus and Magy El-Nasr Seif (Eds.), Working notes of Artificial Intelligence and Interactive Entertainment. AAAI Spring Symposium Series. Menlo Park, CA: AAAI Press. 2002.

[6] Josephine Anstey, Dave Pape, Stuart C. Shapiro, Orkan Telhan and Trupti Devdas Nayak, Psycho-Drama in VR, In The Proceedings of COSIGN 2004, The Fourth Conference on Computational Semiotics for Games and New Media, Split, Croatia, 14th-16th September 2004.

[7] Dave Pape, Josephine Anstey, Margaret Dolinsky, Edward J. Dambik, Ygdrasil:a framework for composing shared virtual worlds, Future Generation Computer Systems Volume 19, Issue 6 (August 2003) iGrid 2002, Pages: 1041 - 1049, 2003

[8] Vikranth B. Rao, Princess Cassie: An Embodied Cognitive Agent in a Virtual World, Advanced Honors Thesis, Department of Computer Science and Engineering, State University of New York at Buffalo, April, 2004

[9] Stuart C. Shapiro and Haythem O. Ismail, Anchoring in a Grounded Layered Architecture with Integrated Reasoning, Robotics and Autonomous Systems 43, 2-3 (May 2003), 97-108

[10] Henry Hexmoor, Johan Lammens, and Stuart C. Shapiro. Embodiment in GLAIR: a grounded layered architecture with integrated reasoning for autonomous agents. In Douglas D. Dankel II and John Stewman, editors, Proceedings of The Sixth Florida AI Research Symposium (FLAIRS 93), pages 325-329. the Florida AI Research Society, April 1993

[11] Henry Hexmoor, Johan Lammens, and Stuart C. Shapiro. An autonomous agent architecture for integrating perception and acting with grounded, embodied symbolic reasoning. Technical Report 92-21, Department of Computer Science, University at Buffalo, Buffalo, NY, September 1992. 22 pages.

[12] Josephine Anstey: The Trial The Trail. Last Accessed: April 2005
URL: http://www.ccr.buffalo.edu/anstey/VDRAMA/TRAIL/index.html

[13] Josephine Anstey: The Trial The Trail. Act 3: The Second Challenge
Last Accessed: April 15th, 2005
URL: http://www.ccr.buffalo.edu/anstey/VDRAMA/TRAIL/SCRIPT/act3.html

[14] Haythem O. Ismail and Stuart C. Shapiro, Conscious Error Recovery and Interrupt Handling. In H. R. Arabnia, Ed., Proceedings of the International Conference on Artificial Intelligence (IC-AI'2000), CSREA Press, Las Vegas, NV, 2000, 633-639.

[15] Haythem O. Ismail and Stuart C. Shapiro, Cascaded Acts: Conscious Sequential Acting for Embodied Agents, Technical Report 99-10, Department of Computer Science and Engineering, University at Buffalo, Buffalo, NY, November 1, 1999.

[16] Trupti Devdas Nayak, Michael Kandefer, and Lunarso Sutanto, Reinventing the Reinvented Shakey in SNePS, SNeRG Technical Note 36, Department of Computer Science and Engineering, University at Buffalo, The State University of New York, Buffalo, NY, April 6, 2004.

[17] W. Lewis Johnson. Agents that learn to explain themselves. In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), American Association for Artificial Intelligence, Menlo Park, CA, 1994, 1257-1263.

[18] Brazier, F.M.T. and Wijngaards, N.J.E. (2001), Designing Self-Modifying Agents, In: Gero, J.S. and Maher, M.L. (editors), Computational and Cognitive Models of Creative Design V, pp. 93-112.

[19] Stuart C. Shapiro, Josephine Anstey, David E. Pape, Trupti Devdas Nayak, Michael Kandefer, and Orkan Telhan, The Trial The Trail, Act 3: A Virtual Reality Drama Using Intelligent Agents, Proceedings of the First Annual Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE-05), AAAI Press, Menlo Park, CA, 2005, in print.

0.10 Appendix D : Server.java

```
/* Java Program to establish socket connections with PMLb of GLAIR and
   communicate through these socket connections (which represent modalities)
   with the lower layers of the GLAIR architecture.
   Written by Trupti Devdas Nayak
   Started: Feb 2005 */

import java.net.Socket;
import java.net.ServerSocket;
import java.io.OutputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/* Class Definition */
public class Server extends JFrame implements Runnable{
    private String Input;
    private JLabel la1, la2, la3, la4, la5, la6;
    private JButton b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12,
                    b13, b14, b15, b16, b17, b18, b19, b20;
    private ServerSocket broker;
    private ServerSocket animation;
    private ServerSocket speech;
    private ServerSocket hearing;
    private ServerSocket vision;
    private ServerSocket navigation;
    private ServerSocket sm;
    private ServerSocket mood;
    private ServerSocket timer;
    private String [] ports = new String [8];
    private Socket brokerConnection;
    private Socket speechConnection;
    private Socket visionConnection;
    private Socket hearingConnection;
```

```
private Socket animationConnection;
private Socket navigationConnection;
private Socket moodConnection;
private Socket timerConnection;
private Socket smConnection;
private OutputStream brokerWrite;
private OutputStream visionWrite;
private OutputStream hearingWrite;
private OutputStream speechWrite;
private OutputStream animationWrite;
private BufferedReader consoleInput;
private BufferedReader clientInput, speechInput, hearingInput;
private String speechLine = null;

/* Constructor Definition */
public Server() throws java.io.IOException{
    /* Set up the socket ports */
    setUpSockets();

    /* Wait for connection from PMLb */
    System.out.println("Waiting for connection from PMLb" + "\n");
    brokerConnection = broker.accept();
    System.out.println("Accepted connection" + "\n");
    brokerWrite = brokerConnection.getOutputStream();

    /* Write socket port number for modalities to broker port
       for establishing modality socket connections */
    establishConn();

    /* Set up input readers */
    clientInput = new BufferedReader(new InputStreamReader
                                     (brokerConnection.getInputStream()));
    hearingInput = new BufferedReader(new InputStreamReader
                                     (hearingConnection.getInputStream()));
    speechInput = new BufferedReader(new InputStreamReader
                                    (speechConnection.getInputStream()));
}
```

```
/* Function which monitors User/Programmer input through the GUI*/
public void ServerLoop() throws java.io.IOException {

    /* Start up the graphics interface */
    startGraphics();

    /* Loop until user/programmer quits*/
    while(true){
        if (Input !=null){
            /* Decide which socket to feed user-input */
            /* Auditory cues are communicated through hearing modality*/
            if((Input == "User_Speaking") || (Input == "Gong_Sounds")) {
                System.out.println("Writing to Hearing socket" + "\n");
                hearingWrite.write((Input + "\n").getBytes());
                Input = null;
            }
            /* Visual cues are communicated through vision modality */
            else if(Input != "Quit")
                {
                    System.out.println("Writing to Vision socket" + "\n");
                    visionWrite.write((Input + "\n").getBytes());
                    Input = null;
                }
            /* User entered Quit */
            else
                {
                    visionWrite.write((Input + "\n").getBytes());
                    /* Close all socket connections cleanly */
                    vision.close();
                    hearing.close();
                    mood.close();
                    navigation.close();
                    broker.close();
                    timer.close();
                    sm.close();
                    animation.close();
```

```
                    System.exit(0);
                }
        }

    }
}

/* Main function to start two processes:
   1. Process to monitor speech socket input
   2. Process to monitor agent-designer input (triggers/cues) through GUI */
public static void main (String[] args) throws java.io.IOException{
    /* Create a new object of class */
    Server example = new Server();
    example.addWindowListener(
                            new WindowAdapter() {
                                public void windowClosing(WindowEvent e)
                                {
                                    System.exit(0);
                                }
                            }
                            );
    /* Start a thread to monitor speech socket */
    (new Thread(example)).start();
    /* Start a loop to monitor the events triggered by agent-designer using the GUI */
    example.ServerLoop();
}

/* Method to print details of which button was pressed (event occurred) */
private class ButtonHandler implements ActionListener {
    public void actionPerformed( ActionEvent e)
    {
        Input = e.getActionCommand();
        System.out.println("Trigger/Cue is " + Input);
    }
}

/* Method run for thread which monitors speech socket for input and
```

```
   pipes the speech Lines through hearing modality using the hearing
   socket (feedback to agent that speech Line has been uttered */
public void run() {
    try
        {
            while(true) {
                speechLine = speechInput.readLine();
                System.out.println("Speech Feedback through
                                    Hearing modality:" + speechLine);
                hearingWrite.write((speechLine + "\n").getBytes());
            }

        }
    catch (java.io.IOException exception) {}

}

/* Method to start the graphical user interface, add buttons, and add action listener
   for notification on events */
public void startGraphics () throws java.io.IOException {

    Container c = getContentPane();
    c.setLayout(new FlowLayout());
    /* Create labels */
    la1 = new JLabel( " Click on one of the Triggers/Cues");
    la2 = new JLabel( "                                    ");
    la3 = new JLabel(" Click on Quit to stop ");

    /* Create buttons */
    b1 = new JButton("U_On_Mound");
    b2 = new JButton("P_On_Mound");
    b3 = new JButton("P_Off_Mound");
    b4 = new JButton("U_Off_Mound");
    b5 = new JButton("P_Near_Whisp");
    b6 = new JButton("P_At_Edge");
    b7 = new JButton("U_At_Edge");
    b8 = new JButton("Gong_Sounds");
```

```
b9 = new JButton("User_Fidgeting");
b10 = new JButton("User_Speaking");
b11 = new JButton("P_Too_Far_From_Mound");
la4 = new JLabel(" Other Modality triggers ");
b12 = new JButton("P_Reverent2_Over");
b13 = new JButton("P_Peeking_Over");
b14 = new JButton("In_Front_Of_User");
b15 = new JButton("U_Hit_Whisp");
b16 = new JButton("P_Look_Around_Over");
b17 = new JButton("Quit");

/* Add all buttons to the content pane */
c.add(la1);
c.add(la2);
c.add(la2);
c.add(la3);
c.add(la2);
c.add(b1);
c.add(b2);
c.add(b3);
c.add(b4);
c.add(b5);
c.add(b6);
c.add(b7);
c.add(b8);
c.add(b9);
c.add(b10);
c.add(b11);
c.add(b12);
c.add(b13);
c.add(b14);
c.add(b15);
c.add(b16);
c.add(b17);

/* Add a action listener for all buttons so an event occurs when
   user/programmer clicks on them */
```

```
    ButtonHandler handler = new ButtonHandler();
    b1.addActionListener(handler);
    b2.addActionListener(handler);
    b3.addActionListener(handler);
    b4.addActionListener(handler);
    b5.addActionListener(handler);
    b6.addActionListener(handler);
    b7.addActionListener(handler);
    b8.addActionListener(handler);
    b9.addActionListener(handler);
    b10.addActionListener(handler);
    b11.addActionListener(handler);
    b12.addActionListener(handler);
    b13.addActionListener(handler);
    b14.addActionListener(handler);
    b15.addActionListener(handler);
    b16.addActionListener(handler);
    b17.addActionListener(handler);
    System.out.println("Graphics initialized" + "\n");
    setSize(600,400);
    show();
}

/* Method to start up sockets on Java side */
public void setUpSockets() throws java.io.IOException {
    broker = new ServerSocket(4515);
    System.out.println("Broker socker: " + broker.getLocalPort() + "\n");

    hearing = new ServerSocket(0);
    System.out.println("Hearing socket: " + hearing.getLocalPort() + "\n");

    vision = new ServerSocket(0);
    System.out.println("Vision socket: " + vision.getLocalPort() + "\n");

    animation = new ServerSocket(0);
    System.out.println("Animation socket:  " + animation.getLocalPort() + "\n");
```

```
        speech = new ServerSocket(0);
        System.out.println("Speech socket: " + speech.getLocalPort() + "\n");

        navigation = new ServerSocket(0);
        System.out.println("Navigation socket " + navigation.getLocalPort() + "\n");

        mood = new ServerSocket(0);
        System.out.println("Mood socket: " + mood.getLocalPort() + "\n");

        timer = new ServerSocket(0);
        System.out.println("Timer socket: " + timer.getLocalPort() + "\n");

        sm = new ServerSocket(0);
        System.out.println("SM socket: " + sm.getLocalPort() + "\n");

        /* Store the port numbers generated in ports[] array */
        ports [0] = hearing.getLocalPort() + "\n";
        ports [1] = vision.getLocalPort() + "\n";
        ports [2] = animation.getLocalPort() + "\n";
        ports [3] = speech.getLocalPort() + "\n";
        ports [4] = timer.getLocalPort() + "\n";
        ports [5] = navigation.getLocalPort() + "\n";
        ports [6] = sm.getLocalPort() + "\n";
        ports [7] = mood.getLocalPort() + "\n";

    }

    /* Method for establishing socket connections between Java
       program and lower layer of GLAIR (PMLb) */
    public void establishConn () throws java.io.IOException {
        /* Communicate port numbers through broker socket connection to PMLb of GLAIR */
        for(int i=0; i<8; i++) {
            brokerWrite.write(ports[i].getBytes());
        }
         System.out.println("Wrote socket port numbers to broker-socket");

         System.out.println("Waiting for Hearing modality connection" + "\n");
```

```
        hearingConnection = hearing.accept();
        System.out.println("Hearing modality connection accepted" + "\n");
        hearingWrite = hearingConnection.getOutputStream();

        System.out.println("Waiting for Vision modality connection" + "\n");
        visionConnection = vision.accept();
        System.out.println("Vision modality connection accepted" + "\n");
        visionWrite = visionConnection.getOutputStream();

        System.out.println("Waiting for Animation modality connection" + "\n");
        animationConnection = animation.accept();
        System.out.println("Hearing modality connection accepted" + "\n");
        animationWrite = animationConnection.getOutputStream();

        System.out.println("Waiting for Speech modality connection" + "\n");
        speechConnection = speech.accept();
        System.out.println("Speech modality connection accepted" + "\n");
        speechWrite = speechConnection.getOutputStream();

        System.out.println("Waiting for  modality connection" + "\n");
        timerConnection = timer.accept();
        System.out.println("Timer modality connection accepted" + "\n");

        System.out.println("Waiting for Navigation modality connection" + "\n");
        navigationConnection = navigation.accept();
        System.out.println("Navigation modality connection accepted" + "\n");

        System.out.println("Waiting for Stage Manager" + "\n");
        smConnection = sm.accept();
        System.out.println("Stage Manager connection accepted" + "\n");

        System.out.println("Waiting for Mood modality connection" + "\n");
        moodConnection = mood.accept();
        System.out.println("Mood modality connection accepted" + "\n");
    }
}
```