Cogent: A Coherence-Driven Cognitive Agent Modeling and Simulation Framework

by

Sunit Sivaraj

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 05, 2018

Keywords: Agent-Based Modeling, Cognitive Agent, Cognitive Coherence, Ethical Decision-Making, Domain-Specific Language

Abstract

Agent based modeling is an effective methodology for understanding complex cognitive systems through generative mechanisms that explain emergent behavior. However, most agent modeling languages use general-purpose imperative language constructs and lack high-level syntactic features necessary for cognitive modeling. In this thesis, we present Cogent, a cognitive coherence-driven agent modeling and simulation platform and demonstrate its use in ethical decision-making by autonomous systems. The underlying strategy is based on parallel constraint satisfaction in terms of a connectionist interactive activation model that implements the theory of cognitive coherence. Agents in the Cogent language are specified by a Domain-Specific Language, which provides the syntax and semantics for creating the decision-making model along with its coherence network. A model in Cogent expresses the constraints of a cognitive architecture through mechanisms such as explanation, deliberation, deduction, and analogy. The language provides the ability to model complex hierarchical network structures while allowing the visualization of the coherence network to gain cognitive affordance into explaining an agent's behavior and decisions. To illustrate the utility of Cogent, we explore a case study in machine ethics and demonstrate how deontological and consequentilist approaches to decision-making in philosophical ethics can be simulated using the Cogent.

Acknowledgments

I would like to thank my advisor, Dr. Levent Yilmaz for providing guidance, patience, confidence, and motivation throughout my graduate study at Auburn. Dr. Yilmaz, not only gave me valuable suggestions, he also helped me to grow and think as a researcher. I would like to express my sincere gratitude towards my advisory committee members Dr. Saad Biaz and Dr. Cheryl Seals for their participation in my advisory committee and their guidance during my graduate studies. I would also like to thank my parents and my sister for their selfless love, support, and belief in me.

Table of Contents

List of Figures

Chapter 1

Introduction

Software agents when simulated can be helpful in understanding a complex problem or scenario. The use of Agent-Based Models (ABMs) or Individual-Based Models (IBMs) for research and management is growing rapidly in a number of fields [1]. Based on the application domain of software agents, the importance of understanding their behavior is critical. Agents address a wide range of problems from simple models like the Schelling Model [2] to flight simulations [3]. Some varieties of agents like intelligent agents are equipped with advanced abilities such as decision making, adaptation, learning, and reasoning. Cognitive architectures are often based on theories of cognitive science and aim at describing human cognitive processes as precisely as possible [4]. There are many cognitive architectures that can be used to develop such intelligent agents. An important problem to consider is the need to express deliberation and reasoning mechanisms explicitly in these agents become more important as the complexity of the problem to be faced by the agent increases.

For demonstrating the importance of the problem, consider Autonomous systems which are increasingly becoming part of both the cyber and physical infrastructures that sustain our daily lives [5]. Instilling confidence and trust in them has never been more important. Autonomous systems like self-driving cars and Unmanned Aerial Vehicles (UAV) can face various situations where they might be tangled between taking decisions which may be challenging even for a human [6]. Consider a scenario where we need to deploy an unmanned vehicle for a search and rescue mission during a natural disaster. There may be many people in need of help and the vehicle may need to make a decision which could save some lives but endanger others. Suppose the agent has a rule that it cannot sacrifice a life to save others, and that it encounters a scenario where sacrificing one person's life can lead to saving many others. In such a scenario, the vehicle may have to take a decision between maximizing the total number of lives saved, thus sacrificing the one person or follow the rules.

The above example can be clearly expressed in as either a deontological [7] or consequentialist [8] approach in ethical decision-making. Deontological and consequentialist perspectives are two approaches to moral decision making. According to the deontological principle, one must act on duties or rules and the actions taken at any point must obey with these rules. The consequentialist approach is contradictory to the deontological approach where the actions taken need to maximize the observed result, which can be maximizing utility or any other parameter based on the scenario. This theory can easily lead to breaking of rules or duties in contrast to the deontological approach but maximizes utility. This example clearly shows how real-world scenarios involve complicated decision making and the need to express deliberation and reasoning mechanisms explicitly in simulation languages modeling such cognitive agents.

Most of the commonly used ABM platforms follow the "framework and library" paradigm, providing a framework along with a library of software implementing the framework and providing simulation tools [1]. Another major category of ABM platforms is the platforms providing a Domain Specific Language (DSL) or higher-level syntax for modeling agents, such as NetLogo [9]. Unfortunately, both these varieties of simulations have challenges of their own. Evaluation OF ABM platforms [10] indicate that the library-based simulation platforms have drawbacks such as the difficulty in use and the insufficiency of tools for observing simulation experiments or visualizing the results of simulations. Although the DSL-based languages provide a higher level syntax for creating agents and other features like visualization of agents in their platform, they lack the ability to generalize. That is, constraints of the agent language make it difficult to define complex behaviors. DSL-based simulation languages do not allow programming language syntax and libraries to be used along with the DSL. This clearly shows the need for simulation platforms that could combine the positive attributes of both the variants of simulation languages discussed.

We address the above problems by developing a simulation platform, called *Cogent*, for creating and experimenting with cognitive agents using a higher-level language. The cognitive model of the cogents, which are agents in the Cogent language, is based on the foundational principles of the theory of cognitive coherence [11]. The cogents can be simulated using a Domain Specific Language (DSL) integrated as a part of the Cogent simulation platform where users can program cogents using the DSL, which provides high-level abstract features based on the theory of cognitive coherence [11]. The features aim to facilitate explicit specification of the perception and deliberation mechanisms. That is, beliefs, evidences, and their relations such as explanation and facilitation are characterized in terms of a programmable abductive model building framework. The DSL of Cogent aims to address the issue of expressing deliberation and reasoning mechanisms explicitly. We validate Cogent's expressive DSL capabilities later in Section 5, where we simulate a Machine Ethics problem similar to the one discussed before in this section. The proposed language and its syntactic constructs can be used for creating autonomous agents and can also be used both as an exploratory and explanatory tool for studying the behavior of these agents in a simulated environment. By examining the behavior of agents in a transparent manner, users can gain cognitive affordance into the behavior of the models and adjust specifications until they are satisfied and instill trust in agent behavior.

The Cogent framework also aims to address the issues faced by the library-based and the DSL-based simulation languages discussed before. Cogent's DSL is carefully designed to not only be expressive but also to allow users to directly connect the programming language syntax to a cognitive theory. That is, the deliberation and reasoning mechanisms are specified using the DSL, and the behaviors executed by agents are Scala programming language functions. The DSL can help programmers implement complex decision-making agents using the DSL while providing them imperative features in terms of Scala functions. This strategy aims to address the tediousness in the library-based languages and the limited available functionality of the DSL-based approaches. The simulation platform also provides facilities for the visualization of the cognitive model defined by an explanatory coherence network using the DSL. Because the cognitive model is a composition of nodes and constraints defined by edges between the nodes, the model is a network. The visualization represents the actions which are classified acceptable or unacceptable based on the input given in the DSL and the result generated by the coherence network via a parallel constraint satisfaction mechanism. This allows users to get a better understanding of the model and can interactively modify the specification to adjust the behavior.

The rest of the thesis is structured as follows. In the next chapter, we overview various existing simulation platforms and cognitive architectures, discussing their features and drawbacks. Chapter 3 introduces the conceptual framework of the Cogent platform and discusses in detail the theory of Coherence and the various types of coherence mechanisms. We also introduce the basic concepts and components of Model Driven Engineering, including Domain-Specific Languages, model transformations, and metamodels. In Chapter 4, we present the implementation details of the Coherence Theory as well as the major components of the Model Driven Engineering framework. We also discuss the syntactic details of the Cogent's DSL and the various features that the platform provides. We demonstrate how Coherence Theory combined with the principles of Model Driven Engineering provides a basis for the Cogent platform. In Chapter 5, we introduce the domain of Machine Ethics and present a problem in the area called the Lethal Strike problem. By using the problem, we validate the platform's capability to model cognitive agents using deliberation mechanisms. In the final chapter, we conclude and discuss both the limitations and potential applications of the Cogent platform. We also discuss potential areas for further research and extension of Cogent.

Chapter 2

Previous Work

Agent-based simulation languages overall aim to simulate a problem through agents, but they also vary greatly in their approach. Based on the simulation language used, the type of functionality of the simulated agents varies as well. That is, although simulation languages can be used to develop a wide range of agents in different domains, the complexity of simulating these agents can vary depending on the formalism of the language used. In this section, we will discuss existing agent simulation languages, their benefits, and limitations. We will also discuss on how Cogent proposes to address the problems faced by these languages.

2.1 Simulation Languages

Agent simulation languages can be classified based on multiple criteria, ranging from programming language, to scalability, development effort, and operating systems [12]. In this section we will classify the languages and explore them as either library-based or DSL-based. library-based simulation languages are where agents are developed using libraries that run over a programming language. DSL-based languages are simulation platforms which have a Domain-Specific Language (DSL) of its own for simulation. We will first discuss library-based languages and then overview DSL-based approaches.

2.1.1 Library-based Simulation Languages

Swarm [13] is a multi agent simulation platform for complex adaptive systems. Agents in Swarm are organized by an object called ”swarm”, where the ”swarm” object can itself be an agent. In Swarm, the different types of agents are separate classes and specific agents are objects of those classes. Each of the agents or objects have separate state variables while sharing common behaviors defined in the class. Swarm allows modelers to build multi-level models. That is, to define a hierarchical Swarm models where one Swarm agent may have multiple Swarm agents within it.

Repast [14] is a Java based software for multi-agent simulation. It allows users to simulate multiple agents that can coordinate with each other. Repast also allows simple agent scheduling, as well as features that enable complex dynamic scheduling for agents. The simulation software allows agents to exhibit behaviors through the execution of basic actions defined as methods of software objects. That is, Repast utilizes a set of software objects to handle the mechanisms used for agent simulation like scheduling and visualization, and new agents can be created by extending these classes. Repast also provides features that enable the specification of agent behavior in terms of state charts and activity flows. As such, it provides a visual DSL to abstract agent behavior.

Repast Simphony [15] is built on top of the Repast library [14] for modeling complex adaptive systems. Repast Simphony was designed with the goal to keep all the core features offered in Repast with a strict separation between model specification, model execution, model visualization, and data storage. All user model objects in Repast Simphony such as networks and time scheduling, are Java objects that can be replaced by custom objects if needed by the modeler. It also offers visualization features such as 2D and 3D visualization of agents.

MASON [16] is a multi-agent simulation toolkit developed in Java. MASON uses a fast, orthogonal, minimal model library which can be modified or extended by a Java programmer [16]. The toolkit consists of three layers. The utility layer consists of various helper functions. The model layer contains the core features for running the agent like scheduling, and the visualization layer allows users to create visualizations and Graphical user Interfaces for the simulation. As compared to Repast, MASON provides sophisticated 3D visualization features and is argued to be faster than Repast [16].

JADE (Java Agent DEvelopment Framework) [17] allows users to simulate agents in compliance with the FIPA [18] standard, which facilitates interoperability between agents developed within the same standard. JADE is also a distributed agent platform, thus running a simulation in multiple machines. JADE allows users to add behaviors from simple atomic ones to complex with sub behaviors by extending specific Java classes which is accompanied with the toolkit. Jadex [19] is also a Java based multi-agent simulation platform that implements the Belief Desire Intention (BDI) model of means-ends reasoning. The BDI model allows users to express simulation instructions in terms of goals, beliefs and plans. Jadex provides a meta-level reasoning mechanism where based on the events passed to the agent appropriate plans are selected from the list of available plans. In a BDI model, causes for actions are only related to desires, ignoring other facets of cognition such as, emotions [19]. We will discuss later on how emotions can be included in our model presented in this work.

All the languages discussed in this section follow a common approach for modeling agents. In all of these languages, agents are modeled by extending a set of classes from a library of a object-oriented programming language. Not only is the simulation procedure in these languages are very similar, they also do not allow human reasoning constructs like beliefs or evidences to be explicitly expressed for modeling agents. They also lack built-in reasoning and decision-making capabilities within agents, and any decision making procedure must be programmed using imperative low-level language constructs. In Jadex, although BDI-based reasoning mechanisms can be expressed, other mechanisms such as analogy and explanation are ignored.

2.1.2 DSL-based Simulation Languages

StarLogo [20] is a modeling environment designed primarily to help non expert users model decentralized systems. Using StarLogo, users can write simple rules for thousands of objects, then observe the patterns that arise from the interactions [20]. StarLogo was extended from the Logo programming language [21] by adding multi-agent capabilities and parallelism to it. StarLogo introduces parallelism in agents in the belief that it would be the most natural way to express the desired behavior of agents. Although multiple agents can be simulated at the same time in StarLogo, those agents can be of only two types, thus limiting its capabilities.

NetLogo [9] is also a multi-agent simulation language. NetLogo is primarily intended for simulating natural and social phenomena [9]. Similar to StarLogo, NetLogo allows agents to be programmed with a DSL. The disadvantage of NetLogo is that it is highly restricted to four types of agents moving and interacting within a 2 dimensional space thus failing to generalize to explicitly define a cognitive model in terms of the language of a cognitive theory.

ReLogo [22, 23] is aimed to create a semantically simple yet powerful package for agent-based modeling by combining the ideas of Repast Simphony [15] and NetLogo [9] through a DSL. ReLogo adds the capabilities of Repast Simphony [15] to the semantic simplicity of Logo, and in doing so contributes a number of concepts to building Logo-like ABMs; object-oriented programming, simple integration of existing code libraries, statically and dynamically typed languages, domain-specific languages, and the use of integrated development environments [22]. ReLogo's DSL allows modelers to program NetLogo type agents involving "patches", "turtles", "links", and "observers" type agents by inheriting already available base classes. Relogo also allows external Java libraries to be added to the model.

Languages in this section follow a different approach in modeling agents. Agents in these languages are modeled using a DSL. Although the languages provide a higher-level syntax for modeling agents, they lack the ability to express reasoning constructs explicitly. Creating agents with complex behaviors that require complex computations can be particularly difficult in StarLogo and NetLogo as they do not allow programming language syntax to be combined with the DSL. We will show how Cogent's DSL not only allows reasoning constructs to be expressed while modeling agents but also allows programming language syntax to be combined with the DSL.

2.2 Cognitive Architectures

A fundamental characteristic of an agent is its potential to make independent deliberations during the problem-solving, conflict resolution and decision-making processes [12].

Cognitive architectures are different from the simulation languages or frameworks discussed in the previous section. Frameworks provide a common base of code and some representational schemes for knowledge, but they do not provide a comprehensive set of memories and processing elements that are sufficient for the development of general agents [24].

Soar [25, 24] is a cognitive architecture based on symbolic and non-symbolic processing. Soar's architecture consists of multiple long-term memories and a short-term memory. The short-term memory is represented as a symbolic graph where objects in them can be represented with properties and relation. There are three long-term memories which are the Procedural memory for encoding procedural knowledge as production rules, the Semantic memory for storing the facts and the Episodic memory for storing past experiences of an agent. Soar provides reinforcement learning to adjust actions of an agent to maximize a reward. Soar also allows agents to express emotions based on a certain appraisal theory. The production rules in Procedural memory of Soar is expressed in the format of mapping a set of conditions to actions which are compared later to the short-term memory. Although this syntax of the language may be good enough to express various real-world scenarios it does not allow human reasoning mechanisms to expressed explicitly.

EPIC (Executive Process-Interactive Control) [26] is a human information-processing architecture that is especially suited for modeling multiple-task performance [26]. EPIC has a production rule cognitive processor along with perceptual motor models. Information in agent's start from a perceptual processor to a cognitive processor and then to the motor processor. There are separate memory modules for storing production rules and declarative knowledge. There is also a Working memory which acts like a Short Term memory. There is no learning mechanism in this architecture. To handle the incapacity to learn, there are models like EPIC-Soar [27], where EPIC is combined with Soar [25]. In EPIC-Soar, EPIC and Soar remain independent programs which communicate using a socket connection. Soar accepts EPIC perceptual and motor processor messages as input to its working memory and returns motor processor commands to EPIC as output [27].

ACT-R [28] is a hybrid cognitive architecture consisting of symbolic and sub-symbolic processing. ACT-R primarily consists of two perceptual-motor modules namely the visual module and the manual motor module for interacting with the real world. Similar to Soar, ACT-R consists of memory modules like the procedural module for storing productions and a declarative module for storing facts. ACT-R does not have an explicit short-term memory but it has buffers for each module described before except for the procedural module. These buffers represent the current state of an agent at any given time. A pattern matching module chooses actions to fire based on the current states of the buffer and the procedural memory. When multiple rules match, ACT-R chooses one based on a sub-symbolic procedure. Productions in the procedural module in ACT-R are expressed in the format of mapping tests on buffer contents to actions. This is clearly similar to how Soar handles its procedural memory.

Clarion [29] is cognitive architecture for modeling a single agent. The components of Clarion are the action centered system to control physical and mental operations (ACS), non-action-centered subsystem (NACS) to store general knowledge, motivational subsystem (MS) to provide underlying motivations for cognition and the metacognitive subsystem (MCS) to monitor the operations of ACS. Clarion differentiates from Soar and ACT-R by the fact that it clearly tries to separate implicit and explicit cognition. This distinction can be clearly seen in the ACS and NACS module. In ACS, the explicit cognition is handled by symbolic processing while the implicit processing is handled by neural networks. The top level of Clarion's ACS maps states to actions while the bottom level uses a neural network associating the states and pattern using learning mechanisms. Similar to Soar and ACT-R and Clarion also does not allow users to express deliberation mechanisms explicitly.

LIDA [30] is an extension of IDA, an autonomous agent used in the United States Navy. LIDA extends IDA by adding perceptual, episodic and procedural learning to it. LIDA works by organizing the cognitive process in a cycle, beginning from perception and ending in an action. LIDA gains new knowledge over several of these cognitive cycles. The cognitive cycle begins with a Perceive phase, where the sensory input of the environment is obtained. The Perceive phase recognizes or establishes unfamiliarity of the sensed environment. In LIDA, an agent's ontology can be defined as a list of objects and classes that the agent can currently recognize. The cognitive cycle's second phase is the Interpret phase, where the precept from the previous phase is moved to a long-term memory and analyzed. Information from the Perceive phase is passed onto the third phase of LIDA known as Act Phase. In the Act Phase, the agent works with the Procedural Memory and the Action Subsystem, where the procedural memory is a collection of procedural rules. In this phase, the context sent from the Perceive Phase is matched with the Procedural Memory and appropriate action is performed.

ICARUS [31] is a cognitive architecture for modeling physical agents that are influenced by the results from cognitive psychology. ICARUS is not to match quantitative data, but rather to reproduce qualitative characteristics of human behavior [31]. The environment in ICARUS is represented using a perceptual buffer. Agents in ICARUS understand the environment by matching conceptual structures against precepts and beliefs. ICARUS also has a belief memory that has higher-level inferences about the agent's situation. The belief memory holds predicates and a set of symbolic arguments of objects that typically appear in the perceptual buffer. The basic activity of an agent in ICARUS is a conceptual inference in which the perceptual buffer is matched with the agent's long-term memory or conceptual memory and adding the result to the belief memory. Goals are an important part of an autonomous system and ICARUS has a goal memory that contains agent's top-level objectives. Goals in goal memory are stored as an ordered list and goals with the highest priority is selected during each cycle. The knowledge to execute the goals are stored in a long-term memory called the skill memory.

All of the Cognitive architectures discussed above, except Clarion, are based on procedural rule system for cognition or decision making. Clarion, along with a procedural rule system, also uses a neural network for implicit processing. Programming agents in production rule systems can be particularly challenging for users as they may have to convert their deliberations into production rules. Allowing users to directly express these deliberations will help them speed up and ease the process of simulation. Our proposed approach tackles the challenges faced by these languages in expressing deliberation constructs by introducing agents that can express various forms of deliberations like explanation and analogy. We work towards simulating autonomous decision making agents where these reasoning constructs can be included within agents.

Chapter 3

Conceptual Framework

In this chapter, we will discuss about the conceptual framework of the Cogent Platform. Agents that are modeled in the Cogent platform are cognitive agents which work on the basis of the Coherence Theory. In section 3.1, we will introduce Coherence theory and the different types of Coherence mechanisms. We will also introduce Model Driven Engineering, the basis on which the Cogent framework works. Model Driven Engineering allows modeler to define cognitive models in a higher level syntax, thus simplifying the process. In Section 3.2, we will introduce Model Driven Engineering and its attributes in detail.

3.1 Coherence Theory

Coherence Theory [11] is a cognitive reasoning technique based on psychological evidences where certain principles are rejected or accepted based on a constraint satisfaction mechanisms. The different types of coherence mechanisms in Coherence Theory are explanatory coherence, deductive coherence, deliberative coherence and analogical coherence. A coherence network is a representation of Coherence Theory where elements like Hypothesis and Evidences are represented as nodes and are connected with edges having weights which are based on if the elements cohere or incohere with each other. Let us first discuss the explanatory coherence in detail. The other forms of coherence are discussed, later in this section.

3.1.1 Explanatory Coherence

Explanatory coherence [32] aims to find a mutual adjustment between different hypothesis and evidences. By explanatory coherence, a hypothesis can be explained or contradicted by other hypothesis or evidences. *Explanatory coherence* also involves Competitions, which are inhibitory connections established between hypothesis that explain a common evidence but do not have any explanation between them. The seven principles of explanatory coherence are [33]:

Symmetry: Explanatory coherence is a symmetric relation, unlike say, conditional probability.

Explanation: (a) A hypothesis coheres with what it explains, which can either be evidence or another hypothesis; (b) hypotheses that together explain some other proposition cohere with each other; and (c) the more hypotheses it takes to explain something, the lower the degree of coherence.

Analogy: Similar hypotheses that explain similar pieces of evidence cohere.

Data Priority: Propositions that describe the results of observations have a degree of acceptability on their own.

Contradiction: Contradictory propositions are incoherent with each other.

Competition: If P and Q both explain a proposition, and if P and Q are not explanatorily connected, then P and Q are incoherent with each other. (P and Q are explanatorily connected if one explains the other or if together they explain something.)

Acceptability: The acceptability of a proposition in a system of propositions depends on its coherence with them.

3.1.2 Deductive Coherence

Deductive coherence discerns a balance between propositions, which can be beliefs or hypotheses. These beliefs, in turn may deduce other beliefs or hypotheses, thus establishing a set of rules. Similar to explanatory coherence, beliefs can also contradict each other. The five principles of Deductive coherence are [33]:

Symmetry: Deductive coherence is a symmetric relation among propositions, unlike, say, deducibility.

Deduction: (a) An axiom or other proposition coheres with propositions that are deducible from it; (b) propositions that together are used to deduce some other propositions cohere with each other; and (c) the more hypotheses it takes to deduce something, the less the degree of coherence.

Intuitive priority: Propositions that are intuitively obvious have a degree of acceptability on their own. Propositions that are obviously false have a degree of rejectability on their own.

Contradiction: contradictory propositions are incoherent with each other.

Acceptance: The acceptability of a proposition in a system of propositions depends on its coherence with them.

3.1.3 Deliberative Coherence

Deliberative coherence [34] finds a mutual adjustment between actions and goals of a problem. While goals can facilitate actions, they can also facilitate each other. Goals can also be incompatible with other goals or actions. The six principles of *Deliberative Coherence* are [34]:

Symmetry: Coherence and incoherence are symmetrical relations: If a factor (action or goal) $F1$ coheres with a factor $F2$, then $F2$ coheres with $F1$.

Facilitation: Consider actions $A1$ ... An that together facilitate the accomplishment of a goal G. Then (a) each Ai coheres with G; (b) each Ai coheres with each other Aj; (c) the greater the number of actions required, the less the coherence among actions and goals.

Incompatibility: (a) If two factors cannot both be performed or achieved, then they are strongly incoherent; (b) If two two factors are difficult to perform or achieve together, then they are weakly incoherent.

Goal priority: Some goals can be desirable for intrinsic or non-coherence reasons.

Judgment: Facilitation and competition relations can depend on coherence with judgments about the acceptability of factual beliefs.

Decision: Decision are made on the basis of an assessment of the overall coherence of a set of actions and goals.

3.1.4 Analogical Coherence

Analogical coherence [33] is the mapping of hypotheses and their corresponding propositions to similar hypotheses and their propositions. Similar hypothesis work collaboratively in a way, if one of them is negatively affected the similarly mapped hypothesis has a negative effect as well. The six principles of *Analogical Coherence* are [33]:

Symmetry: Analogical coherence is a symmetric relation among mapping hypotheses.

Structure: A mapping hypothesis that connects two propositions R(a,b) and S(c,d), coheres with mapping hypotheses that connect R with S, a with c, and b with d; and all those mapping hypotheses cohere with each other.

Similarity: Mapping hypotheses that connect elements that are semantically or visually similar have a degree of acceptability on their own.

Purpose: Mapping hypothesis that provide possible contributions to the purpose of the analogy have a degree of acceptability on their own.

Competition: Mapping hypotheses that offer a different mapping for the same object or concept are incoherent with each other.

Acceptance: The acceptability of a mapping hypothesis in a system of mapping hypotheses depends on its coherence with them.

3.1.5 Parallel Constraint Satisfaction in IAC Networks

The Coherence Network is based on the concept of Interactive Activation Competition Network [35], also called the *IAC* network. The *IAC* network consists of nodes that are connected to each other through bi-directional edges. The nodes can have an activation value between a minimum and a maximum value set by the user. The edges that connect the nodes can also have positive or negative weights associated with them.

To run the network, the $net - input$ values for each node are obtained, then an update function is applied. The $net - input$ of a node is calculated as follows:

$$net - input(j) = \sum_{i=1}^{n} w(i,j) * a(i)$$

In the above equation, $w(i,j)$ is the weight of the connection between nodes i and j. $a(i)$ represents the activation of the node i. The links in an IAC network are bidirectional, hence $w(i,j)$ would be the same as $w(j,i)$. The update action can be given as follows [35]:

$$a(t+1) = \begin{cases} a(t) * (1-\theta) + net - input * (M - a(t)), & \text{if } net - input > 0 \\ a(t) * (1-\theta) + net - input * (a(t) - m), & \text{otherwise} \end{cases}$$

In the above equation, $a(t)$ and $a(t+1)$ represent the activations of a node at time t and time $t+1$ respectively. Values m and M represent the minimum and maximum value set for the network, and θ represents the *decay* parameter. The *decay* parameter controls the speed at which the network settles. The update function is applied for all nodes in the network until all of the nodes settle. The network settles if the difference between activations of all nodes in two successive states is less than the *threshold* set for the network.

3.1.6 Contrastive Hebbian Learning in Coherence Network

In this section, we will discuss the idea of fitting a coherence network to the preferences of a user. Precisely, given a coherence network which settles at a configuration A, we will discuss the procedure that will allow the network to settle in the desired configuration B.

A coherence network in contrast to a neural network instills very little restrictions on how nodes are connected to each other, which reduces the options for applying popular learning algorithms like backpropagation which rely on the constrained architecture of a neural network. Coherence networks also use a symmetric bi-directional edge to connect its nodes, and does not differentiate input and output nodes. The closest similarity to a coherence network among popular network architectures would be a Boltzmann machine [36] as they too have symmetric bi-directional edges, we will now introduce the learning algorithm used in them called the Contrastive Hebbian Learning [37].

Contrastive Hebbian Learning has two phases, which are the positive and negative phase. In the positive phase, the network is run while both the input and output are presented to the network. In the negative phase, the network is run while only the input is presented to the network. The weights of the edges in the network are updated as follows [38]:

$$\triangle W_{ij} = \epsilon * ((a_i^+ * a_j^+) - (a_i^- * a_j^-))$$

The above equation specifies the procedure for updating the weight of an edge between any node i and j. a_i^+ and a_j^+ in the equation are the activations of nodes i and j in the positive phase. a_i^- and a_j^- represent the activations of node i and j in the negative phase. ϵ is the learning rate. The learning algorithm involves updating each edge using the above described procedure.

3.2 Model Driven Engineering

We introduced Cogent's cognitive model in Section 3.1. Modeling the theory of coherence can be a challenging task, as the modeler would require substantial knowledge about the domain. We address this challenge by utilizing the principles of Model Driven Engineering. Model Driven Engineering is an approach to improve the software development process by improving software re-usability, and abstraction in program specification, thus leading to a higher productivity. Model-driven approaches putatively increase developer productivity, decrease the cost (in time and money) of software construction, improve software re-usability, and make software more maintainable [39]. Cogent is built on the principles of Model Driven Engineering, thus sharing some of its prime advantages, such as software re-usability and higher productivity. Figure 3.1 gives an overview of Model Driven Engineering [40]. We will discuss each component of Model Driven Engineering below.

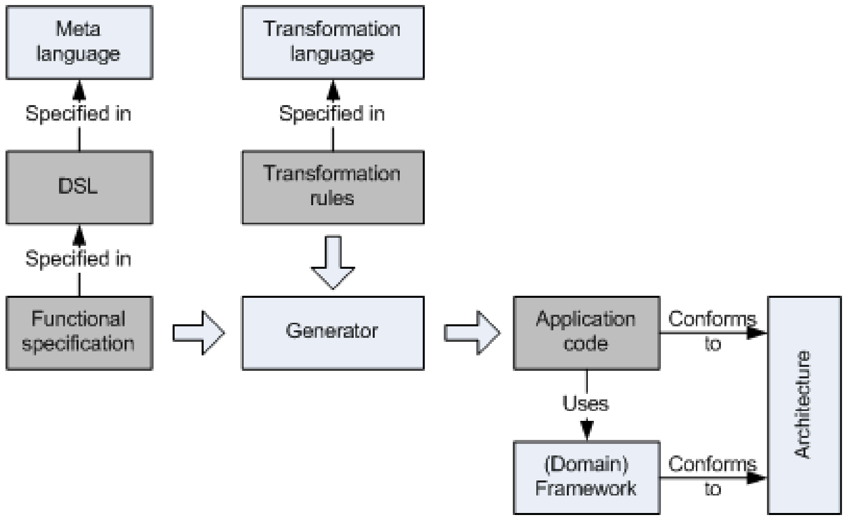


Figure 3.1: Overview of Model Driven Engineering

3.2.1 Model and Domain

The primary component in Model Driven Engineering is a model. ”A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system.” [41]. In the Cogent platform, a model represents a system that incorporates the theory of coherence. Domain describes a bounded field of interest or knowledge [42]. In Model Driven Engineering, models represent a particular domain and the languages to specify these models are called Domain-Specific languages. In Cogent, the domain is the theory of coherence.

3.2.2 Domain-Specific Language

A Domain-Specific Language (DSL), in contrast with a general-purpose programming language, models aspects of a particular problem domain and provides special-purpose constructs tailored to the needs of that domain [39]. The process of creating models using the higher-level DSL is called Domain-Specific Modeling. Developers can use DSL to build applications using elements of the type system captured by metamodels and express design intent declaratively rather than imperatively [43]. A model specified using a DSL is called a Domain-Specific Model (DSM). Domain-Specific languages allow users to specify models in a higher-level syntax pertinent to the domain of the model, thus abstracting the user of any lower level intricacies of the model. Cogent's DSL allows a cognitive model to be defined using keywords such as "explains" and "contradicts", but abstracts the user from how these functionalities are implemented. The syntax details of Cogent's DSL are discussed in detail in Section 4.2.3.

3.2.3 Metamodel

In Domain-Specific Modeling, a model is defined by a DSL, and the DSL is defined by a metamodel. The Domain-Specific Language provides a concrete syntax and the metamodel provides an abstract syntax for modelling. A metamodel is used to define the syntax and semantics of the DSL. More precisely, a metamodel describes the possible structure of models in an abstract way, it defines the constructs of a modeling language and their relationships, as well as constraints and modeling rules but not the concrete syntax of the language [42]. The metamodel itself can be a model defined by a meta-metamodel. meta-metamodel defines a language to define the metamodel. The implementation details of Cogent's metamodel are discussed in Section 4.2.1.

3.2.4 Formal Model

A formal model is the set of statements defined by the DSL that conceptually satisfies an instance of the metamodel definition. A formal model can be considered as the beginning point in the transformation process.

3.2.5 Platform

Platform can be considered as an overall system that acts as a base for running the model. The platform has the task of supporting the realization of the domain, that is, the transformation of formal models should be as simple as possible [42]. The platform provides packages and helper classes that help in the transforming and execution of the formal model. In Cogent, the Scala programming language, transformation classes and the reference implementation form the platform.

3.2.6 Transformation

To transform the higher-level formal model into a lower-level model such as a programming language model, model transformation rules are applied to the formal model. Transformations can be a model-to-model transformation or a model-to-platform transformation. In a model-to-model transformation, a model which conforms to a metamodel is transformed into another model which conforms to a different metamodel. In model-to-platform transformation, a model which conforms to a metamodel is transformed into an artifact acceptable to the platform.

3.2.7 Reference Implementation

Reference implementation refers to the set of classes that provide the functionality of the model specified by the user. The abstract model specified by the user cannot be directly executed as it would lack the necessary procedures to execute. The reference implementation acts as a back-end for the model specified using DSL. In Cogent, the reference implementation would be a set of classes implementing the Coherence Theory. The implementation details of Cogent's reference implementation are discussed in Section 4.2.2.

Chapter 4

Cogent: A Coherence-Driven Agent Simulation Language

4.1 Overview

In this section, we introduce Cogent, a web-based simulation platform developed over the Scala programming environment and the Scalatra [44] web-framework. Cogent is designed based on the principles of Model-Driven Engineering, discussed in Section 3.2. Cogent also incorporates the four different types of coherence discussed in Section 3.1. In this chapter, let us call the agents simulated in Cogent platform as *cogent*. The overall process in simulating cogents in the Cogent Platform is illustrated in Figure 4.1.

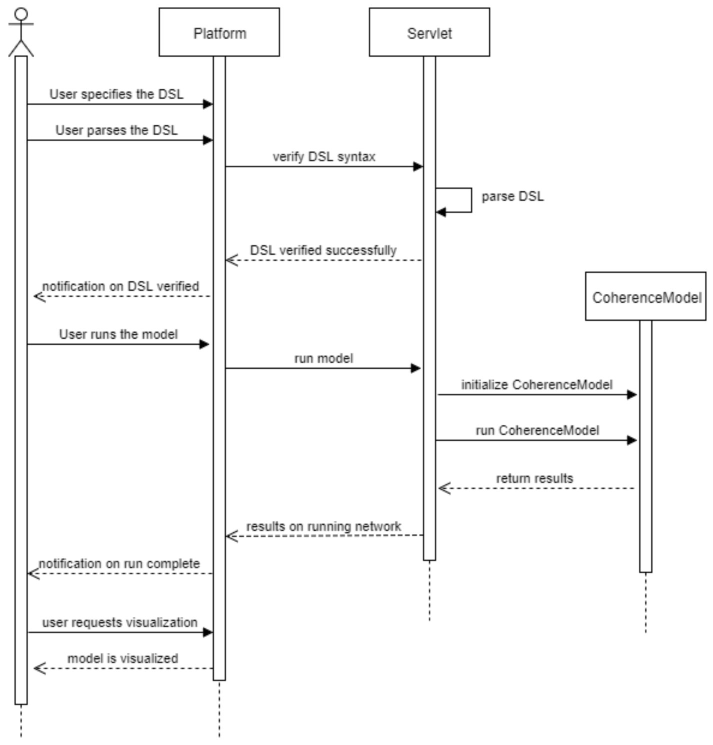


Figure 4.1: Cogent System Sequence Diagram

From Figure 4.1, we can observe that the simulation process begins with the user specifying the DSL in Cogent and parsing it. The DSL syntax is discussed in detail in Section 4.2.3. The DSL is sent from the platform to the servlet. The servlet verifies the syntax of the DSL and responds appropriately to the platform. The metamodel and parsing process of DSL is discussed in detail in Section 4.2.1. After a successful verification of the DSL, the user proceeds to run the model. On receiving the run command, the servlet creates a Coherence Network object. A detailed description on the implementation of the Coherence Network can be found in Section 4.2.2. The results on running the Coherence Network are then propagated back to the user for visualization. A detailed description of the visualization and other features available in the Cogent platform can be found in Section 4.3.

4.2 Cogent's Domain-Specific Language

The DSL of Cogent is intended to ease the process of the agent simulation by hiding intricacies pertinent to the programming language and providing only a simplified syntax necessary to create models. We will begin this section by explaining the metamodel of the DSL. We will further discuss the reference implementation. Finally, we will introduce Cogent's DSL syntax with concrete examples.

4.2.1 Metamodel Implementation

In this section, we will discuss the design and implementation of Cogent's metamodel. Cogent's metamodel and grammar are defined using the *JavaTokensParser* trait in the Scala *combinator* library [45]. The metamodel is designed to capture the various aspects of the DSL, defined by the user during model specification. Figure 4.2 presents the structure of the metamodel design, with each class handling a specific part of the DSL syntax.

- Evidence: This class specifies an evidence, which is a part of *Explanatory Coherence*. The *name* attribute is used to save the name of the evidence specified by the DSL.

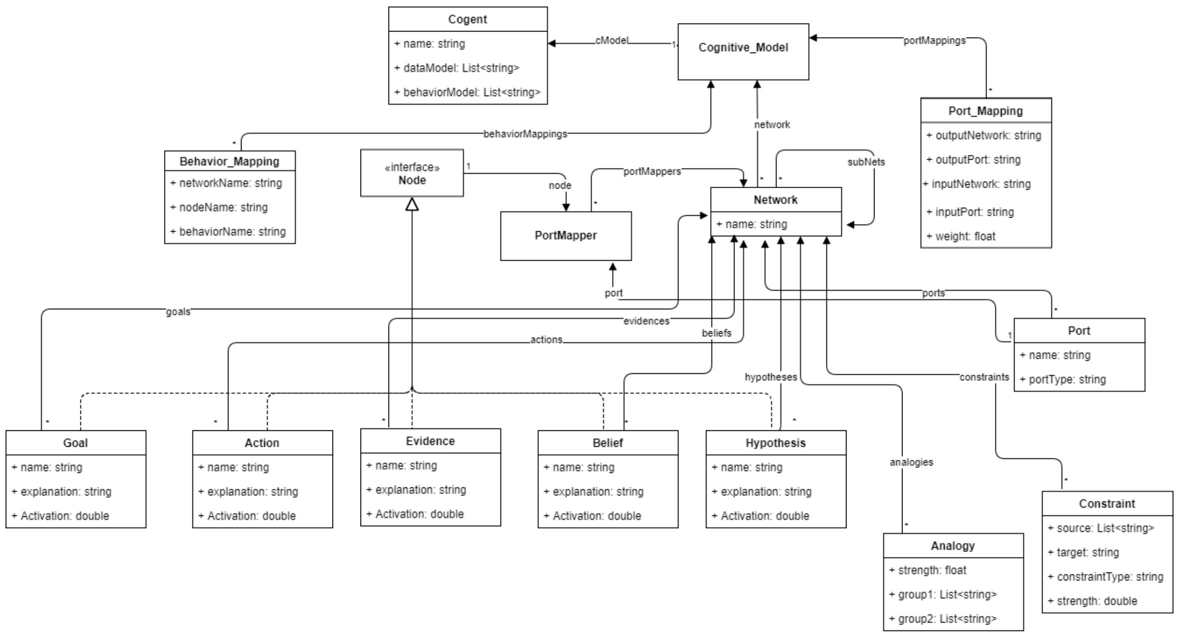


Figure 4.2: Metamodel Implementation

The *explanation* attribute is used to store the description for the evidence specified by the DSL. The *Activation* is used to store the activation of the evidence.

- Hypothesis: This class represents a hypothesis, which is a part of *Explanatory Coherence*. The *name* attribute is used to save the name of the hypothesis specified by the DSL. The *explanation* attribute is used to store the description for the hypothesis specified by the DSL. The *Activation* is used to store the activation of the hypothesis.

- Goal: This class defines a goal, which is a part of *Deliberative Coherence*. The *name* attribute is used to save the name of the goal specified by the DSL. The *explanation* attribute is used to store the description for the goal specified by the DSL. The *Activation* is used to store the activation of the goal.

- Action: This class captures a action, which is a part of *Deliberative Coherence*. The *name* attribute is used to save the name of the action specified by the DSL. The *explanation* attribute is used to store the description for the action specified by the DSL. The *Activation* is used to store the activation of the actions.

- Belief: This class captures a belief, which is a part of *Deductive Coherence*. The *name* attribute is used to save the name of the belief specified by the DSL. The *explanation* attribute is used to store the description for the belief specified by the DSL. The *Activation* is used to store the activation of the belief.

- Analogy: This class aims to represent the analogies, which are a part of Analogical Coherence. *group1* and *group2* are set of hypothesis or evidences. The attribute *strength* denotes the connection strength to be established between the nodes.

- Constraint: This class captures the different type of constraints involved in *Deductive*, *Deliberative* and *Analogical* coherence. The *source* attribute stores values representing the names of a set of nodes. The attribute *target* stores a value representing the name of a single node. The *constraint* attribute stores various constraints, like "explains" or "contradicts", as a string. The *strength* attribute represents the strength of the constraint to be established.

- Network: This class describes the Network specified by the DSL. The network can be a hierarchical or a flat network. The *subNets* attribute can store other network objects, thus representing hierarchical networks. The Network class has attributes to store beliefs, evidences, goals, and hypothesis nodes associated with the Network. The Network class also has an attribute to store the set of ports associated with the Network.

- Port: This class abstracts the port's name and the port's type. The *name* attribute stores the name of the port and the *portType* attribute stores the type of the port as a string.

- PortMapping: This class captures the mapping of ports to the Networks. The attributes *outputNetwork* and *outputPort* are used for storing the output port and the network associated with it. The attributes *inputPort* and *inputNetwork* are used for storing the input port and the network associated with it.

- PortMapper: This class defines the mapping of a node to an input or an output port.
- BehaviorMapping: This class represents the mapping of a behavior to an action node. The attribute *networkName* stores the name of the network in which the action node can be found, the attribute *nodeName* stores the name of the action node that has to be mapped to a behavior. The attribute *behaviorName* stores the name of the behavior. For this mapping to be successful, the behavior name must match the name of the function defined in the behavior section of the DSL.
- CognitiveModel: This class captures the cognitive model of the DSL. This class stores the Network, PortMapping, and BehaviorMapping object.
- Cogent: This class refers to the whole Cogent model, which includes all of the classes defined above. The attributes *dataModel* and *behaviorModel* store the data variables and behavior functions defined in the DSL respectively.

The above classes define the various components of the metamodel. The grammar of the language is defined by extending a class from *JavaTokenParser* trait, and defining various functions using an embedded DSL, thus representing the rules our language. The embedded DSL is a part of the *Combinator* package discussed before, and it can be seen as a very high-level library with multiple operators overloaded. The functions defined using the embedded DSL, can call other functions and they return a specific class as a result of parsing. The transformation from DSL to a Scala model is achieved using the *ParseAll* function, which is a part of the *Combinator* package. Thus, in an event of successful transformation, a *Cogent* object would be returned.

4.2.2 Reference Implementation

The reference implementation is comprised of classes that are used to generate the semantics of a program defined in the language of Cogent. The reference implementation provides classes to create and run the coherence network for a DSL specification. The different classes and their relations can be seen in Figure 4.3. In Section 4.2.1, we discussed the metamodel implementation and transformation from the DSL to *Cogent* class. The *Cogent* class is not executable on its own, hence it will be converted to the classes defined in this section.

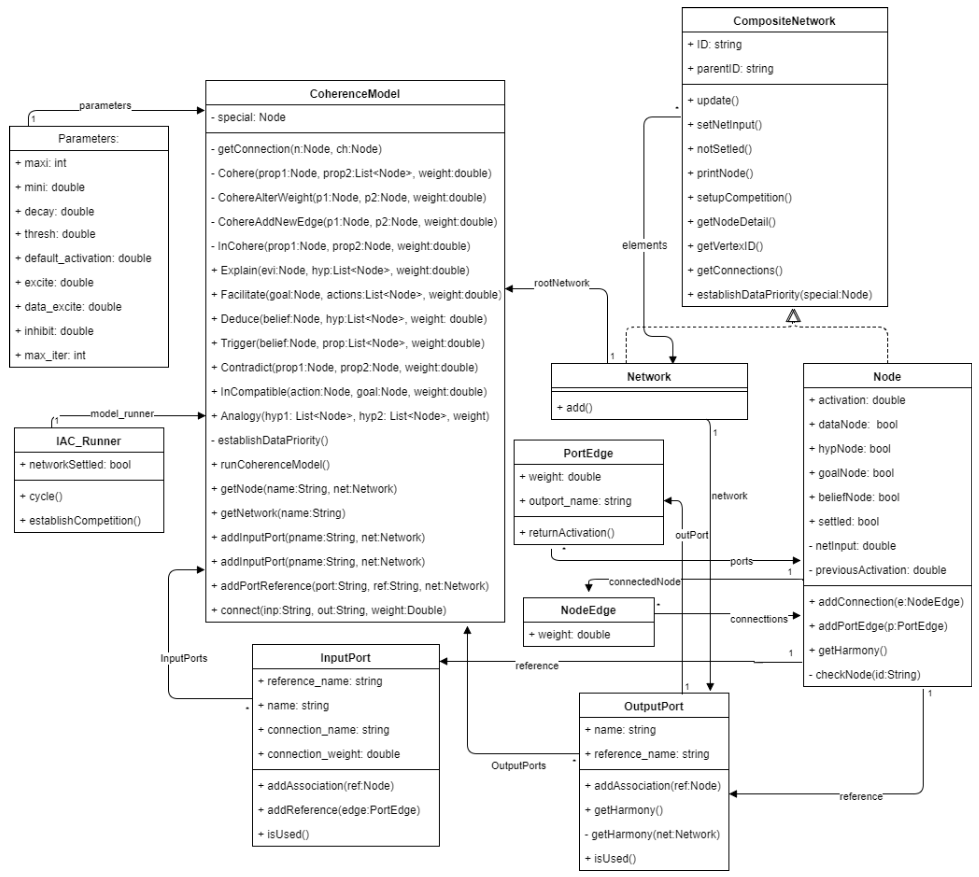

CompositeNetwork
+ ID: string
+ parentID: string
+ update()
+ setNetInput()
+ notSetled()
+ printNode()
+ setupCompetition()
+ getNodeDetail()
+ getVertexID()
+ getConnections()
+ establishDataPriority(special:Node)
CoherenceModel
- special: Node
- getConnection(n:Node, ch:Node)
- Cohere(prop1:Node, prop2:List<Node>, weight:double)
- CohereAlterWeight(p1:Node, p2:Node, weight:double)
- CohereAddNewEdge(p1:Node, p2:Node, weight:double)
- InCohere(prop1:Node, prop2:Node, weight:double)
+ Explain(evi:Node, hyp:List<Node>, weight:double)
+ Facilitate(goal:Node, actions:List<Node>, weight:double)
+ Deduce(belief:Node, hyp:List<Node>, weight: double)
+ Trigger(belief:Node, prop:List<Node>, weight:double)
+ Contradict(prop1:Node, prop2:Node, weight:double)
+ InCompatible(action:Node, goal:Node, weight:double)
+ Analogy(hyp1: List<Node>, hyp2: List<Node>, weight)
- establishDataPriority()
+ runCoherenceModel()
+ getNode(name:String, net:Network)
+ getNetwork(name:String)
+ addInputPort(pname:String, net:Network)
+ addInputPort(pname:String, net:Network)
+ addPortReference(port:String, ref:String, net:Network)
+ connect(inp:String, out:String, weight:Double)
parameters
1
Parameters:
+ maxi: int
+ mini: double
+ decay: double
+ thresh: double
+ default_activation: double
+ excite: double
+ data_excite: double
+ inhibit: double
+ max_iter: int
model_runner
1
IAC_Runner
+ networkSettled: bool
+ cycle()
+ establishCompetition()
elements
*
rootNetwork
1
Network
+ add()
1
network
Node
+ activation: double
+ dataNode: bool
+ hypNode: bool
+ goalNode: bool
+ beliefNode: bool
+ settled: bool
- netInput: double
- previousActivation: double
+ addConnection(e:NodeEdge)
+ addPortEdge(p:PortEdge)
+ getHarmony()
- checkNode(id:String)
PortEdge
+ weight: double
+ outport_name: string
+ returnActivation()
outPort
ports
1
connectedNode
NodeEdge
+ weight: double
connecttions
reference
1
InputPorts
InputPort
+ reference_name: string
+ name: string
+ connection_name: string
+ connection_weight: double
+ addAssociation(ref:Node)
+ addReference(edge:PortEdge)
+ isUsed()
OutputPort
+ name: string
+ reference_name: string
+ addAssociation(ref:Node)
+ getHarmony()
- getHarmony(net:Network)
+ isUsed()
OutputPorts
reference
1

Figure 4.3: Reference Implementation

- CompositeNetwork: As Cogent aims to represent a hierarchical network structure, an effective way to design it would be the Composite Design Pattern [46], which allows a recursive composition of objects. *CompositeNetwork* plays the part of Component class or the abstract class in this design approach. This is an abstract class, which is implemented by the Node and Network class. The Node class acts as a leaf class, and the Network class acts a Composite class.

- Node: This class represents a node in the Coherence network. The various types of nodes in the coherence network, such as a hypothesis or an evidence, are defined using this class. This class has information on the nodes and ports connected to it. This class extends all of the functions defined in the abstract class CompositeNetwork. This class also implements methods such as the *update* method, which is the update operation of an IAC Network, discussed in Section 3.1.5. It also provides methods to add new connections and ports.

- NodeEdge: This class acts as an edge connecting the Node objects. The *weight* attribute represents the weight of an edge connecting two nodes.

- InputPort: This class implements the input port functionality of Cogent. It has a reference to a single node in the network.

- OutputPort: This class implements the output port functionality of Cogent. It has a reference to a node and its network. The OutputPort sends the *harmony* of the network associated with it to the InputPort.

- PortEdge: This class acts as an edge connecting the InputPort and OutputPort. The *weight* attribute represents the weight of an edge connecting the InputPort and OutputPort.

- Network: This class represents the Coherence network. It can contain nodes and other Network objects, thus supporting a hierarchical network structure. Other than implementing the functions from CompositeNetwork, this class also has a function to add new Node or Network objects.

- CoherenceModel: This class provides the functions necessary to establish the Coherence Network, along with the different coherence mechanisms involved in it. *Explanatory Coherence* can be established by using the *Explain* and *Contradict* methods. *Deductive Coherence* can be established by using the *Deduce* and *Contradict* methods. *Deliberative Coherence* can be established using the *Facilitate* and *InCompatible* methods. *Analogical Coherence* can be established using the *Analogy* method. These methods are implemented based on the principles of Coherence Theory discussed in 3.1.

- Parameters: This object provides all the parameters necessary for running the Coherence Network. The Cogent platform allows some of these parameters to be replaced.

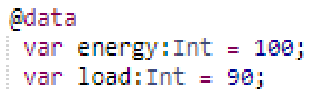
- IACRunner: This class provides the functions necessary to run the Coherence Network. The *cycle* method runs by invoking the *update* method for all the *Node* objects until they settle for a certain threshold.

4.2.3 Domain-Specific Language Syntax

We will discuss the syntax of each part of the DSL further in this section. Although Cogent's DSL is intended to hide the programming language syntax, it also allows users to use Scala programming syntax for some of the sections. We will discuss the structure of each component of the DSL in the following sections.

Data Section

The data section allows the resources of the agent can be defined. The resources, in this case, are Scala variables. These resources can be shared and modified by the behaviors of the agent, as needed. The variables can be compared to common resources needed for the behaviors of the agent, and are defined under the *@data* keyword. An illustration of this section can be seen in Figure 4.4.

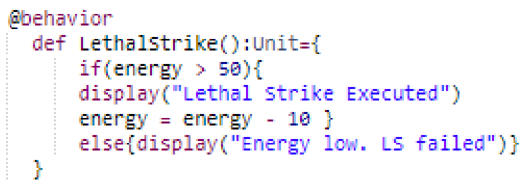


```
@data
  var energy:Int = 100;
  var load:Int = 90;
```

Figure 4.4: Data Section

Behavior Section

The behavior section allows users to define the agent's behaviors. The behaviors are imperative functions that manipulate the agent's resources defined in the data section. Multiple behaviors can be defined for a single agent, and the behaviors are linked to an action node in the cognitive coherence network. All the functions are defined under the *@behavior* keyword. An illustration of this section can be seen in Figure 4.5.



```
@behavior
  def LethalStrike():Unit={
      if(energy > 50){
      display("Lethal Strike Executed")
      energy = energy - 10 }
      else{display("Energy low. LS failed")}
  }
```

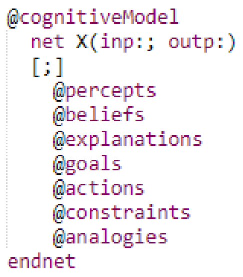
Figure 4.5: Behavior Section

Cognitive Model

A cogent's deliberation and decision-making making model are defined in this section. As discussed before, the decision making in the Cogent language is based on the coherence theory. The network structure, along with various other nodes such as action, explanation, and belief are defined in this section. This section also allows users to add the relationship between various nodes like explanation, contradiction, deduction or triggers. This section is defined under the *@cognitiveModel* keyword. The structure of this section having a single network X can be seen in Figure 4.6.

Network

This section is a part of the Cognitive Model Section discussed before. A network object is defined in this section. There can be multiple networks enclosed inside a single network. The networks contain information of all the nodes and the relationships among them. The network section also takes input and output port information from the user. The syntax allows the input and output ports to be mapped to the nodes in a network. These ports can be used to connect to other networks in the cognitive model. This will be discussed in the Port Mapping section later. An illustration of the structure of a network can be seen in Figure 4.6.

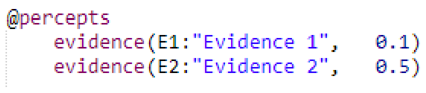


```
@cognitiveModel
  net X(inp:; outp:)
  [;]
    @percepts
    @beliefs
    @explanations
    @goals
    @actions
    @constraints
    @analogies
endnet
```

Figure 4.6: Cognitive Model and Network Sections

Evidence Nodes

The evidence nodes represent data and are associated with activation values, indicating the degree of confidence in respective data sources. The initial activation can also be considered as a form of preference among nodes. That is, if a node has been set with a high initial activation, it would have an advantage over others. A node can still get rejected after the network has settled, as the nodes final activation depends on how it coheres with other nodes. The evidence nodes are defined under the *@precepts* keyword. An illustration of this section can be seen in Figure 4.7.

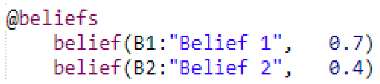


```
@percepts
    evidence(E1:"Evidence 1",   0.1)
    evidence(E2:"Evidence 2",   0.5)
```

Figure 4.7: Precepts Section

Belief Nodes

The belief nodes represent the set of propositions that the cogent has to consider during the deliberation process. The belief nodes are added with the belief name and an initial activation level, which serves the same purpose defined in the previous section. The belief nodes are defined under the *@beliefs* keyword. An illustration of this section can be seen in Figure 4.8.

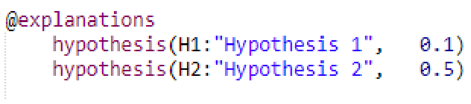


```
@beliefs
    belief(B1:"Belief 1",   0.7)
    belief(B2:"Belief 2",   0.4)
```

Figure 4.8: Beliefs Section

Hypothesis Nodes

Explanatory coherence involves the relation between evidences and hypotheses. Hypotheses are statements that provide a causal explanation for the evidence nodes. Hypotheses are added with the hypothesis name along with an initial activation. The hypothesis nodes are defined under the *@explanations* keyword. Each individual hypothesis is defined using the *hypothesis* keyword. An illustration of this section can be seen in Figure 4.9.

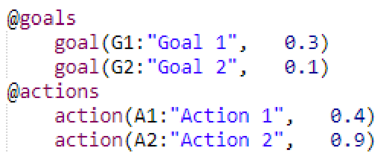


```
@explanations
    hypothesis(H1:"Hypothesis 1",   0.1)
    hypothesis(H2:"Hypothesis 2",   0.5)
```

Figure 4.9: Explanations Section

Goal and Action Nodes

The goal nodes express the different goals that the agent can consider during the means-ends reasoning process. The goal nodes are added similar to previous nodes. They are listed under the *@goals* keyword. The action nodes represent the activities that a cogent can perform to affect the environment. They are mapped to behaviors if needed, and are defined under the @actions keyword. An illustration of these sections can be seen in Figure 4.10.

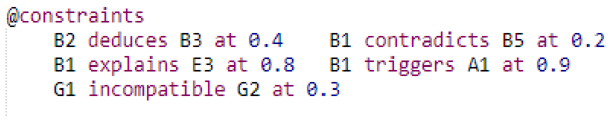


```
@goals
    goal(G1:"Goal 1",   0.3)
    goal(G2:"Goal 2",   0.1)
@actions
    action(A1:"Action 1",   0.4)
    action(A2:"Action 2",   0.9)
```

Figure 4.10: Goals and Actions Sections

Constraints

The constraints among different types of nodes are specified in this section. The constraints include positive ones such as the *explains* relation that connects an evidence and hypothesis, the *deduce* constraint for associating beliefs, the *facilitate* constraint for associating goal nodes, and the *trigger* constraint for associating the goal nodes with the action nodes. The Cogent language also allows negative constraints such as the *incompatible* and the *contradicts* constraints. A specific weight is also declared when setting up constraints to indicate the strength of the relation. An illustration of these sections can be seen in Figure 4.11.

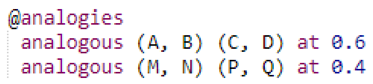


```
@constraints
    B2 deduces B3 at 0.4     B1 contradicts B5 at 0.2
    B1 explains E3 at 0.8    B1 triggers A1 at 0.9
    G1 incompatible G2 at 0.3
```

Figure 4.11: Constraints Section

Analogies

The analogical constraints between nodes can be established in this section. The syntax for adding an analogy constraint is as follows, Analogy $(A, B)(C, D)$ at 0.4, where A, B, C, D can be any of the nodes described before. The analogical expression means that the association or connection between A and B is similar to to the connection between C and D. This section is defined under the @analogies keyword. An illustration of this section can be seen in Figure 4.12.



```
@analogies
 analogous (A, B) (C, D) at 0.6
 analogous (M, N) (P, Q) at 0.4
```

Figure 4.12: Analogies Section

Port Mapping

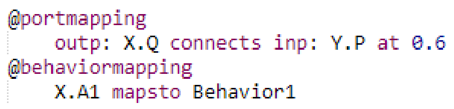
The input and output ports of different networks are mapped in this section. The *connects* statement is used to establish the connection between the ports. A weight must also be provided such that the input and output port is connected through an edge of the weight. When the coherence network is evaluated, the nodes connected to the input port receive an additional information other than the net-input discussed before. The input port receives the harmony of the network associated with the output port through the output port. The harmony of the network can be represented as follows [11]:

$$harmony = \sum_{i=1}^{n} \sum_{j=i+1}^{n} w(i,j) * a(i) * a(j)$$

This harmony is multiplied with a port weight is added to the calculation of the net-input value of the node associated with the input port. This section is declared under the *@portmapping* keyword. An illustration of this section can be seen in Figure 4.13.

Behavior Mapping

In this section, the behaviors are associated with the action nodes. This ensures that, when an action node is activated, the associated behavior is executed. The "mapsto" keyword is used to establish a connection between action node and a behavior name or the function name. This section is declared under the *@behaviormapping* section. An illustration of this section can be seen in Figure 4.13.



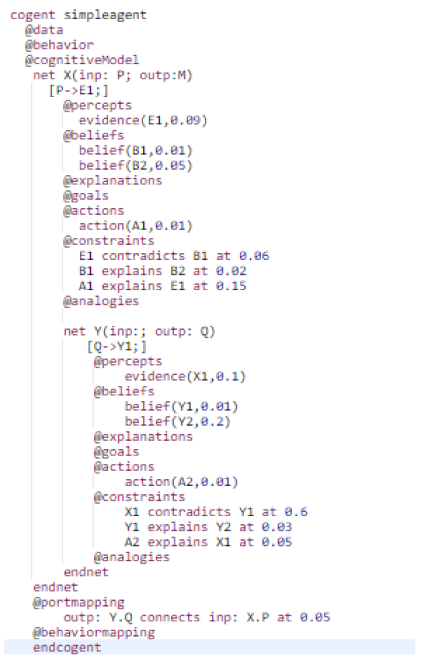
```
@portmapping
    outp: X.Q connects inp: Y.P at 0.6
@behaviormapping
    X.A1 mapsto Behavior1
```

Figure 4.13: Port and Behavior mapping Sections

4.2.4 Hierarchical Network Structure

Cogent's DSL is equipped to handle complex network structures like a hierarchical network, which is a composite model that represents part-whole relation through a modular decomposition of networks. The hierarchical networks are an optional feature of the DSL and need not be applicable for all cases.

In the example shown in Figure 4.14 we present two networks X and Y defined by the DSL. The network Y is encapsulated within the network X. The network X and Y have nodes and constraints of their own; the nodes are associated between networks through ports. When an input port is connected to an output port, the harmony of the network associated with the output port is obtained by the input port and transferred to the node associated with the input port. The hierarchical structure will help users model or experiment complex network configurations.



```
cogent simpleagent
  @data
  @behavior
  @cognitiveModel
   net X(inp: P; outp:M)
     [P->E1;]
       @percepts
         evidence(E1,0.09)
       @beliefs
         belief(B1,0.01)
         belief(B2,0.05)
       @explanations
       @goals
       @actions
         action(A1,0.01)
       @constraints
         E1 contradicts B1 at 0.06
         B1 explains B2 at 0.02
         A1 explains E1 at 0.15
       @analogies

       net Y(inp:; outp: Q)
          [Q->Y1;]
           @percepts
               evidence(X1,0.1)
           @beliefs
               belief(Y1,0.01)
               belief(Y2,0.2)
           @explanations
           @goals
           @actions
               action(A2,0.01)
           @constraints
               X1 contradicts Y1 at 0.6
               Y1 explains Y2 at 0.03
               A2 explains X1 at 0.05
           @analogies
       endnet
   endnet
   @portmapping
       outp: Y.Q connects inp: X.P at 0.05
   @behaviormapping
   endcogent
```

Figure 4.14: Hierarchical Network DSL

4.3 Cogent Platform

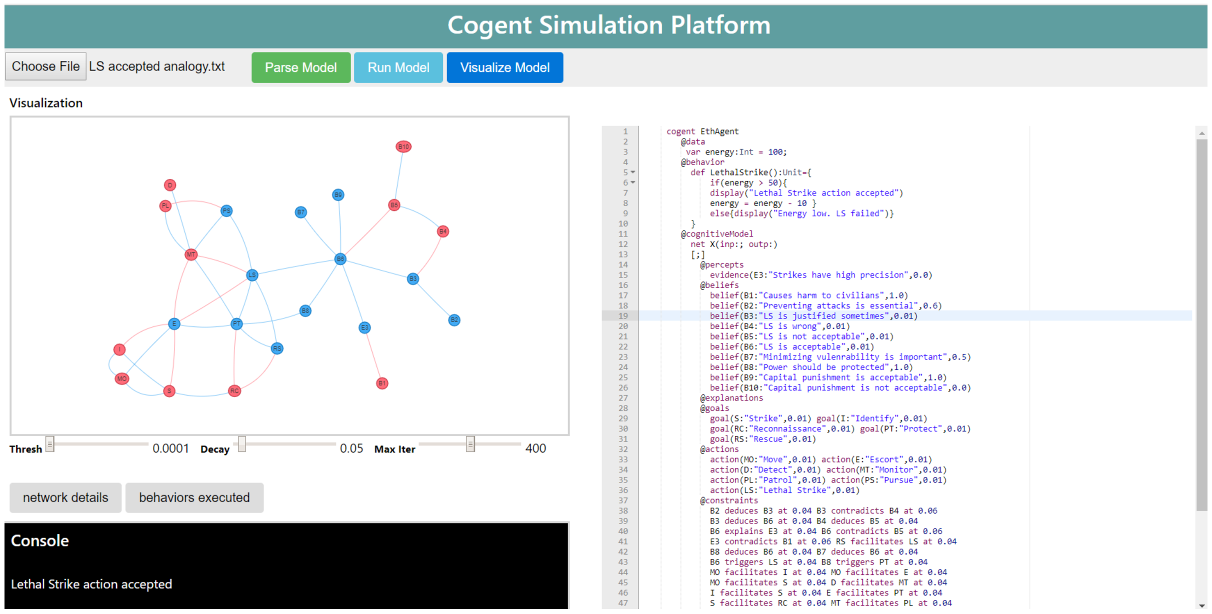
In this section, we will introduce the Cogent Platform and its components. The Cogent simulation and experimentation platform allows users to define cogents using a higher-level language, experiment, and observe them. Figure 4.15 shows the platform and its components.

4.3.1 Editor Section

Cogent includes an editor section similar to a programming language editor. The editor offers features such as syntax highlighting of the keywords discussed in the DSL section. It also highlights imperative programming language keywords used as part of the behavior and data sections of the DSL. The editor in Cogent is a modified version of ACE editor [47].

4.3.2 Console Section

The console section in Cogent comes with two options which can be configured by the user. One option allows viewing node information such as activation level and connections



```
Lethal Strike action accepted
```

```
cogent EthAgent
   @data
    var energy:Int = 100;
   @behavior
     def LethalStrike():Unit={
         if(energy > 50){
         display("Lethal Strike action accepted")
         energy = energy - 10 }
         else{display("Energy low. LS failed")}
     }
   @cognitiveModel
     net X(inp:; outp:)
     [;]
       @percepts
         evidence(E3:"Strikes have high precision",0.0)
       @beliefs
         belief(B1:"Causes harm to civilians",1.0)
         belief(B2:"Preventing attacks is essential",0.6)
         belief(B3:"LS is justified sometimes",0.01)
         belief(B4:"LS is wrong",0.01)
         belief(B5:"LS is not acceptable",0.01)
         belief(B6:"LS is acceptable",0.01)
         belief(B7:"Minimizing vulenrability is important",0.5)
         belief(B8:"Power should be protected",1.0)
         belief(B9:"Capital punishment is acceptable",1.0)
         belief(B10:"Capital punishment is not acceptable",0.0)
       @explanations
       @goals
         goal(S:"Strike",0.01) goal(I:"Identify",0.01)
         goal(RC:"Reconnaissance",0.01) goal(PT:"Protect",0.01)
         goal(RS:"Rescue",0.01)
       @actions
         action(MO:"Move",0.01) action(E:"Escort",0.01)
         action(D:"Detect",0.01) action(MT:"Monitor",0.01)
         action(PL:"Patrol",0.01) action(PS:"Pursue",0.01)
         action(LS:"Lethal Strike",0.01)
       @constraints
         B2 deduces B3 at 0.04 B3 contradicts B4 at 0.06
         B3 deduces B6 at 0.04 B4 deduces B5 at 0.04
         B6 explains E3 at 0.04 B6 contradicts B5 at 0.06
         E3 contradicts B1 at 0.06 RS facilitates LS at 0.04
         B8 deduces B6 at 0.04 B7 deduces B6 at 0.04
         B6 triggers LS at 0.04 B8 triggers PT at 0.04
         MO facilitates I at 0.04 MO facilitates E at 0.04
         MO facilitates S at 0.04 D facilitates MT at 0.04
         I facilitates S at 0.04 E facilitates PT at 0.04
         S facilitates RC at 0.04 MT facilitates PL at 0.04
```

Figure 4.15: Cogent Simulation Platform

associated each node of the coherence network of the cogent. The other option lets the user view the output of the display functions activated by the behaviors executed. To print a value in the console section, the user can call the “display” function as a part of a behavior in the DSL. The section can be used to track the data objects and behaviors and thus help in better understanding the agent’s behavior.

4.3.3 Visualization Section

The Cogent environment also has provision for visualization of the coherence network of individual cogents. The visualization helps the user to verify that the cogent behaves as expected. This would help make sufficient modification to the specification and test the model again until desired behavioral regularities are obtained. Thus the visualization plays a major role in the iterative and incremental building of the model.

In Figure 4.16, we can see the visualization of a hierarchical network with two networks connected through ports. The nodes in the network are represented either by blue nodes indicating that they are accepted and red color nodes indicating the ones rejected. The port

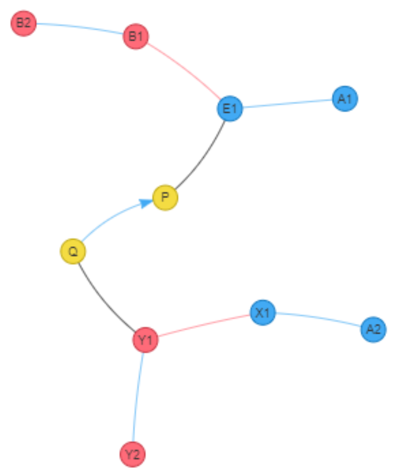


Figure 4.16: Hierarchical Network

nodes are represented by a yellow color node. The links between nodes are also colored based on the relation between the nodes. The nodes with a positive connection, such as *explanation*, would have a blue connection. Nodes having a negative connection such as *contradiction* would have a red connection. While all nodes are connected by an undirected edge, the port nodes are connected by a directional edge where the edge directs from an output port to an input port, representing the flow of harmony from the output port's network to input port's network.

The visualization for a flat network in Cogent has identical properties except that network will not have ports, rather just nodes and connections. An example of this visualization can be seen in Figure 4.17.

4.3.4 Simulation and Experimentation Processes

The simulation of a cogent in the Cogent language begins by the user editing the program in terms of the DSL. The user then parses the DSL to verify the syntax using the "Parse Model" button in the simulation platform. The interface indicates if the syntax of the program is well-formed and correct with respect to the grammar. The user then runs the model by clicking the "Run Model" button which initializes the coherence network and

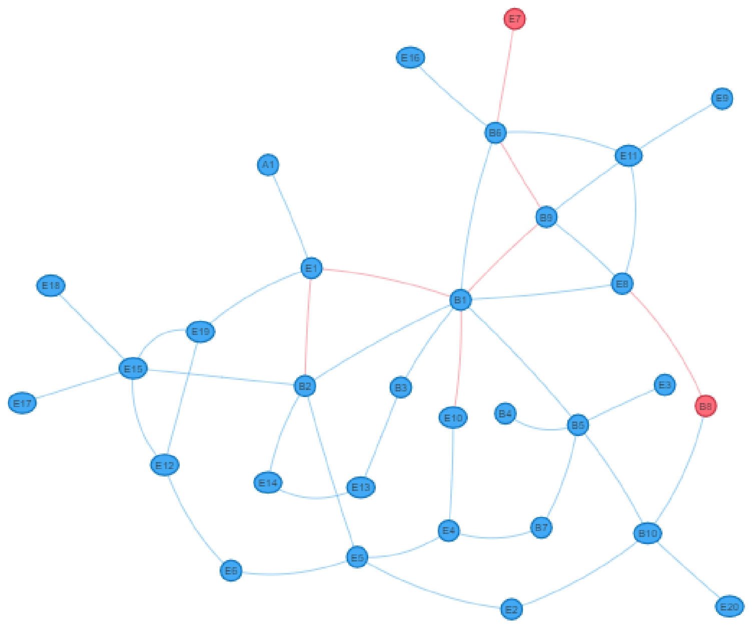


Figure 4.17: Flat Network

simulates the behaviors of the cogent. After a run is completed, the user can view the console section for any messages printed by the behavior. We can also shift the console section to view node details. Finally, the "Visualize Model" button can be clicked to display the visualization of the created agent. The user, repeats the procedure until a satisfied result is obtained. The activity diagram for the simulation and experimentation process can be seen in Figure 4.18

4.4 Contrastive Hebbian Learning Implementation

The Contrastive Hebbian Algorithm is provided as a Scala package. The package is provided so that, users can set up the initial weights for the coherence network connections. These initial weights can be used to specify a model using the DSL. Applying the *Contrastive Hebbian Learning* algorithm to a coherence network can be quite tricky, as a coherence network does not differentiate input and output nodes. We propose to handle this problem by allowing users to experiment with a various combination of inputs and outputs. The

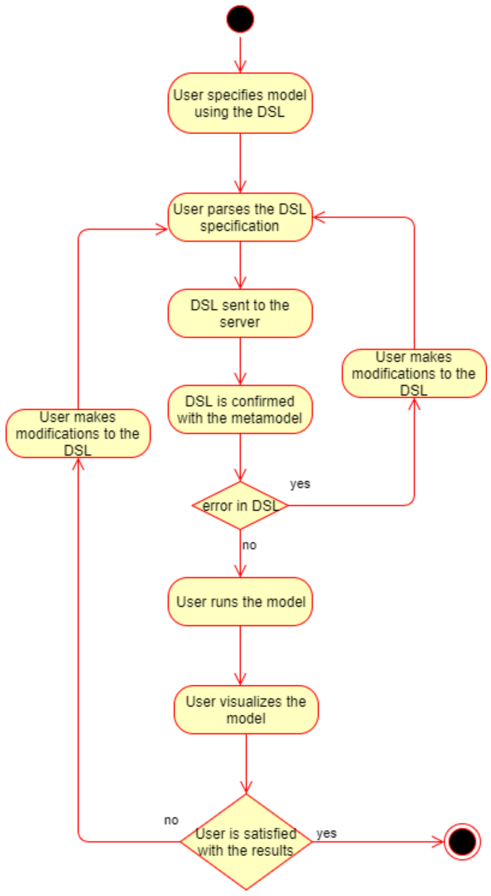

User specifies model using the DSL
User parses the DSL specification
DSL sent to the server
DSL is confirmed with the metamodel
User makes modifications to the DSL
User makes modifications to the DSL
error in DSL
yes
no
User runs the model
User visualizes the model
User is satisfied with the results
no
yes

Figure 4.18: Simulation and Experimentation Process

desired coherence network configuration, or the set of accepted and rejected nodes, can be divided into various combinations of input and output sets to experiment with the learning algorithm. The activity diagram demonstrating the process of applying this algorithm can be seen in Figure 4.19.

It is important to note that *cogents* are still iteratively built using the Cogent platform. The algorithm can only be used to make a guess on the initial Coherence Network design, thus speeding up the development process.

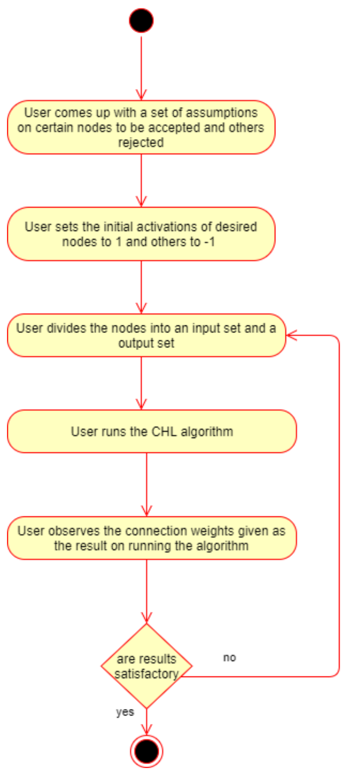

User comes up with a set of assumptions on certain nodes to be accepted and others rejected
User sets the initial activations of desired nodes to 1 and others to -1
User divides the nodes into an input set and a output set
User runs the CHL algorithm
User observes the connection weights given as the result on running the algorithm
are results satisfactory
no
yes

Figure 4.19: Contrastive Hebbian Learning application Process

Chapter 5

Case Study

5.1 Machine Ethics

Machine ethics is the field that concerns the moral behaviors and actions of intelligent computing systems. Although they may sound similar, the field of machine ethics is different from the field of computer ethics. Computer ethics deals with ethical use of computing systems, while machine ethics deals with the moral behaviors of computing systems. Autonomous systems deployed in the real world can face challenges that can result in serious consequences. For instance, driverless systems put machines in the position of making split-second decisions that could have life or death implications [48].

Ethical decision making can be broadly classified into two categories, the *deontological* approach [7] and the *utilitarian* approach to ethics [8]. By deontological approach to ethics, an agent must always abide by the rules set to it. If an agent obeys the deontological approach, then irrespective of the situation, the agent would behave by the rules set to it. The utilitarian approach is contrary to the deontological approach. By the utilitarian approach to ethics, an agent's decision should maximize a utility function. The utility function is typically happiness or welfare of the people that the agent serves or interacts with. Let us illustrate the two ethical decision making approaches by discussing the trolley problem [49]. The problem states the following scenario, a trolley is traveling down a railway track in a high speed, while there are a group of 5 people standing in the current track of the trolley, and one person standing in a side track. Assuming the trolley is operating on schedule, by the deontological approach, the trolley would travel in its current course and harm the five people in its path. On the another hand, if the trolley is designed to follow the utilitarian approach, then it would change its track and hit the single person on the side track.

According to Moor [50], machine ethics can be classified into four categories, which are ethical impact agents, implicit ethical agents, explicit ethical agents, and fully ethical agents.

5.1.1 Ethical Impact Agents

Ethical impact agents are systems that lead to an ethical behavior as a result to a problem it solves. These agents do not exhibit any ethical behavior of their own, but the application and use of these agents lead to an ethical choice for a given problem.

5.1.2 Implicit Ethical Agents

Implicit ethical agents are systems that exhibit ethical behaviors implicitly, as they would be carefully designed to be ethical. These agents will have no knowledge of any ethical constraints or decision making. An example would be an Automatic Teller Machine (ATM) updating the correct balance after a transaction, such that the bank acts fairly towards the customer withdrawing money from the ATM.

5.1.3 Explicit Ethical Agents

In this case, agents are aware of ethical principles and act by it. Ethical decision making mechanisms like deontology and utilitarianism are explicitly programmed into these agents. Machines that are explicit ethical agents might be the best ethical agents to have in situations such as disaster relief [50].

5.1.4 Fully Ethical Agents

A fully ethical agent not only has explicit knowledge of ethical decision making, but also has a level of consciousness to justify their decision. There is a popular argument against these agents, that a machine can never attain full conscience or awareness.

5.2 The Lethal Strike Problem

In this section, we will introduce a problem in the domain of machine ethics, and validate the Cogent platform's capability by developing the problem using it. The Lethal Strike problem [51], is a hypothetical scenario faced by an autonomous Unmanned Ariel Vehicle (UAV), during a search and rescue mission. In the Lethal Strike problem, the UAV may have multiple goals to satisfy, and the UAV can perform a set of actions to satisfy these goals. The UAV has to make decisions between various goals and actions to pursue, as some of the goals and actions may contradict with each other. The UAV has to make decisions based on a set of evidences and beliefs. Its decisions can vary greatly based on changes in these evidences and beliefs. Thus, the UAV is expected to analyze various combinations of goals, actions, beliefs, and evidences during the decision making process.

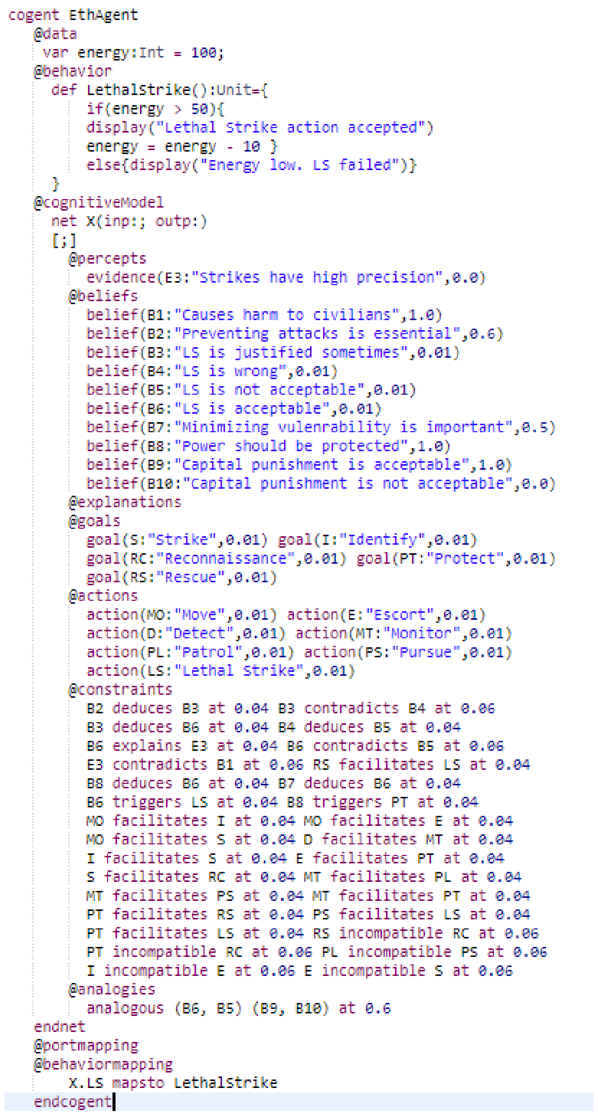
In this case study, we are particularly interested in the ethical decisions taken by the UAV. The UAV can perform many actions, like monitor, patrol, but our major focus would be on the Lethal Strike action. The Lethal Strike action is a scenario where the acts of the UAV can have adverse effect on civilians. Thus, the UAV has to make a decision between performing a Lethal Strike action and maximize the utility, or follow the rules that the UAV's action must not hurt civilians. The first scenario is a consequentialist approach [8] and the latter is a Deontological approach [7] of ethical decision making. From the above description we can see that we need to simulate two UAVs taking two different type of decisions for a single problem.

The Lethal Strike problem is chosen to demonstrate the Cogent platform, as the problem involves the consideration of complex deliberation and reasoning mechanisms between various beliefs, evidences, actions, and goals. We demonstrate the Cogent platform by simulating a scenario where the cogent takes a deontological decision, and another scenario where it takes a utilitarian decision. The cogent in this case acts as an explicit ethical agent. We will demonstrate, how Cogent's DSL allows the problem to be expressed explicitly in terms of deliberation mechanisms. We will also show, how minor changes in some beliefs in our simulation can can lead to different actions to be executed, and various other actions being rejected.

5.2.1 Simulating the Utilitarian Scenario

The DSL program used to simulate the utilitarian scenario for the Lethal Strike problem is given below in Figure 5.1. The DSL is programmed based on the syntax discussed in Section 4.2.3. The model specifies different evidences, beliefs, actions, and goals in their respective sections. In the first scenario, the agent is initialized with high activation on node $B1$, meaning the actions cause harm to civilians. We also have higher activation on nodes $B2$, $B7$, and $B8$. Where, the node $B2$ represents the need to prevent attacks, the node $B7$ represents the need to minimize vulnerability, and the node $B8$ represents the need to protect power of the UAV. We can see the nodes $B2$, $B7$, and $B8$ aim to maximize utility, hence maximizing the UAV's mission of search and rescue. As this configuration supports the utilitarian approach, the belief $B9$ has a higher initial activation, supporting the decision that the UAV can harm civilians to maximize utility. This configuration while activates the Lethal Strike action, it might cause causalities.

The network visualization for this configuration is given below in Figure 5.2, we can see that the Lethal Strike action is activated. From the visualization, it can be observed that, the Coherence Network is established based on the principles of Coherence Theory, discussed in Section 3.1. Nodes that are specified to cohere by using keywords such as "explains", "deduces", and "facilitates", can be seen connected in the visualization with blue coloured edges, indicating they have a positive association. On the contrary, nodes specified to incohere by using keywords such as "contradicts" and "incompatible", can be seen connected with a red coloured edge, indicating they have a negative association. We can also observe competition has been established between various beliefs and action nodes. For instance, the Lethal Strike action competes with the Monitor action as they both facilitate the Pursue goal, but do not facilitate each other. We can also observe the analogical association



```
cogent EthAgent
   @data
    var energy:Int = 100;
   @behavior
     def LethalStrike():Unit={
         if(energy > 50){
         display("Lethal Strike action accepted")
         energy = energy - 10 }
         else{display("Energy low. LS failed")}
     }
   @cognitiveModel
     net X(inp:; outp:)
     [;]
       @percepts
         evidence(E3:"Strikes have high precision",0.0)
       @beliefs
         belief(B1:"Causes harm to civilians",1.0)
         belief(B2:"Preventing attacks is essential",0.6)
         belief(B3:"LS is justified sometimes",0.01)
         belief(B4:"LS is wrong",0.01)
         belief(B5:"LS is not acceptable",0.01)
         belief(B6:"LS is acceptable",0.01)
         belief(B7:"Minimizing vulenrability is important",0.5)
         belief(B8:"Power should be protected",1.0)
         belief(B9:"Capital punishment is acceptable",1.0)
         belief(B10:"Capital punishment is not acceptable",0.0)
       @explanations
       @goals
         goal(S:"Strike",0.01) goal(I:"Identify",0.01)
         goal(RC:"Reconnaissance",0.01) goal(PT:"Protect",0.01)
         goal(RS:"Rescue",0.01)
       @actions
         action(MO:"Move",0.01) action(E:"Escort",0.01)
         action(D:"Detect",0.01) action(MT:"Monitor",0.01)
         action(PL:"Patrol",0.01) action(PS:"Pursue",0.01)
         action(LS:"Lethal Strike",0.01)
       @constraints
         B2 deduces B3 at 0.04 B3 contradicts B4 at 0.06
         B3 deduces B6 at 0.04 B4 deduces B5 at 0.04
         B6 explains E3 at 0.04 B6 contradicts B5 at 0.06
         E3 contradicts B1 at 0.06 RS facilitates LS at 0.04
         B8 deduces B6 at 0.04 B7 deduces B6 at 0.04
         B6 triggers LS at 0.04 B8 triggers PT at 0.04
         MO facilitates I at 0.04 MO facilitates E at 0.04
         MO facilitates S at 0.04 D facilitates MT at 0.04
         I facilitates S at 0.04 E facilitates PT at 0.04
         S facilitates RC at 0.04 MT facilitates PL at 0.04
         MT facilitates PS at 0.04 MT facilitates PT at 0.04
         PT facilitates RS at 0.04 PS facilitates LS at 0.04
         PT facilitates LS at 0.04 RS incompatible RC at 0.06
         PT incompatible RC at 0.06 PL incompatible PS at 0.06
         I incompatible E at 0.06 E incompatible S at 0.06
       @analogies
         analogous (B6, B5) (B9, B10) at 0.6
   endnet
   @portmapping
   @behaviormapping
       X.LS mapsto LethalStrike
   endcogent
```

Figure 5.1: Lethal Strike Problem Specification

set up between nodes $B6$, $B9$, $B5$, and $B10$. The default parameters were used for the parameters *threshold*, *max-iter*, and *decay* for obtaining the discussed result.

5.2.2 Simulating the Deontological Scenario

The deontological scenario is a slight modification to the network configuration discussed in the previous section. The DSL used to simulate this scenario is given below in Figure 5.3. The DSL is programmed based on the syntax discussed in Section 4.2.3. In this case, we

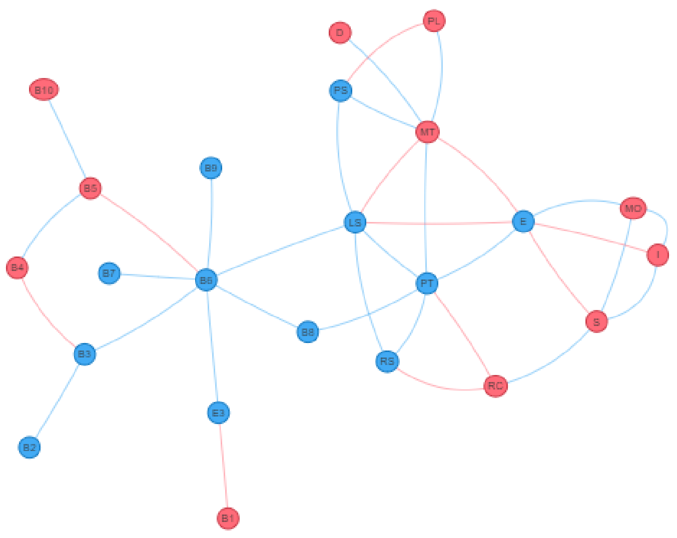
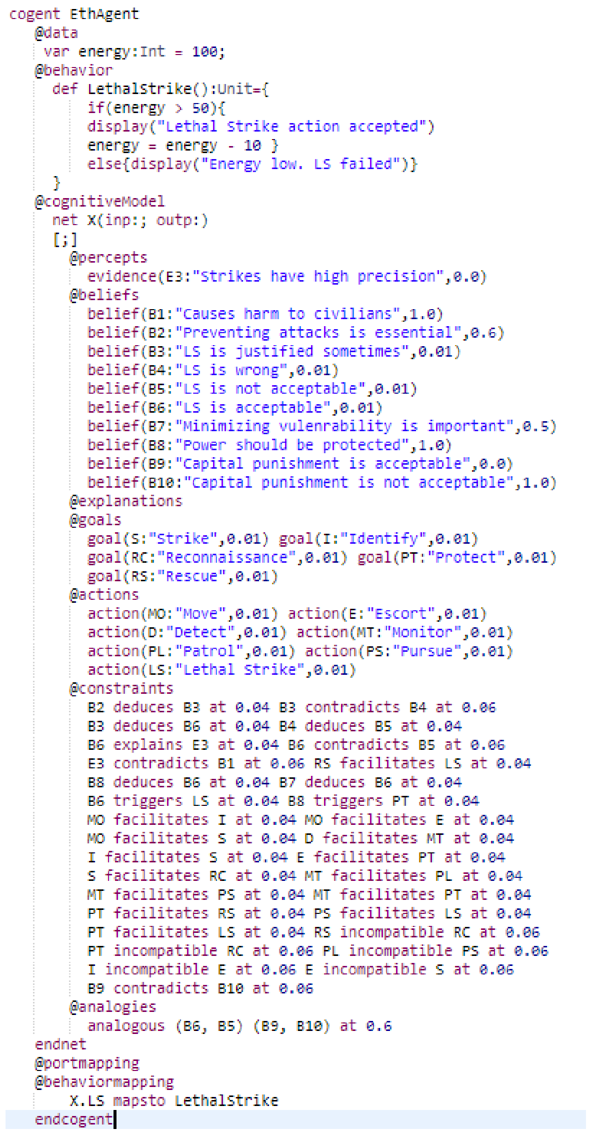


Figure 5.2: Lethal Strike action accepted

reduce the activation of nodes $B2$, $B7$ and $B8$, thus we put utility under lesser importance and show more emphasis on the deontological rule of not hurting causalities. We also update the beliefs $B9$ and $B10$, such that, the UAV values civilian's welfare more than utility.

The visualization of this scenario is given below in Figure 5.4. We can observe that the connections remain the same as before, as the rules were unchanged. As some of the beliefs were changed to reflect a new decision, we obtain a different set of nodes being accepted. We can see that the Lethal Strike node is rejected and other nodes related to maximizing utility are also rejected, thus supporting the deontological idea of protecting civilians. Comparing the two visualizations we observe a set of nodes get accepted or rejected together. The nodes accepted in the First scenario together support an ethical decision and nodes together in the second scenario support another ethical decision. We were able to switch between the two world-views in a transparent manner. As such, our approach of simulation was expressive enough to denote the principles of decision making intuitively. The default parameters were used for the parameters *threshold*, *max-iter*, and *decay* for obtaining the discussed result.



```
cogent EthAgent
   @data
    var energy:Int = 100;
   @behavior
     def LethalStrike():Unit={
         if(energy > 50){
         display("Lethal Strike action accepted")
         energy = energy - 10 }
         else{display("Energy low. LS failed")}
     }
   @cognitiveModel
     net X(inp:; outp:)
     [;]
       @percepts
         evidence(E3:"Strikes have high precision",0.0)
       @beliefs
         belief(B1:"Causes harm to civilians",1.0)
         belief(B2:"Preventing attacks is essential",0.6)
         belief(B3:"LS is justified sometimes",0.01)
         belief(B4:"LS is wrong",0.01)
         belief(B5:"LS is not acceptable",0.01)
         belief(B6:"LS is acceptable",0.01)
         belief(B7:"Minimizing vulenrability is important",0.5)
         belief(B8:"Power should be protected",1.0)
         belief(B9:"Capital punishment is acceptable",0.0)
         belief(B10:"Capital punishment is not acceptable",1.0)
       @explanations
       @goals
         goal(S:"Strike",0.01) goal(I:"Identify",0.01)
         goal(RC:"Reconnaissance",0.01) goal(PT:"Protect",0.01)
         goal(RS:"Rescue",0.01)
       @actions
         action(MO:"Move",0.01) action(E:"Escort",0.01)
         action(D:"Detect",0.01) action(MT:"Monitor",0.01)
         action(PL:"Patrol",0.01) action(PS:"Pursue",0.01)
         action(LS:"Lethal Strike",0.01)
       @constraints
         B2 deduces B3 at 0.04 B3 contradicts B4 at 0.06
         B3 deduces B6 at 0.04 B4 deduces B5 at 0.04
         B6 explains E3 at 0.04 B6 contradicts B5 at 0.06
         E3 contradicts B1 at 0.06 RS facilitates LS at 0.04
         B8 deduces B6 at 0.04 B7 deduces B6 at 0.04
         B6 triggers LS at 0.04 B8 triggers PT at 0.04
         MO facilitates I at 0.04 MO facilitates E at 0.04
         MO facilitates S at 0.04 D facilitates MT at 0.04
         I facilitates S at 0.04 E facilitates PT at 0.04
         S facilitates RC at 0.04 MT facilitates PL at 0.04
         MT facilitates PS at 0.04 MT facilitates PT at 0.04
         PT facilitates RS at 0.04 PS facilitates LS at 0.04
         PT facilitates LS at 0.04 RS incompatible RC at 0.06
         PT incompatible RC at 0.06 PL incompatible PS at 0.06
         I incompatible E at 0.06 E incompatible S at 0.06
         B9 contradicts B10 at 0.06
       @analogies
         analogous (B6, B5) (B9, B10) at 0.6
   endnet
   @portmapping
   @behaviormapping
       X.LS mapsto LethalStrike
   endcogent
```

Figure 5.3: Updated Lethal Strike Model Specification

5.2.3 Behavior Definitions and Executions

Cogent's DSL is designed to bring a balance between the DSL-based and the library-based languages, which are discussed in Section 2.1.1 and 2.1.2 respectively. Cogent platform allows users to define behaviors for their cogents, these behaviors can be any Scala functions. Thus, the cognitive model is specified using the DSL, while the behaviors that these cognitive models execute are specified by Scala programming language syntax. In the above problem, we also defined a behavior in the behavior section of the DSL called the *LethalStrike* behavior.

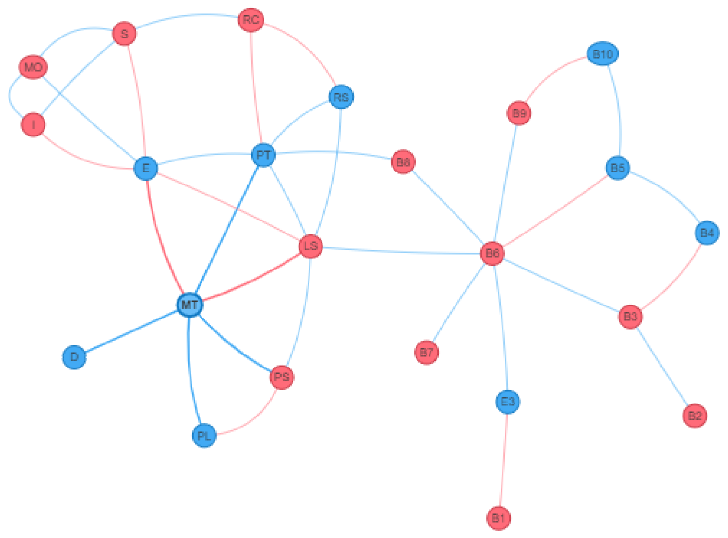
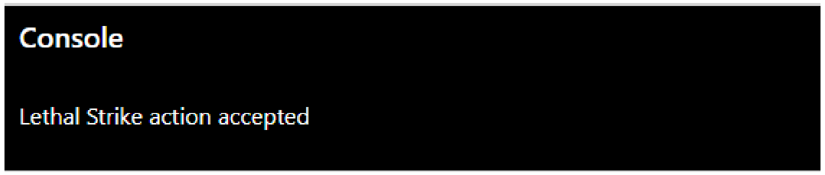


Figure 5.4: Lethal Strike action rejected

This behavior is later associated with the Lethal Strike action in the behavior mapping section of the DSL. We also have a data variable named "energy", which denotes the energy of the UAV. The agent might be expected to expel or reduce this variable when a behavior such as *LethalStrike* is executed. This can be seen in the *LethalStrike* behavior, where we reduce the *energy* variable. The *LethalStrike* behavior additionally checks if the agent has sufficient energy and executes the action. Hence, when the action node gets activated in the Coherence Network, the behavior or the function executes. In case of the DSL specification in Figure 5.3, the behavior fails to execute as the Lethal Strike action is rejected. But, the Lethal Strike action is accepted for the DSL specification in Figure 5.2, thus running the *LethalStrike* behavior. Figure 5.5 shows the console section after running the behavior. Cogent also allows users to view the details of all the nodes in a Coherence Network. Figure 5.6 shows the details of nodes $B3$ and $B4$, for the network configuration in Figure 5.2.

5.2.4 Hierarchical Configuration of the Problem Specification

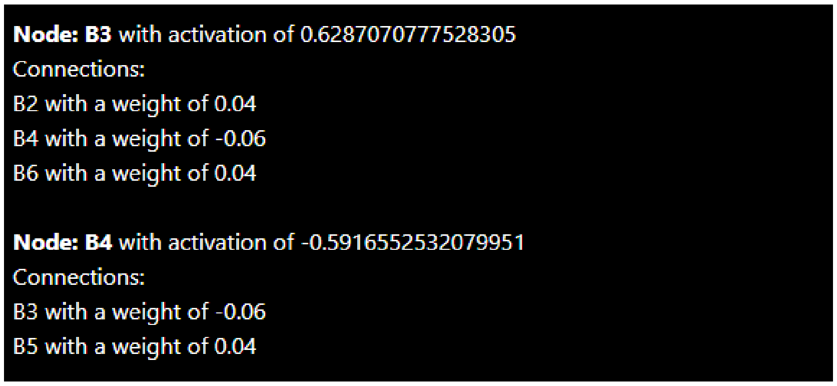
The two scenarios of the Lethal Strike problem discussed before are flat networks. That is, the scenarios do not involve any hierarchy or sub-networks. Hierarchical networks can be



```
Console

Lethal Strike action accepted
```

Figure 5.5: Console Output



```
Node: B3 with activation of 0.6287070777528305
Connections:
B2 with a weight of 0.04
B4 with a weight of -0.06
B6 with a weight of 0.04

Node: B4 with activation of -0.5916552532079951
Connections:
B3 with a weight of -0.06
B5 with a weight of 0.04
```

Figure 5.6: Node Details

used in a scenario where a node A's activation has to depend on the *harmony* of a group of nodes. In such a case, the group of nodes can be specified as a separate network and connected to the node A using ports. The Lethal Strike problem can also be extended into a hierarchical network configuration. For instance, the acceptability of the belief $B9$, which states that capital punishment is acceptable, may depend on a set of other beliefs, hypotheses, and evidences. This is analogous to a real-world scenario, where a jury decides to award capital punishment to a criminal, based on the consideration of a set evidences and hypotheses that are irrelevant to the trial itself.

5.2.5 Observations and Remarks

In the previous sections, we have demonstrated Cogent's capabilities by using the Lethal Strike problem. Although, the Lethal Strike problem involved the consideration of a complex model with various beliefs, evidences, goals, and actions, the Cogent platform's powerful DSL allowed us to create a cognitive model by directly expressing these deliberation mechanisms. Cogent's approach to cognitive modeling and specification is contrast to the common procedural approach to modeling cognitive agents. As discussed in Section 2.2, the procedural approach to cognitive modeling may be less intuitive and hard to express complex cognitive agents. But in the case of Cogent, we not only expressed the Lethal Strike problem intuitively using deliberation mechanisms, we were also able to shift between two entirely different ethical decision scenarios by modifying certain beliefs intuitively. This clearly demonstrates Cogent's ability to perform powerful cognitive modeling using a higher-level DSL. We also demonstrated Cogent's ability to combine Domain-Specific Language with the Scala programming language in creating the *LethalStrike* behavior with programming language syntax, while creating the cognitive model using the DSL.

Chapter 6
Conclusions

The Cogent platform allows users to model, simulate, and experiment with complex cognitive agents, the theoretical foundations of which are based on Coherence Theory. The modeling process for these cognitive agents is orchestrated by the principles and ideologies of Model-Driven Engineering. Cogent allows cognitive modeling to be performed iteratively and interactively using its powerful Domain-Specific Language (DSL) and features like visualization. Cogent's DSL is designed to be expressive enough to allow modelers to specify complex cognitive models explicitly using deliberation and reasoning mechanisms, while also allowing to specify the behaviors of these agents using the Scala programming language syntax.

6.1 Summary

In this thesis, we introduced Cogent, a web-based agent simulation and experimentation platform for simulating cognitive agents. In Chapter 2, we introduced various simulation platforms and cognitive architectures, discussing their properties and drawbacks. Most of these simulation platforms and cognitive architectures, do not allow users to express deliberation mechanisms explicitly to model agents. Hence, modeling complex decision-making agents using these platforms can be difficult and time-consuming. We discussed how Cogent addresses the problem by combining cognitive modeling with the principles of Model-Driven Engineering such as Domain-Specific Language (DSL) and transformation. In Chapter 3, we introduced the principles of Coherence Theory and Model-Driven Engineering, showing how they act as a backbone of the Cogent platform. Cogent allows users to express cognitive agents intuitively by connecting users to agents, using real-world deliberation ideas like beliefs and goals. This is achieved by an expressive DSL of Cogent that allows to model agents involved in complex scenarios that include conflicts and dilemmas. In Chapter 4, we discussed the implementation and syntax details of Cogent's DSL and the transformation from a DSL specification to a reference implementation of Coherence Theory. We saw how Cogent's DSL design allowed us to model cognitive agents intuitively. We also saw how the DSL allowed us to define an agent's behaviors using programming language syntax, thus finding a balance between the library-based and the DSL-based languages. Based on the DSL and the reference implementation, Cogent uses a reasoning mechanism by constraint satisfaction, accepting certain principles and rejecting others. In Chapter 4, we also introduced the various user interface features of the platform. Cogent promotes iterative and incremental development of cognitive agents by providing capabilities such as visualization of the agent, acting as a tool to help users to understand the simulated agent. In Chapter 5, we demonstrated Cogent's cognitive modeling capability by expressing the Lethal Strike problem explicitly through deliberation mechanisms, while defining the agent's behavior using Scala language syntax. Cogent's powerful modeling capabilities were also apparent when we shifted between two entirely different ethical decision scenarios by modifying the DSL intuitively.

6.2 Limitations of Cogent

In this section, we will discuss some of the limitations faced by the Cogent platform, and discuss briefly on how to address them. Cogent verifies the syntax of the DSL that the user specifies but does not verify the DSL's semantics. Scenarios such as the user trying to connect two nodes that aren't defined will lead to run-time errors. This is an implementation limitation in Cogent, and the issue can be solved by implementing a type checking mechanism for Cogent. Cogent also does not provide an analysis on how much the initial activation of each evidence and belief can be changed to a point where we get a completely new set of propositions being accepted. An analysis of such kind may be essential as it would give the user a higher confidence in the model. Such an analysis may act as a measure of robustness of the model and can be used as an evaluation metric to compare various models. To handle this limitation, we need to implement a meta-level reasoning mechanism for the Cogent.

6.3 Application Areas of Cogent

We previously demonstrated Cogent using the Lethal Strike problem, which is a problem in the domain of Machine Ethics. However, Cogent can be applied to a problem in any domain where a cognitive decision-making process is necessary. Cogent can be used in a wide variety of ways, from modeling autonomous systems to modeling real-life scenarios faced by people. Cogent can be used to model a higher-level decision-making process of systems such as Unmanned Ariel Vehicles and driverless cars. These autonomous systems can be modeled using propositions such as beliefs and goals, but their actions would depend on these defined propositions and evidences that the system observes. Cogent can also be used to simulate real-life scenarios that involve complex reasoning and decision-making. For instance, a jury in a trial may face contradictory propositions such as hypothesis and evidences. In such a case, Cogent can be used as a decision support system, by using it to model the problem faced by the jury, expressing all hypotheses, evidences, beliefs, and actions. Cogent could give the set of propositions that contradict and support each other, accepting certain propositions and rejecting others. Cogent's simulation results can help the jury to see a higher-level overview of the problem and aid them in the decision-making process.

6.4 Future Research

6.4.1 Multi-Agent Simulation Platform

Currently, Cogent allows users to model and experiment with just a single agent. In many scenarios, users may desire to simulate multiple coordinating agents in an environment.

Agents simulated in a multi-agent environment will be more complex, as their actions, not only depend on the environment but also other agents.

Cogent is envisioned to be expanded into a multi-agent simulation platform. The multi-agent platform can allow users to model multiple cognitive agents that interact with each other. Similar to a cogent, these cognitive agents will have their own cognitive models that utilize various deliberation mechanisms, such as beliefs and hypotheses. The platform can also allow users to define advanced behaviors for an agent, allowing it to interact with other cognitive agents. Like a cogent, these agents can modify the environment defined by the user. To support such a multi-agent environment, we need to add new features to the current Cogent platform. Each agent in this platform can be simulated as a separate process with their own cognitive model. As these processes share the variables that represent the environment, the access to them will have to be restricted using semaphores to avoid race conditions. The proposed multi-agent platform is expected to be more powerful and would allow users to simulate complex scenarios such as social simulation.

6.4.2 Hybrid Approach to Model Agents

Currently, Cogent allows users to define cognitive agents in an iterative top-down fashion. This approach allows users to specify how these agents must function, thus allowing the user have greater control over their behavior. On the contrary, a bottom-up approach allows agents to be built from training data. The bottom-up approach can be used to give an overall view of the model, thus guiding the user in the top-down modeling.

Although we provide a package that would help users to develop initial configuration for the Coherence Network, it can be integrated with the top-down process of modeling. Cogent's modeling process can be extended to a hybrid approach by integrating the top-down approach and the bottom-up approach to modeling. Thus, the proposed platform will allow agents to be specified using a DSL, while they can also to be created simultaneously from training data. The proposed approach is expected to enhance the modeling experience, as the user can now revise the top-down specification and the bottom-up data, during the iterative modeling process.

6.4.3 Extending Cogent as a Cognitive Architecture

Most cognitive architectures use a symbolic approach to cognition, this approach might be non-intuitive and challenging, as users may have to convert their deliberations into production rules. Cogent's approach is relatively different, as its Domain-Specific Language allows users to model cognitive agents using deliberation mechanisms, such as beliefs and evidences, thus helping users define complex cognitive agents intuitively. Although, the specification of agents in Cogent is intuitive, agents modeled by cognitive architectures such as Clarion are more powerful. Cognitive architectures have multiple memory modules and provide the capability to model agents that learn gradually from experience. We propose to extend Cogent into a cognitive architecture, while also not sacrificing Cogent's capability of modeling agents using deliberation mechanisms.

Currently, agents in Cogent are limited to only a short-term memory, and they cannot learn progressively through experience. We propose to extend the capability of Cogent by allowing it to model agents with a long-term and a short-term memory. The short-term memory can store the current attributes of the problem such as various beliefs, hypotheses, and their activations. The long-term memory can store the attributes from past simulation results. The short-term memory will primarily help the agent in solving the current problem, while the long-term memory will help the agent learn progressively.

Cogent can also be extended by providing an implicit and an explicit cognition, similar to the Clarion cognitive architecture. The explicit cognition will be guided by a Coherence Network, while the implicit cognition will be guided by a feed-forward neural network. The explicit cognition is proposed to work on the short-term memory, while the implicit cognition will work on the long-term memory. Coherence networks are hard to train with multiple training instances. On the contrary, neural networks have proved to work well with large volumes of training instances. Hence we propose to use a feed-forward neural network for the implicit cognition. The advantage of Coherence Networks is that they are more explainable than neural networks, hence, we intend to use the Coherence Network approach for the explicit cognition. After each simulation, the results will be transferred from the short-term memory to the long-term memory, and the neural-network would be updated using the backpropagation algorithm. Agents can now make decisions based on the results obtained by both implicit and explicit cognition. The proposed architecture will allow users to model powerful agents that learn progressively and are also explainable, thus providing greater confidence to users.

Bibliography

[1] S. Railsback, S. Lytinen, and S. Jackson, "Agent-based simulation platforms: Review and development recommendations," *Society for Modeling and Simulation International*, 2006.

[2] T. Schelling, "Micromotives and macrobehavior," *Norton*, 1978.

[3] M. Pechoucek and D. Sislak, "Agent-based approach to free-flight planning, control, and simulation," *IEEE Intelligent Systems*, 2009.

[4] C. Adam and B. Gaudou, "Bdi agents in social simulations: a survey," *The Knowledge Engineering Review*, vol. 31, no. 3, p. 207238, 2016.

[5] L. Yilmaz, "Verification and validation of ethical decision-making in autonomous systems," in *Proceedings of the Symposium on Modeling and Simulation of Complexity in Intelligent, Adaptive and Autonomous Systems.* Society for Computer Simulation International, 2017, p. 1.

[6] A. Grogan, "Driverless trains: It's the automatic choice," *Engineering & Technology*, vol. 7, no. 5, pp. 54–57, 2012.

[7] K. Arkoudas, S. Bringsjord, and P. Bello, "Toward ethical robots via mechanized deontic logic," in *AAAI Fall Symposium on Machine Ethics*, 2005, pp. 17–23.

[8] J. Gips, "Towards the ethical robot," *Android epistemology*, pp. 243–252, 1995.

[9] S. Tisue and U. Wilensky, "Netlogo: A simple environment for modeling complexity," in *International Conference on Complex Systems*, 2004.

[10] R. Tobias and C. Hofmann, "Evaluation of free java-libraries for social-scientific agent based simulation," *Journal of Artificial Societies and Social Simulation*, vol. 7, no. 1, 2004.

[11] P. Thagard, *Coherence in Thought and Action.* MIT Press, 2000.

[12] S. Abar, G. Theodoropoulos, P. Lemarinier, and G. OHare, "Agent based modelling and simulation tools: A review of the state-of-art software," *ELSEVIER*, 2017.

[13] N. Minar, R. Burkhart, C. Langton, M. Askenazi *et al.*, "The swarm simulation system: A toolkit for building multi-agent simulations," 1996.

[14] N. Collier, *RePast: An Extensible Framework for Agent Simulation*, Social Science Research Computing University of Chicago, 2003.

[15] M. J. North, N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko, "Complex adaptive systems modeling with repast simphony," *Complex Adaptive Systems Modeling*, vol. 1, no. 1, p. 3, Mar 2013.

[16] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan, "Mason: A multi-agent simulation environment," *Society for Modeling and Simulation International*, 2005.

[17] F. Bellifemine, A. Poggi, and G. Rimassa, "Jade a fipa-compliant agent framework," in *Fourth International Conference on Practical Application of Intelligent Agents and Multi-Agent Technology*, 1999.

[18] "Foundation for intelligent physical agents," specifications. [Online]. Available: http://www.fipa.org

[19] L. Braubach, W. Lamersdorf, and A. Pokahr, "Jadex: Implementing a bdi-infrastructure for jade agents," 2003.

[20] M. Resnick, "Starlogo: an environment for decentralized modeling and decentralized thinking," in *Conference companion on Human factors in computing systems.* ACM, 1996, pp. 11–12.

[21] S. Papert, "Windstorms," 1980.

[22] J. Ozik, N. Collier, J. Murphy, and M. North, "The relogo agent-based modeling language," in *Winter Simulation Conference*, 2013.

[23] J. Ozik, *Relogo Getting Started Guide*, Repast Development Team, 2013.

[24] J. Laird, *The Soar Cognitive Architecture.* MIT Press, 2012.

[25] ——, "Extending the soar cognitive architecture," in *Artificial General Intelligence*, 2008.

[26] D. E. Kieras and D. E. Meyer, "The epic architecture for modeling human information-processing and performance: A brief introduction," MICHIGAN UNIV ANN ARBOR DIV OF RESEARCH DEVELOPMENT AND ADMINISTRATION, Tech. Rep., 1994.

[27] J. Rosbe, R. S. Chong, and D. E. Kieras, "Modeling with perceptual and memory constraints: An epic-soar model of a simplified enroute air traffic control task," SOAR TECHNOLOGY INC ANN ARBOR MI, Tech. Rep., 2001.

[28] J. Anderson, D. Bothell, M. Byrne, S. Douglass, C. Lebiere, and Y. Qin, "Mason: A multi-agent simulation environment," *Society for Modeling and Simulation International*, 2005.

[29] S. Ron, "The clarion cognitive architecture: Extending cognitive modeling to social simulation," in *Cognition and Multi-Agent Interaction*, S. Ron, Ed. New York: Cambridge University Press, 2005, ch. 4.

[30] S. Franklin and F. Patterson Jr, "The lida architecture: Adding new modes of learning to an intelligent, autonomous, software agent," *pat*, vol. 703, pp. 764–1004, 2006.

[31] P. Langley and D. Choi, "A unified cognitive architecture for physical agents," in *Proceedings of the National Conference on Artificial Intelligence*, vol. 21, no. 2. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006, p. 1469.

[32] P. Thagard, "Extending explanatory coherence," *Behavioral and brain sciences*, vol. 12, no. 3, pp. 490–502, 1989.

[33] P. Thagard, C. Eliasmith, P. Rusnock, and C. P. Shelley, "Knowledge and coherence," *Common sense, reasoning, and rationality*, vol. 11, pp. 104–131, 2002.

[34] P. Thagard and E. Millgram, "19 inference to the best plan: A coherence theory of decision," *Goal-driven learning*, p. 439, 1995.

[35] J. L. McClelland and D. E. Rumelhart, "An interactive activation model of context effects in letter perception: I. an account of basic findings." *Psychological review*, vol. 88, no. 5, p. 375, 1981.

[36] E. Aarts and J. Korst, *Simulated annealing and boltzmann machines.* New York, NY; John Wiley and Sons Inc., Jan 1988.

[37] X. Xie and H. S. Seung, "Equivalence of backpropagation and contrastive hebbian learning in a layered network," *Neural computation*, vol. 15, no. 2, pp. 441–454, 2003.

[38] R. C. O'Reilly, "Biologically plausible error-driven learning using local activation differences: The generalized recirculation algorithm," *Neural Computation*, vol. 8, no. 5, pp. 895–938, 1996.

[39] S. W. Liddle, "Model-driven software development," in *Handbook of Conceptual Modeling.* Springer, 2011, pp. 17–54.

[40] "Model driven engineering tools compared on user activities," http://www.theenterprisearchitect.eu/blog/2009/02/04/roles-in-model-driven-engineering, accessed: 2017-02-13.

[41] J. Bézivin and O. Gerbé, "Towards a precise definition of the omg/mda framework," in *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on.* IEEE, 2001, pp. 273–280.

[42] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, *Model-driven software development: technology, engineering, management.* John Wiley & Sons, 2013.

[43] D. C. Schmidt, "Model-driven engineering," *COMPUTER-IEEE COMPUTER SOCIETY-*, vol. 39, no. 2, p. 25, 2006.

[44] "Scalatra," http://scalatra.org/, accessed: 2017-02-22.

[45] "Api documentation of scala.util.parsing.combinator," http://www.scala-lang.org/api/2.12.3/scala-parser-combinators/scala/util/parsing/combinator/JavaTokenParsers.html, accessed: 2017-02-22.

[46] A. Shvets, M. Pavlova, and G. Frey, "Design patterns explained simply," https://sourcemaking.com/design_patterns/composite, accessed: 2017-02-22.

[47] "Ace: High performance code editor for the web," https://ace.c9.io/, accessed: 2017-02-22.

[48] C. Allen, W. Wallach, and I. Smit, "Why machine ethics?" *IEEE Intelligent Systems*, vol. 21, no. 4, pp. 12–17, 2006.

[49] J. J. Thomson, "The trolley problem," *The Yale Law Journal*, vol. 94, no. 6, pp. 1395–1415, 1985.

[50] J. H. Moor, "The nature, importance, and difficulty of machine ethics," *IEEE intelligent systems*, vol. 21, no. 4, pp. 18–21, 2006.

[51] L. Yilmaz, A. Franco-Watkins, and S. Kroecker, Timothy, "Coherence-driven reflective equilibrium model of ethical decision-making," in *CogSIMA*, 2016.

Appendices

Appendix A

Grammar Implementation using Scala combinator

```
class CoherenceNet extends JavaTokenParsers {


    def cogentspec: Parser[Cogent] =
        "cogent"~>ident~dataspec~bspec~cmspec ^^ {case name~data~behavior~cm
        => Cogent(name.toString(), data, behavior, cm)}


    def dataspec: Parser[List[String]] =
        "@data"~>repsep(datastmt, ";") ^^ (List() ++ _)


    def datastmt: Parser[String] = stringLiteral ^^ {case stmt => stmt}


    def bspec: Parser[List[String]] =
        "@behavior"~>repsep(datastmt, ";") ^^ (List() ++ _)


    def bstmt: Parser[String] = stringLiteral ^^ {case stmt => stmt}


    def cmspec: Parser[CogModel] = "@cognitiveModel"~>
        netspec~opt(inoutmappinglist)~opt(behmappinglist)<~"endcogent"
        ^^ {case netspec~inoutmappinglist~behmappinglist
        => CogModel(netspec, inoutmappinglist, behmappinglist)}


    def inoutmappinglist: Parser[List[Mapping]] =
        "@portmapping"~>rep(inoutmappingmember) ^^ (List() ++ _)

    def inoutmappingmember:Parser[Mapping] =
        "outp:"~>ident~"."~ident~"connects"~"inp:"~ident~"."~ident
        ~"at"~floatingPointNumber ^^ {case outNet~"."~outPort~"connects
        "~"inp:"~inpNet~"."~inpPort~"at"~value =>
        Mapping(outNet.toString(),outPort.toString(),inpNet.toString()
        ,inpPort.toString(), value.toFloat)}

    def behmappinglist: Parser[List[BMapping]] =
        "@behaviormapping"~>rep(behmappingmember) ^^  (List() ++ _)

    def behmappingmember: Parser[BMapping] = ident~"."~ident~"mapsto"~ident
        ^^ {case netName~"."~nodeName~"mapsto"~behName =>
        BMapping(netName.toString(), nodeName.toString(),
        behName.toString())}

    def netspec: Parser[Network] = "net"~>ident~opt(portlist)~
        opt(portmapper)~evidencelist~belieflist~hypothesislist
        ~goallist~actionlist~constraintlist~analogylist~opt(netlist)
        <~"endnet" ^^ {case ident~portlist~portmapper~evidencelist
        ~belieflist~hypothesislist~goallist~actionlist~constraintlist
        ~analogylist~netlist => Network(ident.toString(), portlist,
        portmapper, evidencelist, belieflist, hypothesislist,
        goallist, actionlist, constraintlist, analogylist, netlist)}

    def netlist: Parser[List[Network]] = rep(netspec) ^^ (List() ++ _)
    def portlist : Parser[List[Port]] = "("~>inportlist~outportlist
        <~")" ^^ {case inportlist~outportlist => inportlist ::: outportlist}

    def inportlist: Parser[List[Port]] = "inp"~":"~>
        repsep(inportmember, ",")<~";" ^^ (List() ++ _)

    def outportlist: Parser[List[Port]] = "outp"~":"~>
        repsep(outportmember, ",") ^^ (List() ++ _)

    def inportmember : Parser[Port] = ident ^^
        {case name => Port(name.toString(), "in")}

    def outportmember : Parser[Port] = ident ^^
        {case name => Port(name.toString(), "out")}

    def portmapper: Parser[Map[Port,Node]] = "["~>inportmapper
        ~outportmapper <~"]" ^^ {case inportmapper~outportmapper=>
        inportmapper ++ outportmapper}

    def inportmapper: Parser[Map[Port,Node]] =
        repsep(inportmap, ",")<~";" ^^ (Map() ++ _)

    def outportmapper: Parser[Map[Port,Node]] =
        repsep(outportmap, ",") ^^ (Map() ++ _)

    def inportmap : Parser[(Port,Node)] = ident~"->"~ident ^^
```

```
    {case src~"->"~tgt => (Port(src.toString(), "in"),
    Evidence(tgt.toString(), Some(""), 0.2))}

def outportmap: Parser[(Port,Node)] =  ident~"->"~ident ^^
    {case src~"->"~tgt =>(Port(tgt.toString(), "out"),
    Hypothesis(src.toString(), Some(""), 0.2))}

def evidencelist: Parser[List[Evidence]] =
    "@percepts"~>rep(evidencemember) ^^ (List() ++ _)

def evidencemember: Parser[Evidence] =
    "evidence("~>ident~opt(explanation)~","~floatingPointNumber
    <~")" ^^ {case ident~expl~","~value =>
    Evidence(ident.toString(), expl, value.toDouble)}

def explanation : Parser[String] =
    ":"~>stringLiteral^^{case expl => expl}

def belieflist: Parser[List[Belief]] =
    "@beliefs"~>rep(beliefmember) ^^ (List() ++ _)

def beliefmember: Parser[Belief] =
    "belief("~>ident~opt(explanation)~","~floatingPointNumber
    <~")" ^^ {case ident~expl~","~value =>
    Belief(ident.toString(), expl, value.toDouble)}

def hypothesislist: Parser[List[Hypothesis]] =
    "@explanations"~>rep(hypothesismember) ^^ (List() ++ _)

def hypothesismember: Parser[Hypothesis] =
    "hypothesis("~>ident~opt(explanation)~","~floatingPointNumber
    <~")" ^^ {case ident~expl~","~value => Hypothesis(
    ident.toString(), expl, value.toDouble)}

def goallist: Parser[List[Goal]] =
    "@goals"~>rep(goalmember) ^^ (List() ++ _)

def goalmember: Parser[Goal] =
    "goal("~>ident~opt(explanation)~","~floatingPointNumber<~")
    " ^^ {case ident~expl~","~value => Goal(ident.toString(),
    expl, value.toDouble)}

def actionlist: Parser[List[Action]] =
    "@actions"~>rep(actionmember) ^^ (List() ++ _)

def actionmember: Parser[Action] =
    "action("~>ident~opt(explanation)~","~floatingPointNumber<~")"
    ^^ {case ident~expl~","~value =>
    Action(ident.toString(), expl, value.toDouble)}

def constraintlist: Parser[List[Constraint]] =
    "@constraints"~>rep(constraintmember) ^^ (List() ++ _)

def constraintmember: Parser[Constraint] =
    basicconstraint | compoundconstraint

def basicconstraint: Parser[Constraint] =
    ident~"explains"~ident~"at"~floatingPointNumber ^^
    {case src~"explains"~tgt~"at"~value =>
    Constraint(src.toString :: List(), tgt.toString,
    "explains", value.toDouble)} |
    ident~"contradicts"~ident~"at"~floatingPointNumber ^^
    {case src~"contradicts"~tgt~"at"~value =>
    Constraint(src.toString :: List(), tgt.toString,
    "contradicts", value.toDouble)} |
    ident~"deduces"~ident~"at"~floatingPointNumber ^^
    {case src~"deduces"~tgt~"at"~value =>
    Constraint(src.toString :: List(), tgt.toString,
    "deduces", value.toDouble)} |
    ident~"facilitates"~ident~"at"~floatingPointNumber ^^
    {case src~"facilitates"~tgt~"at"~value =>
    Constraint(src.toString :: List(), tgt.toString,
    "facilitates", value.toDouble)} |
    ident~"incompatible"~ident~"at"~floatingPointNumber ^^
    {case src~"incompatible"~tgt~"at"~value =>
    Constraint(src.toString :: List(), tgt.toString,
    "incompatible", value.toDouble)} |
    ident~"triggers"~ident~"at"~floatingPointNumber ^^
    {case src~"triggers"~tgt~"at"~value =>
    Constraint(src.toString :: List(), tgt.toString,
    "triggers", value.toDouble)}
    def compoundconstraint : Parser[Constraint] =
        srclist~"explains"~ident~"at"~floatingPointNumber ^^
        {case srclist~"explains"~tgt~"at"~value =>
        Constraint(srclist,tgt,"explains",value.toDouble) }

    def srclist: Parser[List[String]] =
        "["~>repsep(ident, ",")<~"]" ^^ (List() ++ _)

    def analogylist :Parser[List[Analogy]] =
        "@analogies"~>rep(analogymember) ^^ (List() ++ _)

    def analogymember: Parser[Analogy] =
        "analogous"~hyplist~evilist~"at"~floatingPointNumber ^^
        {case "analogous"~hyplist~evilist~"at"~value =>
        Analogy(hyplist, evilist, value.toDouble)}

    def hyplist: Parser[List[String]] =
        "("~>repsep(ident, ",")<~")" ^^ (List() ++ _)

    def evilist: Parser[List[String]] =
        "("~>repsep(ident, ",")<~")" ^^ (List() ++ _)
}
```