The Design of a Tool for Teaching Hierarchical Control for Robot Navigation

by

Hao Ren

BCom / BSc,

The University of Auckland, New Zealand, 2005

AN ESSAY SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University Of British Columbia

(Vancouver)

December 2008

Abstract

The main goal of this project is to introduce a tool to teach students about hierarchical robotic control. The applet is designed to let students of Artificial Intelligence write controllers in Java and experiment with the behavior of different controllers with the help of a programming environment which includes a debugger and example controllers. One of the example controllers makes use of value iteration, and the controller is intelligent enough to avoid traps. The design of the interpreter used to process Java is discussed in detail, such as why the interpreter BeanShell is selected, and how to use BeanShell. Finally, software design for the teaching applet is presented for future developers who will finalize the applet for the use of AISpace.org.

Table of Contents

List of Figures

Acknowledgments

I would like to thank my supervisor David Poole for helping me to complete the essay; I appreciate the support from my parents and my friends Lloyd Markle, Ali Akhavan, Marisol Flores Garrido, Massih Khorvash, Sancho McCann, Hoyt Koepke, Cara Koepke, Dan Ray and most importantly God.

Chapter 1

Introduction

With "Good Old Fashioned Artificial Intelligence and Robotics" (GOFAIR), an intelligent agent typically features one level of controller that interacts with the body (Figure 1.1). Mackworth et al. [6] attempts to reason that a robot controller can be more effectively viewed as hierarchical (Figure Figure 1.2), where lower layers are typically "reactive and synchronous on continuous state spaces" and upper layers are "deliberative and asynchronous in symbolic, discrete spaces".

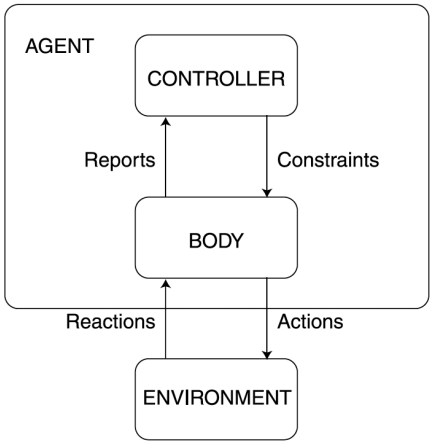


Figure 1.1: Single layer of controller for GOFAIR (Mackworth et al. [6])

The main goal of this project is to introduce a tool to teach students about hierarchical robotic control. The tool is designed to be easy to use by providing an

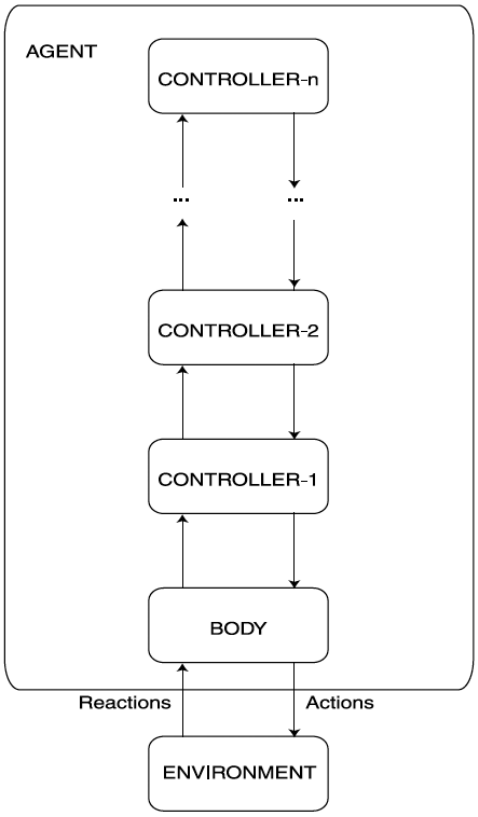


Figure 1.2: Hierarchical agent controller (Mackworth et al. [6])

environment where movements of the robot can be tracked and replayed, and all the inputs and outputs of the different levels of the controller can be monitored by a user of the applet (Section 3.2.5). We also want to emphasize the importance of hierarchical control as seen in examples shown in Chapter 2, and how easy it is to replace/add one layer of the controller to add functionality.

Work described here focuses on a simulated environment that consists of walls, locations (to be used by robot as goals), and a robot moving in constant motion (Figure 1.3). A prototype applet is already written as can be seen in Figure 2.2 on page 6, and we attempt to improve on the prototype. A detailed explanation on the background of the prototype "robot navigation" applet can be found in Chapter 2 of Poole & Mackworth's text book [8], and a general discussion of the applets for

AISpace can be found in Knoll et al. [5].

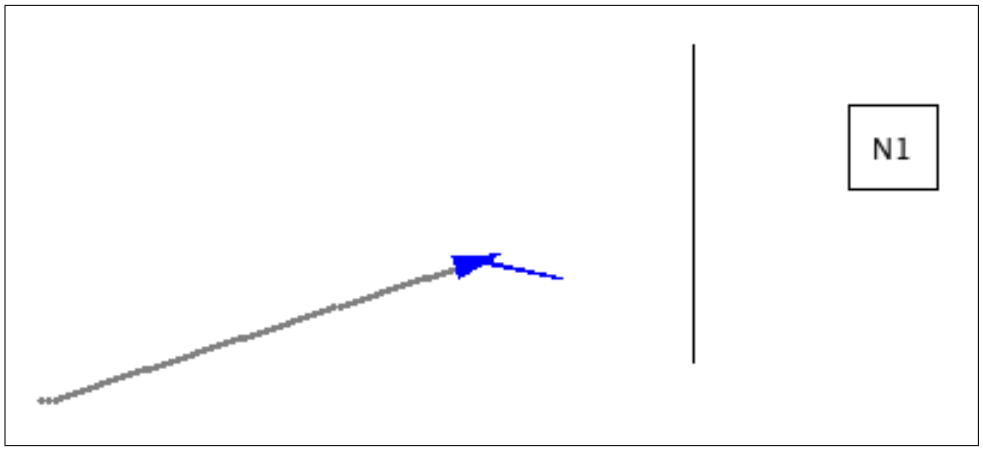


Figure 1.3: An example of the simulated environment containing a wall, a location (N1) used by the robot as goal location, and a moving robot

Chapter 2

Sample Controllers

From a user's perspective, one needs to design a controller and then use the robot navigation applet to run the controller to see how it works. In this section, several such controllers are introduced as examples, all of which has already been implemented and tested.

For the purpose of letting users to easily understand the power of hierarchical control, we intend to introduce a simple controller algorithm called the basic controller (Section 2.1). This basic controller is based on the controller of the prototype applet[1]. Then, we will introduce a more complicated controller based on value iteration which is a Dijistra-like algorithm that solves the shortest path problem. This value iteration based controller adds one additional layer to the basic controller, so it demonstrates how to modify an existing hierarchical controller by reusing and/or modifying existing layers .

2.1 Basic Controller

This first controller is called the basic controller (Figure 2.3) because all the robot tries to do is to go towards the nearest not yet reached goal, and whenever the robot bumps into a wall, it just turns left. This works well for some combination of walls and goals (see Figure 2.1), but the robot gets trapped by specially placed configurations of walls' positions (see Figure 2.2).

[1] http://aispace.org/robot/

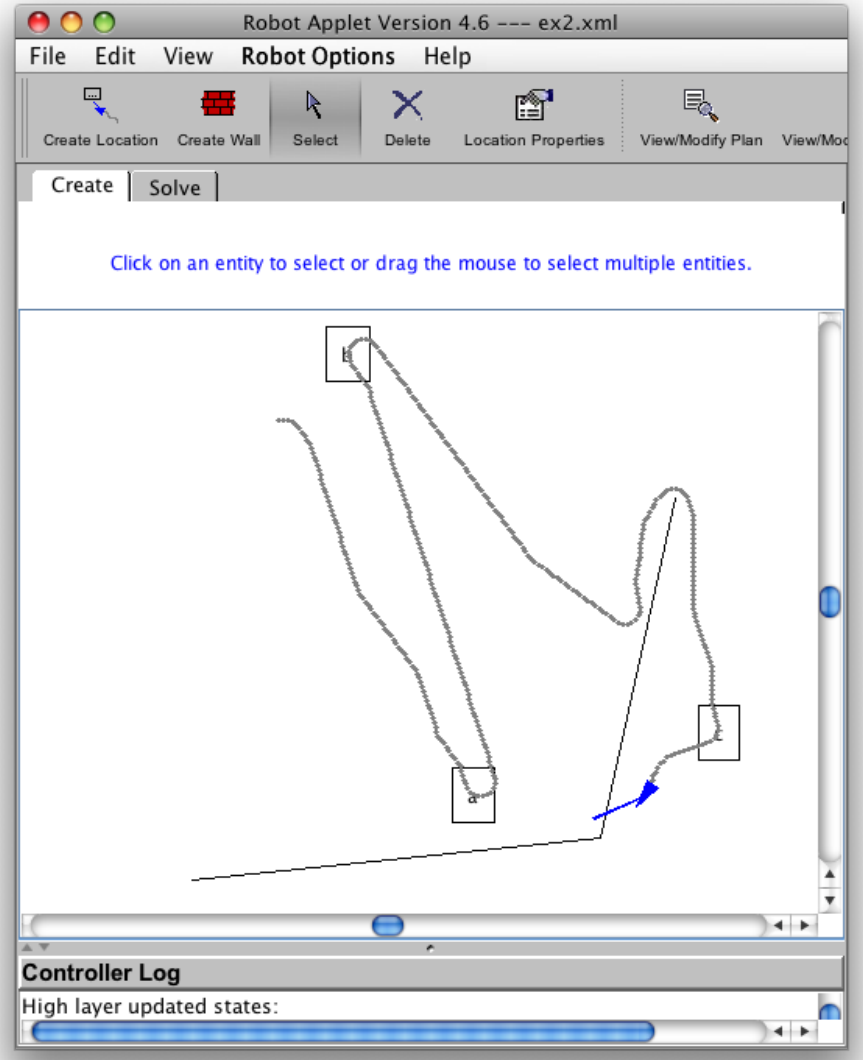


Figure 2.1: Robot successfully reaching all goals controlled by basic controller

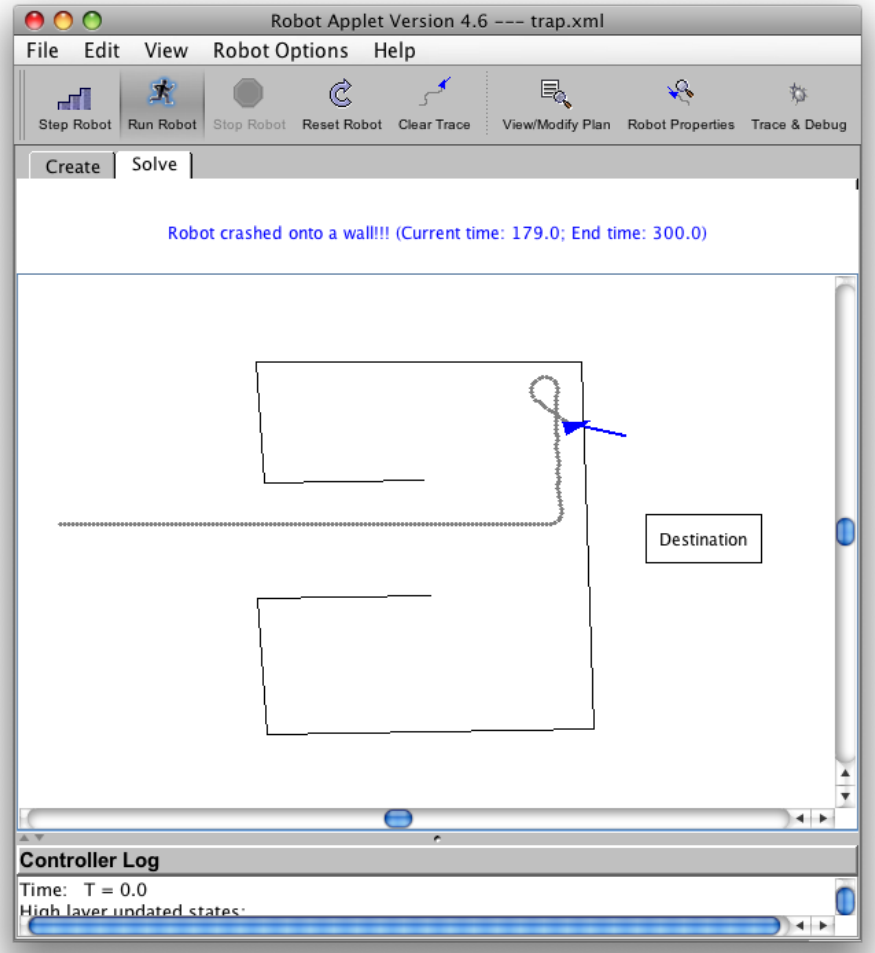


Figure 2.2: Trapped robot in prototype applet.
This prototype applet snapshot shows that the Robot is trapped when controlled by basic controller. Even without the crash, the robot will loop at the corner in the top right.

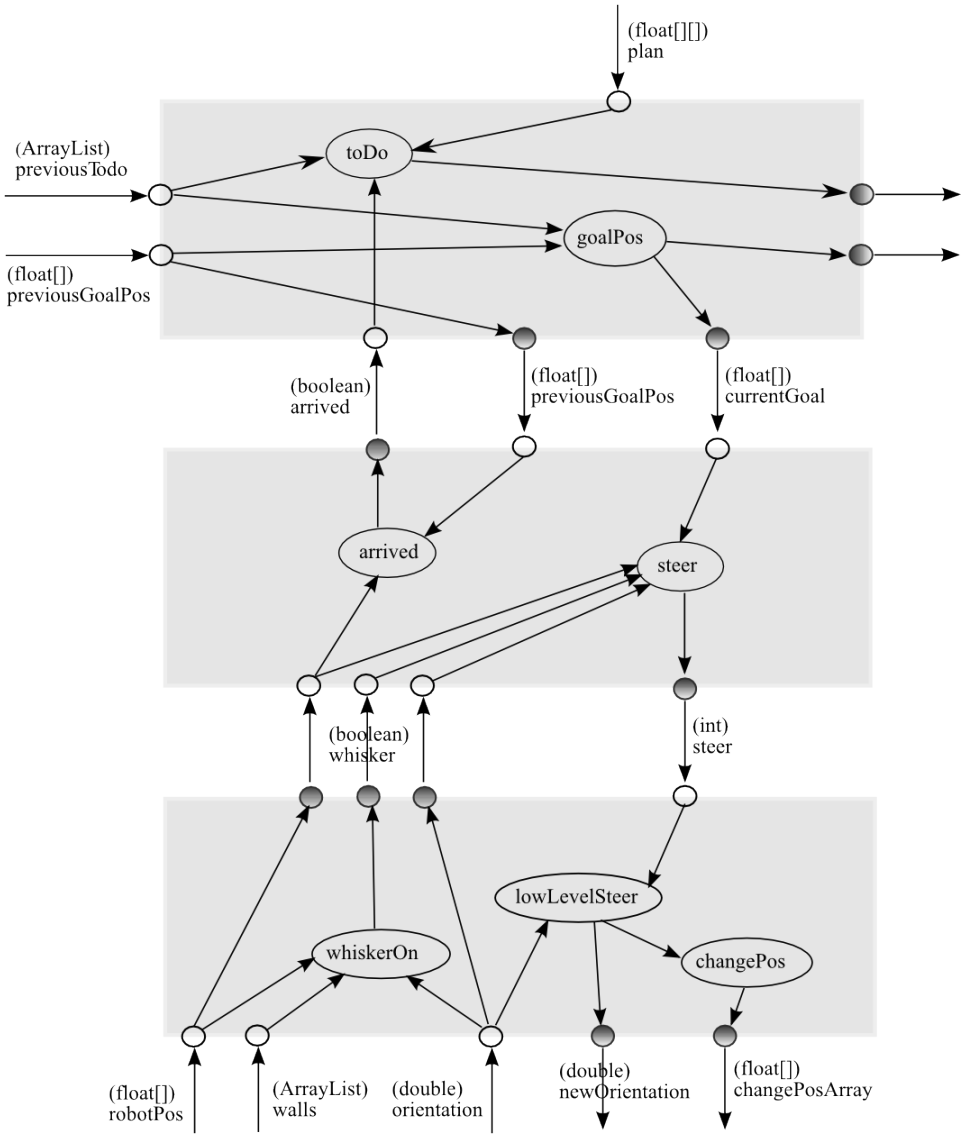


Figure 2.3: Basic controller

2.1.1 Description of Layers

The controller is split into three layers:

Top layer This layer decides the next goal for the robot to reach.

Middle layer Given a goal and current location, this layer decides what *rough* direction to turn the robot (in terms of left, straight or right).

Bottom layer This layer detects if the whisker of the robot is in touch with wall, and gives the *exact* new orientation and change in robot position.

For more information about this controller, readers are encouraged to read chapter 2 of Poole & Mackworth's textbook [8].

2.2 Using Value Iteration

We will introduce a new controller in this section to demonstrate how to modify and reuse an existing controller to create more intelligent behaviors. Differences between the controllers in Figure 2.4 demonstrate the parts that are modified/reused from the basic controller to create the value iteration controller.

For the more complicated value iteration (VI) controller, VI is integrated into the controller at higher leveled layers (Figure 2.5, page 11). VI can be used to find an optimal policy, and the policy tells us what is the optimal action to take at any given state. Readers interested to read further about VI and reinforcement learning are suggested to read the journal paper by Kaelbling et al. [4], the textbook by Ghallab et al. [3], and the original Dynamic Programming book by Bellman [1].

We use VI as a shortest path algorithm, and any other algorithm such as Dijistra's algorithm [2] can be used to replace VI.

The lower two layers from the basic controller (Figure 2.3) are extensively re–used with slight modifications for this controller, as only the higher layers make use of VI.

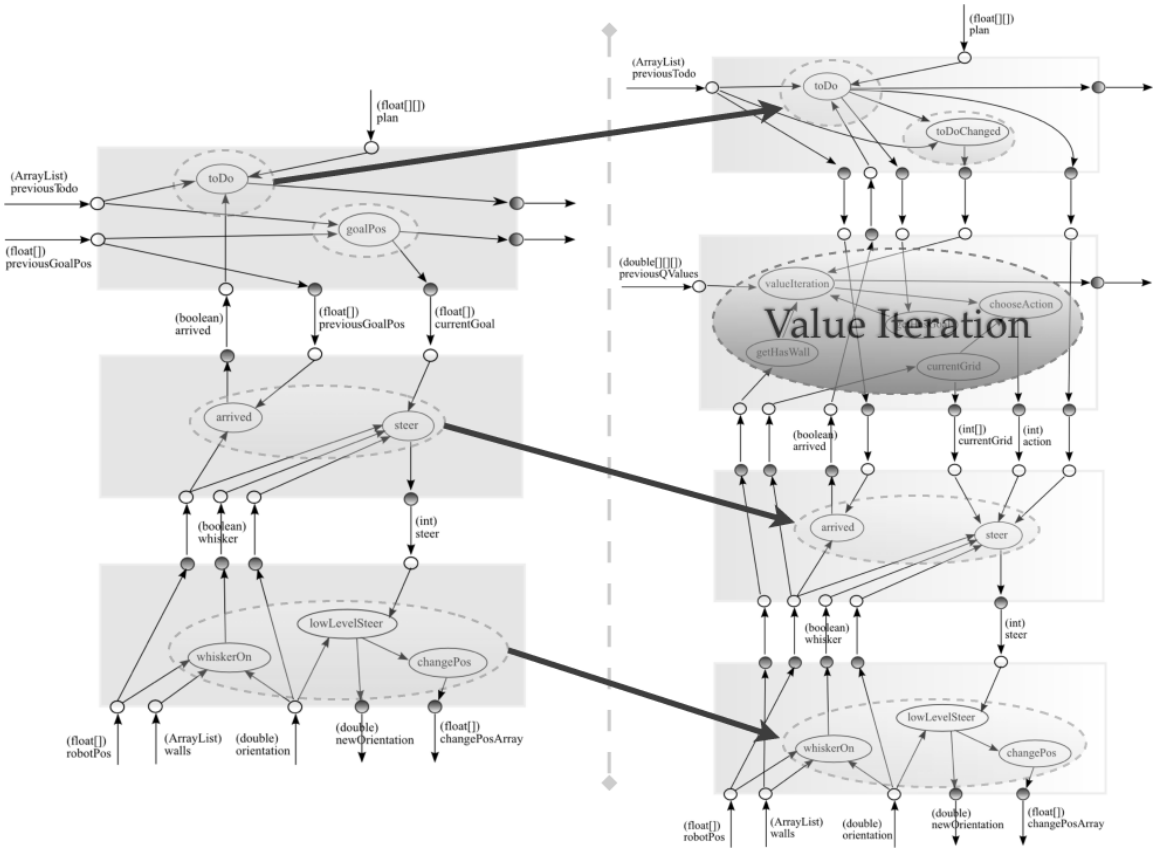


Figure 2.4: Differences between the basic controller and the value iteration controller

The controller on the left is the basic controller, and the one on the right is the VI controller. The three bold arrows point toward the functions that are reused and modified from the basic controller. In the right side image, the second layer from the top is a completely newly added layer.

2.2.1 Markov Decision Process

The controller's design is based on Fully Observable Markov Decision Processes (FOMDPs) with finite states and actions for an infinite time horizon. For more background introduction on a similar system, please refer to Ren et al. [9]. The system here is deterministic and the outcomes of all actions are fixed.

Since each action has an associated cost, and only goals have rewards, we will use a non-standard Markov Decision Processes (MDP) consisting of the following elements:

- a set of *states* S (Section 2.2.2);
- a set of *goal states* $\Gamma \subseteq S$ (Section 2.2.2);
- a set of *actions* A (Section 2.2.3);
- *cost function* $c : S \times A \times S' \rightarrow \mathbb{R}$ (Section 2.2.3), cost of performing an action;
- *reward function* $R : \Gamma \rightarrow \mathbb{R}$ (Section 2.2.4), reward of reaching a goal state;
- *state transition function* $S \times A \rightarrow S$.

Value iteration is used to find the optimal policy (i.e. the optimal action to take for each state).

Each action has a cost, and the only rewards are at goal locations, so the optimal policy obtained from the greedy algorithm (value iteration) will result in a shortest path to the goal states.

The following sub-sections explain how the elements for the MDP are instantiated in our example domain.

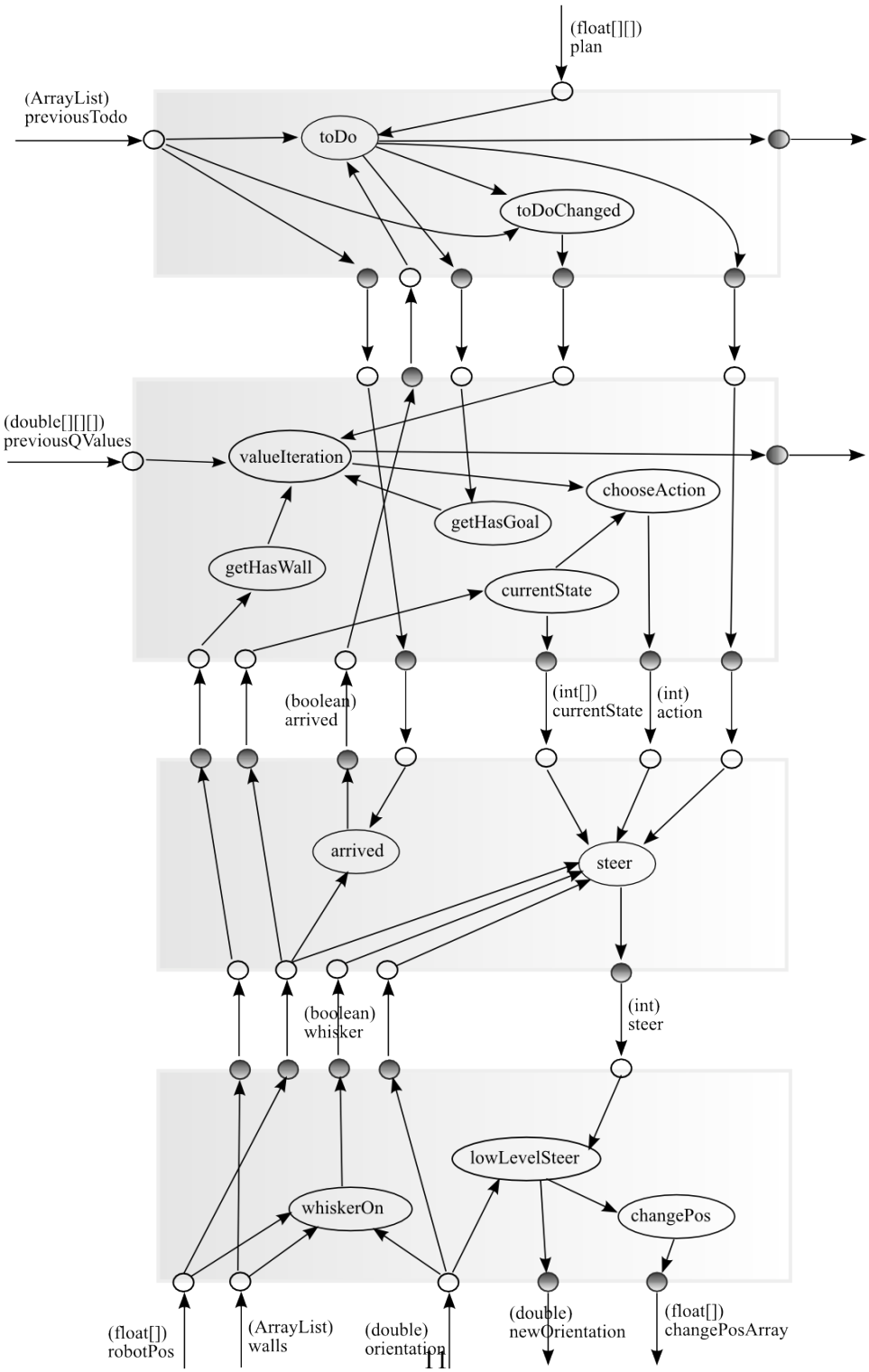


Figure 2.5: Value iteration controller

2.2.2 States

The robot situated environment is divided up into a grid world. The grid world cannot actually be seen in the simulated environment, and it is only an artificial construct to show the states in the MDP. See Figure 2.6 for a drawing of the conceptual grid world, where each square is a state. Figure 2.6 helps the reader to understand the states but the squares are not actually present in the simulated environment.

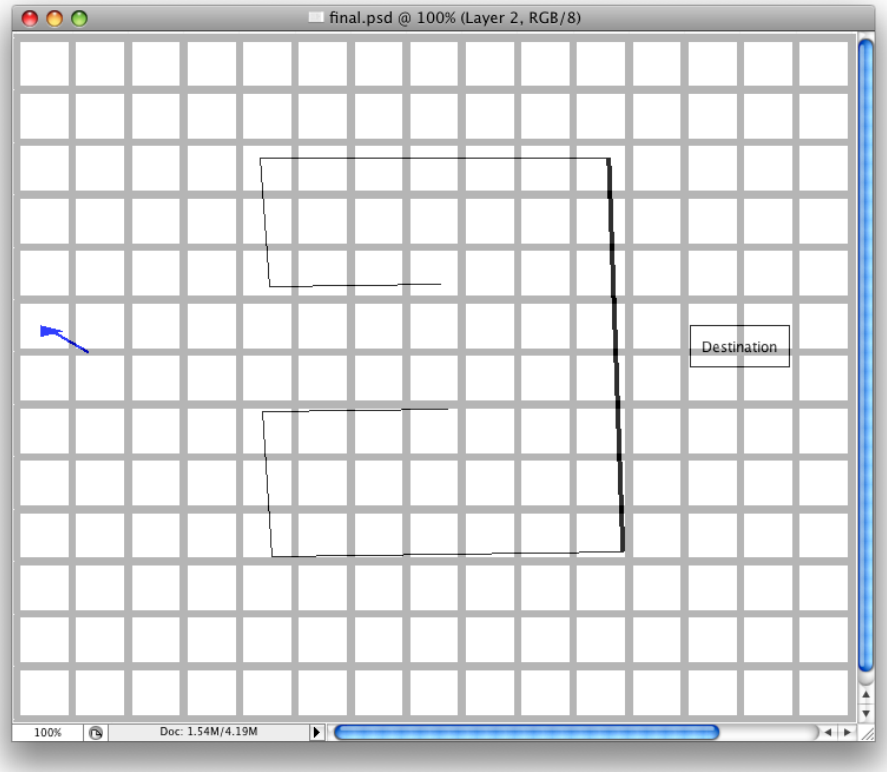


Figure 2.6: Converting robot navigation applet into grid world.
Note that even though the "Destination" (i.e. goal) is drawn as a rectangle, the goal position is just a point at the center of the rectangle. The same idea goes for the robot, even though the robot is drawn as a triangle, we only need to track its central position. The robot's whisker is still a line segment.

A state may contain one or more of the following elements:

1. Segment of wall;
2. Goal position.

As an example, in a 6×6 grid world, properties of a state positioned at row three and column four of the grid world can be shown as follows:

the states's row location (int) 3

the state's column location (int) 4

if the state contains wall (boolean) false

if the state contains goal (boolean) true

We assume that a state cannot have both a wall and a goal at the same time. If this happens, we can either increase the number of states or shift the position of the states to have different states containing the wall and the goal.

The robot cannot go into a state that contains a segment of wall. Within a state, the robot can go up, down, left and right to get out of a state or inaction to stay within the same state (more about actions in Section 2.2.3).

When value iteration is completed, each state would be assigned an optimal action.

2.2.3 Actions, Cost Function

The actions (Figure 2.7) that the robot can perform in a state are:

- left $\leftarrow$;
- right $\rightarrow$;
- up $\uparrow$;
- down $\downarrow$;
- inaction (stay in current state).

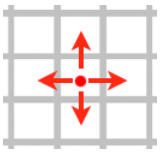


Figure 2.7: Actions
The available actions are left, right, up, down and inaction.

The actions described so far ($\leftarrow$, $\rightarrow$, $\uparrow$, $\downarrow$ and inaction) are high level actions taken in the grid world, and these actions tell which state the robot should go next. But the robot also has to steer within a state, which involves lower leveled maneuvers in the simulated environment.

As can be seen from the VI controller (Figure 2.5) between the relationships of `chooseAction` in the second layer from the top, `steer` from the third layer and `lowLevelSteer` from the fourth layer, actions described in this section exist in a higher level than the actions introduced in the basic controller (Section 2.1). The following list is a summary of the possible return values from these three functions:

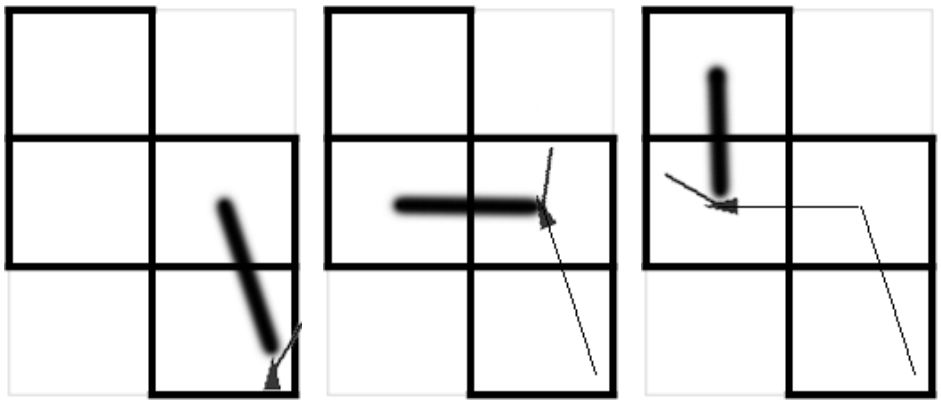
chooseAction $\leftarrow$, $\rightarrow$, $\uparrow$, $\downarrow$ and inaction

steer left, right, straight

lowLevelSteer The new orientation of the robot as an angle between 0 to 360°

This high level action from `chooseAction` tells the robot which state to go next. For example in Figure 2.8 (a), the optimal action for the current state is to go up. The `steer` function finds out which direction the robot needs to rotate to get to the middle of the target state, and in the example (Figure 2.8 (a)) the direction is left. Finally, based on the direction from `steer`, `lowLevelSteer` calculates the new orientation, which is the current orientation (90°) + change in orientation to turn left (18°) = 180°. The change in orientation is selected to be 18° because the robot's animation seems to have smooth movements with that number. If the `steer` function says to turn right, `lowLevelSteer` will change direction by $-18°$.

Each action is associated with a cost to ensure that the controller finds an optimal path. For example, this cost for an action can be set to 0.9.



(a) chooseAction:up; steer:left; lowLevelSteer: $90+18=108^\circ$

(b) chooseAction:left; steer:left; lowLevelSteer: $135+18=153^\circ$

(c) chooseAction:up; steer:right; lowLevelSteer: $180-18=162^\circ$

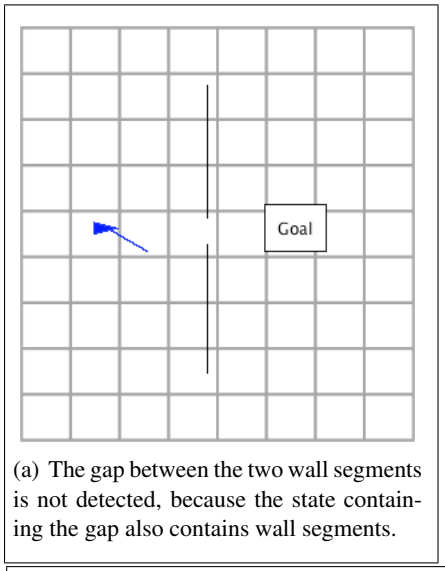
Figure 2.8: An example to show robot actions using VI.

2.2.4 Goal States and Reward Function

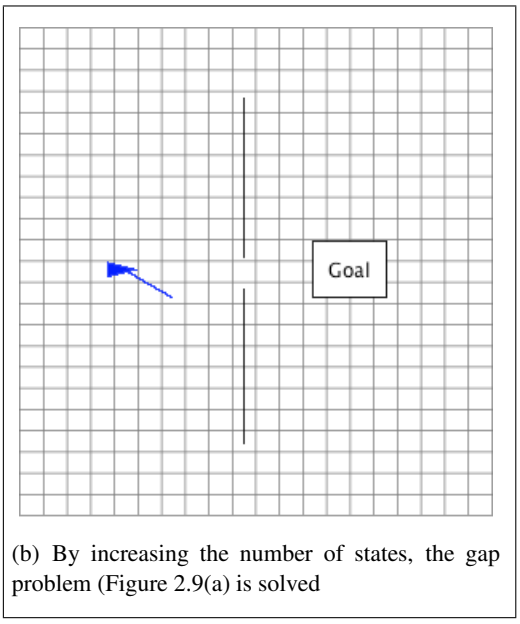
A state with a goal position inside has a reward value of for example 10. A state with a wall segment is considered not beneficial and impenetrable, so such states always have a reward value of zero. This assumption about impenetrable wall states has the drawback that if there is an optimal path that goes through a small gap between walls, the algorithm may not find the path since it cannot see the gap (Figure 2.9(a)), but the problem can be solved by increasing the number of states (Figure 2.9(b)).

As discussed in Section 2.2.2, we assume that a state cannot have both a wall and a goal at the same time. If this assumption causes problems (see Figure 2.9(c)), the number of states in the grid world can be increased to put goals and walls into different states (Figure 2.9(d)). Another possible solution is to shift the states, so that walls and goals can be split into different states.

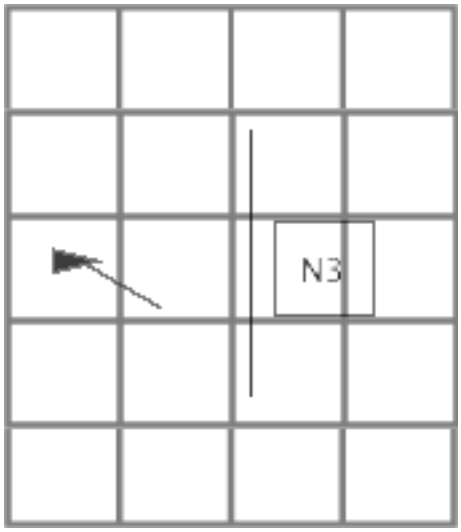
Robot only wastes time traversing in the spaces outside of the grid world, so they also have a reward value set fixed to zero.



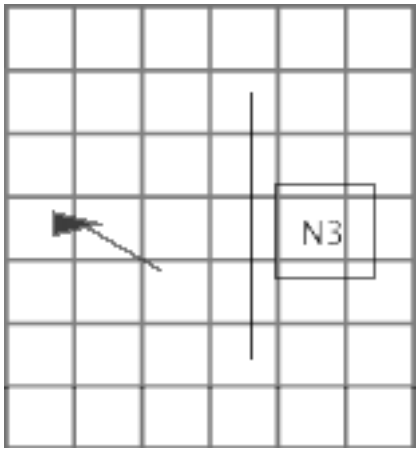
(a) The gap between the two wall segments is not detected, because the state containing the gap also contains wall segments.



(b) By increasing the number of states, the gap problem (Figure 2.9(a) is solved



(c) Problem with goal and wall in the same state. Note: The goal N3's position is at the center of the box for N3



(d) Increased number of states to solve the problem in 2.9(c)

Figure 2.9: The issues with VI controller and the solution.
In both cases, use of a smaller square block size for the states solved the problem. The user of the designing environment may need to experiment with multiple state sizes to solve the problem.

Processing Goal States in the Top Layer

When there are multiple goals, and the robot is required to visit all goals only once, as soon as the robot reaches a goal, that goal will be removed from the goals' list. This is detected by the `toDoChanged` function (see VI control diagram in Figure 2.5), and so value iteration can be performed again while ignoring the reached goal, and the robot can move on towards the next goal.

2.2.5 Characteristics of the Reinforcement Learning Controller

This controller successfully avoids the trap (as shown in Figure 2.2) that posed a problem for the basic controller. Even though modifications to the basic controller may solve the problem, the robot may be trapped again in a different unanticipated environment that is designed to trap the robot. The VI controller is intelligent enough to learn for an unknown situation, and can plan optimally for any arbitrary environment.

VI for the high level control is only called when a goal is reached, which saves computation time.

The problem with small gaps between walls (Figure 2.9(a)) and the problem of a goal being in the same state as a wall segment (Figure 2.9(c)) can both be solved by increasing the number of states in the environment (Figures 2.9(b) and 2.9(d)) or by shifting locations of the states.

Due to the amount of calculations, another possible drawback with VI controller is that the speed of planner is reduced as the state number is increased. But this does not seem to be significant problem as a 150×150 grid world only takes about one second[2] each time the `valueIteration` function is executed to find a new policy.

Please note that the current implementation of the VI controller assumes that the environment does not change, i.e. the positions of the walls and goals do not change during the execution of the controller. As discussed in Section 2.2.6 the implementation of the top layer of controller, a new path is only calculated when a goal is reached. If the environment changes often, there should be mechanisms added to the controller to detect the change in the positions of the walls and/or

[2]Hardware: 2.4 GHz Intel Core 2 Duo, 2GB RAM

goals.

To improve the VI controller, one suggestion is to separately detect changes in goals' and walls' positions. Changes of goals can be detected by modifications to the `toDoChanged` function in the top layer (Figure 2.5). Changes of the walls can be detected by a new function called `wallsChanged` in the value iteration layer, and when any one of the walls' position is changed, `wallsChanged` should be able to trigger `valueIteration` function to generate a new policy. These additions are excellent assignment exercises to help people understand the controller better.

2.2.6 Detailed Description of Each Function

To further clarify the controller (Figure 2.5), here is a detailed description of the functions for the VI controller. We will now explain each one of the functions with its input parameters and output.

Top Layer (Figure 2.5)

This layer keeps track of the target goal positions, which goals the robot still needs to visit, and the current target goal. An example of the the functions' output values are shown in Figure 2.10.

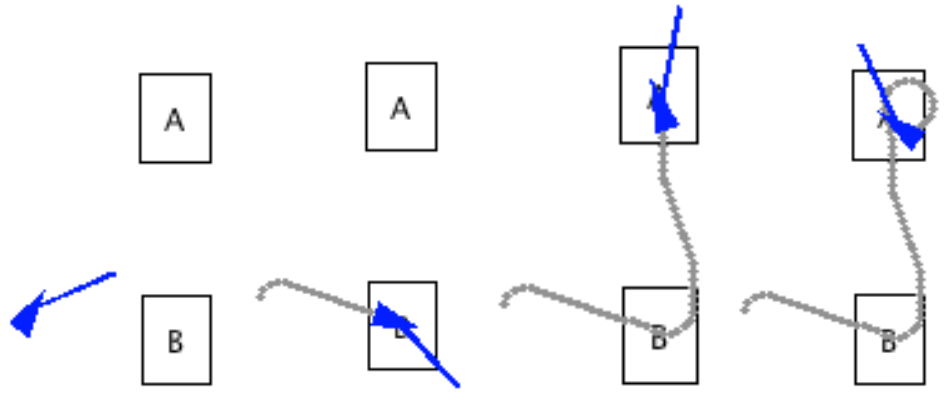
***

toDo keeps track of which goal positions are still not reached. The group of goal positions are recorded in this function as a to–do list. When a goal is reached, it is then removed from this list of target goals (i.e. to–do list) which is stored within the `toDo` function.

Input Parameters For each input, we will list the variable type, variable name and then a description.

- *ArrayList* **previousTodo**: Output of the `toDo` function from the previous time–step[3].

[3]Definition of time–step can be found from Section 3.2.4



(a) plan: A, B; output of *toDo*: B, A; output of *toDoChanged*: false

(b) plan: A, B; output of *toDo*: A; output of *toDoChanged*: true

(c) plan: A, B; output of *toDo*: null; output of *toDoChanged*: true

(d) plan: A, B; output of *toDo*: null; output of *toDoChanged*: false

Figure 2.10: An example showing values of the output of the top layer functions for the VI controller.
`Plan` stores the locations of all the target goals regardless of a goal is reached or not.

- *float[][]* **plan**: The entire list of goal locations. This list includes the goals that are reached already.
- *boolean* **arrived** If the robot has reached its current goal location.

Output: Output type: *ArrayList*. Description: The list of target goals not yet reached. The type used is `ArrayList` because it supports dynamic arrays that can grow or shrink as needed[4], where as standard arrays are of fixed length and cannot grow or shrink. So, with `ArrayList`, when a goal is reached, we can easily delete it from this list. Each goal is stored as a *float[]* with length of two, e.g. [7837.0, 4031.0].

***

[4]http://java.sun.com/javase/6/docs/api/java/util/ArrayList.html

toDoChanged By comparing the output of the `toDo` function from the current and the previous time–steps, this function finds if the to-do list from the `toDo` function is changed. If the to-do list is changed, the output of the `toDoChanged` function will then trigger the `valueIteration` function to generate a new shortest path leading to the next goal.

The output of the `arrived` function is *not* used as the trigger because when a robot stays at the final destination, it has `arrived` at the destination, but it will not need to use `valueIteration` to generate a new path.

Normally, when a goal is reached, the goal is deleted from the to–do list. But we always need the output of `toDo` for the `steer` function to find which direction the robot should go (straight, left, or right). So when the final destination is reached, it is not removed from the to–Do list, so that we can have a non–empty list. `toDoChanged` specializes in finding out if the to–do list is changed so that `toDo` can output the actual list for `steer` function.

Input Parameters

- *ArrayList* **previousTodo**: Output of the `toDo` function from the previous time–step.
- *ArrayList* **toDo**: Output of the `toDo` function from the current time–step.

Output: A *boolean* that is true if the to–do list of goals from the `toDo` function is changed. When a goal is reached, the output of this function is only true for one time step, and this is to enable `valueIteration` to generate a new policy.

==========

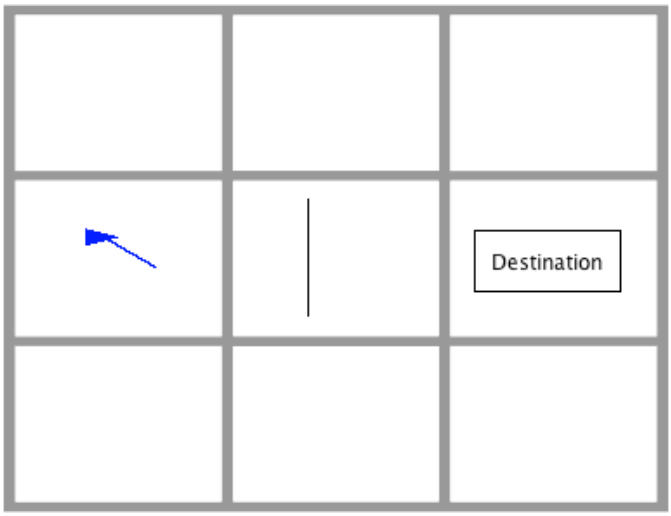
Value Iteration Layer (second layer in Figure 2.5)

By using value iteration, this layer finds an optimal policy in the grid world, and so the optimal actions for all the states.

***

getHasGoal finds out which states contain goals. Given the output of `toDo`, which contains the goal locations in a numerical form (e.g. {[7787.0,5181.0]}), `getHasGoal` records goal locations in a different format that says if a state in the grid world contains a goal.

For an example of the output of `getHasGoal`, please see Figure 2.11(c).



(a) The real environment with wall positioned between the two points : {(2438.14, -1743.59); (2438.0, -1708.66)}, goal from to-do at: (2495.67, -1726.66)

FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

(b) Output of getHasWall

FALSE	FALSE	FALSE
FALSE	FALSE	TRUE
FALSE	FALSE	FALSE

(c) Output of getHasGoal

Figure 2.11: Example outputs for the getHasWall and getHasGoal functions

Input Parameters

- *ArrayList* **toDo**

Output A two dimensional boolean *boolean[][]* indicating if there is a goal in each of the row and column position of the grid world.

***

getHasWall finds out which grids contain walls. The output contains the same information as `walls`, but in a different format that says if a state in the grid world contains a wall segment.

Input Parameters

- *ArrayList* **walls**: The locations of all the walls.

Output A two dimensional boolean *boolean[][]* indicating if there is a wall segment in each of the row and column position of the grid world.

***

currentState finds out the current state location of the robot.

Input Parameters

- *float[]* **robotPos**: Position of the robot, e.g. [2388.3333, -1726.0].

Output *int[]* The current index of the row and column location of the robot.

***

valueIteration finds out the the Q-value for each of the state in the grid world (i.e. each square). The `valueIteration` function is triggered to re–learn Q-values when the to–do list is changed.

Input Parameters

- *double[][][]* **previousQValues**: The Q-values from the previous time step. If `valueIteration` is not triggered by the output of `toDoChanged`, this old set of Q-values will be passed on to the next time step, since it is not necessary to learn new Q-values.
- *boolean[][]* **hasWall**
- *boolean[][]* **hasGoal**
- *boolean* **todoChanged**

Output: A three dimensional double array (*double[][][]*) which indicates the Q-value for a certain row, column position and action in the grid world.

***

chooseAction Based on the Q-values from the `valueIteration` function, this function finds out the optimal actions for the current state.

Input Parameters

- *double[][][]* **qValues**
- *int[]* **currentStateLocation**

Output: An *integer* representing the optimal action for the current state, for example, $NORTH(\uparrow) = 0; EAST(\rightarrow) = 1; SOUTH(\downarrow) = 2; WEST(\leftarrow) = 3; INACTION = 4;$

==========

Middle orientation Layer

This layer acts as a liaison between the VI and the bottom layers. Given the optimal action for the current grid from the VI layer, this layer finds out the general direction action for the robot to head towards. Based on the current location, this layer also calculates if the robot reached a goal and passes that information to the top layer.

***

steer Based on the current conditions of robot location, orientation, whisker sensor and goal location, this function determines which direction the robot should steer towards.

Input Parameters

- *float[]* **robotPos**: Position of the robot, e.g. [2388.3333, -1726.0].
- *double* **orientation**: Current orientation of the robot in degrees, e.g. 90^o.
- *boolean* **whisker**: True if the whisker of the robot is touching a wall, false otherwise.
- *int[]* **currentState**: see output of `currentState`. Based on the current state and the optimal action, the controller can calculate to steer the robot towards the middle of the next wanted state.
- *int* **action**: The optimal action for the current state. $NORTH(\uparrow) = 0; EAST(\rightarrow) = 1; SOUTH(\downarrow) = 2; WEST(\leftarrow) = 3; INACTION = 4;$
- *ArrayList* **toDo**: see output of `toDo`. If there is a goal from the to-do list, then the robot should steer towards the goal.

Output There are three possible directions for steering: left, right, straight.

***

arrived This function determines if the robot has arrived its current target goal location.

Input Parameters

- *float[]* **robotPos**: Position of the robot, e.g. [2388.3333, -1726.0].
- *ArrayList* **previousToDo**

Output `boolean`, true if the robot has arrived a goal in the to-do list from the previous time step. As long as the reached goal stays in the to-do list and the robot stays at the goal, this output will be true. But if the goal is removed from the to-do list (i.e. via `toDo` function), output of `arrived` will normally be false during the next time step.

==========

Bottom Orientation Layer

This layer handles low level real time interactions with the environment.

***

lowLevelSteer Compared to `steer`, `lowLevelSteer` offers much more finer grained control over the direction that the robot should steer toward.

Input Parameters

- *double* **orientation**: Current orientation of the robot in degrees, e.g. 90^o.
- *int* **steer**

Output The function returns the new value for the orientation of the robot in terms of degrees of angle. This orientation is the final orientation, but not the change in orientation, because the final orientation could be useful to calculate the change in position of the robot (see function `changePos`).

***

whiskerOn finds out if the robot's whisker is in contact with any of the walls.

Input Parameters

- *ArrayList* **walls**: The locations of all the walls. Each wall contains the location of the two end points. (e.g. {[7787.0,5181.0],[7987.0,5181.0]})
- *float[]* **robotPos**: Position of the robot, e.g. [2388.3333, -1726.0].
- *double* **orientation**: Current orientation of the robot in degrees, e.g. 90^o.

Output True if the robot's whisker is touching a wall.

***

changePos based on the new orientation and speed of moving one pixel per time step, `changePos` finds the exact change in location for the robot to take.

Input Parameters

- *double* **newOrientation**

Output The change in x and y location of the robot.

Chapter 3

System and Graphical User Interface (GUI) Design

There is an existing robot navigation applet prototype (Section 3.1) which relies on prolog as the main underlying programming language. But most users of the applet are believed to be more familiar with the Java programming language rather than Prolog, therefore it will be more effective for them to learn how to write a controller by saving the time spent on learning Prolog. The controller for the new applet will be based on Java, so one of the main focuses in designing the new applet is to assist users to program in Java. The tool is designed to assist people to write a controller, so a "Layer Editor" (Section 3.2.1) is added to help with the visualization and the creation of the hierarchical controller.

Another design goal is to help the user to examine details of the controller during the execution of the simulated environment. The debugging facilities are designed so that the user can look at the arguments passed between all the functions at any chosen time step. This ability to check values of arguments allows the user to examine in detail the inner workings of the controller, so during the execution of the controller, the user can have a thorough understanding of what is going on.

3.1 Prototype Applet

The new applet follows the design of the existing prototype applet. The prototype applet can be found from: `http://aispace.org/robot/`. Designers and programmers of the robot navigation applet are encouraged to view all the applets from AISpace (`www.aispace.org/downloads.shtml`) and the look and feel document (`http://www.aispace.org/lookAndFeel.shtml`) to understand the design.

3.1.1 Menu Options

There are five menu options (Figure 3.1 and Figure 3.2). Since most of the options will not be changed, except for the "Edit" option, we will not discuss them here, and detailed explanations on these menu options can be found from: `http://aispace.org/robot/help/general.shtml#menu`.

To change the the controller, the user of the applet will need to use the "Edit" menu which we will concentrate on talking about.

The "Edit" menu option (Figure 3.1) will bring up the options to edit a layer of controller source code (Figure 3.2(b)). This "Edit" menu is sufficient for a controller with three fixed layers, but insufficient for an arbitrary number of layers. An improved "Edit" menu will be talked about in Section 3.2.1.

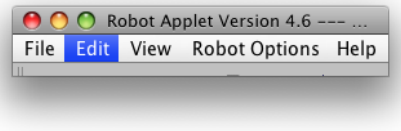
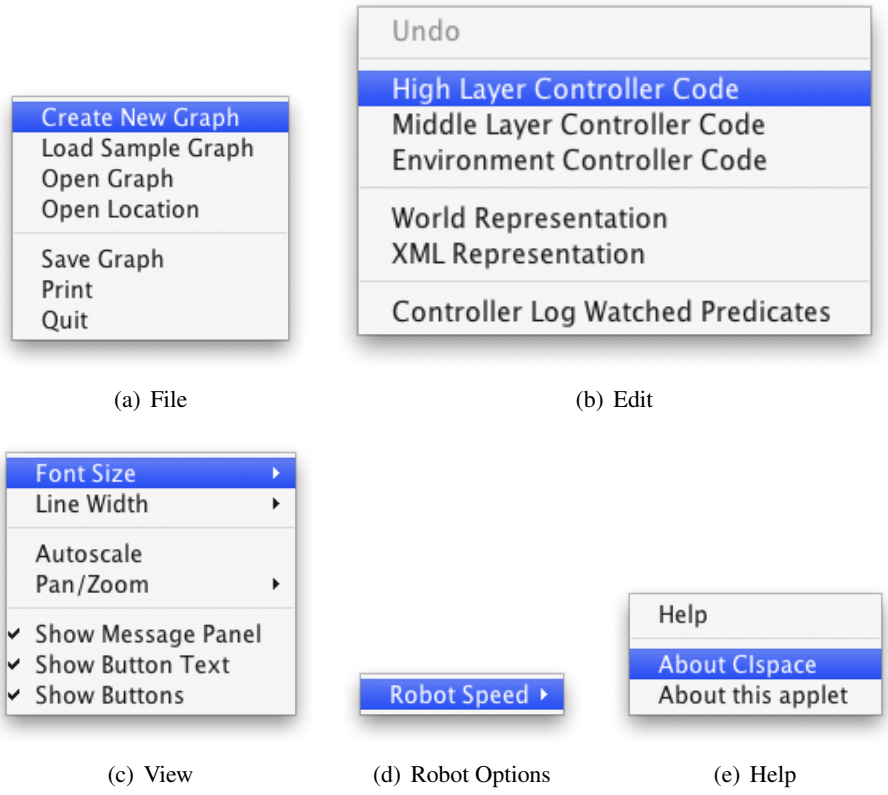


Figure 3.1: Main menu



(a) File (b) Edit

(c) View (d) Robot Options (e) Help

Figure 3.2: All the file menu options from the prototype applet.

3.1.2 Tab Panels

In addition to the menu options, there are two major tabs/panels in the prototype applet, one is called "Create" (Figure 3.3), and the other called "Solve" (Figure 3.4). The "Create" tab allows users to create an environment by adding/deleting walls or goal locations. As can be seen from the buttons across the middle of Figure 3.4, the "Solve" tab allows the user to control the execution of the robot applet (the buttons: "Step Robot", "Run Robot", "Stop Robot", "Reset Robot" etc), view the properties of the current execution plan ("View/Modify Plan") and to obtain trace/debug information.

Figure 3.3: Create tab

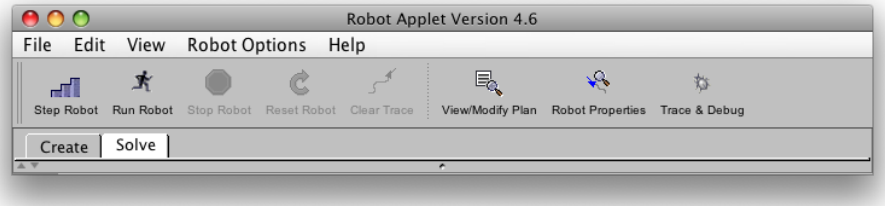


Figure 3.4: Solve tab

3.2 New Applet Design

For the new design, menu options "File", "View", "Robot Options" and "Help" and creating a simulated environment under the "`Create`" tab will be the same as the previous prototype design.

The “Edit” menu option and the “Solve” tab , however, will be significantly modified. The major differences are that an arbitrary positive number of layers is allowed for the controller, the function code is in Java rather than Prolog, and there are some modifications for the debugger.

3.2.1 Layer Editor

Under either of the modes “Create” or “Solve”, the layer editor can be brought up by selecting “Layer Editor” from “Edit” option under main menu as shown in Figure 3.5.

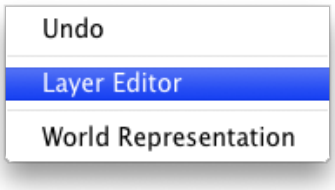


Figure 3.5: The new drop down Edit menu

The layer editor (as shown in Figure 3.6) allows user of the applet to create and modify layers, edit functions within a layer, and edit the interactions between the functions by editing the arcs.

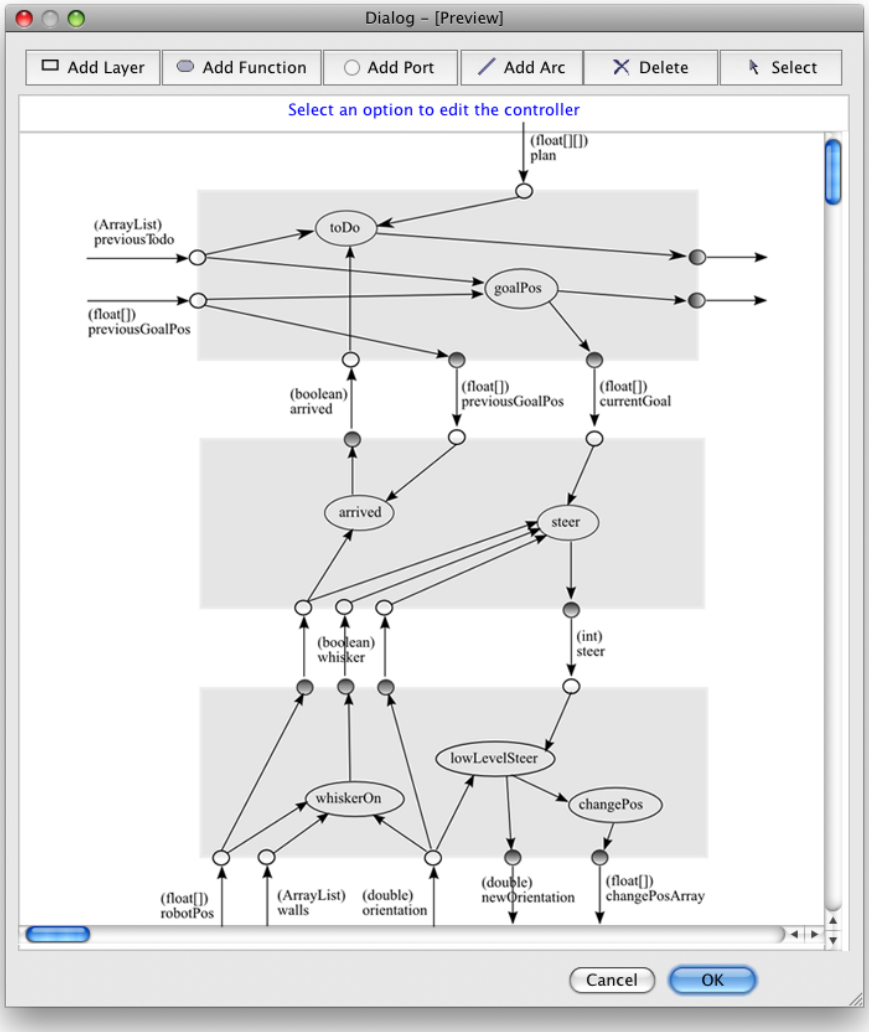
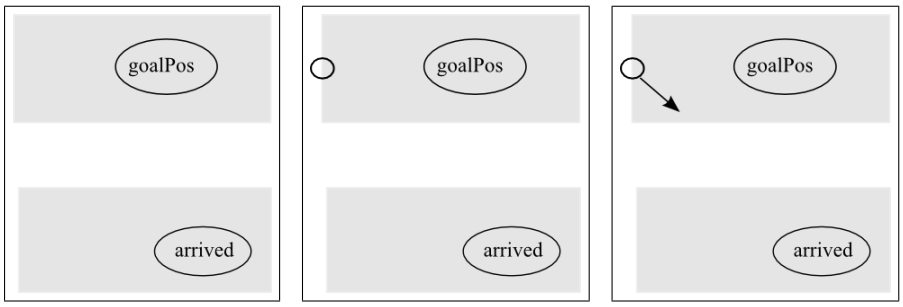
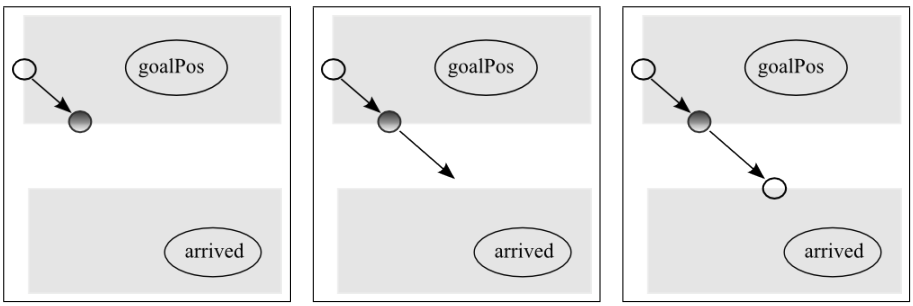


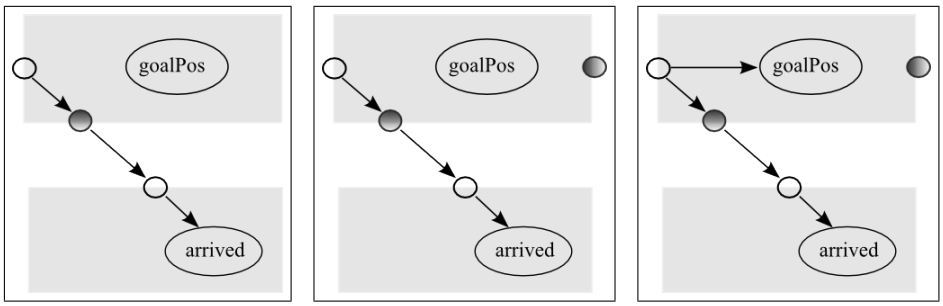
Figure 3.6: Layer editor



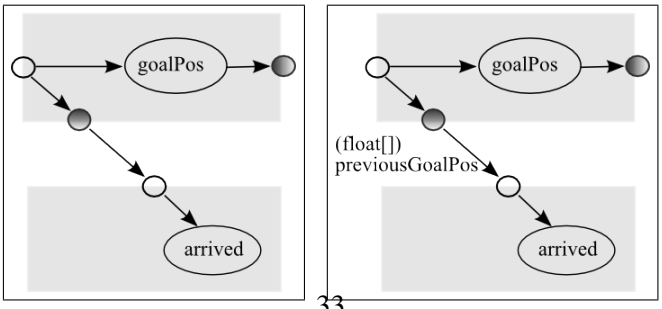
(a) Before adding any arcs or ports (b) Creating a port (c) Creating an arc



(d) Port added automatically (e) Continuing dragging the arc from Figure 3.7(b) (f) Port added automatically



(g) Continuing dragging the arc from Figure 3.7(b) (h) Create the pairing output port for the port created in Figure 3.7(b) (i) Creating arcs to pass parameters between different time steps



(j) Creating arcs to pass parameters between different time steps (k) The arc's type and name can be shown near any of the arc segments

Figure 3.7: An example used to illustrate how to create arcs. This example is extracted from the original Basic Controller from Figure Figure 2.3.

Adding/Editing Ports

A port can be added to a layer by the "Add Port" button at the top of layer editor (Figure 3.6). For example, compared to Figure 3.7(a), a port is added in Figure 3.7(b).

Definition 1 *A* port *resides at the edge of a layer and passes a parameter in or out of a layer.*

An input port passes a parameter into a layer, and is shown as a hollow circle (Figure 3.8). Output port passes a parameter out of a layer, and is shown as a dark circle (Figure 3.8).

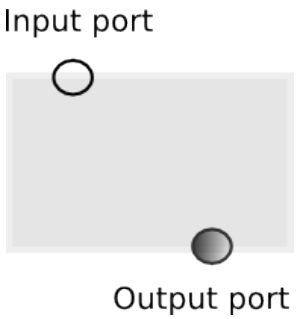


Figure 3.8: Input and output ports

By using the "Select" button in layer editor (Figure 3.6) and then clicking on a port, a user can change the type of the a port to either "Input" or "Output" as shown in Figure 3.9.

Matching Ports When a variable needs to be passed from one time step[1] to another time step, an input port on the left of a layer can receive the variable from the previous time step, and another port on the right side of the layer can be used to pass the current value of the variable. This can be seen as the two ports shown in Figure 3.10.

Definition 2 *A pair of* matching ports *passes the same variable between time*

[1] Definition of time–step can be found from Section 3.2.4

Figure 3.9: Editing a port

steps. They appear graphically at the same horizontal height as seen in Figure 3.10.

The ports can be declared to be delivering the same variable through different time steps by using the port editor shown in Figure 3.9. An example showing the steps to create matching ports can be seen from Figures 3.7(g) to 3.7(k).

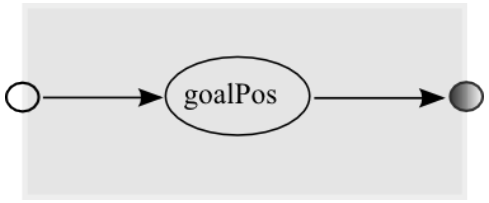


Figure 3.10: Two ports deliver a variable through out different time steps `goalPos` function is the same as in the basic controller Figure 2.3 as described in Section 2.2.6

Arcs

An arc passes output from one function[2] to the input parameter of another function. For example, in Figure 3.7, the arc passes information on `previousGoalPos` between the functions `goalPos` and `arrived`.

"GUI arcs" or simply "arcs" are represented by the `ArcVariable` class as described in Section 5.1. An arc graphically shows the flow of a variable passed between functions, and an `ArcVariable` is the actual storage of the object being passed around.

Multiple arcs coming out of a function are stored as one object of type `ArcVariable`.

If a function passes a variable through different time steps, then the arc(s) going to the output port uses one `ArcVariable` object, and the arcs going to the input port uses a different `ArcVariable`. E.g. the two arcs in Figure 3.10 are represented by two different `ArcVariable`s.

Adding/Editing Arcs .

We will use the standard AIspace way to create arcs, for example, the method of creating arcs from the belief and decision networks applet found at
`http://aispace.org/bayes/index.shtml`

If the two functions being connected are in different layers, e.g. functions `goalPos` and `arrived` Figure 3.7, the additional ports and arcs between the two functions will be added automatically. i.e. when the user drags the arrow end of the arc across different layers, the programming environment will automatically add I/O ports at the bottom and top of the layers. As an example, when the arc (Figure 3.7(g)) between the input port (output of function `goalPos`) and `arrived` is added, arc's segments and ports are automatically added as shown in Figures 3.7(b) to 3.7(g).

An arc can be edited by clicking on "Select" button from "Layer editor" (Figure 3.6), and then edit the arc using the arc dialog (Figure 3.11).

[2]Functions are explained in Section 3.2.2

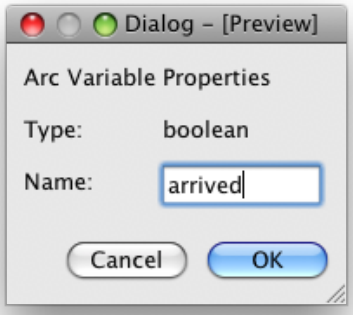


Figure 3.11: Dialog to modify a selected arc. Note that the name is used as the name of the input parameter for the function taking the input. A name is necessary because a name is not given from the function where the arc comes from (there is no name for a variable returned from a method in Java).

Adding Layers

To add a layer to the existing controller, the user can click on the "Add Layer" button, then the frame for a new layer will appear at the top of the controller editor panels (Figure 3.12). To change the position of the layer, user can drag the new layer to a new position.

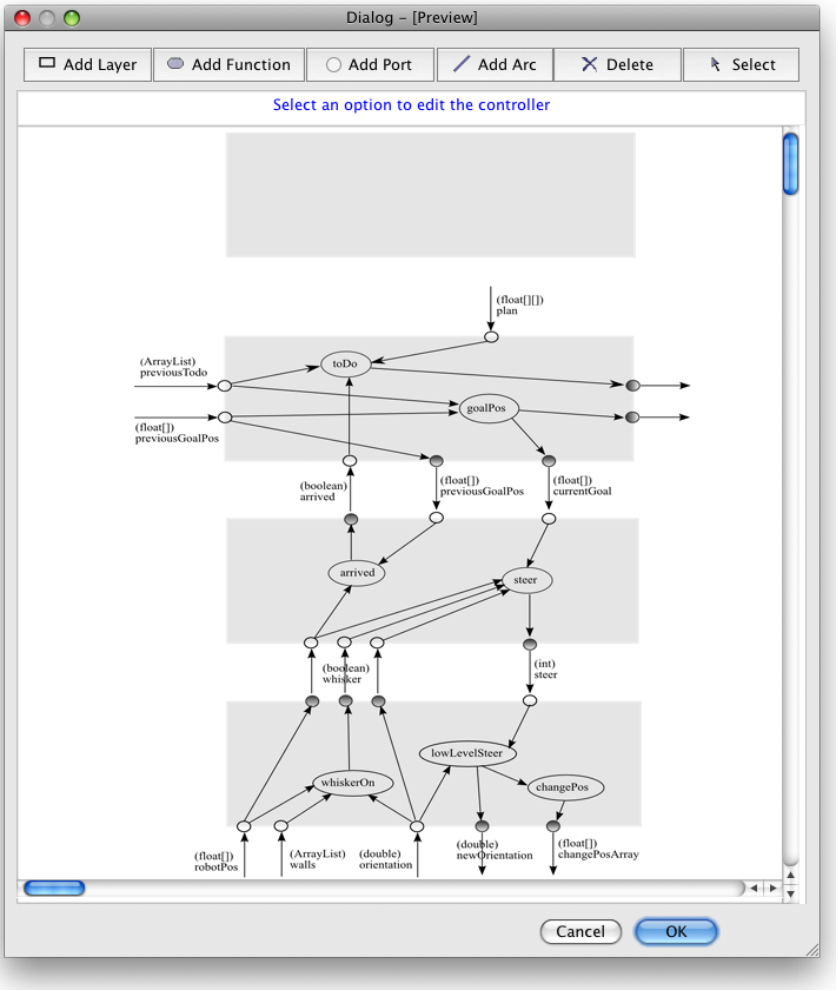


Figure 3.12: After "Add Layer" is clicked, a new layer appears at the top. This controller is based on the basic controller Figure 2.3

3.2.2 Function Editor

A function is very similar to a Java method, in fact, when the Java interpreter is run for the user written controller code, a function is transformed to a Java method.

As shown in Figure 3.13 with an example using the `arrived` function from the basic controller (Section 2.1), a function takes inputs and returns one single output.

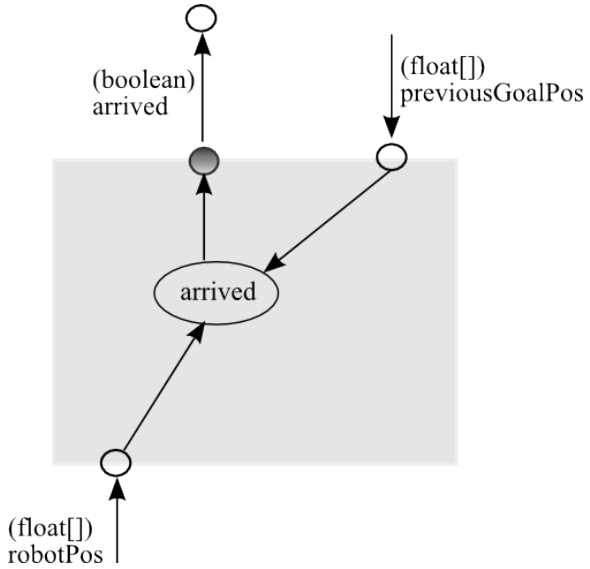


Figure 3.13: "arrived" as an example function
"arrived" function as represented as the oval shape is a function that takes parameters from arcs with variable names: arrived, previousGoalPos, robotPos, that are of type: boolean, float[], float[] respectively.

If a function is selected (e.g. by clicking on the "Select" button from "Layer Editor" and then click on a function), the function editor dialog will pop up (as shown in Figure 3.14).

Users can change the output type of the function, which is equivalent to "return type" in Java. If the actual return type of the function differs from the specified return type from the "Output variable type" field, an error will be thrown, and the controller will not be executed.

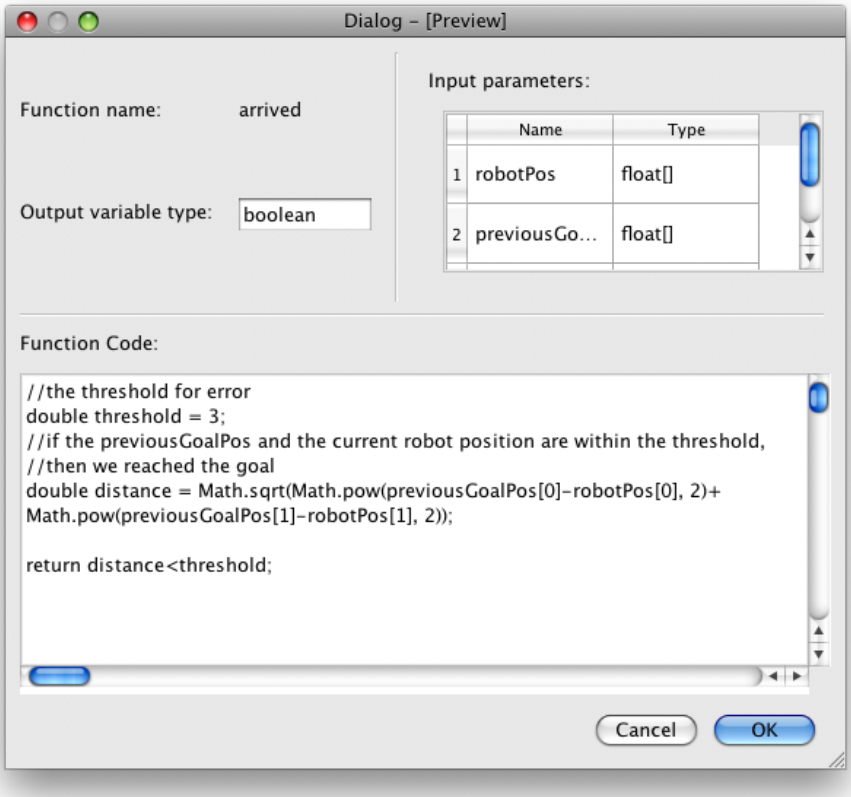
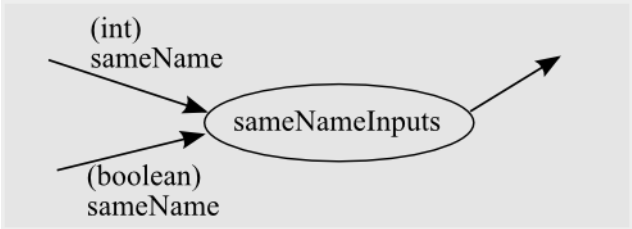


Figure 3.14: Function editor

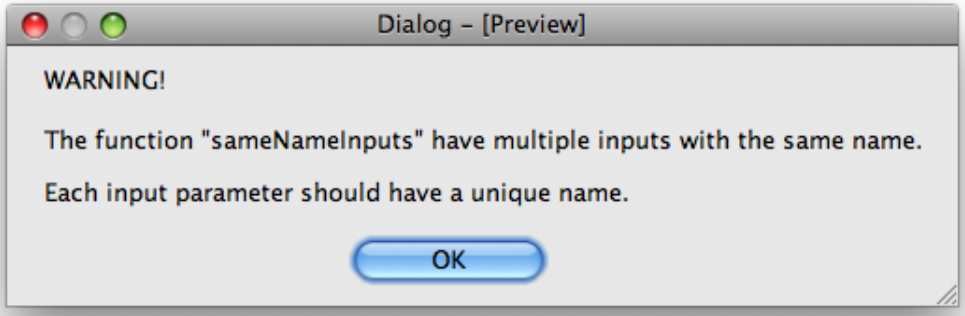
Input(s) of Function

Source code can also be modified in the "Function editor". The input parameters take the variable names and types given by the input arcs. These input parameters are processed by the programming environment, and shown in the function editor dialog for the users' convenience. If one of the input arcs for a function is edited or removed from layer editor (Figure 3.6), the "Input parameters" list in the function editor (Figure 3.14) will reflect the change which is processed by the applet, but the user of the applet needs to change the code for the function to make use of the change.

If two inputs of a function have the same variable name (Figure 3.15), an error message will be shown before controller is executed and the Java interpreter is called.



(a)



(b)

Figure 3.15: Function with multiple inputs of the same name.
The error message (Figure 3.15(b)) will appear if input arcs causing the problem are added to the function. The error message will appear again if the user clicks on the "Run Robot" button (Section 3.2.5, Figure 3.18).

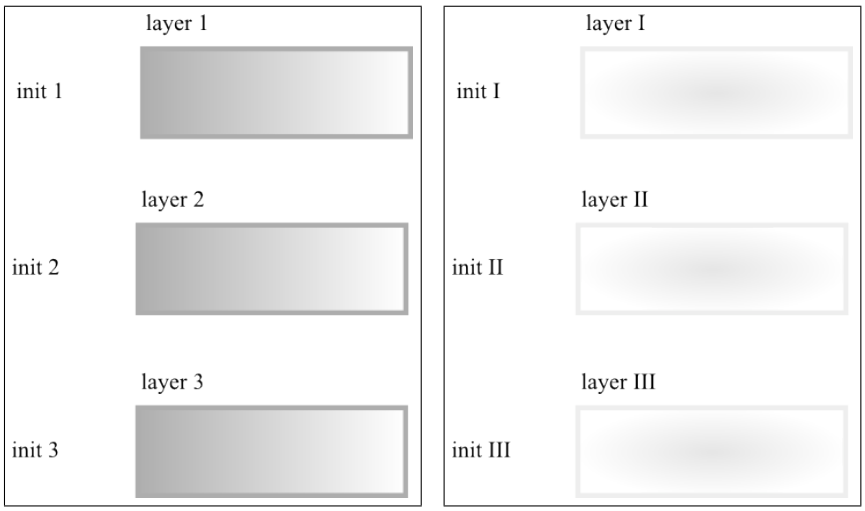
3.2.3 Layer Variables

A controller may need to initialize data to be used by multiple functions. As opposed to global variables for the entire controller, each layer may have its own set of *layer variables* (see Definition 3). As shown in Figure 3.16, layer variables allow designers of a robot navigation controller to easily copy an existing layer to a new controller.

Definition 3 *A* layer variable *has its scope within a layer and is shared and accessible by all functions within a layer* .

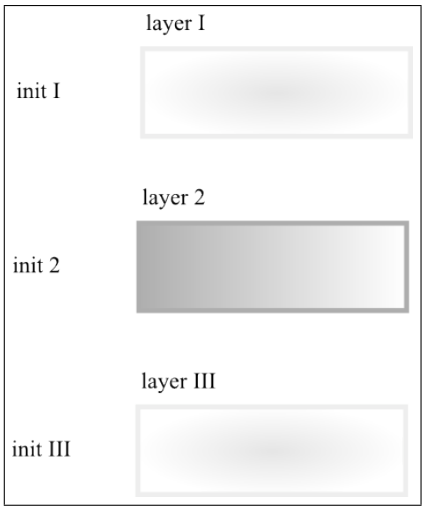
Layer variables can be edited by selecting a layer, and then using the dialog editor shown in Figure 3.17.

The programming environment will later add the layer variables to each of the functions in the layer as local variables.



(a) Controller 1

(b) Controller 2



(c) Controller 3, using layers from both controllers 1 and 2

Figure 3.16: An example showing the use of layer variables.
This example shows how a new controller (Figure 3.16(c)) might be created based on two existing controllers (Figures 3.16(a) and 3.16(b)). If global variables are defined for the entire controller instead, the two controllers 1 and 2 are likely to have different global variables, and there will be problems when adding an existing layer (e.g. layer 2 of controller 1) to the new controller (controller 3). We assume that the different layers have the same ports between the layers, so that the user do not need to edit the ports during the integration of controllers 1 and 2.

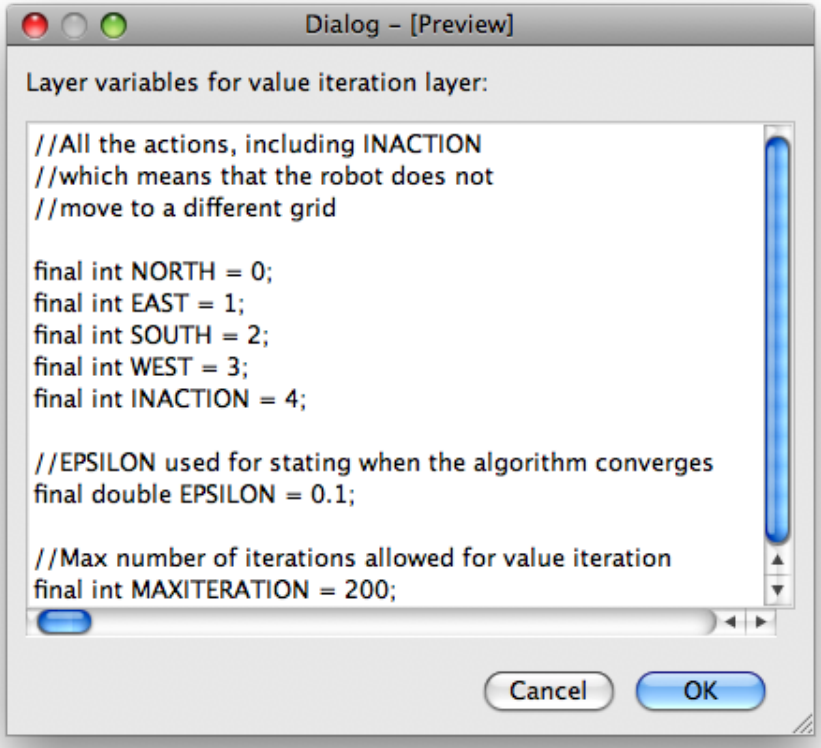


Figure 3.17: Editor to modify layer variables.
The layer variables shown here are for the value iteration layer of the VI controller (Figure 2.5).

3.2.4 Executing the Controller and Selecting a Time Step

We use the "Solve" tab panel to execute the running of the robot. For the old applet, please see Figure 3.4. For our new design, please see Figure 3.18. "Run Robot" button is used to run the simulated environment.

Definition 4 *A* time step *is a unit of simulated time between each simulation cycle, and a unique state is normally associated with a time step.*

User can select a time step by using spin box (Figure 3.19(a)), back/forward buttons, or the slider (Figure 3.18), and then the applet will show the snapshot of the simulated environment for that particular time step (Figure 3.19(b)). The number (≥ 0) in the spin box (as seen in Figure 3.19(a)) in Figure 3.18 indicates the current time step. If the robot has already been run, i.e. the elapsed number of time steps is ≥ 1, the spin box can be used to select a past time step.

The backward/forward buttons goes backward/forward by single time steps respectively.

The slider provides fast access to past time steps, with the left most side of the slider being time step 0, and right most side of slider being the final time step when the slider is clicked.

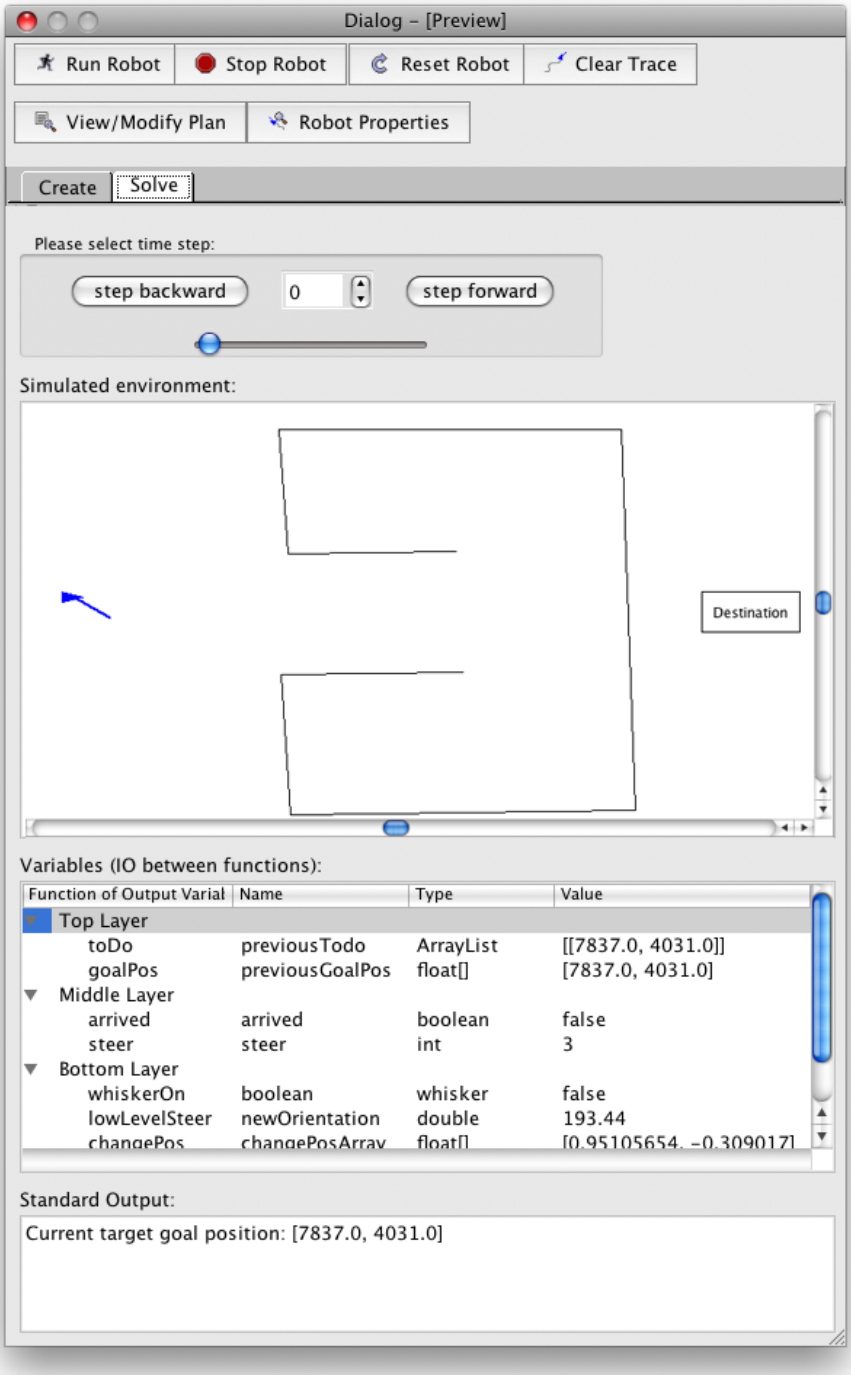
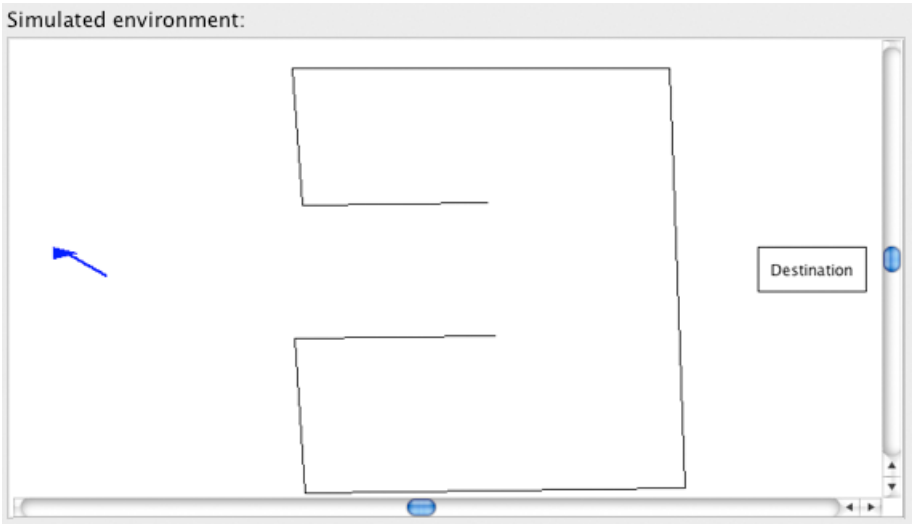


Figure 3.18: Execution dialog with debug functionality

(a) Spin box



(b) A snapshot of the simulated environment

Figure 3.19: Explaining the elements of the execution dialog from Figure 3.18

Figure 3.19(a) shows the spin box positioned just under the text “Please select time step:” in Figure 3.18. Figure 3.19(b) shows a snapshot of the simulated environment.

3.2.5 Debugging

The debugging facilities on the new "Solve" tab (Figure 3.18) allow the user to examine the values of variables at a selected time step.

All the parameters between the functions are shown near the bottom of the "Solve" tab, values of the variables are processed by the programming environment. The values of the parameters are printed by using `variableName.toString();`. The programming environment should have the `toString()` method overridden for predefined classes such as `Arraylist`.

Standard Output

Any print out from the "Standard Output" will be dumped to the `Standard Output` field at the bottom of the execution dialog as shown in Figure 3.18. The example in the figure shows the goal position that the robot is aiming at, and this goal position is useful when there are multiple goals.

This output trace is cleared when the "Reset Robot" button (Figure 3.18) is clicked. This field will not be changed when any of the controls under the text of `Please select time step` is clicked (i.e. "step back", "step forward" buttons, spin box and the slider).

A Note on the Dynamic Change of the Environment

In order to facilitate the debugging tools, the final implementation of the applet should remember all the changes in the environment during execution. E.g. when the simulated environment is running, and the user changes the position of wall(s) or even the robot by dragging these types of objects around, the applet needs to remember all the changes of the locations of objects in the environment.

3.2.6 Random Inputs

A random variable editor (Figure 3.20) is provided to add random variable inputs. This random variable editor dialog can be brought up by clicking on "Add Random" button from "Layer editor" (Figure 3.6). Note that the "Add Random" does not yet exist in Figure 3.6, but should be added in the final implementation of the applet. After the random variable arc is added to the controller, the arc can then be

linked to a function, and be used (top left of Figure 3.21).

The user can have the option to select the type for a random variable, and these types are from the available random variable types offered by Java: boolean; double; float; int; long and Gaussian (normally distributed *double* value with mean 0.0 and standard deviation 1.0).

The user can set the seed so that the same trace of a random variable can be used if the robot is reset. An example of a trace for a `boolean` random variable is $true, false, false, false, true, false$. This same trace may be beneficial if the user needs to modify some aspects of the controller to see how the robot behaves but do not want a random variable to affect the behavior of the robot.

For an example on the use of a random variable, the basic controller (Section 2.1) is modified to include a random controller as shown in Figure 3.21.

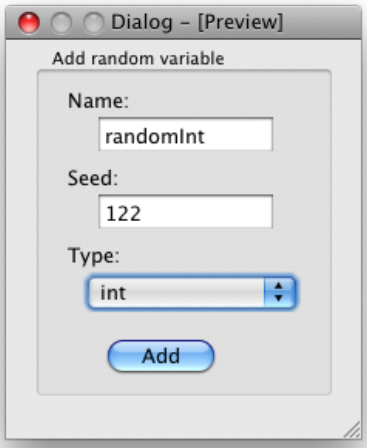


Figure 3.20: Editor to add random input

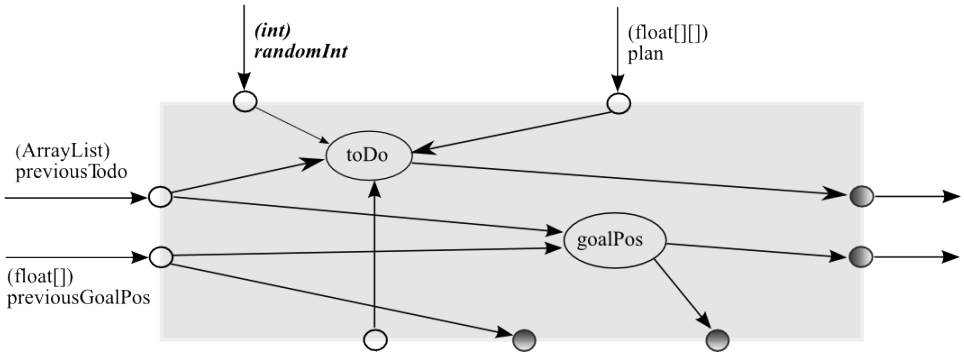


Figure 3.21: Basic controller with random input.
See randomInt at top left. Only the top layer of the controller is shown here, for the whole controller, please refer to Figure 2.3 on page 7.

Chapter 4

Interpreter

One of the key differences between the new robot navigation applet and the prototype version is the language used by users to program the controllers. The proposed version uses Java versus Prolog used in the previous version. Most users of the applet are believed to be more familiar with the Java programming language rather than Prolog, therefore it will be more effective for them to learn how to write a controller by saving the time spent on learning Prolog.

4.1 Introducing BeanShell – www.beanshell.org

BeanShell is selected as the Java interpreter for the applet. From BeanShell's homepage [7]:

> "BeanShell is a small, free, embeddable Java source interpreter with object scripting language features, written in Java. BeanShell dynamically executes standard Java syntax and extends it with common scripting conveniences such as loose types, commands, and method closures like those in Perl and JavaScript."

The benefits of using BeanShell for the robot navigation applet are explained in the following sections.

4.2 Small Size

Size is an important factor in selecting the interpreter, because during loading of the applet's homepage, the "Start Applet" button only appears when the entire applet is loaded. For example, the prototype applet at url: http://aispace.org/robot/. If the applet takes a long time to load, users may think that there is problem with the application. The size of BeanShell interpreter is only around 280 kb, therefore it does not dramatically increase the size/loading time of the applet.

4.3 Ease of Use

BeanShell fully supports the standard Java syntax, and we use BeanShell from the application to execute Java code dynamically at run-time.

4.4 Feedback on Error for Debugging

BeenShell returns useful error messages, which in turn can be useful for our debugging trace as described in Section 3.2.5. In BeanShell, "exception handling using try/catch blocks works just as it does in Java" [7]. We can have an exception handler around each of the function calls within the controller, and based on the exceptions thrown, we can can inform the user of the programming environment what has gone wrong.

For example, for code of the `arrived` function in the middle layer of the basic controller (Figure 2.3), if one assigns a `double` value of `1.4` to `String` variable `abc`, an error message will appear as follows:

```
Error occurred at "arrived" function in the "Middle" layer:

String myString = 1.4

Target exception: java.lang.ClassCastException: Cannot cast
primitive value to object type:class java.lang.String
```

BeanShell dynamically interprets Java, so the code is not compiled and errors occur

Listing 4.1: Example of making and calling a function by using BeanShell

```
1  import bsh.Interpreter;
2
3  ....//code for class and method declarations omitted
4
5  Interpreter interpreter = new Interpreter();
6  //the concatenation function as a string
7  String concatFunctionStr =
8     "public String concatFunction(String a, String b){return a+b;}"
          ;
9
10 //handles EvalError exception
11 try{
12    //declares the function in the interpreter
13    interpreter.eval(concatFunctionStr);
14    //use and run the function
15    System.out.println(
16      interpreter.eval("concatFunction(\"left\", \"right\");"));
17 }
18 catch(Exception e){
19    System.out.println(e.toString());
20 }
21
22 ....
```

at run time after the buttons "Run Robot" or "Step Forward" (Figure 3.18) from the "Solve" tab panel are clicked.

4.5 Using the BeanShell Interpreter

If Eclipse IDE is used, the developer can import the BeanShell jar file (bsh-2.0b4.jar) as one of the "Referenced Libraries".

Then the interpreter can be used after putting

```
import bsh.Interpreter;
```

in a Java source file, as shown in line 1 of Listing 4.1.

The example in Listing 4.1 declares a function in the interpreter by passing the function as a string (lines 7,8) to the interpreter (line 13). The the function is called at line 15. The output of the code is:

```
leftright
```

Chapter 5

Back-End Software Design

This chapter describes the back–end design of the software, and for the GUI design, please refer to Chapter 3.

The design of the software is object-oriented and organized by the classes that represent Arc Variable, Function, and Controller. Each of these classes will be discussed in the following sub–sections. The design is based on Assumption 1.

Assumption 1 *Within a time step, the relationships between functions are* acyclic

This assumption is a constraint "that the graph of how one variable depends on another has to be acyclic" [8]. This constraint ensures that a function can be called when all of its inputs are available, and there will not be deadlock cycles where function *A* waits for the output of function *B* which in turn waits for the output of function *A* (Figure 5.1).

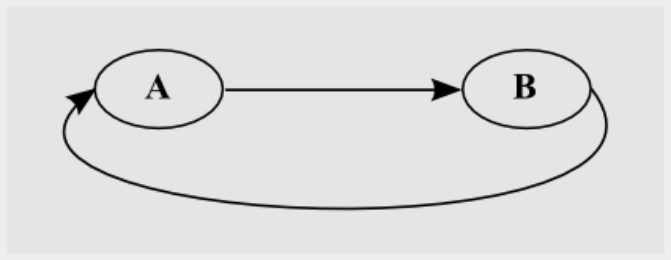


Figure 5.1: An example of a deadlock cycle.

5.1 Arc Variable

The `ArcVariable` class represents variables of the GUI arcs (Section 3.2.1) that link input/output between functions.

"GUI arcs" or simply "arcs" is a GUI feature that links functions and ports, where as an instance of `ArcVariable` contains the actual variable that is passed by a GUI arc.

The `ArcVariable` class provides all the setter and getter methods for the type and object of the variable.

5.1.1 Arc Variable Between Time Steps

Special care needs to be taken for `ArcVariable`s that link different time steps. Examples of `ArcVariable`s between time steps can be found from Figure 5.2, where GUI arcs with labels 1, 3, 4, and 5 are stored by one `ArcVariable` between time steps, and GUI arc with label 5 is stored by another `ArcVariable` between time steps.

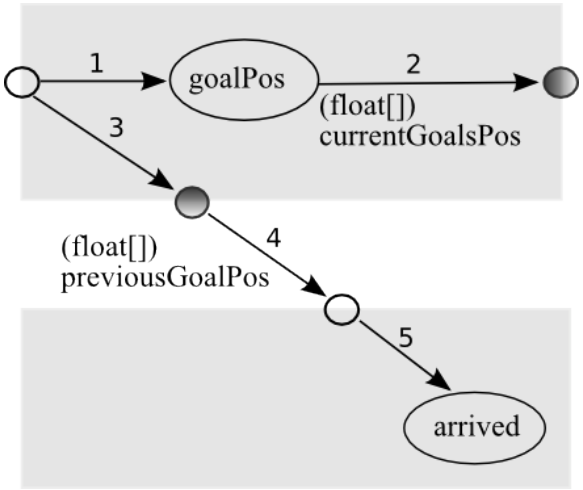


Figure 5.2: Examples of "ArcVariable" between time stes.
Extracted from Basic Controller in Figure 2.3 on page 7. Arcs with labels 1, 3, 4, and 5, although pictorially different, are represented by the same `ArcVariable` object in the back–end program. Arc with label 2, however, is stored in a different `ArcVariable` object.

An `ArcVariable` between time steps has a special boolean tag called `betweenTime`, and an `ArcVariable` between time steps is processed differently compared to other `ArcVariable`s.

5.1.2 Details of the ArcVariable class

The fields and methods of the `ArcVariable` class are documented below. Readers not interested in the detailed documentation of the class can skip this subsection.

Field Summary

The following list contains type, name and descriptions of global variables.

- `boolean` **betweenTime**: True if this `ArcVariable` is between time steps
- `ArcVariable` **previous**: If this `ArcVariable` is between time steps, and if this `ArcVariable` holds the value for the current time step (e.g. `ArcVariable` for arc with label 2 in Figure 5.2), `previous` holds the value for the `ArcVariable` from the previous time step (e.g. for `ArcVariable` of arc 2, `previous` hold `ArcVariable` for arc 1). Otherwise, `previous` is `NULL`.
- `Object` **ioObject**: The object that the `ArcVariable` is passing between functions.
- `String` **ioType**: The type of the object that the `ArcVariable` is passing between functions
- `Interpreter` **interpreter**: An instance of the BeanShell interpreter. This interpreter can be used to declare and execute the functions written by the users of the applet, e.g. in Listing 4.1, the function `concatFunction` is declared by calling`interpreter.eval(concatFunctionStr)`, where `concatFuncionStr` contains the string for the function.
- `String` **name**: The name of the `ArcVariable`.

Constructor Summary

- `ArcVariable`(String ioType, String name, Interpreter interpreter, boolean betweenTime, ArcVariable previous): Constructs a new `ArcVariable` based on the type of the object that the `ArcVariable` holds, the name of the arc, an instance of the BeanShell interpreter, if the `ArcVariable` is a *between time steps* `ArcVariable`, and if there is a matching `ArcVariable` from the previous time step (e.g. in Figure 5.2, with `ArcVariable` for arc 2, `previous` holds the `ArcVariable` for arc 1).
- `ArcVariable`(String ioType, String name, Interpreter interpreter, boolean betweenTime)
- `ArcVariable`(String ioType, String name, Interpreter interpreter)

Method Summary

The getters and setters for the global variables:

- `boolean` **isBetweenTime()**
- `void` **setBetweenTime**(boolean betweenTime)
- `ArcVariable` **getPrevious()**
- `void` **setPrevious**(`ArcVariable` previous)

Other methods:

- reset(): This method is called after every time step to reset the value of the object (`ioObject`). Special caution needs to be taken for `ArcVariable`s between time steps.
- declareIO(): Declares the object that the `ArcVariable` is holding within the BeanShell interpreter environment.

5.2 Function

The `Function` class represents a function and how it is to be executed. Detailed documentation of the `Function` class can be found from Section 5.2.5.

Definition 5 *A* function *in the programming environment obtains input parameters from the arcs linking to it, contains code that is used to process the inputs, and computes an output.*

Fields of the `Function` class include: (please refer to Figure 5.3 for an example, and the labels used below are from that figure)

- input `ArcVariable`s; (see label 1 and 2)
- one output `ArcVariable`; (label 5)
- Java code for the function; (label 4)
- the function name; (label 3)

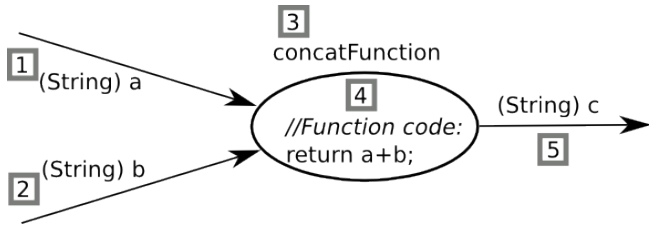


Figure 5.3: An example function before conversion to Java code. The function has two input `ArcVariable`s and one output `ArcVariable`. The function simply returns a concatenation of the two input String. User uses the design environment (e.g. editors shown in Figure 3.6, and Figure 3.11) to name the variables, and we assume that the input variables will all have different names. If the name are not unique, an error message (Figure 3.15(b)) will be shown to the user of the programming environment (as discussed in Section 3.2.2).

The `Function` class mainly put together the Java code of a function to be run by the interpreter, then runs the code in the interpreter, so that the function can be called later on.

Listing 5.1: Final Java code made from information of graph from Figure 5.3 the text for the code is stored in a String called funcJavaCode

```
1 public String concatFunction(String a, String b){
2         return a+b;
3 }
```

5.2.1 Converting a Function to Java Code

An example here will illustrate how to put code together. Let us say there is a function called `concatFunction` that concatenates two Strings together. As shown in Figure 5.3 there are two input arcs each having String type, and one output arc being String type as well.

The final code combines the knowledge from the graph shown in Figure 5.3 to produce code as shown in Listing 5.1.

5.2.2 Passing a Function to the Interpreter

Using the code generated as shown in Listing 5.1, assuming the String for the code is called `funcJavaCode`, we can let the interpreter know the function by calling

```
Interpreter interpreter;
interpreter.eval(funcJavaCode);
```

Please refer to Section 5.2.5 the method called `makeInterpreterCode()` for details on how `funcJavaCode` is made.

5.2.3 When to Run Function

Before the function can be executed, designer of the programming environment needs to make sure that all the inputs of the function are ready.

The following method ensures that all inputs are ready for a function: Except for input `ArcVariables` between time steps that just reset its value to the previous time-step, reset all other `ArcVariable`'s object to null at the end of each time-step; During the next time step, if a function has none of the input `ArcVariable` having a null object, then we know that the function has all its inputs ready, and we can run that function.

5.2.4 Using Interpreter to Run the Function

Designer of the programming environment can call the function as shown in the following code segment, and set the object of the output `ArcVariable`.

```
outputArcVariable.setIoObject((Object)interpreter.eval(methodCall));
```

The above code assumes the output `ArcVariable` is called `outputArcVariable`, and there is a setter method called `setIoObject` that sets the object of an `ArcVariable`, and the function calling string called `methodCall`, which is generated using information from graph in Figure 5.3 on page 58.

`methodCall` for our example (Listing 5.1) is `"concatFunction(a, b);"`, where the inputs *a* and *b* are named by the user of the programming environment when they created the controller.

5.2.5 Details of the Function class

The fields and methods of the Arc class are documented below. Readers not interested in the detailed documentation of the `Function` class can skip this subsection.

Field Summary

The following list contains type, name and descriptions of global variables.

- `Interpreter` **interpreter**: An instance of the BeanShell interpreter.
- `String` **code**: The code for a function of the controller. The code look like a Java method, but does not have the signature of a method. e.g. For the example from Listing 5.1, `code` would be: `return a+b;`
- `ArcVariable[]` **inputArcVariables**: An array pointing to all the input `ArcVariable`. For example, in Figure 5.3, `ArcVariable`s for arcs with labels 1 and 2.
- `ArcVariable` **outputArcVariable**: The output `ArcVariable`. For example, in Figure 5.3, the `ArcVariable` for arc with label 5.
- `String` **name**: The name of the function.

Constructor Summary

- Function(ArcVariable[] inputArcVariables, ArcVariables outputArcVariable, String code, String name, Interpreter interpreter): Constructs a new `Function` based on the given input `ArcVariables`, one output `ArcVariable`, the textual code for the function, the name of the function, and an instance of the BeanShell interpreter.
- Function(String code, String name)

Method Summary

The getters and setters for the global variables:

- `String` **getCode**()
- `void` **setCode**(String code)
- `ArcVariable[]` **getInputArcVariables**()
- `void` **setInputArcVariables**(ArcVariable[] arcVariables)
- `String` **getName**()
- `void` **setName**(String name)
- `ArcVariable` **getOutputArcVariable**()
- `void` **setOutputArcVariable**(ArcVariable outputArcVariable)

Other methods:

- `boolean` **getHasAllInputs**(): finds out if the function has all its input are ready from the input `ArcVariables`. This information is useful in determining if a function is ready to be run by the interpreter.
- `String` **makeInterpreterCode**() creates Java code used for interpreter to run the function code.

- `void` **declareFunction**(): declares the function with the code made by `makeInterpreterCode()` in the BeanShell interpreter.
- `Object` **execute**(): executes the function and returns the object returned by executing the function in the interpreter.
- `boolean` **isLastFunction**(): detects if the last function in the controller is being executed. This assumes that there is always a function in the controller called `changePos()`, and it is the last function to be executed. `changePos()` specifies the change in location of the robot (examples `changePos()` can be see in the basic controller (Figure 2.3) and VI controller Figure 2.5). The designer of programming environment can change how this method is currently implemented to let the user of the applet modify this, so that it is not hard coded.

5.3 Controller

The `Controller` class overlooks the operations between `ArcVariables` and `Functions` and keeps track of how many time-steps have elapsed.

The main functionality offered by the `Controller` class is to run the controller one time-step at a time. For each time-step, the controller attempts to run all the necessary functions.

Firstly, to run a function, we find out if all the input `ArcVariable` of the function are set and ready, then we run the function in the interpreter, so that the object of the output `ArcVariable` can be set the the appropriate object returned from the interpreter. Each function only needs to be run once, since the graph is acyclic as stated in Assumption 1.

We know all the functions are executed after the function called `changePos` is executed, because `changePos` is the final function which gets the change in position of the robot.

Finally, once all the functions are executed, with the exception of some of the `ArcVariables` between time steps, we can reset all the `ArcVariables` by setting the object of the `ArcVariables` to null.

5.4 Controller Application

The final class that brings together user input such as the graph shown in Figure 2.3 and the previously discussed classes (`ArcVariable`, `Function`, and `Controller`) is the Controller Application class. This class creates all the `ArcVariable`s and `Function`s from the user input, and makes use of the `Controller` class to run the controller.

5.5 The Absence of Layer Class

As the reader may have already noticed, there is no mentioning of a `layer` class in this section. "Layer" is only a concept that is used to help the user of the applet to conceptualize the hierarchical controller but is not necessary in the back–end. The "Layer" concept should instead be present in the user interface part of the design, and a "layer" in the UI should contain and distinguish between input/output ports. For example, in Figure 3.8 on page 34, an input port is shown in hollow circle, and output port is shown as filled circle.

Chapter 6

Conclusion

In this design document, we have described a proposed new robot navigation applet. We proposed the layout of the GUI based on factors such as a Java implementation for a controller, arbitrary number of layers for a hierarchical controller, and the revamped debugging functionalities. The users should have all the necessary tools to assist them to design a hierarchical controller, and all the information to help them to understand what's going on inside the controller during the execution of the simulated environment.

The basic controller from the prototype controller is tested with the new applet, furthermore, we have designed and implemented a hierarchical controller based on value iteration which reuses many elements from the basic controller. We hope the user can appreciate the ease to change an existing controller to accommodate a more complex behavior of the robot. The value iteration based controller is intelligent enough to avoid certain traps.

In Chapter 4, we have analyzed the benefits of using BeanShell as the Java interpreter such as the small size (Section 4.2), ease of use (Section 4.3), and BeanShell's ability to provide feedback on error for debugging purposes (Section 4.4). We explained how to use BeanShell for this applet and provided code to demonstrate on its use.

Finally, we provided a detailed explanation of the back–end design for the applet. The design is object–oriented, and so the description of the design was based on the classes such as `Arc`, `Function`, `Controller` and `Controller Application`.

Bibliography

[1] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957. → pages 8

[2] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. URL http://jmvidal.cse.sc.edu/library/dijkstra59a.pdf. → pages 8

[3] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*, chapter 16, pages 411–433. Morgan Kaufmann Publishers, draft edition, June 2003. → pages 8

[4] L. Kaelbling, M. Littman, and A. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996. → pages 8

[5] B. Knoll, Kisyński, G. Carenini, C. Conati, A. Mackworth, and D. Poole. Aispace: Interactive tools for learning artificial intelligence. In *Proceedings of the AAAI 2008 AI Education Workshop*, Chicago, IL, July 2008. URL http://www.aispace.org/papers/Knoll2008.pdf. → pages 3

[6] A. Mackworth and Y. Zhang. A formal approach to agent design: An overview of constraint-based agents. *Constraints*, 8(3):229–242, 2003. ISSN 1383-7133. doi:http://dx.doi.org/10.1023/A:1025697810124. → pages v, 1, 2

[7] P. Niemeyer. Beanshell's homepage, October 2008. URL www.beanshell.org. → pages 51, 52

[8] D. Poole and A. Mackworth. *Artificial Intelligence: Computational Foundations of Intelligent Agents*. Unpublished, 2008. → pages 2, 8, 54

[9] H. Ren, A. A. Bitaghsir, and M. Barley. Safe stochastic planning: Planning to avoid fatal states. In *Proceedings 3rd International Workshop on Safety and Security in Multiagent Systems (SASEMAS), 5th International Joint*

Conference on Autonomous Agents and Multiagent Systems, Hakodate, Japan, May 2006. → pages 10