Humans aren't enough: Providing access for simulated participants to behavioral experiment software

Vladislav D. Veksler (vdv718@gmail.com)
DCS Corp, U.S. Army Research Laboratory

Norbou Buchler
U.S. Army Research Laboratory

Christian Lebiere, Don Morrison
Carnegie Mellon University

Abstract

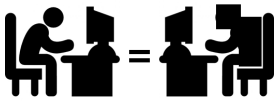
Behavioral studies often warrant the inclusion of computational participants in addition to humans. However, connecting computational cognitive or AI frameworks to GUI-based software developed for human use is extremely difficult. This results in researchers either (1) diving into software code to append an API for computational participants, (2) developing two separate versions of task code – one for human and one for computational participants, (3) cherry-picking research tasks that already include both a GUI and an API, or (4) finding a way to publish the research "as is" without the potentially useful results from running simulated participants on task. The seemingly minor nuisance of the API-GUI dichotomy in today's world of software development is, in fact, responsible for reduction in scientific progress. This work proposes a *functional-essence* approach to software development, and the use of STAP (Simple Task-Actor Protocol) as a standard UI interaction language, for overcoming the API-GUI dichotomy and enabling access to the same software for both human and computational participants. We envision the adaptation of the proposed methodology to enable selection of off-the-shelf behavioral tasks, decorative templates, and cognitive/AI frameworks for a more efficient path to research results.

Keywords: behavior research methods; simulations; cognitive modeling; synthetic users; simulated humans; standards

Introduction

Cognitive Science research often warrants the inclusion of both human and computationally simulated participants in the same experiments. Computational participants may be necessary in behavioral studies for deriving baseline performances or behavioral predictions, for model comparison, or for performance evaluation. Simulated human users may also be required in multi-user tasks and multi-player games, especially where adding human participats becomes prohibitively expenses. Similarly, in studies focused on AI development, human users may be required for establishing or validating behavioral predictions, functional Turing testing[1], performance evaluation, and to participate as teammates/adversaries/filler in multi-agent scenarios. Research efforts in fields involving human-computer interaction – decision aids, education and training aids, automation, and human-agent teaming, among others – all warrant the ability for computational agent access to human user interfaces.

Providing access for computational agents to task software that was designed for human participants is often a complicated process. The status quo is either (1) to develop two separate versions of a given task – one for human participants and another for computational ones, or (2) to develop two separate interfaces to the task – a GUI (graphical user interface) for human participants and an API (application programming interface) for computational ones. Oftentimes software development is interface-centric to such a degree that it makes more sense to go with the first of these options. However, even in the case where both GUI and API access is available for the same underlying task logic, the two interfaces often provide dissimilar experiences for human and computational users, which can be detrimental to the very purposes of providing access to both user types.



To make matters worse, enabling machine access to a task is often a one-off process. That is, there is no standard means through which a computational user may connect to human-centric software. Most task-software APIs are unique, and connecting a computational agent to ten different tasks often requires ten separate and non-trivial software development efforts. If a research effort requires connecting three different agent frameworks to these ten tasks (e.g. for model comparison), the effort is further tripled.

The seemingly minor nuisances of the API-GUI dichotomy and of dealing with varied and often idiosyncratic task interfaces has become a major limiting factor for progress in behavioral research. A given empirical study may benefit from a complementary computational simulation, but the benefits often do not merit the effort needed to try and connect a cognitive or behavioral simulation framework to the software used in the human study. In another case, a new computational model of cognitive processes may benefit from evaluation across a wide variety of task software where behavioral data are known and prior modeling results exist, but the development effort needed to parse each task interface (or to redevelop each task for the purposes of having access to their interfaces) is prohibitive. The development effort needed for

[1] Whereas the traditional Turing test is an evaluation of confusion between human and computational participants in the context of verbal communication, functional Turing testing is more general in that it examines human/computer confusability in any domain of interest.

exhaustive model comparison across a battery of behavioral tasks is almost always prohibitive, as is the effort needed to evaluate a decision aid or a cognitive training dummy or any agent software. Lowering the bar to entry for connecting computational agent frameworks to a battery of behavioral software – the same software that is employed in human studies – would enable a better quality of research.

The goal of the current work is to enable plug-and-play interconnectivity between the task software employed in human studies and varying computational agent and behavioral/cognitive modeling frameworks. We will refer to this goal as *generalized task access*.

In this paper we describe a *functional-essence* approach to task development as a necessary component for enabling generalized task access, outlining the benefits of the proposed approach for enabling symmetric human and computational user access across tasks. We describe the pluses and minuses of some existing methodologies for generalized task access. Finally we propose use of the Simple Task-Actor Protocol (STAP) as a standard API that adheres to the functional-essence approach and enables generalized task access.

A functional-essence approach to task development

Oftentimes spacing, layout, button styles, font types, sizes, colors, and other visual aspects of the graphical user interface (GUI) are tangential to the purpose of the task being performed. For example, in a simple addition task the participant may be presented with two numbers and required to type the sum of those two numbers in a textbox and click a "Submit" button. The style of the "Submit" button, the font style, background color, and spacing are all arbitrary choices from the perspective of users participating in this task. As long as all aspects are clearly visible to the participant, the functional essence of the task is preserved regardless of whether the numbers are displayed in "Times", "Tahoma", or "Courier".

There is, of course, work that may specifically focus on the differences between "Times", "Tahoma", and "Courier". However, even in the cases where font-type differences are central to the research or to task functionality, there may be many other visual specifications for the GUI (e.g. button size and color, margin size, button spacing) that are arbitrary choices from the perspective of task goals.

The problem is that there is no clear separation between task-essential affordances and arbitrary design choices in either the source code or the display interface of most software (unfortunately, the use of CSS in HTML is no exception, see *Current approaches to generalized task access* section below). Moreover, many programming paradigms force the developer to make arbitrary design choices throughout the development process. Even in the cases where design decisions aren't forced on the developer explicitly, these are implicitly warranted so as to avoid an "ugly" or "dated" look and feel, as this may give human users the suggestion that the software is poorly supported and insecure.

This state of UI development is not problematic when designing behavioral study software exclusively for human participants, but it becomes highly limiting when simulated participants are warranted. It is not realistic to expect that behavioral researchers will be developing computational models and agents that are capable of dealing with all of the visual information that human participants are exposed to. Instead, researchers either pick and choose task software that already provides an Application Programming Interface (API), or develop task software with this capability. An API provides agents with task-essential information without any arbitrary style choices, reducing the noise in the interface and enabling more robust, generic, useful computational models and agents.

For example, if the task of interest is a game of chess, the 8x8 chessboard layout is task-essential, whereas board size in pixels is not, all 'white' pieces being distinguishable via some common feature from 'black' pieces is task-essential, whereas whether that distinguishing feature is color and whether that color happens to be white is not, and so on. Thus, whereas human data collection may involve human participants interacting via a rich visual display and fancy graphics, behavioral simulation software may interact entirely over a stripped-down API comprising only figure identifications and grid locations.

The result of the GUI/API duality is that tasks created for human participants are not readily accessible to computational participants, and vice versa. In the scenario where human-accessible tasks have API's for access by computational models and agents, those API's often provide human and computational participants with different information and capabilities, sometimes hindering computational participants as compared to their human counterparts, and sometimes giving them an unfair advantage, making it impossible to run a fair human-model comparison, or to gather correct behavioral predictions.

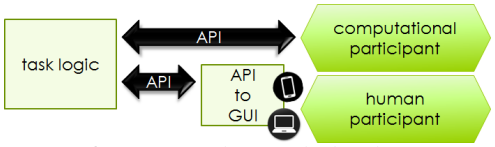


Figure 1. API-first approach to task development. This approach focuses on separating out the functional-essence of the task from the tangential design choices that can be relegated to the API-to-GUI layer.

We propose a task-development methodology that flips the GUI-first-API-second process on its head. We propose an API-first approach, where tasks are developed for interaction with machine participants, and human interaction is enabled via a separate API-to-GUI software layer (see Figure 1). The focus on a machine-readable API serves to separate out the functional-essence features of the user interface from non-essential visual design choices. The non-essential design choices may be appended for human participants via optional

templates without increasing any noise in the UI for computational participants.

It may be impossible to completely separate out functionally essential interface features from those that are decorative. For example, if the task describes a visual object, and the color of the object is important, the developer may still be required to specify object size, even if it is irrelevant to task goals. It may also be the case that developers disregard the functional essence doctrine, employing the methodology we are proposing here in unintended ways. This, however, is not a reason to abandon the goals being pursued here. More importantly, putting the functional essence approach at the core of the API and the surrounding technology will serve to push developers in the intended direction, reducing the noise in the interface and enabling more robust agent development.

The *functional essence* API-first concept is similar to a setup that may be employed in web applications (web-apps) where server-side scripts execute task logic and pass task-essential information in a standard machine-readable format (HTML) either to computational participants or to human participants. Human participants in this setup require a separate standard API-to-GUI software layer (the web browser) to interpret and interact with the HTML-formatted task interface. Additionally, CSS templates may be used to make non-essential visualization choices for human participants, but can be ignored by the computational participants.

This described web-app setup makes for a good example to ground the proposed functional-essence approach. To be clear, however, we are not arguing for HTML and GET/POST requests as the solution for enabling generalized task access. HTML focuses more on document formatting than on interactive task functionality, and requires client-side JavaScript code for many task-types. In short, existing web-app methodology is not compatible with the suggested functional-essence approach (we will discuss this in more detail in the *Current methods for generalized task access* section below).

The proverbial "baby" among the web-app "bathwater" is the use of a standard API (i.e. W3C standards) and a cross-platform API-to-GUI software layer (i.e. web browsers). The described API-first approach is only possible if the API is standardized; otherwise, the API-to-GUI layer must be reimplemented separately for each task. A standardized API provides the additional benefit of lowering the bar of entry for connecting varying computational model/agent frameworks to multiple tasks.

In sum, a functional-essence, API-first approach to task development would enable near-symmetric experience for human and computational participants. In the cases where a standard UI interaction language may be employed, a functional-essence approach could greatly reduce development costs for both GUI and computational model and agent development. This approach would provide a solution for generalized task access, and promote a library of tasks that may be used in a plug-and-play fashion for behavioral research with both human and computational participants.

On a final note, the functional-essence approach is necessary, but not sufficient for generalized task access. The other necessary component for this blue-sky vision is a standard API that is generic enough to represent a majority of potentially desired tasks, and simple enough that any research group could add an interpreter for it within their computational model or agent framework. The following sections describe some ways in which current methodologies for generalized task address, or fail to address these challenges, and propose the use of Simple Task-Actor Protocol (STAP) as the API designed specifically to address these challenges.

Current approaches to generalized task access

Screen pixels

The most universal approach for connecting computational models and agents to GUI-based software is to change nothing in task software, relying on near-future developments in computer vision for parsing pixel-by-pixel display information. There have, in fact, been many prior attempts for cross-task agent evaluation that relied on pixel-by-pixel screen reading, including a PEBL-based cognitive decathlon (Mueller, 2010) and the Atari game set used to examine Google's DeepMind AI (Mnih et al., 2015). In theory, this seems like the best and only approach for providing a symmetric experience for human and computational users. It may be even be argued that any imposition of abstraction in the form of an API could prevent the kind of deep synergy between perception, cognition, and action that might be a key enabler of the robustness of human behavior.

In practice, however, there are two major problems with this approach. First, computer vision does not promise to be good enough in the near future to enable behavioral researchers to connect computational participants to GUI-centric software in a plug-and-play fashion. At best, we may see rapid advances in computer vision research that enable screen-scraping libraries for popular computer languages. However, the integration and customization of computer vision libraries with the models/agents developed for behavior research will require non-trivial expertise and development efforts.

Second, as we have mentioned in the previous section, the pixel-by-pixel approach makes no distinction between task-essential and tangential visual information. The lack of such a distinction will lead researchers to custom-build computational models and agents for the specific idiosyncrasies of a given interface. The overly task-specific nature of such models and agents will lead to more arduous development, less generic and less useful results, and brittle simulations that may fail when faced with UI perturbations as slight as font changes.

HTML and web-app technology

Web application technology is the golden standard for machine-readable cross-platform interface interaction specification. Without the need for scraping screen pixels, a com-

putational user can view the same documents as a human user. These documents are translated into visual format for human users via standard API-to-GUI software (web browsers). Computational agents are able to parse such documents directly, as these are in machine-readable format.

The web-app technology comprises multiple standards, including HTML (hyper-text markup language) for specifying document format and interactive elements, CSS (cascading style sheets) for specifying additional custom style choices, and JavaScript for specifying additional custom interaction and content. The more recent version of these standards is often referred to as HTML5.[2]

In fact, HTML5 enables much of what we want to accomplish in this paper in providing universal access to human and machine task participants. First, it is a well-documented, standard, cross-platform, machine-readable format. Second, style-sheets (CSS) allow developers to completely separate visualization choices from actual task logic. Finally, a major strength of HTML5 is that it enables just about any visualization and event capturing that a task might require. Moreover, as human experiment participation over the web has become more prevalent, more and more behavioral research experiment software is already being written in HTML5 format.

An obvious question becomes – why are there no computational modeling and agent frameworks that are able to connect to and interact with off-the-shelf web-apps? The answer to this question has to do in part with the complexity of HTML5, and in part with the fact that this technology does not aid much with the separation of signal and noise in the interface.

Unfortunately for our purposes, HTML5 is based on HTML, which, at its root, is not a task-interface standard, but rather a document-markup standard. That is, HTML is all about structure and references, in the service of formatting. The addition of JavaScript brought client-side logic, interactions, and often task-irrelevant visualization choices. As the web matured to allow in-browser applications, CSS was added to separate out visualization choices from task logic, but actual web applications often employ HTML, CSS, JavaScript, and separate server-side code interchangeably to accomplish various task-essential functionality and decorative visualizations.

There are myriads of ways to indicate the same information and functionality in HTML5. For example, the <**select multiple**> tag gives the participant multiple boolean choices. The same thing is often accomplished via a series of <**input type**=checkbox> tags. These two control types are *functional synonyms* – they only differ in how they appear, not in their function. Whether a developer decides on one method over another is a matter of taste, rather than a matter of user affordances.

HTML5 contains many functional synonyms. To make matters worse, oftentimes web development involves the addition of JavaScript code to provide the same functionality that may be provided via an existing tag. For example, the above is often accomplished via a series of <**button**> tags with "onclick" attributes that employ JavaScript to change the style of the <**button**> when it's clicked. Oftentimes the class of the button is changed rather than its style attribute, or a new class is appended to the list of classes, depending on what the CSS specifications allow. Oftentimes, it is not a <**button**> tag, but an <**input**> or <**div**> or <**span**> tag that is employed for the same purpose. Sometimes each user choice is immediately sent to the server where task logic resides, sometimes the user has to click the Submit button for the information to be sent, and sometimes no information is sent at all because some or all of task logic is embedded in HTML5. To interact with this set of boolean choices, and to comprehend that it is a decision-point in the first place, a computational participant must be able to account for a virtually infinite set of such programming choices, and to consider other surrounding HTML tags (e.g. <**form**>, <**div**>).

This overabundance of task-development methodologies makes it very difficult to develop computational cognitive agents/models that will interact meaningfully across different web-tasks. To put it plainly, developing a model/agent that can interact meaningfully with off-the-shelf HTML5 web-apps would be a Herculean effort.

Another non-trivial problem is that HTML is bulky (as compared to protocols such as JSON), and thus it is not a preferable format for real-time task-to-user display updates. The GET/POST requests used to send data from user back to the task in modern web-apps are not efficient for real-time task interactions either. This greatly limits the use of HTML5 for dynamic simulations and massive faster-than-real-time parameter searches.

Despite these shortcomings, HTML5 web-app development provides a strong foundation for separation of functional affordances from tangential visual design choices (e.g. CSS), using a machine-readable API to interface between task and users (i.e. HTML), and using widely available API-to-GUI software (i.e. the web browser) to enable human interactions with said API.

Generic game playing protocols

There have been several efforts to create standard API's for generic game playing (GGP; Thielscher, 2010). For example, GDL (game description language; Thielscher, 2010) describes which moves are legal, which objects an agent 'sees', and what the goals are, but the translation from these descriptions to a graphical display suited for human users seems difficult to achieve. The GDL website includes a visualization (API-to-GUI layer) for this protocol, though it is not clear how simple or general it is (http://gamemaster.stanford.edu).

[2]We employ the term HTML5 to signify the combination of HTML, JavaScript, and CSS because this bundle has become a standard in practical use, but the W3C HTML5 specifications (https://www.w3.org/TR/html5/) are actually semi-agnostic about scripting and style languages. The type attribute of the <**script**> tag defaults to "text/javascript", and the type of the <**style**> tag defaults to "text/css", but JavaScript and CSS are not actually a part of official W3C HTML specifications.

Another API, VGDL (video game description language; Schaul, 2013) describes a top-down description of a game grid. VGDL API-to-GUI layer is simple and generic, as it is based on Pygame (a popular graphics and interactions library for the Python programming language), and the description language itself lends itself to direct translation to a graphical display.

GGP protocols were all developed with the purpose of computational agent interaction, and thus they all subscribe to the functional-essence task development methodology. The major problem with all GGP protocols is that these focus entirely on game-play, and do not consider non-game tasks that may be of importance to behavioral researchers. For example, there is no way that a grid-description language like VGDL can describe a generic form display with buttons, checkboxes, selectors, and text inputs.

Moreover, these protocols lack a standard means for specifying stylistic choices that is often desired by task developers. Functionally-tangential stylistic choices may be viewed in theory as excessive frills. In practice, however, design choices turn out to be extremely important in conveying standard interaction functionality to human participants (e.g. clickable links are usually blue and underlined), and in providing users and experiment participants with confidence that the task software is up to date, bug-free, and secure. In this sense, the HTML/CSS generic approach and flexibility provides something that GGP protocols tend to lack.

Summary of current limitations to generalized task access

In the sections above we discuss the limitations of some current approaches to generalized task access for human and machine agents. We highlight pixel-by-pixel screen-scraping, HTML5 web apps, and Generic Game Play protocols, as these are among the most successful and well-known approaches in the domain. There exist many other current and past approaches for enabling human users and various computational agent interactions with task software. These include real-world embodiment (i.e. robotics), virtual-world embodiment (e.g. Minecraft, Second Life, Unity 3D, Unreal engine, Gazebo, Visual Robotics Toolkit, OpenAI Gym), and OS-specific primitive interaction capture routines (e.g., Neth, Patton, Banas, Schoelles, & Gray, 2008).

Almost all of these approaches are overly generic to the point that they suffer from the same noisy-display problem as pixel-scraping. Task-essential information is not distinct from task-tangential visual noise, which hinders model and agent development. In the cases where task-essential information may be siphoned via an API, there are often issues of symmetry between computational and human user experiences (e.g. virtual world API's provide different information to human users than they do to computational agents).

There are also many approaches to generalizing task access that are tied to some specific behavioral simulation software framework. That is, the task development process is somewhat altered, such that the GUI for human use includes hooks that translate it for computational users, as long as those computational users are built on top of a specific AI or cognitive theory and framework. The reason such methodology is not widely adopted is that tying exploratory research a priori to a specific theory and system is, at best, inefficient, and, at worst, it is bad science.

Of the methods discussed, HTML5 and GGP protocols seem to hold the most promise for generalized task access. As we mention above, HTML5 is overly generic in its ability to describe display and interactions, whereas GGP protocols are not generic enough. However, HTML5 provides a good example for side-loading task-tangential style information via CSS and a widespread API-to-GUI software layer (i.e. the web browser), and the API-first focus of GGP protocols enables the reduction of noise in display. Optimally, to achieve generalized task access we would like to marry some of the elements of GGP protocol methodology with those of web-app technology.

Simple Task-Actor Protocol (STAP)

The Simple Task-Actor Protocol (STAP) is a standard format for serializing and communicating task-essential user-interface (UI) changes and user interactions. STAP messages are consistent regardless of whether the user is a human or a computational participant. STAP message format is agnostic of the operating system and programming paradigm employed on either the task-software side or the user/agent - software side. The aim of STAP is to address *generalized task access*. This section provides a brief introduction to STAP and some examples of STAP UI interactions (based on the latest major release, STAP 7.0). The full specifications for STAP, along with task and agent software sample code, may be found in the STAP and stapjs github repositories [see the links at vdv7.github.io/stap].

Much like GGP protocols, STAP aims at the API-first approach, where messages specifying UI changes contain only task-essential information. Unlike GGP protocols, and much like HTML, STAP enables the specification of a separate style-sheet for task-tangential GUI decoration. Also in keeping with HTML, and unlike GGP protocols, STAP enables specification of standard software UI elements (e.g. buttons, text inputs, vector graphics). Unlike HTML, STAP UI options are minimal, avoiding functional synonyms where possible, and enabling a clear separation between task logic and UI functionality.

STAP is meant to maximize the symmetry between human and computational user experiences, but also to enable experiences that are unique to these two groups of users – faster than real-time or slower than real-time simulations for computational models and agents, and aesthetic design choices for human visualization. STAP is meant to be a suitable interface language for both symbolic and graphics-intensive tasks, allowing for consistent interpretation for both task types. Finally, STAP is meant to be simple, consistent, and easily parsable by all modern programming languages, such that

STAP interpretation modules may be added easily, piecemeal to existing AI and computational cognitive modeling frameworks.

STAP messages adhere to a widely employed standard called JSON (JavaScript Object Notation; json.org). JSON is a simple data format that can contain hierarchies and sequences of values. It is likely that all computational model and agent frameworks will be able to interact with task software via JSON-compliant messages, since almost every modern programming language has core libraries that can serialize to and deserialize from JSON strings.

As an example, our recent work involved connecting three popular cognitive frameworks to STAP-compliant tasks – ACT-R (Anderson, 2007; Anderson & Lebiere, 1998), Soar (Laird, 2012), and PyIBL (Gonzalez & Dutt, 2011; Morrison & Gonzalez, 2016) examples. These frameworks are developed in LISP, JAVA, and Python, respectively (moreover, the models written for these systems are often executed on all three major PC operating systems). All three of these programming languages include libraries for serializing and deserializing JSON messages into native types, making it very easy to recognize and parse STAP messages.

For human users it is possible to customize any look and feel features (that do not change the functional-essence of the task) via optional templates. In the case of the API-to-GUI software layer in the stapjs github repository [github.com/vdv7/stapjs], the optional look and feel may be specified via standard CSS. For example, Figure 2 shows a sample GUI generated via the same STAP messages (adding "Hello, World!" text and a container titled "Click a button" with two buttons), but rendered via three different templates.

Figure 2. Same text and buttons UI rendered via three different templates.

Summary

This paper proposes the *functional-essence* approach to software development as a necessary component for enabling symmetric task access for human and computational participants to the same applications. At the core of this approach is the separation of functional affordances necessary for the software to run in intended ways and arbitrary decorative choices. We argue that the decorative choices sprinkled throughout the software may serve an important purpose for human participants, but actually detract from the ability to connect computational participants to the same software.

Additionally, we suggest the use of the Simple Task-Actor Protocol (STAP) as a standard UI language that promotes the functional-essence methodology and enables generalized task access. Much like web-app technology (i.e. HTML5), STAP is machine-readable, platform-independent, and enables specification of any GUI elements and functionality needed for behavioral study software. Unlike web-app technology, and more akin to generic game-playing protocols, STAP is an API-first approach that focuses entirely on describing functionally-relevant UI properties (though enabling specification of decorative choices via separate style templates).

We envision that the use of STAP and the functional-essence approach to experimental software development in behavioral sciences will enable a faster path to more rigorous scientific research. Specifically, we envision a library of STAP-compliant task software and a library of commonly used decorative templates for running behavioral experiments with human users. Furthermore, we envision that major computational cognitive architectures and AI frameworks will include modules for interacting with STAP UI primitives, enabling behavioral scientists to quickly and easily develop models and agents to interact with the same task software that human participants (and other models) are able to interact with. Future work includes the development of such modules for ACT-R, Soar, and other major behavioral/cognitive modeling frameworks, the development of a battery of STAP-compliant behavioral experiments that may be employed to assess human and computational participants alike, and further improvements to the stapjs API-to-GUI software layer and templates.

References

Anderson, J. R. (2007). *How can the human mind exist in the physical universe?* Oxford University Press.

Anderson, J. R., & Lebiere, C. (1998). *The atomic components of thought*. Mahwah, NJ: Lawrence Erlbaum Associates Publishers.

Gonzalez, C., & Dutt, V. (2011, oct). Instance-based learning: integrating sampling and repeated decisions from experience. *Psychological review*, *118*(4), 523–51. doi: 10.1037/a0024558

Laird, J. E. (2012). *The Soar cognitive architecture*. MIT Press.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., … Others (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.

Morrison, D., & Gonzalez, C. (2016). *http://pyibl.ddmlab.com.* Retrieved 2016-01-01, from `http://pyibl.ddmlab.com/`

Mueller, S. T. (2010). A partial implementation of the BICA cognitive decathlon using the Psychology Experiment Building Language (PEBL). *International Journal of Machine Consciousness*, *2*(02), 273–288.

Neth, H., Patton, E. W., Banas, S., Schoelles, M. J., & Gray, W. D. (2008). Integrated Semantic and Visual Aspects of Online Information Search. In *30th annual meeting of the cognitive science society2*. Austin, TX.

Schaul, T. (2013). A video game description language for model-based or interactive learning. In *Ieee conference on computatonal intelligence and games, cig.* doi: 10.1109/CIG.2013.6633610

Thielscher, M. (2010). A general game description language for incomplete information games. In *Proceedings of the twenty-fourth aaai conference on artificial intelligence (aaai-10)* (pp. 994–999).