

Using Artificial Neural Networks in the Visual Programming of Autonomous Robots

Shawn M. Best

Philip T. Cox

Abstract

The use of Artificial Neural Networks (ANNs) to control autonomous robots has been quite extensively studied. Also, in recent years researchers have begun to investigate the notion of programming such robots using visual programming languages based on various programming and robot control models. Some of this work has focused on developing visual programming-by-demonstration (PBD) systems.

Here we extend the latter approach by proposing a visual PBD environment for autonomous robots based on ANNs. Within this environment, sensor-to-motor rules, called sensorimotor maps, are programmed by employing ANNs to match sensor outputs to actuator inputs. The goal is to create a programming environment in which the end-user is not required to have any knowledge of the underlying control model, ANN programming in this case. In this regard, the current proposal appears more promising than previous attempts using the subsumption model

I Introduction

Visual language designers have at their disposal a growing array of tools with which they can assess *a priori* the effectiveness of a proposed language [3, 8]. One of the most frequently used is a list of criteria, the “cognitive dimensions,” proposed by Green and Petre [8]. Many of the criteria in this catalogue relate to the notion of “concreteness” — making explicit, directly observable or manipulable, the concepts that the programmer or problem solver has to deal with. In particular “closeness of mapping” stands out. A language scores well in this dimension if it provides objects and constructs that mirror the objects and constructs of the problem domain. One implication of this is that a problem domain which is itself concrete is a prime candidate for a language that maps closely to that domain. For example, a “smart home” language for building systems to control appliances, heating, air conditioning, telephone systems, locks and so forth might include graphic representations of these items embedded in networks expressing their dependencies and functions.

The concrete domain we focus on here is the control of free-ranging autonomous robots, examples of which range from the highly sophisticated rover vehicles currently exploring Mars (<http://marsrovers.jpl.nasa.gov/home/index.html>) to the simple robots that can be built with LEGO Mindstorms kits.

The behaviors of a robot can vary from simple reactive ones that accomplish simple low level tasks such as path-following, to high level ones requiring deductive skill. Low-level behaviors provide the foundation for the higher-level ones, and can be implemented with simple, efficient control models requiring modest resources. It is this kind of robot behavior that we are interested in here.

The research reported here lies at the intersection of several established fields: visual programming, end-user programming, programming by demonstration, robot control and artificial neural networks. It is a continuation of previous work by Cox *et al.* [5, 6], the goal of which was a visual programming environment in which the user could implement reactive behaviors for autonomous robots by interacting with a simulated robot in a simulated environment, with as little knowledge of, or interaction with, the underlying control model as possible. The results fell somewhat short of this goal primarily because the underlying control model, the subsumption architecture due to Brooks [4], could not be completely hidden from the user.

In the following we take a similar approach to [5] but using Artificial Neural Networks (ANNs) instead of subsumption. As a result, in the programming environment we propose, the details of the control model are far less intrusive.

I.1 Related work

Although programming robots of various kinds has been intensively studied for many years, applying visual programming techniques to the problem is a relatively recent idea. Most of the current research in this area has arisen from the visual robot programming competition held at the 1997 Visual Languages Symposium [1]. A brief summary of the most closely related ones follows a short description of subsumption below.

Brooks’ subsumption model consists of parallel behaviors implemented as finite state machines (FSM) in a hierarchy in which higher level behaviors have priority over lower-level ones. Higher level behaviors inhibit the output signals, or suppress the input signals, of lower-level ones, if several behaviors attempt to control the same actuator.

Altaira [12] is a rule-based visual language for programming reactive robot controllers. The programmer defines robot behaviors by building transformation rules using visual

representations of the robot and environment. Although its underlying model is a version of Brooks' subsumption model, its rule syntax is heavily dependent on the particular robot for which it was designed, so it is difficult to see how it could be expanded into a general robot programming environment.

More recently, Pfeiffer *et al.* have developed Isaac [13], a visual language in which rules are triggered by sensors, and affine transformations are applied to compute displacements of objects internal and external to the robot. These displacements are then mapped to the actuators to create a response. The rules that govern behaviors may become unmanageable as the complexity and number of the behaviors increase.

Cox *et al.* have developed three visual language systems for programming autonomous robots. The first, Visual Subsumption Language, implemented a very simplified version of the subsumption model in a forms-based interface incorporating simple data flow diagrams to program FSM transitions [7]. Visual Behavior Based Language provided an accurate visual representation of the full subsumption model [6], consisting of FSM graphs and message flow graphs to capture the network of FSMs, suppressors and inhibitors.

In [5], a two-part system was proposed, using programming-by-demonstration (PBD) techniques in order to achieve a close mapping to the problem domain. The first part, the Hardware Definition Module (HDM), is used for building simulated sensors, actuators and other parts, and assembling them into a simulated robot and simulated environment. In the second part, the Software Definition Module (SDM), the user builds subsumption-based control programs by interacting with the simulated robot. SDM was only superficially described, and dealt only with the FSM level of subsumption. In [2] Banyasad fleshed out the definition of SDM, providing a more suitable formulation of the subsumption model, and details of the FSM-building interface. He also extended the programming-by-demonstration process to allow hierarchies of behaviors with inhibition and suppression to be built.

Although this system achieves its goal to some extent, its interface does not completely hide the underlying subsumption model. The user is forced to deal with details such as creating and labelling FSM states, and drilling down through hierarchies of lower level behaviors in order to suppress or inhibit values. These seem to be inevitable consequences of the subsumption model.

The use of ANNs for robot control has been studied for some time. For instance, Mitchell and Keating [11] developed simple mobile robots that use unsupervised ANNs in a reactive controller. These robots learn to avoid obstacles through trial and error by interacting with their environment. Low *et al.*, [5] developed a simulation environment that employs unsupervised ANNs in conjunction with other mechanisms, to support obstacle avoidance and goal orientation

for a robot in a dynamic environment. The ANN is used to fine-tune actuator output at the lower level of the architecture. Kostelnik *et al.* [9] describe a simulation environment for mobile LEGO robots that include the use of unsupervised ANNs to cluster the output signals from sensors in order to generate semantic knowledge of the environment. The control architecture also uses supervised ANNs, called multilayered perceptrons, with Q-learning (a reinforcement learning algorithm) for the behaviors associated with the subsumption-based controller.

The goal and organization of these architectures provides a framework for self-learning robots.

Our work has the same goal as [5] and takes a similar architectural approach, two modules (HDM and SDM) achieve generality by allowing a wide range of simulated robots to be built, and closeness-of-mapping, by allowing the control program to be constructed by directly manipulating the simulated robot. However, in order to rectify the problems with the previous system, described above, we have explored the use of a control model based on ANNs

2 Background

The class of robots we consider consists of machines with actuators and sensors. An actuator is a device with which the robot can affect its environment. A sensor is a device via which the robot can gather information about its the environment. A robot has a body to which actuators and sensors are attached.

An ANN is composed of a number of computing units called *nodes* that are interconnected by *synaptic weights*. The network of nodes forms a parallel processor that adapts in response to *input vectors*. Learning is achieved by changes in the synaptic weights that connect each node to its neighbors. The activation of nodes is governed by an activation function, which can be either linear or non-linear, and produces an output signal that is propagated in parallel to neighboring nodes. The activation function that contributes to the resulting signal, called an *output vector*, determines whether the network is best suited for linear or non-linear classification problems.

ANNs are categorized into different taxonomies based on their computational capabilities, which includes their computational dynamics and architectural configurations [13]. A *single-layered perceptron* consists of an input layer of nodes that is fully connected to an output layer of nodes. Perceptrons are linear classifiers, so they are suited to classifying linearly separable patterns in a problem domain. Supervised learning techniques are used to train perceptrons. Training is an iterative process that involves presenting the ANN with an input vector, from which it generates an output vector, and a desired output vector, known as a *class*. If the actual output and desired output do not match, an error-correction algo-

rithm is used to propagate weight changes from the output layer back to the input layer. When presented with new input vectors, a perceptron uses *generalization* to classify unknown patterns to their best matching class.

Multilayered perceptrons have at least a depth of two since they have at least one layer of nodes between the input and output layers. Multilayered perceptrons are particularly useful for classifying patterns that are not linearly separable in a problem domain, and are considered universal classifiers. For multilayered perceptrons, a back-propagation algorithm is used to propagate weight changes through to the hidden layers in response to training.

In the following, we use perceptrons that are automatically transformed from single-layered perceptrons to multilayered perceptrons as the classification problem reaches non-linearity, which is likely to occur as the number of desired outputs increases.

3 Robot programming environment

The robot programming environment we propose here includes the HDM described in [5] together with a similar SDM, but in our case, based on ANNs. We will describe the proposed programming environment by leading the reader through a worked example of its use, demonstrating how the SDM would be used to build a control program for a robot car to follow a centre-line on a track, to change direction after detecting an obstacle, and to wander randomly in search of the track markings if for some reason the robot has “lost sight” of them. This environment provides a simulation of the robot interacting with a simulated environment. For simplicity, we restrict our attention to two-dimensions, but the simulation environment and other concepts we describe could clearly be generalized to 3D.

Our example is similar to that used in [5], namely, a world consisting of sections of track traversed by a small, wheeled robot similar to ones that could be built using LEGO Mindstorms robotics kits (<http://mindstorms.lego.com/>). The robot has two wheels each with its own motor, one on either side at the back; two odometer sensors, one at each wheel; one unpowered wheel at the front that swivels like a furniture castor, an array of three infrared sensors across the underside for watching the track, and two bump sensors at the front. Each odometer has variable resolution, which means that we can set it so that its value changes in the specified increments.

3.1 Line following

In our first example we show how Line Following behavior is built in this system. We want to program the robot to drive continuously along the track using the black centre line as a guide.

To begin, we create a new workspace in the SDM by selecting an appropriate menu item, and name it **LEGO Car**. The

workspace has a frame, initially empty, called **Virtual Environment** (Figure 2) in which the simulation will take place. Next, via an appropriate menu selection, we open a floating palette called **HDM Palette** (see Figure 1) which contains objects such as track pieces previously defined in the HDM, and drag ones we need into the **Virtual Environment** panel. In our example, we arrange track pieces to create the environment shown in Figure 2, a cyclic track with no intersections and two obstacles which we will use in later examples. Created environments can, of course, be saved and loaded.

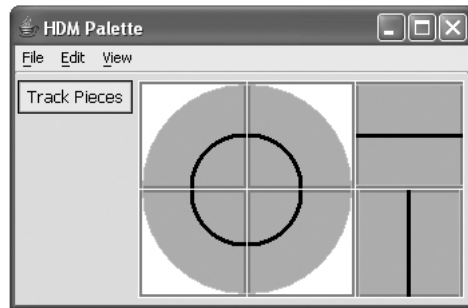


Figure 1: HDM Palette with track pieces

Now we drag the car icon from the **HDM Palette** into the **Virtual Environment** panel. This initiates the programming process, causing a new frame called **Behavior Builder** to appear at the left end of the workspace window as shown in Figure 2. We close **HDM Palette** since it is no longer required.

The **Behavior Builder** provides an interface to create new behaviors and to manipulate existing ones by specifying how they interact with one another, as we shall see later. By selecting a menu item, we open another window called **Control Panel**, which contains various playback controls that we will use later to control the robot’s interactions with its environment. At this point, since no behaviors have been defined on the **Behavior Builder** frame, the **Control Panel** buttons are disabled.

To begin training the robot, we click in the empty **Behavior Builder** panel. This creates a new behavior, represented as an empty text box into which we enter the name “Line Following” and terminate typing by pressing the return key. As soon as the new behavior is named, a floating window entitled **Robot Abstraction** opens, displaying the robot configuration as an enlarged overhead view of the robot with all the significant components, that is sensors and actuators, greyed out. At this point, the environment appears exactly as depicted in Figure 2. By clicking on them, we select the sensors and actuators we will need in order to follow the centre line of the track; namely, the three infrared sensors and the two motors. Components that are not selected remain grey. We also need to choose values to be output to the motors.

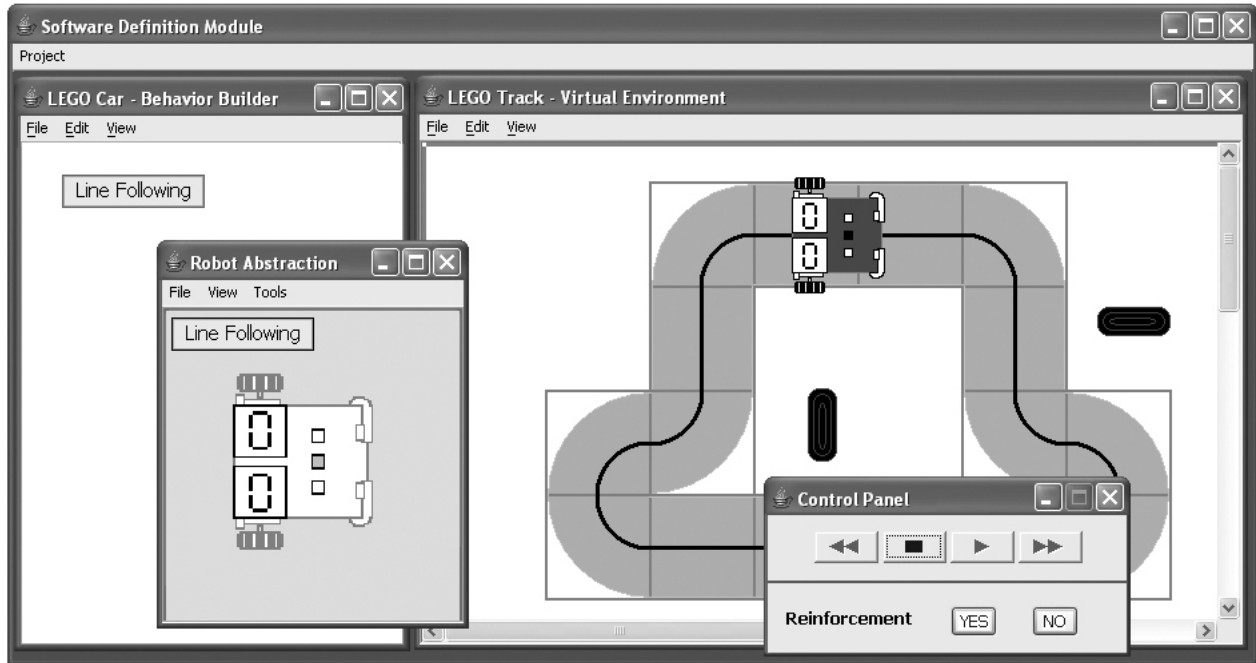


Figure 2: Software definition module with construction elements for Line Following

Since the robot is currently centred on a straight section of track, we choose 5 for both motors so the car will move straight ahead. Note that, because the **Robot Abstraction** window is open, the controls in the **Control Panel** are disabled, since the simulation cannot run while we are adjusting the settings of the robot.

The act of creating a new behavior, described above, also creates a corresponding ANN, which will compute a function for all possible combinations of sensor values. However, since it initially has no user-specified training data, all its sensorimotor maps are *generalizations*; that is to say, the function it computes is arbitrary. By setting the motor values to 5 given the sensor values $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$, we have begun the training process, and the ANN will modify its weights as described in Section 2 to ensure that the output function includes this particular *trained* sensorimotor map.

Next we close the **Robot Abstraction** window and start the simulation by clicking the **▶** button on the **Control Panel**. The robot moves forward at a speed of 5. When it encounters the right curve, the sensor inputs will change to $\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$, the ANN will compute corresponding motor outputs and the robot will behave accordingly; however, because this computation does not correspond to an input-output combination specified by the user, an alarm will sound and the **YES** and **NO** buttons in the **Control Panel**, which have been inactive to this point, will flash. If we approve of the robot's behavior, we click **YES**, which reclassifies the current sensorimotor map from *generalized* to *trained*, and the simula-

tion continues. If we click **NO**, the simulation stops, resets to the point where the sensor values changed, and the **Robot Abstraction** window appears again with the current values for sensors displayed. We provide appropriate values for the motors, dismiss the **Robot Abstraction** window, and resume the simulation.

At any point in the training process, we can view existing sensorimotor maps by selecting a menu item on the **Behavior Builder** frame, opening a new window called **Sensorimotor Maps** which is separated into two panels as shown in Figure 3. This window contains two panels labelled **Trained** and **Generalized** which respectively contain the sets of trained and generalized sensorimotor maps. The union of these two sets is the input-output function computed by the current state of the ANN. The generalized sensorimotor maps are drawn in grey to emphasize the fact that they may not be correct. If we spot a generalized map that we know is correct, we can drag it into the **Trained** list, thereby reducing the time required to build the list of trained maps associated with a behavior. Note that this reclassification by dragging can be done while the simulation is running, and that if we wish to correct earlier training errors, we can drag maps from the **Trained** panel to the **Generalized** panel.

There are, therefore, three ways that a map can become classified as trained. The user can explicitly specify it via the **Robot Abstraction** window, accept it by clicking the **YES** button during simulation, or drag it from **Generalized** to **Trained** in the **Sensorimotor Maps** window. Only the first of

these three actions requires that the ANN be updated. If the **Sensorimotor Maps** window is open at the time, the system updates it by recomputing generalizations; otherwise the recomputation occurs when the window is next opened.

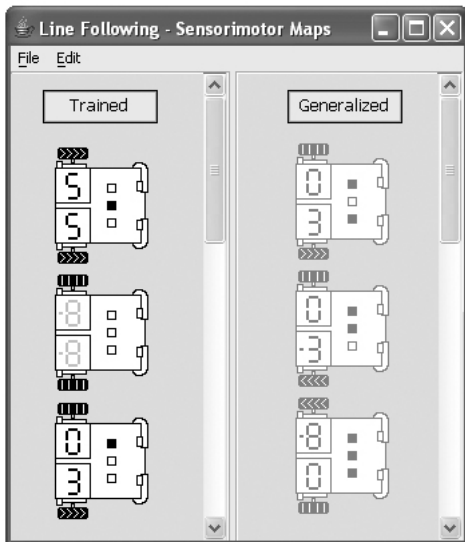


Figure 3: Sensorimotor maps window

It is important to note that we may consider a behavior to be completely specified, even if some sensorimotor maps are not trained. For example, during the programming of Line Following, we know that the robot is off-course as soon as the middle sensor turns off and one of the outer ones turns on, and take immediate steps to correct it. Hence when the robot is finally following the track without intervention, there will be many sensor patterns which will not have arisen, and therefore many maps not trained. Hence, when we are fully satisfied with the behavior we have programmed, we *finalize* it by choosing an appropriate menu item. This causes the training of the ANN to be finalized as follows. For each combination X of input values corresponding to a generalized map, train the ANN with input X and output \otimes for each actuator. The value \otimes signifies “no value”, meaning that the behavior sending this value to an actuator is not attempting to control the actuator.

3.2 Collision Detection

In our second example we add a collision detection behavior, which determines the robot’s actions should it encounter an obstacle on the track. In this case, the robot should perform a series of distinct actions; stop, back up a certain distance, about face. The components involved in these behaviors are the bump sensors, the odometers, and the motors.

With the Line Following behavior complete and the simulation stopped, we add a new behavior named “Collision Detection”. The **Robot Abstraction** window appears as

before, and we select the bump sensors and motors, since these are the only devices required for the first action, stopping. Since the bump sensors are currently not activated, we set the motor outputs to \otimes .

After closing the **Robot Abstraction** window, we move one of the obstructions that we placed in the environment earlier, on to the track where the robot will encounter it. We start the simulation, and the robot, under control of the Line Following behavior, starts moving along the track and encounters the obstruction, triggering the bump sensors. As before, we are warned that the ANN has started to use a generalized map. This time, however, the behavior we observe results from Line Following, since it is using a trained sensorimotor map, and trained maps have priority over generalized maps which send output to the same actuators. Hence the robot will drive through the obstruction, illustrating the value of using a simulation rather than a real robot.

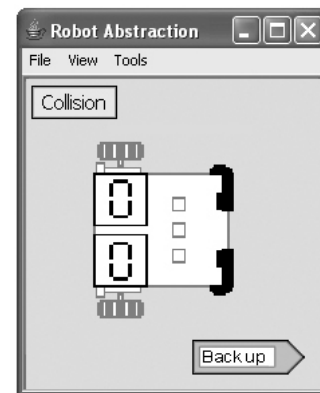




Figure 4: Collision Detection

We click the button. The simulation resets to the point where the bump sensors were activated, and the **Robot Abstraction** window opens displaying the robot with bump sensors turned on. We set the motors to 0, and via a menu selection, add the icon  to the window, then type **Backup** in its text box (see Figure 4). This icon is the system actuator GOTO, the value of which is the name of a behavior to which to transfer control.

On closing the **Robot Abstraction** window, we see that the **Behavior Builder** appears as in Figure 5, where the icon  is a *subsumption* which will transmit the value from its left input to its output whenever the top input is \otimes , and otherwise will output the value of its top input.

When we resume simulation, the Collision Detection behavior outputs 0 to the motors, which subsumes the output from Line Following, thereby causing the robot to stop. It also outputs Backup to GOTO. Since no Backup behavior exists, the simulation stops, a new behavior named Backup is

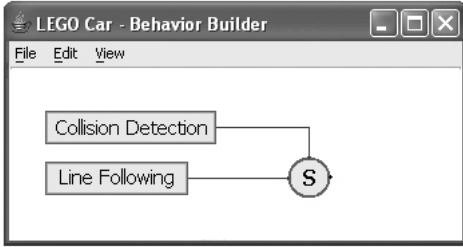


Figure 5: Subsumption generation

automatically created in the **Behavior Builder** frame, and the **Robot Abstraction** window opens. We select the odometers, which initially read 0, and set their resolution to 2. We select the motors, set them to -2 (see Figure 6) and close the window. When the simulation is started, the robot backs up till the odometers change (a distance of 2), at which point we set the motors to 0, and add a GOTO actuator and give it the value “Turn”, to initiate the third and last action in the sequence. Note that this adds the GOTO to the previously trained map, with the value \otimes .

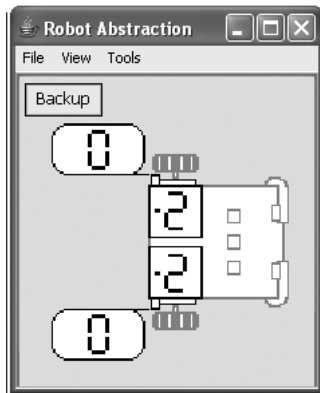


Figure 6: Backing up

Once the Turn behavior has been created and programmed in a similar manner, we finalize it, which finalizes all three behaviors in the sequence. At this point, the **Behavior Builder** frame contains the three behaviors that achieve obstacle avoidance, chained together and subsuming Line Following, as shown in Figure 7.

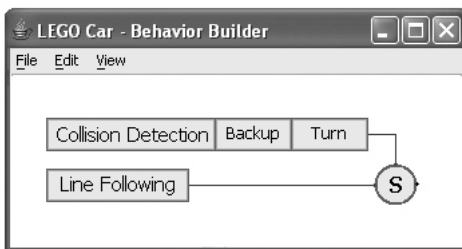


Figure 7: Behavior sequence

Our robot will now follow the track, and if it meets an obstacle, it will stop, back up, turn around and resume Line Following. However, the process of backing-up and turning may well leave the robot off the track, especially if the obstacle is on a corner. In the next section we introduce a final behavior to remedy this problem.

3.3 Wandering

The wandering behavior consists of an action, chosen randomly from “turn clockwise”, “turn counter-clockwise” and “move forward”.

We place the robot off the track, define a new behavior called “Wander”, and in the **Robot Abstraction** window select the odometers and motors. Via an appropriate menu selection, we add a random generator, a system actuator represented by \textcircled{R} . A configuration dialog opens, in which we specify that the random generator should produce a number in the range $[0,2]$ every 5 seconds. (Figure 8). The box to the

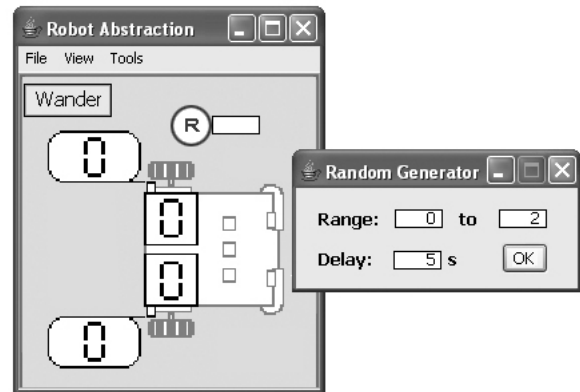





Figure 8: Random generator and configuration dialog

right of the random icon displays its value. When the configuration dialog is closed, one random number is generated and displayed. We suppose that in this case, it generates 0, and arbitrarily decide that this value indicates a clockwise turn. Accordingly, we set the right motor to -2 , the left to 2, and the resolution of the odometers to 3. When the simulation is started, the robot turns clockwise until the odometers register 3.

Simulation and programming continues in the usual way, defining the sensorimotor maps for the other two values of the random sensor.

Note that, at any time during this process, the robot may run into an obstacle or across the track markings, which will cause either the Collision Detection or Line Following behaviors to start producing motor values not equal to \otimes . If this happens, the current sensorimotor map of the Wander behavior must be trained, since otherwise we would have been asked to accept or reject the current action before the obstacle or markings were encountered. Hence we have two trained

maps competing for control of the same actuators. In this situation, the simulation stops, and the dialog shown in Figure 9 opens. The panels labelled Action A and Action B correspond to the two conflicting sets of actuator inputs. Let us assume that A corresponds to the outputs from the existing Collision Detection and Line Following behaviors. If we click the  button for A, the configuration of behaviors is first temporarily altered by inserting a subsumption so that Collision Detection and Line Following will override Wander, then the simulation continues. We can stop it by clicking the  button for A, which resets the simulation. Similarly we can “test drive” the other behavior by clicking the  for B. Once we have decided which is the correct choice, we click the corresponding button. In this example we would choose A, so that Collision Detection and Line Following subsume Wander.

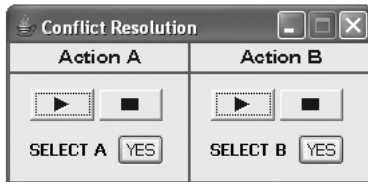


Figure 9: Resolving conflicts

If we were to attempt to finalize Wander before it had encountered the situation outlined in the previous paragraph, the system would refuse, and tell us that we must continue to test the behavior until such conflicts arise. It would, of course, be more helpful than that, and tell us what situations we should explore: for example, “position the robot so that the IR sensors are activated.”

The final configuration of the three behaviors is shown in Figure 10.

4 Concluding remarks

We have described a visual programming environment for building control programs for autonomous robots, using PBD techniques. Our goal is a system in which the user can program by directly interacting with a simulated robot, with little or no knowledge of the underlying control model.

Our proposed programming environment constructs control programs consisting of collections of ANNs, each implementing a behavior, and subsumption nodes that resolve conflicts when two ANNs attempt to control the same actuator.

A careful inspection of the steps involved in building a program shows that at no point is the user required to know the details of the control model. The ANNs are completely hidden, and the only view the user has of the control structure is a window displaying the behavior and subsumption icons and connections between them. However, except for

clicking in this window to create a new behavior, the user does not have to interact with it. Subsumptions, connections and behavior sequences are all created automatically, as a consequence of the user’s direct manipulations of the robot and environment.

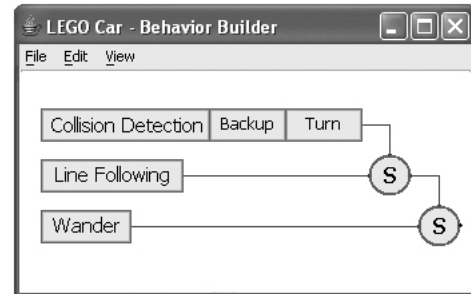


Figure 10: Three-layered control

5 Acknowledgements

This work was partially supported by Natural Sciences and Engineering Research Council of Canada Discovery Grant OGP0000124.

6 References

- [1] A. L. Ambler, T. Green, T. D. Kimura, A. Reppenning, T. Smedley, 1997 Visual Programming Challenge Summary, *Proceedings of the 1997 IEEE Symposium on Visual Languages*, Capri, 1997, pp. 11-18.
- [2] O. Banyasad, *A Visual Programming Environment for Autonomous Robots*, MCS Thesis, Dalhousie University, 2000. (<http://www.cs.dal.ca/~pcox/theses/OBanyasad.pdf>)
- [3] A.F. Blackwell, First steps in programming: A rationale for Attention Investment models, *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, Washington DC, 2002, pp. 2-10.
- [4] R.A. Brooks, A Robust Layered Control System for a Mobile Robot, *IEEE Journal of Robotics and Automation*, RA-2(1), 1986, pp. 14-23.
- [5] P.T. Cox, and T. Smedley, Building Environments for Visual Programming of Robots by Demonstration, *Journal of Visual Languages and Computing*, 11(5), Academic Press, 2000, pp. 549-571.
- [6] P.T. Cox, C. Risley, T. Smedley, Toward Concrete Representation in Visual Languages for Robot Control, *Journal of Visual Languages and Computing*, 9(2), 1998, pp. 211-239.
- [7] P.T. Cox, J. Garden, M. McManus, T. Smedley, Experiences with Visual Programming in a Specific Domain —Visual Language Challenge ’96, *Proc. of Symposium on Visual Languages*, Capri, Italy (Sept 1997), 254-259.

- [8] T.R.G. Green, and M. Petre, Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework, *Journal of Visual Languages and Computing*, 7(2), 1996, pp. 131-174.
- [9] P. Kostelnik, M. Hudec, and M. Samulka, Distributed Learning in behavior Based Mobile Robot Control, *Intelligent Technologies - Theory and Applications*, Eds. P. Sincak et al., IOS Press, 2002.
- [10] K. H. Low, K. Leow, and M. Ang Jr. A Hybrid Mobile Robot Architecture with Integrated Planning and Control, in *Proceedings of 1st International Joint Conference on Autonomous Agents & MultiAgent Systems (AAMAS'02)*, vol. 1, 2002, pp. 219-226.
- [11] R.J. Mitchell, and D. A. Keating, Neural Network Control of a Simple Mobile Robot, in *Concepts for Neural Networks, a survey*, Eds. Landau, L.J. and Taylor, J.G., Springer Verlag, 1997, pp. 95-108.
- [12] J.J. Pfeiffer, Jr., Altaira: A Rule-based Visual Language for Small Mobile Robots, *Journal of Visual Languages and Computing*, 9(2), 1998, pp. 127-150.
- [13] Pfeiffer J., Vinyard R., and Margolis B. A Common Framework for Input, Processing, and Output in a Rule-Based Visual Language, *IEEE Symposium on Visual Languages*, 2000, pp. 217-224.
- [14] J. Sima, and P. Orponen, "A Computational Taxonomy and Survey of Neural Network Models", *Neural Computation*, 12(12), 2001, pp. 2965-2989.