DASH User Guide 6/2016

This document provides a short overview to creating and running DASH[[1]](#footnote-1) agents. DASH agents are intended to simulate human behavior in a variety of situations in DETER and other environments where group decision-making is mediated by computers. For example, they have been used to model observed behavior in:

* Responding to phishing email
* Downloading and using security software such as Tor
* Making decisions to control a power plant.

In situations such as these, human behavior might differ from the optimal, as may be defined according to decision-theoretic measures or accepted best practice, and these differences impact the behavior of systems under test. This may happen because the typical user of a system has an incorrect or incomplete model of the system’s behavior or of its security, or because humans inevitably make mistakes, particularly if their attention is taken with other tasks.

DASH agents model this behavior using a dual-process cognitive architecture.

* **Rational Behavior:** This module contains sub-modules for reactive planning and for projection using mental models.
* **Instinctive Behavior**: A second module models instinctive behavior and other reasoning that humans are typically not aware of

The combination of these two modules can account for the effects of cognitive load, time pressure or fatigue on human performance that have been documented in many different domains. The combination can also duplicate some well-known human biases in reasoning, for example the confirmation bias. The DASH platform includes support for teams of agents that communicate with each other and a GUI to control agent parameters and view the state of both modules as the agent executes.

**This guide**:

* First it goes through an example of a DASH agent and:
  + Shows how to run an agent
  + Explains what the agent does
  + Explains how to examine its state and modify its behavior
* The guide then moves into more detail on the representation used for the rational and instinctive modules and shows how to program new agents that combine these behaviors.

For comments or questions about DASH and to obtain a copy for research purposes, please contact Jim Blythe at [blythe@isi.edu](mailto:blythe@isi.edu), or 310-448-8251.

1. Installation 3

2. An example mail reading agent 3

3. The rational Module and top-level agent behavior- *modify* 5

4. Goal-driven reactive behavior – *change?* 7

4.1 programming goal-driven agents 8

5. inter-agent communication 8

6. the instinctive module 10

How this works – change? 10

# 1. Installation

DASH agents are written in python, so first you will need to download python. In order to do so:

* Go to <https://www.python.org/downloads/>
* Choose python2 and Mac, Linux, Windows, or another version compatible with your computer

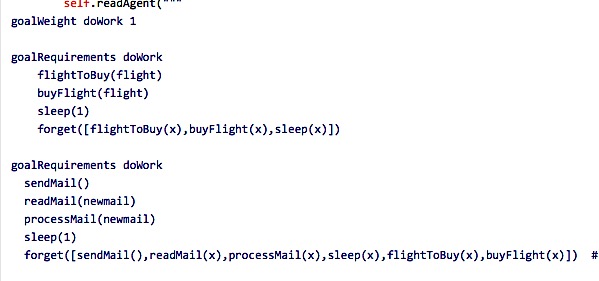
In order to get the code, go to GitHub:

* https://github.com/cuts/webdash/
* If you do not already have an account, you will need to create one

To run the code:

* Go to a command line interface like linux or terminal.
* Once you pull in the git repository, move it to the directory that contains DASH
* Type: python ‘name of file’ and it should start to run
* You may also choose to use Pycharm or another application which runs python if you choose to do so.

# 2. An example mail reading agent

We introduce DASH and illustrate some of its capabilities using an agent that wants to buy an airline ticket. In our first version, this agent both reads and sends mail. When processing mail, the agent "reads" the message and discovers whether or not a flight is requested. If so, it has a choice of 4 destinations (New York, London, Shanghai, and Paris), which are randomly chosen with each recurrence of the code.

In this case, the agent has 2 sets of goals (seen in the figure to the right):

* Buying a flight, if there is one available
* Reading, sending, and processing mail

After each run through the code, the agent forgets what has already been done, so that it will continue to check whether there are flights available and buy a different one. Below is an example of the code running on terminal. In this example, a flight is not currently available, so the agent skips to the next goal, which is to send mail, after which the agent is able to buy flight tickets:



**Sleeps and forgets everything so it can repeat the process.**

**🡨**

**So, begins to send mail**

**🡨**

**Chooses random destination:**

**New York . 🡪**

**Adding known false: signals no flights are available. 🡪**

*Jim to look over and update if necessary*

The mail reading agent makes use of several modules of DASH which are introduced and explained in more detail in subsequent sections. These include the rational module, the mental models library within the rational module, the instinctive module and inter-agent communication.

At the heart of the DASH architecture is a dual-process model, consisting of the rational and instinctive modules. In this approach, both modules may be engaged in deciding which action to choose, and may collaborate or compete. In normal operation, the instinctive module produces a suggested action, put in working memory, that the rational module usually accepts. This reflects human behavior since, most of the time, humans are not thinking deeply about the actions they decide but are following habit. Sometimes a surprising observation or request breaks the agent out of habitual action. In DASH, this can happen in two ways:

* First, the instinctive system may have low confidence in its chosen action – in this example, an email might look suspicious – in which case it will signal to the rational module that it should provide a second opinion.
* Second, the rational module might be triggered to override the suggestion of the instinctive module for a number of reasons, and will begin more detailed processing even though the confidence of the instinctive module’s selection has not reduced.

In general, the balance between the two modules in terms of the final action decision can be changed in a number of ways, including physiological changes that may lead the rational system to be more active, or, for example, changes in cognitive load caused by other tasks – high cognitive load may suppress the rational module.

In the rest of this guide, we describe the instinctive model and the rational module in more detail so the reader can understand how the agent’s knowledge is represented and can modify it.

* Section 3 covers the instinctive module
* Section 4 gives an overview of the rational module.
* Section 5 covers mental models and how they are used for decision-making.
* Section 6 covers goal-driven behavior in the rational module, which can follow workflows while responding to changes in the environment if needed, even when they may lead to changes in the workflow or its being dropped entirely.
* Section 7 provides some more examples of agents and tips and tricks for providing required behavior.

# 3. The Rational Module and top-level agent behavior – *change?*

The rational module deals with decision-making that humans perform consciously and deliberatively, including planning and assessing alternative actions. In the next sections we describe support in DASH for goal-driven behavior, including following workflows, and we describe support for mental models in the following section. Here we focus on how the rational and instinctive modules combine to make decisions about actions and the top-level agent API.

Figure 3 below shows the overall architecture of the agent. On each cycle, the agent takes in new inputs and chooses an action to perform, or it may choose not to perform any action.

* The instinctive module first applies rules to the new information and updates the items in working memory with new items that have high activation.
* The rational module then picks a recommended action.

It may choose a suggested action placed in working memory by the instinctive module, or it may decide on further deliberation. By default, this is determined by an activation strength threshold: if the suggested action with highest activation is too low, deliberation will occur.

When the rational module engages in deliberation, it enters a planning process automatically, reasoning based on its working memory about its most important goals and about actions that can achieve them. As part of this process, it may reason about the results of alternative actions according to its own mental model, which may be different from the agent’s actual environment or from that of the instinctive module. The framework to support this reasoning is described in the next two sections, but here we note that the framework or the rational process can easily be customized by providing general python clauses to describe action selection, goal elaboration, or mental models.

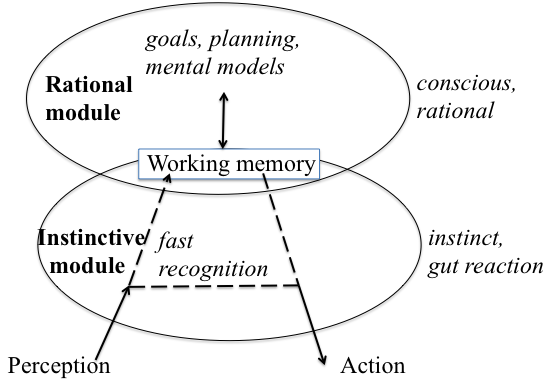


Figure 3. The instinctive module and rational module combine to make action decisions, and share information that is posted in working memory by either module.

# 4. Goal-driven reactive behavior

Much of the time, human online behavior is influenced by *workflows*, for example when performing group tasks within an organization such as purchasing equipment or performing individual tasks such as taking pictures at an event, uploading and distributing them. While following workflows, we are aware of events in our environment that might cause us to change our plans.

To support this behavior, DASH provides a framework of goals and sub-goals that are used to achieve them, with an elaboration process that bottoms out into primitive actions that may be chosen as the next agent action. The agent can perform coherent goal-driven activity by maintaining a top-level goal over time and working towards it. However the framework effectively re-computes the goal and actions on each decision cycle, giving the agent a chance to respond to changes that might require a change in plans and to combine its plans with other less plan-driven activity such as taking a break.

## 4.1 Programming goal-driven agents – *update for straight python programming*.

*Will work through the email-reader in more detail showing how to build up a goal tree and associate it with code.*

### Underlying Python definition

Within python, a series of goals and sub-goals, written as primitive actions, are used to establish the goal of the agent as a whole, in this case a mail reading agent. Below is an example of these goals:

goalWeight doWork 1

goalRequirements doWork

flightToBuy(flight)

buyFlight(flight)

sleep(1)

forget([flightToBuy(x), buyFlight(x), sleep(x)])

goalRequirements doWork

readMail(newmail)

processMail(newmail)

sleep(1)

forget([readMail(x), processMail(x), sleep(x), flightToBuy(x)])

transient doWork # Agent will forget goal’s achievement or failure as soon

as it happens

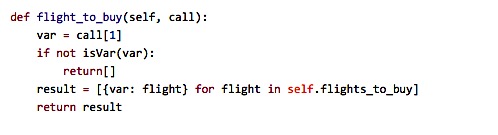
The goal, doWork, has a goal weight of 1. In this case, there are two ways to achieve that goal. Either the agent can buy a flight or send mail. First, the agent checks if there is a flight available, if there is not it will switch to sending mail in order to complete its goal requirements.

Alternatively, goals can also be implemented by other goal requirements clauses, or, if there is not a goal requirement, then the goal activity must be associated with python code. In order to associate the code, you must use primitive actions call to establish the code that will be run for each of the primitive actions.

Here is an example in which these primitive action methods need to return binding lists for the variables that show the way in which an action can succeed. Otherwise, it will return an empty list.

**Here, no flights are available, so an empty list is returned**

**🡨**



**Here, there is a flight, so a result is produced**

**🡪**

# 5. Inter-Agent Communication

The *mailReader* agent in the examples above, that responds to emailed requests to buy flight tickets, is designed to respond to email messages sent to it by the corresponding *mailSender* agent, which wants to buy plane tickets. The messages are passed through a communications hub that accepts connections from several agents, routes communications between them and implements a simple shared state consisting of variables that all agents can query and set. This section describes the process and API for agent communication and shared state, using the *mailReader* and *mailSender* agents as an example.

In order for agents to communicate, an instance of the hub program must be started before the agent programs. In our example, this is done by starting the mail\_hub.py program from the Tutorial directory in the repository before running both *mailReader* and *mailSender*.

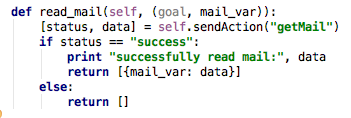
Each agent contains an instruction to register with the hub as part of its initialization routine. For example, mailReader03.py contains the line:

self.register([‘mailagent@amail.com’])

This causes the agent to look for the communications hub when it starts up, on the default port and host (localhost). A hub can easily be started by running the world\_hub.py python program from the DASH repository, but in this case the generic hub has been specialized to provide a mail service, which is why the ‘register’ command accepts an email address. This is done by overriding the method ‘processRegisterRequest’ on the hub subclass – for details see mail\_hub.py.

On success, the ‘register’ method binds the ‘id’ instance variable in the agent to a value that is unique for the running instance of the hub.

Once registered with the hub, primitive actions may use the agent method ‘send\_action’ to inform the hub of an action being taken in the world and get the result. The default world hub does not define any actions, but the mail hub subclass responds to the ‘getMail’ and ‘sendMail’ actions. ‘getMail’ takes no arguments and returns a list of mail that has been sent to the agent, while ‘sendMail’ takes a list of mail messages as its argument and returns success or failure with no other information. These are implemented by overriding the ‘processSendActionRequest’ method as can be seen in mail\_hub.py. The first argument to ‘sendAction’ is the name of the action as a string and the second is any auxiliary data that will be processed with the action. For example, the primitive action that implements ‘readMail’ in the mailReader03 is shown below: it simply calls sendAction(‘getMail’) and binds a goal variable to the results:



# 6. The Instinctive Module - *modify*

The dual-process model of human behavior includes two distinct systems of reasoning: one that makes fast, instinctive decisions based on its perception of the world and one that performs slower, more conscious and deliberative reasoning. According to this theory, humans are generally aware only of the rational system, while the instinctive system is constantly suggesting decisions and is more frequently involved in our outward behavior. In the psychological literature, these systems are often called respectively “system 1” and “system 2”, in order to reduce any prejudice from their naming as to which is more likely to offer correct decisions or has general control [Stanovich & West 00]. Here, we will refer to them as the “instinctive” module and “rational” module for ease of reference.

In DASH, the instinctive module is represented with a set of statements about the world that have activation levels, and a set of if-then rules that act to change the activation level of statements on the right-hand side of the rule based on the levels of statements on the left-hand side. When these rules are chained together, the result is a form of spreading activation [Anderson 00].

Here, for example, are some of the rules in use in the mailReader agent:

**if** url(ID,Url) and short(Url)

**then** ok(followLink(Url,ID)) **at** 0.4

**if** doNotReply(ID) **then** ok(followLink(Url,ID)) **at** -0.6.

**if** url(ID,\_) **then** ok(followLink(Url,ID)) **at** -0.5.

Each rule begins with “if” and has three pieces, separated by the words “then” and “at”. The first piece, before “then” is the rule precondition, which uses variables and logical connectives to specify a pattern that might match many facts in the instinctive module’s memory. For example, the first rule will match any Url that is short found in an email message, binding the variable ID to the email message and Url to the Url. The definition of “short” is given elsewhere. For matches to this pattern, the rule changes the activation strength of its consequent, in this case ok(followLink(ID)), specified between the words then and at in the rule. Semantically, this can be interpreted as a suggestion that the agent follow the link.

The final number after the word at, 0.4 in this case, is its activation modifier. All facts in the instinctive modules knowledge base have an activation strength, where a strength of 0 implies a neutral attitude and increasingly positive or negative numbers imply an increasingly positive of negative attitude, respectively. When a rule is applied, it increments the activity level of its consequent by the product of its activation modifier and the activation strength of its precondition. When the precondition is an atomic fact, this is the activation strength of the fact. The activation strength of a conjunction is the minimum of the strengths of its components, and the activation strength of a disjunct is the maximum of its components. The facts with the highest absolute value of activation strength are placed in the working memory buffer where the rational module can access and act on them.

Figure 1 shows how the three rules shown above contribute to the strength of the fact ok(followLink(1,1)), where the first argument to followLink represents the first url seen in the email message, and the second argument is the Id of the message.

## How this works – change?

When the agent is run within the python program, this program initializes a connection to the communication hub server after querying the agent for its id. The python controller sends a message to the communications hub containing the action. The communications hub implements a hash table of variables and values and a list of messages for each agent, and updates these structures based on the messages. After asserting an action’s result and before querying for the next action, the python controller queries the communication hub for any messages for the agent. These are asserted directly into the agent’s database and removed from the message list in the communication server.

*Think and talk about how state update is done from the hub (not just the result of actions explicitly performed).*

1. Deter Agents Simulating Humans [↑](#footnote-ref-1)