2011

The LIDA Framework as a General Tool for AGI

J. Snaider

R. McCall

S. Franklin

Recommended Citation
Snaider, J., McCall, R., & Franklin, S. (2011). The LIDA Framework as a General Tool for AGI. 10.1007/978-3-642-22887-2_14

The LIDA Framework as a General Tool for AGI

Javier Snaider[1] , Ryan McCall[2], Stan Franklin[3]
[1-2] FedEx Institute of Technology #403h, 365 Innovation Dr., Memphis, TN 38152,
[3] FedEx Institute of Technology #312, 365 Innovation Dr., Memphis, TN 38152,
[1]jsnaider@memphis.edu, [2]rmccall@memphis.edu, [3]franklin@memphis.edu

Abstract. Intelligent software agents aiming for general intelligence are likely to be exceedingly complex systems and, as such, will be difficult to implement and to customize. Frameworks have been applied successfully in large-scale software engineering applications. A framework constitutes the skeleton of the application, capturing its generic functionality. Frameworks are powerful as they promote code reusability and significantly reduce the amount of effort necessary to develop customized applications. They are well suited for the implementation of AGI software agents. Here we describe the LIDA framework, a customizable implementation of the LIDA model of cognition. We argue that its characteristics make it suitable for wider use in developing AGI cognitive architectures.

Keywords: AGI framework, software framework, computational framework, cognitive architecture, design patterns, LIDA model

1 Introduction

Artificial General Intelligence (AGI) aims at producing agents exhibiting human-level intelligence and beyond. Any successful AGI agent must surely be implemented using a sophisticated cognitive architecture — but which one to choose? A comparative table of cognitive architectures currently lists twenty-nine candidates [1]. If every AGI research group is focused on their own control architecture, how can the field of AGI progress?

Superficially, these architectures seem quite different from one another in their structure, and they use vastly different terminology. However, closer inspection reveals much similarity between the function of the modules of one architecture and those of another once the common meanings of different vocabulary are mapped onto an accepted ontology. The beginnings of such an ontology have been proposed [2]. Once the similarity of function among modules becomes apparent, the architectures themselves seem less different in structure, and perhaps, more amenable to implementation using a common software framework. Such a common underlying framework might likely result in a "tree" of cognitive and/or AGI architectures with branches at every point of difference. Architectures would be quicker to implement due to code reuse, and easier to analyze and compare.

Here we propose such an underlying computational software framework for AGI and offer, as an example, one based on the LIDA cognitive architecture. The advantages of using such a framework were stated just above. A possible

disadvantage is that to use any such framework the developers must commit to the underlying assumptions of the LIDA architecture upon which this framework is based. We will argue that this software framework requires commitment to only a minimal set of assumptions, one that is not too onerous for other AGI research projects.

In recent years, an enormous number of computational frameworks have appeared in the software engineering world. See for example [3, 4]. This is not by chance but is due to the advantages of using frameworks. They promote code reuse and significantly reduce the amount of effort necessary to develop customized applications. Intelligent software agents aiming for general intelligence are complex systems and as such are difficult to implement and to customize. We will argue that ideas from frameworks are well-suited for the implementation of such generally-intelligent software agents. Here we describe the LIDA framework, a customizable implementation of the LIDA model of cognition. While the LIDA model provides a conceptual ontology for general models of cognition [2], we hope that the LIDA software framework might provide a customizable computational framework with which to more economically develop AGI architectures, as well as to more easily analyze and compare them.

We begin this paper by describing the general characteristics of frameworks and the advantages of using them to implement generally intelligent agents. Then we sketch the LIDA model and outline the LIDA framework. Next we describe the main components of the framework in some detail, and, finally, we summarize the minimal assumptions required for an AGI using this framework and draw some conclusions.

2 Frameworks

A framework is a reusable implementation of all or part of a software system. In many cases, a framework constitutes the skeleton of the application, capturing its generic functionality. The framework specifies a well-defined application programming interface (API) that is implemented generically using abstract classes, interfaces, and generic, customizable module design. This hides the complexity of its code from the user. Most frameworks are based on object-oriented languages because the major properties of OO, data abstraction, inheritance, information hiding and polymorphism, complement the goals of frameworks.

The core idea of a framework is to have a generic design as well as a base implementation of a complex software system. The user of the framework then only needs to "fill in the blanks" with problem or domain-specific elements. This is, perhaps, the major advantage of using frameworks: users can concentrate their efforts on the specifics of the problems, and reuse the generic mechanisms implemented in the framework. This also speeds up the development of the new application and makes it less error-prone because part of the system has already been produced and tested.

Frameworks, in general, promote the use of proven design patterns and good practices in software development [5, 6]. This leads to better application designs, more manageable maintainability and easier extension of the application. The

framework's API also provides a higher level of abstraction at which to define the application. This API is composed of elements with names, characteristics and behaviors. They form a specific language among users of the framework, which facilitates a concise and clear description of the application.

2.1 Frameworks and Cognitive Architectures

Ideas from frameworks can be applied outside the domain of enterprise applications. In particular, cognitive systems aiming for general intelligence tend to be complex and sophisticated. This creates a barrier that makes them difficult to learn, implement, and customize. The use of frameworks can mitigate these issues.

Cognitive architectures are complex from two points of view: the theory behind it tends to be inherently complicated and, consequently, any software implementation is also very complex. Cognitive architectures are typically composed of several modules with different functionalities and, in many cases, with different algorithmic implementations. This makes implementing software agents based on them a very hard task. Developers have to spend a lot of time and energy re-implementing common functionality for each new agent implementation. Code reuse between architectures has been difficult in general because of lack of standardization and ill-defined modules.

Frameworks are ideal tools with which to solve many of the problems that implementations of generally intelligent systems entail. A framework for AGI systems allows developers to focus on their particular algorithms instead of implementation details common to many agents. The architecture can be understood more quickly because the framework's API itself provides a higher level of abstraction than unitary code. The API supplies a set of high-level concepts for elements of the architecture. These concepts abstract the complexity of the implementation and allow more effective and accurate communication between researchers.

3 The LIDA Model and its Architecture

The LIDA model [7-9] is a comprehensive, conceptual and computational model covering a large portion of human cognition[1]. Based primarily on Global Workspace theory [10, 11] the model implements and fleshes out a number of psychological and neuropsychological theories. The LIDA computational architecture is derived from the LIDA cognitive model. The LIDA model and its ensuing architecture are grounded in the LIDA cognitive cycle. Every autonomous agent [12], be it human, animal, or artificial, must frequently sample (sense) its environment and select an appropriate response (action). More sophisticated agents, such as humans, process (make sense of) the input from such sampling in order to facilitate their decision

[1] "Cognition" is used here in a particularly broad sense, so as to include perception, feelings and emotions.

making. The agent's "life" can be viewed as consisting of a continual sequence of these cognitive cycles. Each cycle constitutes a unit of sensing, attending and acting. A cognitive cycle can be thought of as a moment of cognition, a cognitive "moment."

We will now briefly describe what the LIDA model hypothesizes as the rich inner structure of the LIDA cognitive cycle. More detailed descriptions are available elsewhere [13, 14]. During each cognitive cycle the LIDA agent first makes sense of its current situation as best as it can *by updating its representation of its current situation, both external and internal*. By a competitive process, as specified by Global Workspace Theory [10], it then decides what portion of the represented situation is most in need of attention. Broadcasting this portion, the current contents of consciousness[2], enables the agent to chose an appropriate action and execute it, completing the cycle.

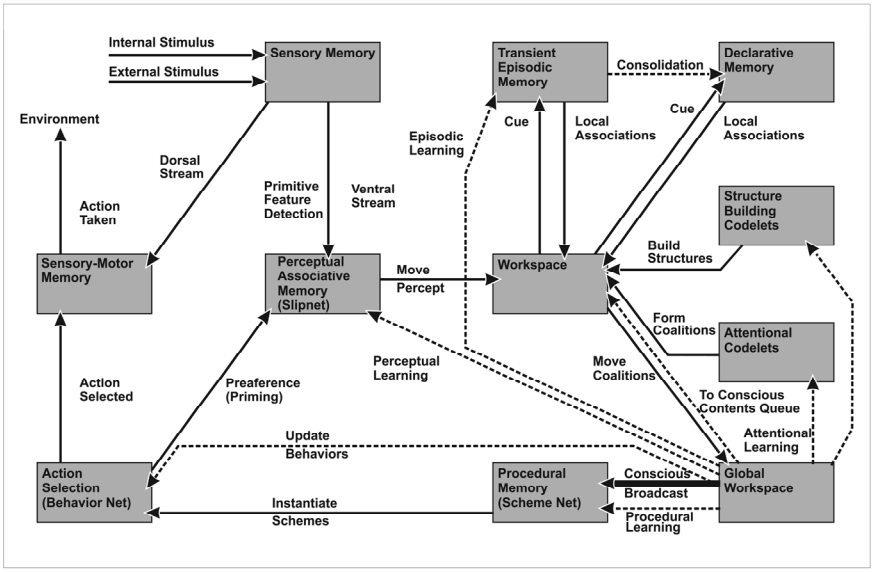

Figure 1. The LIDA Cognitive Cycle Diagram

Thus, the LIDA cognitive cycle can be subdivided into three phases, the understanding phase, the attention (consciousness) phase, and the action selection phase. Figure 1 should help the reader follow the description. It starts in the upper left corner and proceeds roughly clockwise. Beginning the understanding phase, incoming stimuli activate low-level feature detectors in Sensory Memory. The output is sent to Perceptual Associative Memory where higher-level feature detectors feed in to more abstract entities such as objects, categories, actions, events, etc. The resulting percept moves to the Workspace where it cues both Transient Episodic Memory and Declarative Memory producing local associations. These local associations are

[2] Here "consciousness" refers to functional consciousness [15]. We take no position on the need for, or possibility of, phenomenal consciousness.

combined with the percept to generate a Current Situational Model, which represents the agent's understanding of what is going on right now.

Attention Codelets[3] begin the attention phase by forming coalitions of selected portions of the Current Situational Model and moving them to the Global Workspace.

A competition in the Global Workspace then selects the most salient, the most relevant, the most important, and the most urgent coalition whose contents become the content of consciousness. These conscious contents are then broadcast globally, initiating the action selection phase. The action selection phase of LIDA's cognitive cycle is also a learning phase in which several processes operate in parallel (see Figure 1). New entities and associations, and the reinforcement of old ones, occur as the conscious broadcast reaches Perceptual Associative Memory. Events from the conscious broadcast are encoded as new memories in Transient Episodic Memory. Possible action schemes, together with their contexts and expected results, are learned into Procedural Memory from the conscious broadcast. Older schemes are reinforced. In parallel with all this learning, and using the conscious contents, possible action schemes are recruited from Procedural Memory. A copy of each such is instantiated with its variables bound and sent to Action Selection, where it competes to be the behavior selected for this cognitive cycle. The selected behavior triggers Sensory-Motor Memory to produce a suitable algorithm for the execution of the behavior. Its execution completes the cognitive cycle.

The Workspace requires further explanation. Its internal structure is composed of various input buffers and three main modules: the Current Situational Model, the Scratchpad and the Conscious Contents Queue [16]. The Current Situational Model is where the structures representing the actual current internal and external events are stored. Structure-building codelets are responsible for the creation of these structures using elements from the various submodules of the Workspace. The Scratchpad is an auxiliary space in the Workspace where structure-building codelets can construct possible structures prior to moving them to the Current Situational Model. The Conscious Contents Queue holds the contents of the last several broadcasts and permits LIDA to understand and manipulate time-related concepts [16].

4 The LIDA Framework

Based on all these ideas, we have been developing the LIDA software framework, a generic and customizable computational implementation of the LIDA model. It is implemented in Java, a strong and proven object oriented language.

The main goal of this framework is to provide a generic implementation of the LIDA model, easily customizable for specific problems or domains. As mentioned before, this has several advantages: it speeds up the implementation of new agents based on the LIDA model and shortens the learning curve to produce such implementations.

[3] A codelet is a small piece of code that performs a specific task in an independent way. It could be interpreted as a small part of a bigger process, similar to an ant in an ant colony.

The framework permits a declarative description of the specific implementation. The full architecture of the software agent is specified using an XML formatted file; this is similar to other frameworks where the use of declarative description files are common [4, 17]. In this way, the developer does not need to define the entire agent in Java; he can just define it using this XML specification file.

Another important goal of the framework is its ready customization. The customization can be done at several levels accordingly with the required functionality. At the most basic level, developers can use the LIDA configuration file to customize their applications. Several small pieces in the framework can also be customized implementing particular versions of them. For example, new strategies for decaying or codelets can be implemented. Finally, more advanced users can also customize and change internal implementation of whole modules. In each case, the framework provides default implementations that greatly simplify the customization process.

The framework was conceived with multithreading support in mind. Biological minds operate in parallel and so should artificial ones. *LIDA-tasks*, encapsulations of small processes together with a dedicated task manager, implement multithreading support that allows for a high level of parallelization. Finally, the LIDA framework implementation adheres to the most important design principles [5] and best programming practices.

4.1 Framework Outline

The LIDA framework defines several data structures and procedures (algorithms) and is composed of several pieces. Its main components are *modules*, interconnected elements that represent modules in the LIDA model. Another main component is the task manager that controls the execution of all processes in the framework. These processes are implemented by small, demon-like processors called LIDA-tasks. LIDA-tasks can be executed on separate threads by the *LIDA task manager* in a way that is almost transparent for the user. NodeStructures are core elements that constitute a main data structure in the framework. Finally, several supporting tools were implemented such as a customizable GUI, logging, and an architecture loader that parses an XML file with the definition and parameterization of the application.

Modules. For each main component of the LIDA cognitive model we define a module in the framework. In particular, each box in Figure 1 is implemented as a module in the framework. For example, the Sensory Memory, Workspace and Action Selection are all modules in the framework. All modules have a common interface (API) but also each one has its own API that defines its particular functionality. Modules can have submodules which are modules nested inside another module. For example, the Workspace in LIDA has several submodules such as the Current Situational Model.

Most modules in the LIDA framework are domain independent. For each of these, the framework provides a default implementation. For example, the Episodic Memory is implemented using a sparse distributed memory [18] and the Action Selection module by a heavily-modified behavior network [19]. Developers can use these

implementations and customize some of their parameters. Some modules however, are domain specific. In particular, Sensory Memory and Sensory-Motor Memory must be specified by the user. Nevertheless, the framework supplies default implementations for these modules from which users can extend their own domain-specific implementation.

For a more advanced customization of the framework, users can also implement their own version of any of the modules. Implementing a module's corresponding interface ensures its compatibility with the rest of the framework. For example, Episodic Memory could be implemented using a database. The default classes provided in the framework simplify the creation of alternate implementations of modules.

Modules need to communicate with other modules. To implement this, we use the observer design pattern [5]. In short, a module, called the "listener," which receives information from another "producer" module, can register itself to the producer as a listener. Each time the producer has something to send, it transmits the information to all of its registered listeners. There are numerous instances of listeners being used in the framework. Each listener type is implemented with its own interface. One module can be registered as a listener of several other modules. Also a module can be producer and listener of other modules at the same time. This pattern has several advantages; mainly, the listener and the producer do not need to know each other's internal structure and implementation, they only need to satisfy the particular listener interface. The arrows in Figure 1 are implemented as listeners in the framework.

Fundamental Data Structures. Another important piece of the framework is a data structure called the NodeStructure. A NodeStructure is a graph structure, containing nodes and the links between them. It constitutes the main representation of data in many framework modules. Several use NodeStructures to represent their internal data and, while other forms of representation are used in the framework; the NodeStructure functions as a representational "common currency" between many modules.

NodeStructures greatly assist in creating graph structures as they manage the low-level operations needed to add, remove, or retrieve particular Nodes and Links. Links are defined to connect a source Node with either another Node or a Link. These graphs are used for conceptual representation of object, actions, and events, the basic data representation in the LIDA model [20].

Nodes, Links and other LIDA elements such as coalitions, codelets, and behaviors, have activation. The activation can represent different things, but generally it represents the importance of the element. Elements can also have an additional "base-level" activation for learning. All activations are excited or decayed using "strategies". These are implementations of the strategy design pattern which allows for customizable behavior; in this case they specify the way activation of each element is excited or decayed.

Other basic data structures in the LIDA framework include bit vectors for the two episodic memory modules, schemes in Procedural Memory, coalitions for the Global Workspace, and behaviors in Action Selection. Each has an interface and a base implementation. Some are tied to specific module implementations; nonetheless, they are general enough that they could be used in other implementations as well.

LIDA-tasks. Modules need to perform several tasks in order to achieve their specific functionalities. The framework provides LIDA-tasks, encapsulations of small processes. A LIDA-task has an algorithm, a time of execution and a status. A module can create several LIDA-tasks to help it perform its function. A LIDA-task can run one time or repeatedly. A task that passes activation is an example of the former, while a structure-building codelet is an example of the latter. Some LIDA-tasks are likely to be fundamental for many AGI agent implementations, such as a task to pass activation. Others are implementation dependent and can be specified by the user. An example of this is a feature detector for a unique feature of a specific domain.

The execution of LIDA-tasks is delegated to the LIDA task manager. This important piece of the framework has the responsibility of scheduling and executing all the tasks of the application. It maintains a pool of threads, so several tasks can be executed at the same time. The task manager maintains a task queue which it uses to schedule LIDA-tasks for execution. Each position in the task queue represents a discrete instant in simulation time, which we call a *tick*. Ticks are numbered along the simulation, for example tick 1, tick 2, and so on. All tasks are scheduled to be executed at a specific tick. So if a single LIDA-task scheduled for tick *t* is enqueued in position *t*. All tasks scheduled for a particular tick are executed before the task manager advances to the next tick. Additionally, a parameter representing milliseconds called *tick duration*, can be set to ensure that *tick duration* milliseconds passes before the task manager moves onto the next position in the queue. With this mechanism, the whole simulation can run at different speeds, in slow motion, or even step by step.

4.2 Framework Tools

The current version of the LIDA framework features several useful tools. The first is a customizable GUI consisting of a main GUI application and a series of GUI panels which display such things as the content of modules, running tasks, parameter values, etc. A properties file allows users to add their own GUI panels as well as configure which panels are used and where they appear in the GUI window.

The Java logging API is used throughout the framework, recording important activities as they occur. Every log is made with one of several *levels* of severity. A dedicated GUI panel for Logging is part of the standard framework GUI. It allows the user to view logs of particular levels for specific modules or all modules.

An architecture loader allows agents to be specified via XML file. The loader reads this file and constructs an agent with modules, parameters, and initial tasks based on the file's specification. This utility obviates the need to specify agents by hand and allows for quick interchange of modules, connections between modules, change of parameters, etc.

Finally an *element factory*, implementing the factory design pattern [5], provides a centralized, configurable way to obtain new Nodes, Links, and Codelets. The excite and decay strategies used by objects created by the factory can be configured and changed dynamically. Factory support for additional object is planned in future versions.

4.3 Underlying Assumptions of the LIDA Framework

Even though originally intended for the LIDA model, the framework's general structure and functionality could be used to implement other general architectures as well. The scaffolding provided by the framework can benefit such implementations. This is an interesting but unexplored side of this framework.

There are a few requirements that any cognitive architecture using the framework should adhere to. Broadly, the architecture must be composed of interconnected modules, be able to divide their functionality into small tasks, and use a graph-like data structure as the main conceptual representation.

The first assumption is not a problem for most AGI cognitive architectures because in general they are structured in this way already. The second assumption is also common among cognitive architectures but the inherent asynchronous nature of this framework's task model may require a refactoring for some architectures. Nonetheless, a task can perform the whole operation of a module instead of a small part of it. This fact further relaxes this constraint.

Finally, the chosen common currency for communication between modules in the framework is the NodeStructure. This graph data structure can be used to represent a wide range of data types. It is inherently appropriate to represent connectionist data but symbolic constructs can also be represented. Other representation data types, such as images or sensors raw data, can be internally referenced by a Node in this structure. This is not directly supported by the current version of the framework however future versions of the framework will address this limitation.

In summary, there are few basic assumptions that architectures need to address in order to use this framework as a foundation for its implementation. Nonetheless, we believe these constraints are not prohibitively tight, making this framework a viable and general tool for AGI.

5 Conclusions

The LIDA software framework allows the creation of new intelligent software agents and experiments based in the LIDA model. Its design and implementation aim to simplify this process and to permit the user to concentrate in the specifics of the application, hiding the complexities of the generic parts of the model. It also enforces good software practices that simplify the creation of complex architectures. It achieves a high level of abstraction permitting several ways and levels of customization with a low level of coupling among modules. Supplemental tools such as a customizable GUI and logging support are also provided. The result is a powerful and customizable tool with which to develop LIDA based applications and, perhaps, many others as well. Much work is still needed to improve the performance of the framework and to add functionality. Learning mechanisms should be implemented in several modules and improved versions of Procedural Memory and Action Selection modules are in development.

References

1 BICA. Comparative Table of Cognitive Architectures. 2011 [cited 2011; Available from: http://bicasociety.org/cogarch/architectures.htm].
2 Franklin, S., A Foundational Architecture for Artificial General Intelligence, in Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms, Proceedings of the AGI Workshop 2006, B. Goertzel and P. Wang, Editors. 2007, IOS Press: Amsterdam. p. 36-54.
3 Singh, I., Stearns, B., and Johnson, M., Designing Enterprise Applications with the J2EE(TM) Platform (2nd Edition). 2002: Prentice Hall.
4 Walls, C. and Breidenbach, R., Spring in Action. Second Edition. 2007, Manning Publications.
5 Gamma, E., Helm, R., Johnson, R., and Vlissides, J.M., Design Patterns: Elements of Reusable Object-Oriented Software. 1995, Addison-Wesley Professional.
6 Alur, D., Crupi, J., and Malks, D., Core Java EE Patterns: Best Practices and Design Strategies, 2nd Edition. 2003: Prentice Hall.
7 Franklin, S. and Patterson, F.G.J., The LIDA Architecture: Adding New Modes of Learning to an Intelligent, Autonomous, Software Agent, in IDPT-2006 Proceedings (Integrated Design and Process Technology). 2006, Society for Design and Process Science.
8 Ramamurthy, U., Baars, B.J., D'Mello, Sidney K., and Franklin, S. LIDA: A Working Model of Cognition. in *7th International Conference on Cognitive Modeling*. 2006. Trieste: Edizioni Goliardiche.
9 Baars, B.J. and Franklin, S., Consciousness is computational: The LIDA model of Global Workspace Theory. International Journal of Machine Consciousness, 2009. 1(1): p. 23-32.
10 Baars, B.J., A Cognitive Theory of Consciousness. 1988, Cambridge: Cambridge University Press.
11 Baars, B.J., The conscious access hypothesis: origins and recent evidence. Trends in Cognitive Science, 2002. 6: p. 47–52.
12 Franklin, S. and Graesser, A.C., Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents, in Intelligent Agents III. 1997, Springer Verlag: Berlin. p. 21–35.
13 Baars, B.J. and Franklin, S., How conscious experience and working memory interact. Trends in Cognitive Science, 2003. 7: p. 166–172.
14 Franklin, S., Baars, B.J., Ramamurthy, U., and Ventura, M., The Role of Consciousness in Memory. Brains, Minds and Media, 2005. 1: p. 1–38.
15 Franklin, S., IDA: A Conscious Artifact? Journal of Consciousness Studies, 2003. 10: p. 47–66.
16 Snaider, J., McCall, R., and Franklin, S., Time Production and Representation in a Conceptual and Computational Cognitive Model. Cognitive Systems Research, in press.
17 Sun_Services, FJ-310-EE6 Developing Applications for the Java(TM) EE Platform 2010: Sun Microsystems Inc.
18 Kanerva, P., Sparse Distributed Memory. 1988, Cambridge MA: The MIT Press.
19 Maes, P., How to do the right thing. Connection Science, 1989. 1: p. 291–323.
20 McCall, R., Franklin, S., and Friedlander, D. Grounded Event-Based and Modal Representations for Objects, Relations, Beliefs, Etc. in *FLAIRS-23*. 2010. Daytona Beach, FL.