DASH User Guide

This document provides a short overview to creating and running DASH[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 or 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. One module within the system models rational behavior, containing sub-modules for reactive planning and for projection using mental models. 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 begins by going through an example of a DASH agent, showing how to run an agent, cause it to start executing by a remote command, 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, or 310-448-8251.

1. Installation

The cognitive module of a DASH agent is implemented in prolog while the graphics and agent communications are implemented in Java. If you are using DASH on a

[1] Deter Agents Simulating Humans

Deter node, it is already installed and you can run the examples developed in the rest of this guide using the shell scripts that are indicated.

In order to install and run DASH on your own computer, you will need to install SWI Prolog and provide its library path to Java. Go to http://www.swi-prolog.org/download/stable to install the appropriate version of SWI Prolog for your platform. On a mac, this will normally be installed into /opt/local/.

Download and uncompress the DASH zip file dash.zip. In the directory this creates, you will need to edit the file 'bot.sh' ('bot.bat' in windows) to point to the installed location of your SWI prolog library. Open the file and change the line

SWILIB=/opt/local/lib/swipl-5.10.4/lib

to point to your installation library directory, which will contain jpl.jar and a platform-specific subdirectory. Change the second line to name this directory, e.g.

SWIOS=i386-darwin10.7.0

At this point, you should be able to run DASH agents. The rest of the user guide will use a few pre-defined agents as examples.

A more detailed description for windows is forthcoming.

2. An example mail reading agent

We introduce DASH and illustrate some of its capabilities using an agent that reads a set of mail messages and decides whether to follow the web links in the messages, some of which may be malicious. Experiments in this domain typically involve multiple agents, including at least one receiver and sender, so that all stakeholders can be modeled in addition to network effects. However, the mail reader can be run alone with a set of "pre-delivered" mail and without the communications machinery, and we will use this mode initially.

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, and the instinctive module. It does not typically use the reactive planning capabilities in DASH since they are not normally associated with reading mail, but we will show how to extend the agent to include this

This section will briefly introduce the basic agent cycle and a simple DASH user interface (the DASHBoard). To begin running the agent, run the shell command "mailReader.sh" after installing DASH according to the instructions in the previous

section. This script file starts the agent and brings up the user interface with the command-line argument "-gfx". It is possible to start the agent without the interface by omitting this argument from the file.

The initial DASHBoard for the interface will look like this:

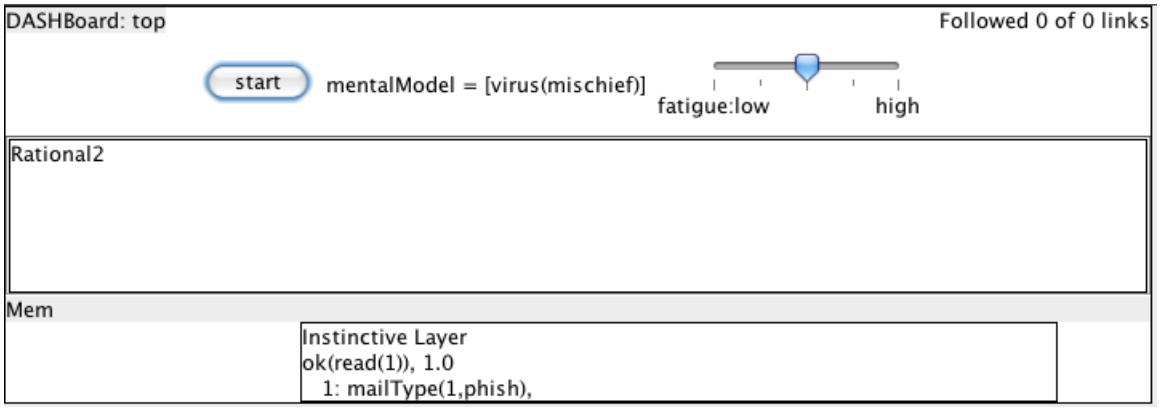


Figure 1. The initial DASHBoard view of the mailReader agent

The top left of the DASHBoard shows the agent's name and the top right contains a short status message programmed by the agent designer, in this case showing the number of web links followed out of the total number of email messages seen. Below this the "start" button will send a message to the agent to begin its cycle. The other two controls in the upper white area describe the current mental model in use by the agent and show the fatigue level, which the user can change through the slider. The author of the agent can control which variables are shown to the experimenter in this way.

The lower three rows in the window can show the working state of the two modules, labeled "Rational" for the rational module and "Instinctive" for the instinctive module, as well as the working memory ("WM") that is produced by the instinctive module and may be accessed and modified by the rational module. The use of these areas will be explained in more detail in the following sections.

At this point, click on the "start" button. This begins the agent cycle, in which the agent selects an action, performs it in the world, observes the results and again selects an action, repeating the cycle. In this case, the mailReader agent chooses to look at its first email while it still has unread emails in its list. If it is currently looking at an email that contains a URL, it decides whether to follow the url.

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 and compete. In normal operation, the instinctive module produces a suggested action, put in working memory, which the rational module usually accepts. This models the fact that, 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.

Figure 2 shows the result of the mailReader agent's decision-making about an email. In this case the rational module was engaged because the instinctive module has a low confidence in its decision when compare with the activation level of the rational module. In turn, the rational module consults its mental model and projects, or plays forward, the likely consequences of following the link according to the model. After doing this, it decides not to follow the link. The information shown in the DASHBoard for the instinctive and rational modules will be explained in more detail in their respective sections below, but for now we provide a brief summary.

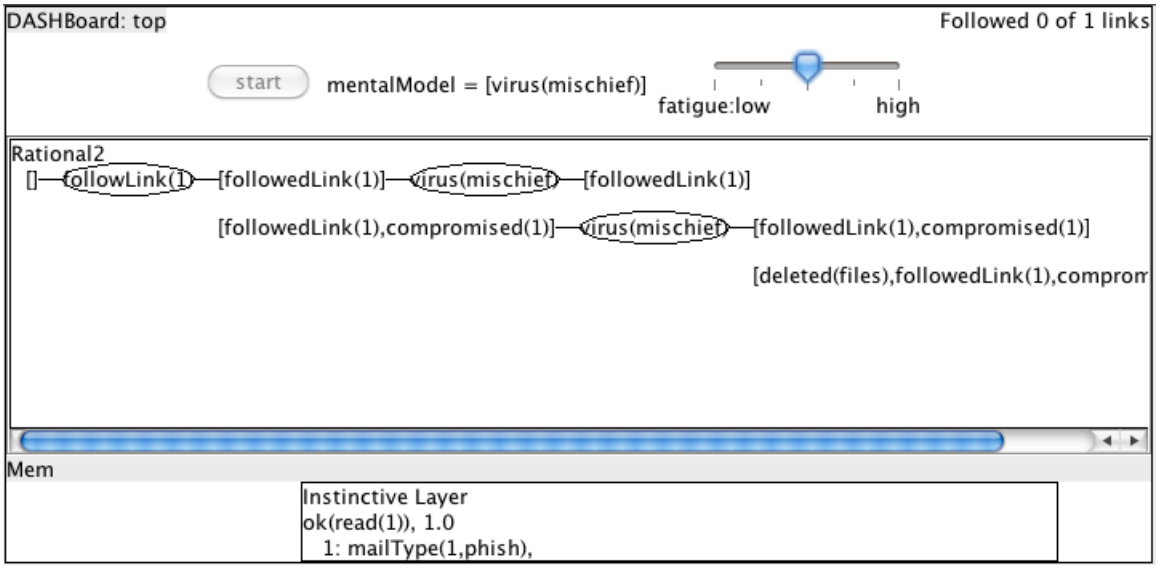


Figure 2. The mailReader DASHBoard after deciding, via the rational module, not to follow a URL.

The logical term in working memory, "ok(read(1))", was added by the instinctive module based on a simple rule and is interpreted by the agent as an impulsion to read the email message with id 1. Before the action is chosen and returned to the agent body, the rational agent may review it and decide to avoid it. However this is not typical, reflecting human behavior, which is instinctive most of the time.

The remaining lines in the instinctive module's window show that the module computed an activation strength of 0.4 for the action to follow the link in that email. The activation strength is the sum of a number of values coming from different forward-chaining rules, and the indented lines show the information that was used by those rules. Each of these rules is based on observed behavior reported in the literature. First, the strange form of the url, with a ".ua" domain that otherwise looks like a the flickr domain, led to a negative association with the action. Second, the fact that the url was short was seen as a positive and almost remove the negative association from the url. Finally the email contained the word "friend", which was viewed as positive. However the total strength of 0.4 was not high enough for the term to be placed in working memory as a suggested action.

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. The next section covers the instinctive module, and 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 Instinctive Module

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.

4. The Rational Module and top-level agent behavior

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 prolog clauses to describe action selection, goal elaboration or mental models.

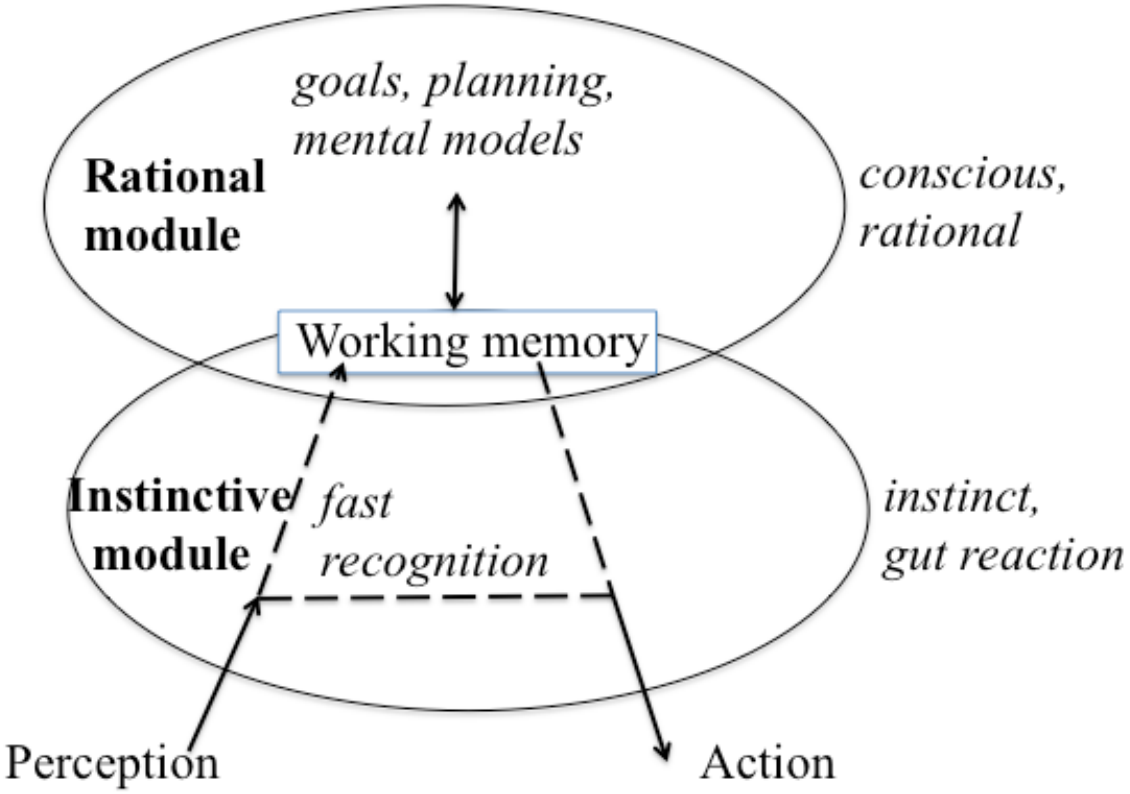


Figure 3. The instinctive module and rational module combine to make action decisions, and share information that is posted in working memory by either module.

4.1 Agent API

DASH agents are designed to be run from java, although they can also run directly in prolog. When an agent is run from the shell script as described in section 2, a java loop is executed that creates a prolog sub-process in which the agent rational and instinctive modules run. The java-based main loop repeatedly queries the prolog agent for its next chosen action, executes it through the 'perform' method, informs the agent of the result of the action and then again queries it for a new action. Behavior such as web access or operating other code can be attached to the actions by creating a new class with a new 'perform' method and registering it with the agent.

5. 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.

5.1 Programming goal-driven agents through the wizard interface

In this section we illustrate how a DASH agent can form and follow a plan when needed to achieve its goal, but can break out of the plan if the goal is achieved unexpectedly or pre-empted and return to it later from where it left off. The goal framework is represented as a prolog file but is simple to create using the Wizard, as we show here. On start-up, the initial wizard screen contains space for a list of goals as shown in Figure 4. To run the wizard as shown here, run the shell command "wizard.sh" after installing DASH according to the instructions in section 1.

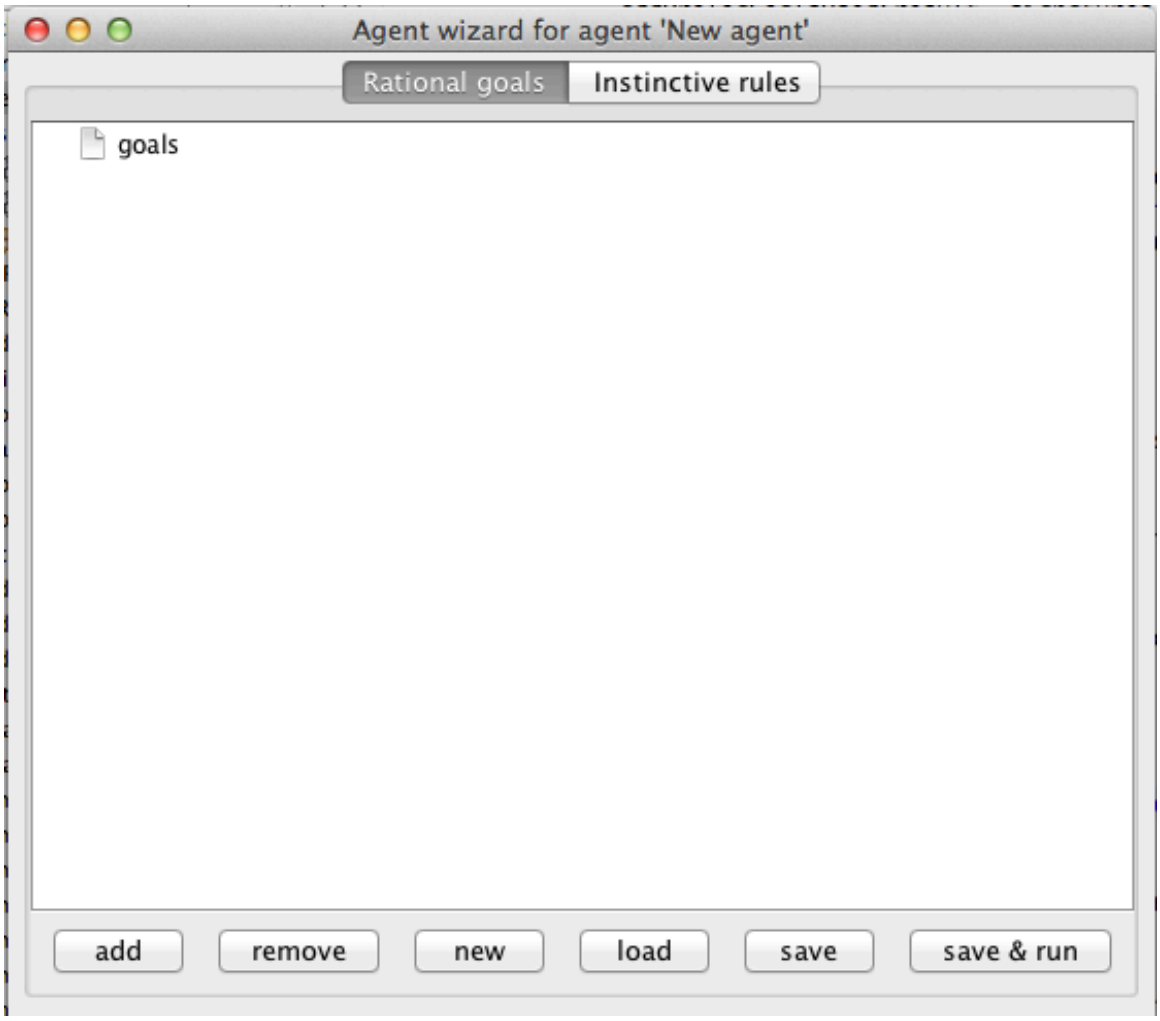
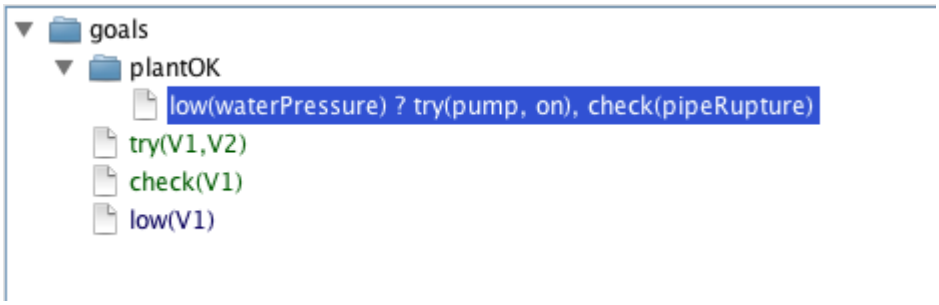


Figure 4. The initial wizard screen, with no information about goals.

We illustrate an agent that controls a power plant and that has a long-term high-level goal to maintain the health of the plant. Under normal operating conditions this may require no action, or rote behavior suggested by the instinctive module. When the sensors indicate a potential problem, however, the agent may need to draw from workflows, perhaps in manuals or from training, that perform some further tests and then prescribe a course of action that should bring the plant back to normal operating conditions.

We represent the long-term behavior with the explicit goal 'plantOK'. To create a goal using the Wizard, select the 'goals' node and click 'add'. The new goal has a default name that can be edited, leading to this view:

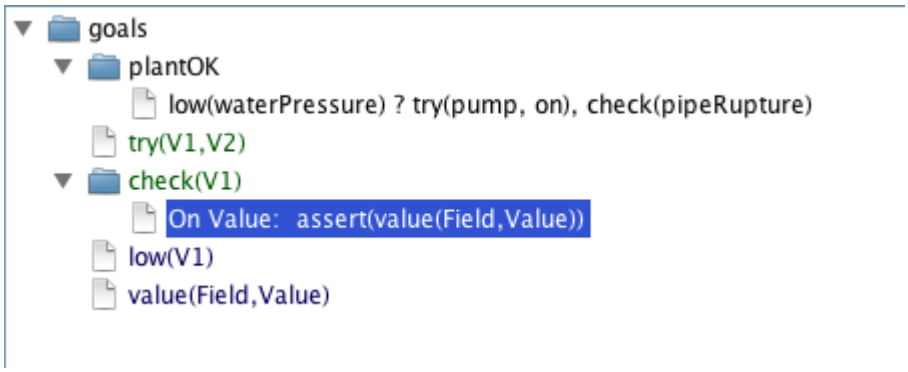
Next, we add ways to achieve the 'plantOK' goal when it is not true. If there is low water pressure, for example, the standard approach is to check the pump is on, and turn it on if it is not, then check for a pipe rupture if this has not solved the problem. The general approach of checking if an object is in the correct state and re-setting it if not occurs frequently, so we will encapsulate it in a new subgoal called 'try(Object,Value)'. Note that capitalized words denote variables in Prolog and also in DASH. To add the rule, select the 'plantOK' node, click 'add', and edit the node to describe the actions as conditions as shown in the next screenshot:



The wizard adds three new goals in response to the new decomposition rule. The first two, in green, are the sub-goals involved in the decomposition, 'try(Object,Value)' and 'check(Object)'. The wizard has chosen variable names, which can be edited. These sub-goals are in green because they currently have no decompositions into lower-level goals, so the DASH agent will treat them as 'primitive' goals and send them as suggested actions to the agent executive. This is the desired behavior for 'check', which directs the agent to check a value on the console for the plant. We will add decomposition rules for the 'try' goals, at which point the wizard will see that the goal is not primitive and change its color to black.

The last goal added by the wizard is in blue, indicating it is a pattern for a fact in the agent's knowledge base rather than an action to achieve. The wizard infers this from the use of 'low(waterPressure)' as a condition for the decomposition rule.

When the agent sends an action like 'check(pipeRupture)' to the executive, this triggers an action in the agent's world that will have a result. In general the action might be to scan a port or move the agent's body, and users can attach arbitrary code to the executive in order to implement different agents. This code will send a result value back to the DASH agent on completion, and we must describe how the agent should update its beliefs based on this result. For the 'check(X)' action, the result is the observed value from a sensor that should be stored in the agent's knowledge base to reason about future goals. This is achieved with an update rule attached to the action, which can be added through the wizard by selecting a primitive node and choosing 'add update rule', leading to the view below:

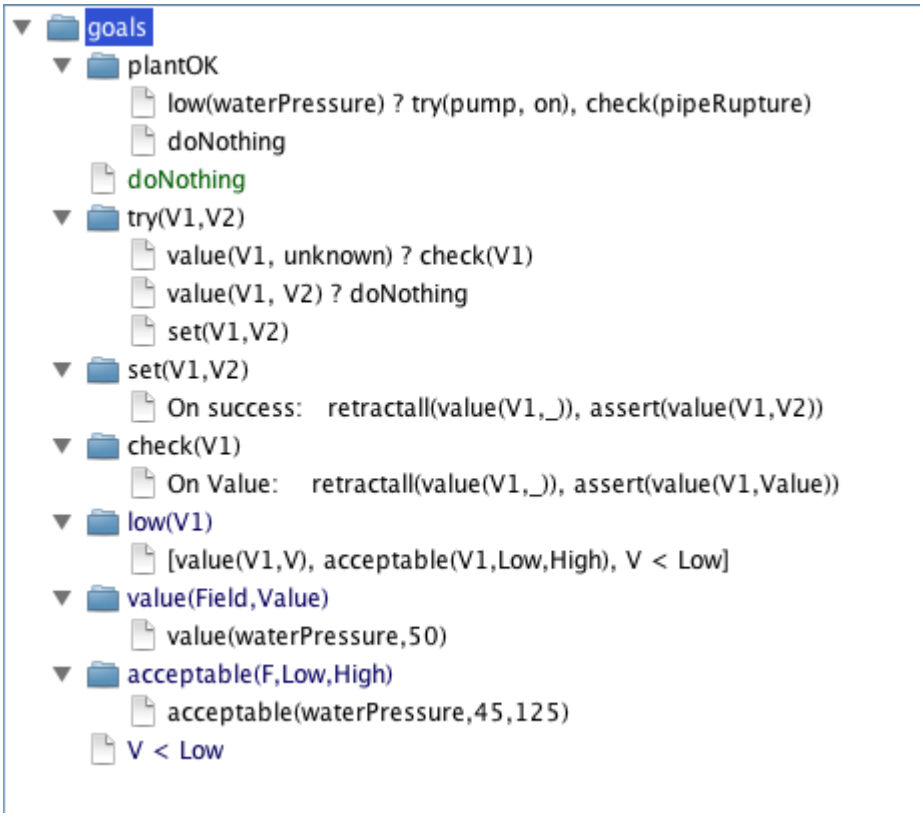


'assertValue' is a built-in command that sets a field to have the given value. The wizard adds a new node for the fact group 'value(Field,Value)' that it infers from the update rule and removes any other value for the field from its knowledge base.

The wizard can be used to add individual facts to the fact group nodes, e.g. 'value(waterPressure,30)' and rules for inferring facts, e.g.

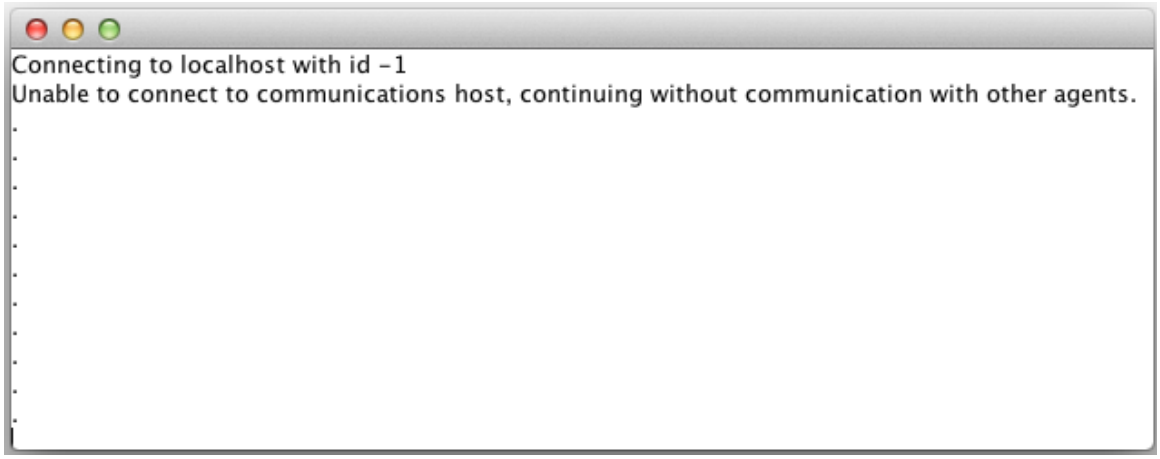
low(Field) if value(Field,Value), acceptable(Field,Low,High), Value < Low.

In the view below, the wizard has been used to add this rule and a definition for the 'try' subgoal. The wizard automatically added a node for the 'acceptable' fact group, which we used to add an acceptable range for waterPressure of 45 to 125.



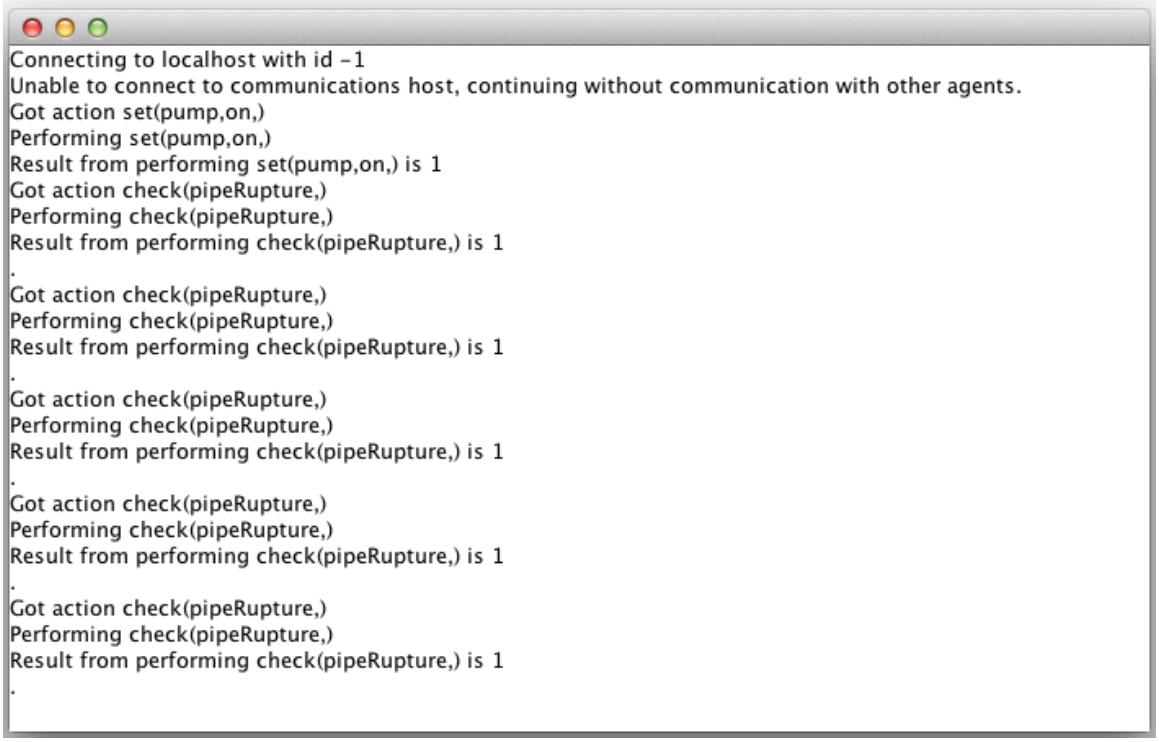
Running the agent from within the wizard

The agent definition just given through the Wizard is fully functional, and can be run using the 'save & run' button in the bottom row. When this is selected, a new window appears with the results of running the agent for 10 steps (by default) and then quitting:



The output indicates that the agent is not connected to a communications server and so is running standalone. The dots below the text indicate a series of 'doNothing' actions, since the agent believes the plant has acceptable water pressure.

If the waterPressure is set to 30 in the 'value' node, the agent's plan is triggered for below normal pressure, and the new window looks like this:



This time, the agent first turns on the pump, and then (since the problem is not solve) continually checks for a pipe rupture. Note that at present there is no implementation for the 'check' and 'set' actions in the agent, that would normally connect it to a simulated or real plant. In normal operation, turning on the pump might increase the water pressure so that the agent would complete its plan. Otherwise, this simple definition can be modified to stop re-checking for a pipe rupture.

Underlying Prolog definition

The wizard generates a prolog file that encodes the goals and prolog rules. This can be edited by users, for example to modify or update prolog clauses. The file generated by the wizard shown above is as follows:

```
% -*- Mode: Prolog -*-
% Agent file created automatically by the wizard.
:-style_check(-singleton).
:-style_check(-discontiguous).

goalRequirements(plantOK,[try(pump, on), check(pipeRupture)]) :-
low(waterPressure).
goalRequirements(try(V1,V2),[check(V1)]) :- value(V1, unknown).
goalRequirements(try(V1,V2),[doNothing]) :- value(V1, V2).
goalRequirements(try(V1,V2),[set(V1,V2)]).
goalRequirements(plantOK,[doNothing]).

goal(plantOK).
```

```
goalWeight(plantOK, 1).
primitiveAction(set(_,_)).
primitiveAction(doNothing).
primitiveAction(check(_)).
subGoal(try(_,_)).

updateBeliefs(set(V1,V2),1) :-   retractall(value(V1,_)),
assert(value(V1,V2)).
updateBeliefs(check(V1),Value) :-    retractall(value(V1,_)),
assert(value(V1,Value)).
% Generic update rule
updateBeliefs(_,_).

:-dynamic(low/1).
low(V1) :- value(V1,V), acceptable(V1,Low,High), V < Low.
:-dynamic(value/2).
value(waterPressure,30).
:-dynamic(acceptable/3).
acceptable(waterPressure,45,125).
:-dynamic(</2).
id(1).
```

Each goal is declared in the agent file, along with a goal weight that allows the agent to choose between alternatives. In the agent's prolog-based representation the syntax is as follows:

```
goal(plantOk).  % Agent's top-level goal
goalWeight(plantOk,1).
```

The weight might depend on other features, in which case the goalWeight might be implemented with a set of prolog clauses.

We might wish to arrange that the instinctive module provides the right action in normal operation, but a workflow is required in other cases. The 'goalRequirements' clause is used to specify a set of sub-goals or actions that can be used in a workflow to achieve some goal:

```
goalRequirements(plantOk,
                 [try(pump, on), check(pipeRupture)])
   :- low(waterPressure).
```

Here, the two-action plan will be chosen when the water pressure is known to be low. The first step tries the feedwater pump and the second step will check for a pipe rupture if this fails. The '`try`' sub-goal is a generic goal that looks up the appropriate sensors and setters and sets the value if it is not already correct. As well as having its own goalRequirements clause, it is declared as a subgoal with the clause:

```
subGoal(try(_,_)).
```

(In prolog syntax, the underscore refers to a variable whose name is not important since it is not used in the clause.)

The '`check`' action is not a subgoal but a primitive action that can be returned to the java body. This must be declared with a `primitiveAction` clause:

```
primitiveAction(check(_)).
```

With this information in place the agent is able to create and follow a very simple workflow. Refer to the file nuclearController.pl in the DASH distribution for the full agent, defining the `try` subgoal.

As an example of the use of reactive planning, we note that the plan to achieve `plantOk` has two sub-plans that appear to be executed in sequence. However, if the feedwater pump sub-plan succeeds, the check for a pipe rupture is unnecessary. This is taken care of, though, because when the sub-plan succeeds, the water pressure returns to normal operation and the `goalRequirements` clause is no longer active. If the agent does not pick an action from the instinctive module, it will have other goals or sub-plans, essentially abandoning the longer plan for water pressure in reaction to the changed environment.

6. Inter-Agent Communication

The mailReader agent, that responds to an email message in the examples above, is designed to respond to email messages sent to it by the corresponding mailSender agent. 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 briefly describes the process and API for agent communication and shared state.

Although it is possible to run the prolog files defining an agent directly in a prolog environment, inter-agent communication relies on two java programs, one that controls each agent and manages primitive action choice in the agent's BDI perception-action cycle, and one that runs a communications hub to which the agents connect. In order for agents to communicate, the hub program must be started before the agent programs. This is started with the command 'comms.sh' in the top-level directory of the DASH release. The DASHBoard runs agents with their java controller and it is also run by default using the 'bot' command-line script found in the top directory of the DASH release, e.g. "./bot.sh mailSender". When the agent starts, it will look for the communications hub on a fixed port, and register with it

using a unique id number. This number is the value of the 'id(X)' query within the prolog part of the agent. In both mailReader.pl and mailSender.pl, the id number is set to be a constant.

Once registered with the hub, three primitive actions can be used by the agents to communicate and to query and set shared variables. Examples can be seen in the mailSender and mailReader agents:

callPerson(ID,Term) [used in mailSender] will send a message consisting of the prolog term bound the Term to the agent registered with id ID. Both variables must be bound when this primitive action is chosen.

commSet(Var,Value) [shown as an example in mailSender] will set the value of a variable in the shared state implemented in the hub program. Var must be bound to a constant and Value must be bound when this primitive action is chosen, e.g. 'commSet(message,1)'.

commGet(Var,Value) [shown as an example in mailReader] will query the value of a variable in the shared state implemented in the hub program and bind Value to its value. Var must be bound to a constant when this primitive action is chosen. Value can be unbound and will be bound to the value after the action is completed.

How this works

When the agent is run within the java controlling program, this program initializes a connection to the communication hub server after querying the agent for its id. The java program then repeatedly queries the prolog program for its next primitive action with the query do(A), handles performing the action and then informs the agent of the result by asserting result(A,R) into the prolog database of the agent. If the primitive action is recognized as a send, set or get action, the java 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 java controller queries the communication hub for any messages for the agent. These are asserted directly into the agent's prolog database and removed from the message list in the communication server.