Visual Programming for Robot Control

Philip T. Cox   Trevor J. Smedley

Faculty of Computer Science, Dalhousie University, Halifax, Canada

Abstract

The primary advantage of visual programming languages is that they directly represent the structure of algorithms and data, thereby enhancing the programmer's ability to build and comprehend programs. If the programming domain consists of physical objects with observable behaviour, such as a robot and its environment, then incorporating the obvious visual representations of these objects directly into the programming process may further increase the programmer's effectiveness and accuracy.

We propose a robot programming system consisting of two parts; a definition module with which to describe the structure, function and visual representation of a specific robot, and a programming module that uses this description to enable programming by direct manipulation. We describe the visual editors that constitute the first of these modules, discuss the underlying structure generated by it, and briefly show how this structure can be used in the second module.

1 Introduction

The primary advantage of visual programming languages is that they provide direct representations of software structures such as algorithms and data. This is in contrast to traditional textual programming languages, where such multi-dimensional structures are encoded into one-dimensional strings according to some intricate syntax. Visual languages remove this layer of abstraction, allowing the programmer to directly observe and manipulate complex software structures. Such directness of representation, termed "closeness of mapping", is seen by Green and Petre as an important factor in enhancing the programmer's ability to build and comprehend such structures [10], and is well supported by practical experience, for example [13, 21].

While some software development systems fully address the visualisation of algorithms [18], there are many that provide visual representations of data and processes in specific domains. One obvious example is the tools some development systems provide for building graphical user interfaces and other common application parts [2, 17, 18, 24]. These tools usually consist of a library of classes that implement the functionality of the domain (e.g. windows, menus), together with WYSIWYG editors for viewing and manipulating concrete representations of instances of these classes. Some of these systems also provide some visual programming capability directed specifically at the target domain: for example, message-flow programming, which is well suited to the kinds of interactions that occur between interface elements or between interface elements and data repositories [11]. Another example of domain-specific visual programming is represented by tools for building telephony software.

This work was partially supported by NSERC grants OGP0121312 and OGP0000124.

It appears that although visual representation of algorithms can significantly enhance the coding task, the process of building domain-specific software is likely to be more efficient and productive if the programming is done via direct manipulation of well-understood, concrete representations of domain-specific entities.

In recent work on visual languages for programming robot control, we explored this principle in a sequence of experiments moving from general visual programming towards more direct visualisation [6]. The robot used as the target consists of a Programmable Brick [20], also referred to as a "handyboard" embedded in a LEGO car. The handyboard reads signals from two touch sensors mounted on bumpers at the front of the car, and five infrared sensors symmetrically placed across the underside, three grouped together in the centre, and two at the outside edges. The handyboard controls two motors at either side of the car. The vehicle runs on a LEGO track made up of straight sections, curved sections, and crossings, all suitably coded with black tape to be read by the sensors.

The final system reported in [6], Visual Behaviour-Based Language (VBBL), is based on the subsumption architecture for robot control due to Brooks [3]. The top level of a subsumption model control specification consists of a network of behaviours connected by message-flow links, where behaviours are defined by finite state machines (FSM). In VBBL these message-flow diagrams and FSMs are explicitly represented and edited. However, even though these constructs are directly related to robot control, for programming a *particular* robot VBBL is still too abstract since it supplies no representations of the actual robot.

As a result of our experiences with VBBL, our current goal is a robot control system which is not only general enough to apply to any robot, but allows as much of the programming as possible to be done by directly manipulating representations of the robot under consideration. Clearly, in order to satisfy these apparently contradictory criteria, we will need more than just the traditional programming language embedded in a software development system. One possibility is a system consisting of two parts: first, a hardware definition module (HDM) for defining the structure and function of a robot, and a simulated environment; and second, a software definition module (SDM) which uses the model constructed in HDM to provide various capabilities, including the building and executing of robot control programs.

In the following, we explore this idea, concentrating on HDM, then show how the model output by HDM could be used in SDM for programming robot control by direct manipulation. First we give a brief survey of robot control, taken from [6].

2 Robot Control

Robots divide into two categories. The first category is exemplified by industrial robots such as those used in assembling products, or in analysing samples in a laboratory. Such robots have effectors but no sensors, perform repetitive and possibly quite complex sequences of actions, always operate in a fully defined environment and require no problem-solving capabilities. They are normally programmed using derivatives of standard programming languages possibly combined with direct “recording” of skilled operator movements, for example RAPL-3 [8], a derivative of structured Basic.

Robots in the second category have sensors as well as effectors, and operate in environments which are partly unknown and possibly changing. Such a robot senses various values in its environment and computes values for its effectors in order to attain some goal. A simple example is the LEGO car used in our challenge problem: complex examples are the Mars Rover [23], and autonomous submarine vehicles [12]. Such robots must exhibit reactive behaviour coupled with some form of problem-solving capability. This can be achieved by simply programming sensor/effector feedback using an ordinary programming language: however, such an approach would require a control program for each new problem-solving activity to be built “from the ground up.”

Much research has been devoted to problem-solving systems for robots with feedback. One of the earliest applications of logic to robot control resulted in the plan-representation language STRIPS developed for Shakey the Stanford robot [9]. More recently, possibly encouraged by the considerable progress in efficient execution of logic programs, logics have been developed that deal with real-time events generally [15], and robot control specifically [5, 16].

The established logic-based planning approach to the control of free-ranging robots was challenged in 1986 by Brooks’ subsumption architecture, a control model which attains quite sophisticated problem-solving behaviour via a hierarchy of simple, independent but interacting behaviours [3]. Although subsumption deals with many of the more direct aspects of robot control, such as avoiding obstacles and following paths, it is less useful for more sophisticated planning tasks because the hierarchy of behaviours becomes too complex. Consequently, many hybrid architectures have been proposed combining reactive and planning control. For example [1] combines subsumption, learning and planning under the supervision of a central controller. The Saphira architecture described in [14] uses a local world model to formulate plans used to supervise and guide reactive behaviour.

Proponents of logic programming also argue that the low-level reactive control afforded by subsumption can be satisfactorily addressed in logic [19], providing a uniform framework for high-level planning and low-level behaviour.

3 Defining Robot Hardware and Environment

Programming by direct manipulation of images is unlikely to be useful unless it is applied to a domain of objects with a well understood visual appearance, subject to well understood direct manipulations. Rearranging and deforming objects in space has this property. We will therefore confine our attention to robots moving and transforming objects in 2D or 3D space. Although displaying and manipulating representations is significantly harder in three dimensions than in two, the principles behind our proposed approach are independent of the number of dimension of the domain. For ease of presentation, therefore, we will consider only examples in two dimensions.

We assume that a physical robot consists of a collection of *parts*, some to gather information from the robot’s environment, others to act on the environment in some way, and the remainder to simply provide connecting structure. We call these parts *sensors*, *effectors* and *blocks* respectively. We assume that a set of primitive *external functions* are available for reading values from sensors, and sending values to effectors.

We confine our attention here to the problem of programming reactive control. We do not address other aspects of control programming such as deductive problem solving, vision or the low-level computational problems arising from mechanics of the robot hardware.

3.1 Hardware Definition Module (HDM)

HDM is intended to be an extension to the application framework of an existing application development system, so that the underlying programming language and tools are always available. This is because even seemingly trivial robot control tasks usually require some symbolic computation. Clearly, this also gives expert programmers access to the robot control structures at any level from the hardware external functions to the structures built by HDM. As a basis for discussion we assume the underlying system to be Prograph CPX, and will use some Prograph constructs in HDM. We therefore expect the reader to have some familiarity with the Prograph language, described briefly in [7] and fully in [18].

With HDM, the programmer defines classes that capture the function and appearance of the parts of a robot, then builds a representation of the robot from instances of these classes. We will use the LEGO car example to illustrate this process. In the discussion, we will frequently refer to familiar user interface items such as menus, dialogues and drawing tools, but for simplicity will not show them in figures or discuss in detail how they operate.

On starting HDM, a single empty window called untitled workspace is displayed, together with a floating *component palette* consisting of a revolving sequence of panels, each containing some category of available classes. The workspace window can be populated with various kinds of objects each of which has at least one visual representation or *icon*. For example dragging a class from the palette creates an instance of the class. Graphic objects are created in HDM using drawing tools similar to those found in any object-based drawing software, or can be pasted from elsewhere.

Each graphic object has one or more *published attributes*. On receipt of an appropriate message, an object can send the value of an attribute to the message sender. Similarly, in response to a message it can change the value of a published attribute, which may alter its icon. We use this message-flow mechanism in defining properties of the robot and its environment.

3.2 The robot environment

In building a robot control program, a programmer specifies procedures which use information about the robot's environment to compute values used to change the environment. Since our aim is to make it possible to do at least part of this programming by direct manipulation of a representation of the robot, we must also provide a representation of the environment in which the robot operates. What we need is a way to define values at points in space for properties the robot will measure or attempt to change. To this end, we consider that the environment consists of a set of *tiles*. A tile has an icon of any shape, and has associated with it a *property* and a function that maps any point on the tile to a value for the property. This function could simply be a constant, or arbitrarily complex, with parameters other than just the point, such as time. The LEGO car requires functions which depend only on location and have a small number of values, so we will restrict our attention to building such tiles.

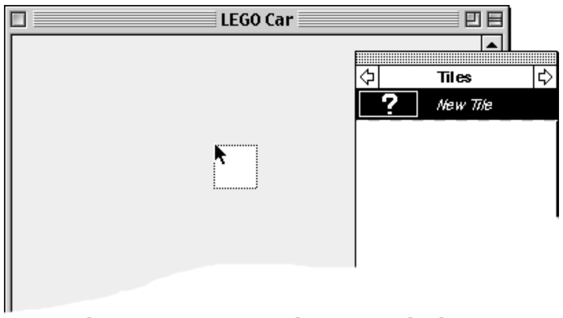


Figure 1: HDM workspace window

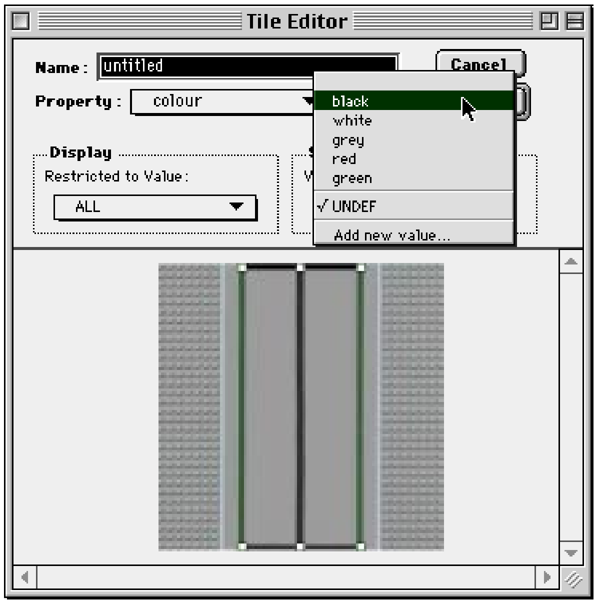


Figure 2: Setting the value of a region

Suppose we have started HDM and saved the workspace as LEGO Car. The item New Tile in the component palette represents the root class of the tiles class hierarchy in the underlying framework. Dragging this into the workspace as in Figure 1 creates an instance and opens the Tile Editor window (shown in Figure 2). In this editor we define the characteristics of the newly created tile: that is, we assign it a property and define the function that computes the value of the property at points on the tile.

The Property popup is used to assign a property to the tile being edited. This popup lists all properties defined in the workspace, and also includes an item Add new property… which can be used to create and name a new property, adding it to the workspace and assigning it to the tile under consideration.

The function that determines values for the property associated with the tile we are editing, is defined by assembling a collection of regions in the lower part of the Tile Editor window, where each region is associated with a value for the property.

Any number of regions in the window can be simultaneously selected. The popup Value of Selected Regions in the Selection control group is used to assign a value to all selected regions.

In our example, we will create a tile corresponding to a straight section of LEGO track. First we select the property colour for the tile, then paste an image of the grey plastic part of the track section into the lower part of the window. We choose the value grey from the Selection popup to assign it to the region.

Each LEGO track section has white and green stripes along the edges of the roadway section, so we add graphics for these and assign them the values white and green. Finally, our track also has an added strip of black tape down the centre plus crosswise strips at the ends for the infrared sensors to detect. We add graphics for these, select them all and assign black to them, as shown in Figure 2.

Both the popups displaying property values in this window can be used for adding a value to a property, in which case the popup is set to the new value, thereby affecting either the set of displayed graphics or the value associated with each of the currently selected graphic items.

Clicking. the OK button closes the tile editor, returning to the workspace which contains our new tile instance. So that we can repeat straight sections when building tracks, we use this tile as a prototype to create a class, by dragging it into the component palette. This results in a dialogue where we name the new tile class Straight. When this dialogue is closed, the new track class is added to the Tiles panel of the palette, represented by a reduced form of its icon. After creating four tile classes corresponding to the four kinds of LEGO track sections, the palette appears as in Figure 3.

We did not name the tile we created in Tile Editor. Most HDM entities can be named, but since these names are mainly required for locating structures in the underlying model if one is programming at a low level, we will not bother to name them in our examples.

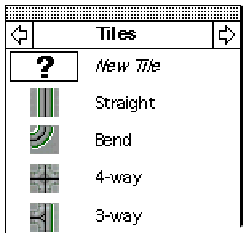


Figure 3: New tile classes in the component palette

To build a robot environment in the workspace, instances of tile classes are created and arranged. Tiles may be butted or overlaid. If two tiles for the same property overlap, the upper tile defines the property in the overlap region. If overlapping tiles correspond to different properties, they do not affect each other.

By default, in every robot environment there is a value defined everywhere for the property `location`. There is also a property called `velocity` defined at every point by a function of two parameters, the point itself and a point on the robot: this function computes the relative velocity of the two input points.

3.3 Sensors

Depending on how a robot's on-board controller operates, it may not be possible to directly access a single physical sensor, as is the case with the five infrared sensors on the LEGO car. We therefore recursively define a *sensor* to be either a simple sensor or a compound sensor, which is a collection of sensors. This is important only in HDM where the connection to the physical robot is defined, and does not affect the view of the robot that SDM presents to the user.

3.3.1 Simple sensors

Creating a simple sensor is similar to creating a tile, that is, New Sensor is dragged into the workspace from the Sensors panel of the component palette. This opens a Sensor Editor window (shown in Figure 4) with three panels; from top to bottom, the *specifications*, *head* and *body* panels. In the specifications panel the Function popup menu in the External Mapping area can be used to choose the external function with which to read the value of the corresponding hardware sensor. The sensor we build will be used as a prototype for a class, so we will not select an external function for it since then every instance of the class would refer to the same physical sensor.

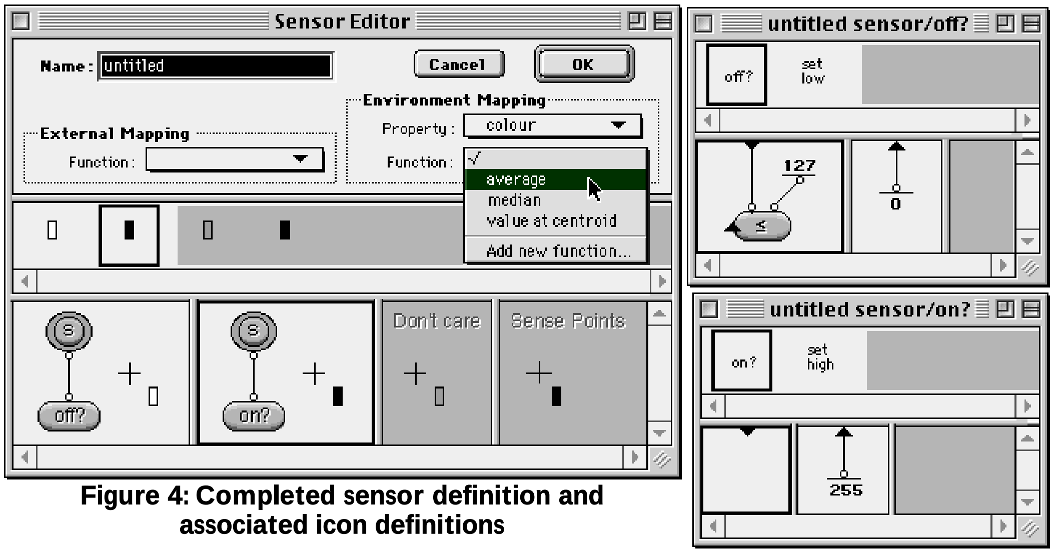
The head and body panels are used to define two mappings; a function from the sensor value received from the hardware sensor to the visual representation, and an inverse. Note that since the first mapping is not necessarily 1-1, the second cannot be a true inverse. It is used to compute a plausible sensor value in situations where the user sets the sensor icon manually. This is necessary since the sensor value is stored in the internal model of the sensor built by HDM, and can be accessed for lower-level programming. These functions are defined by a sequence of cases, similar to the definition of a method in Prograph, arranged horizontally in each of the panels. The head panel shows "thumbnails" consisting of an icon for each case, while the body panel consists of panes containing the "bodies" of the cases. The head panel is used for easily locating a case in the sequence. Selecting a case in the head panel also selects the corresponding item in the body panel, and scrolls it into view.

The editor initially displays one case, at the left end of each of the two panels. The body of a case contains a *graphic origin* consisting of a pair of crosshairs, and a network of icons including one occurrence of Ⓢ, representing the sensor value. The graphic origin provides a reference to ensure proper registration of the graphics that will be constructed in each case. Aside from graphic representations corresponding to values, each sensor also has a "don't care" representation, used when we define rules for robot behaviour, constructed in a pane to the right of the last case. Note that the `Don't care` pane has a graphic origin for this purpose. Dragging the graphic origin in one case will not move those in other cases, however, all graphic objects in the same case will move with it.

The physical infrared sensors of our LEGO car return integer values from 0 to 255: however, for the purposes of this example, we want to partition the 256 possible values into two, indicating whether or not the sensor is over a black strip on the track. We therefore want to map sensor values on to two representations for the sensor. First, we modify the body of the existing case as shown on the lower left of Figure 4, building a network of icons and adding the graphic ▯. The icon labelled `off?` is a *local* operation, defined by a sequence of cases.

Clicking in the head panel between the first case and the grey trailer creates a new case like the initial case. The second-case and "don't care" graphics are created, and after further editing, we obtain the sensor definition and associated icon definitions in Figure 4.

The semantics of sensor definitions and local operations is a logic/dataflow hybrid. Links normally represent unification, but may be marked with an arrow to indicate dataflow. To illustrate, let us suppose that a sensor value of 200 is being mapped to an icon by the definition in Figure 4. The first case is tried, which leads to execution of the local `off?`. The first case of `off?` applies since the arrow on its input indicates that data must flow in. The operation ≤ fails, however, so the second case is attempted, but also fails because data is expected to flow out. This leads to the failure of the first case of the sensor, and thence to the execution of the second case. The first case of the local `on?` succeeds immediately because it requires only



Name: untitled
Property: colour
Function:

Figure 4: Completed sensor definition and associated icon definitions

that data flows in. As a result, the icon ▮, is selected.

The sensor definition is completed by defining its reaction to a simulated environment, assuming that a simple sensor computes a simulated sensor value from the value of one property of the environment. First, we create a region in the Sense Points area of the body panel defining exactly which points of the environment the sensor collects values from. Since the infrared sensor reads all the values under its rectangle, we simply copy the graphic from any case, thereby defining a region consisting of all points in this rectangle. The Sense Points graphic consists only of black pixels, indicating the points that are read by the sensor.

Finally, using the two popup menus in the group of controls labelled Environment Mapping, we choose the property colour and the function with which to compute the sensor value as shown in Figure 4. The Function popup lists existing functions for the chosen property, as well as an option Add new function… which can be used to add a new environment mapping function.

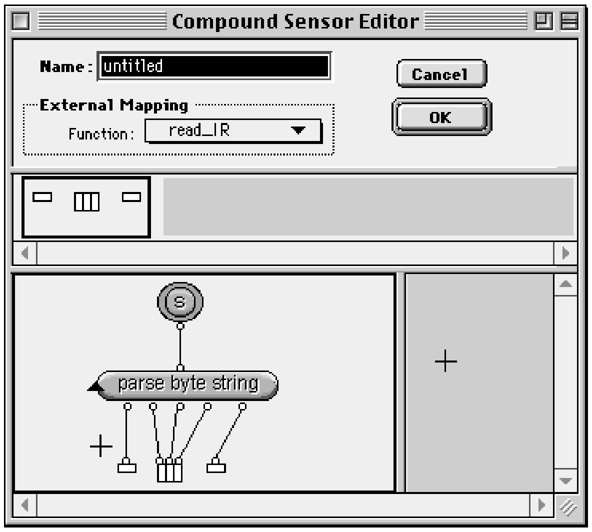
Clicking the OK button dismisses the Sensor Editor, returning to the workspace. As mentioned above, we need several sensors like the one we have built, so we use it as a prototype for a new class Infrared Sensor.

In this example, each case of the sensor corresponds to exactly one graphic representation, and the sensor value consists of simple data. This may be insufficient for some applications. In such a situation, the appropriate correspondence between sensor value and icon would be maintained by two message-flow links attached to the two graphic elements as discussed at the end of Section 3.1.

The concept of icons which change their appearance depending on some internal state has been used in various other visual programming systems, for example [4, 22].

3.3.2 Compound Sensors

To construct the compound sensor corresponding to the five infrared sensors on the LEGO car, we again invoke the sensor editor by dragging New Sensor into the workspace. We then drag five instances of Infrared Sensor into the first case body, positioning and orienting them appropriately. As each of the constituent sensors is deposited, it becomes part of the graphic in the case, and a terminal appears above it corresponding to its sensor value



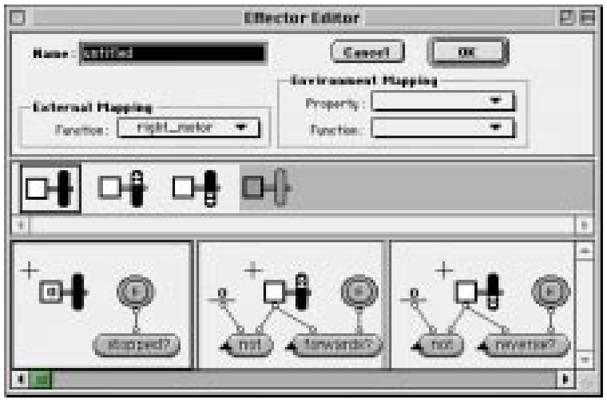
Name: untitled
Function: read_IR

Figure 5: Definition of compound sensor

Next we select read_IR as the external function that will deliver values to the sensor, and complete the case as shown in Figure 5. The local operation parse byte string maps the sensor value to the values of the individual sensors and *vice versa*. In the forward direction of the mapping, each of the constituent sensors receives a value with which it determines its own icon. Note that we have rotated two of the Infrared Sensor instances 90°. This matches their orientation on the physical robot.

3.4 Building Effectors

An effector is similar to a sensor in that it has an iconic representation defined as a collection of graphic items that varies depending on an underlying value (*effector value*), and is associated with an external function. Not surprisingly, editing an effector is similar to editing a simple sensor as in Section 3.3. The Effector Editor window in Figure 6 shows the definition for the right drive effector of our LEGO car. In this example, the icon consists of three parts; a text box representing the motor, a rectangle for the driveshaft, and a rounded rectangle for the wheel. The effector value is represented by the icon Ⓔ. The exact representations of the motor and wheel depend on the speed and direction components of the effector value. The first case defines the correspondence for a stopped drive, while the other cases define it for forward and reverse movement. Each of the locals stopped?, forwards? and reverse? either checks that the direction component of the effector value is correct, or sets it correctly, depending on whether the definition is used to choose a graphic representation for an effector value or *vice versa*. These locals also ensure that the speed component of the effector value matches the display in the motor text box. During construction of these diagrams, whenever a terminal is created on a graphic, a popup appears from which the appropriate published attribute is selected.



Name: untitled
Function: right_motor

Figure 6: The Effector Editor

To define the relationship between the effector and the environment, we choose the property velocity in the Environment Mapping section. In the popup of functions available for this property is the special item Direct which, rather than defining a function, allows us to define one directly. Choosing this item adds small banners labelled velocity to each of the cases of the definition.

Switching to the *environment mapping view* preserves the graphic from the regular view, and adds a *velocity control* and two *attribute indicators*. The control consists of a vector and a reference axis meeting at a point called the *origin*. The indicators display the current values of the length of the vector (speed) and the angle between the vector and the axis (angle). We drag the origin of the control to a point at the centre of the wheel; then drag the arrowhead, changing the angle and magnitude of the vector; and finally, wire the speed indicator to the motor icon, as shown in Figure 7. This last action constrains the magnitude of the vector to 0.

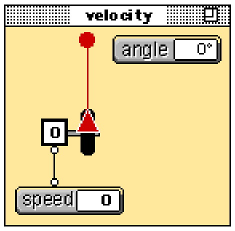


Figure 7: Defining environment mappings

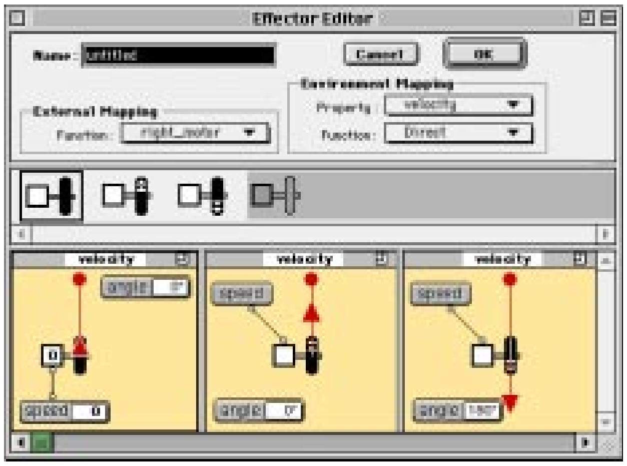
In a similar fashion we edit the environment mapping view in each of the other cases, obtaining the definition shown in Figure 8. When the speed indicator is wired to the motor icon in the second and third cases, its value display disappears since the motor icon display is empty.

Just as a sensor has a set of sense points defining which points of the environment it observes, an effector has a set of *effect points* defining the points in the environment it acts on. The effect points can be defined in the Effector Editor in the same way as sense points are defined (see Figure 4), or as a consequence of directly programming the environment mapping as in Figure 7, which defines a single effect point as the point on the effector that coincides with the origin of the velocity control.

The left drive effector is defined by horizontally inverting a copy of the right drive effector and changing the external function, horizontally inverting each of the icons in the definition.

3.5 Components

In Section 3.3 we took a hierarchical view of sensors since control programs may group simple sensors for communication. For different reasons, it is important to consider a robot as a hierarchical structure. For example, in an assembly-line robot consisting of a chassis and an articulated arm, the arm is not just a rigid item, but consists of parts that move in relation to each other. For this reason we will view a robot as a component, where a *component* is recursively defined as either a basic part or a set of components together with a constraint that specifies how the constituent components are related to each other. A *part* is either a sensor, an effector or a block, where a *block* is a maximal collection of graphic objects rigidly constrained together. The assembly-line robot, therefore, is a component consisting of one part, a fixed chassis, and one component, an arm, one end of which is constrained to a specific point on the chassis. The arm in turn is made up of other parts and components constrained together in such a way that the arm can pivot at its joints.



Name: untitled

Environment Mapping
Property: velocity
Function: Direct

External Mapping
Function: right_motor

Figure 8: Final definition of environment mapping

In above definition of component, the intention of the definition of "block" is to eliminate irrelevant detail. For example, a definition of the LEGO car need not deal with the individual plastic bricks that make up the chassis. Our definition of block suppresses such detail. The car does, however, have a single, centre-mounted, swivelling back wheel which the definition would treat as a block separate from the chassis, even though it contributes nothing to modelling the functionality of the vehicle. We will omit it from our description.

In drawing applications the "group" operation fixes the relative positions a set of graphical objects. In HDM grouping is generalised to *constraining*, where any constraint is applied to a set of objects. For example, one can select two overlapping objects, choose the constraint *pivot*, then click the cursor at a point in the intersection of the objects to define the pivot point. The LEGO car is a simple structure which we build by arranging in the workspace the compound sensor from Section 3.3.2, the right and left drive effectors from Section 3.4, and a graphic representing the chassis, as shown in Figure 9. With all these items selected, we choose the constraint *lock*, which specifies that the four components must remain in the same relative positions and orientations.

We now save the LEGO Car workspace, which saves to a file the components in the workspace window, and all the classes and properties we have defined.

3.6 Programming the Robot

In this section we give a simple example to illustrate how the model produced by HDM can be used to program the robot. First we load the LEGO Car workspace into SDM. Next we assemble an environment by dragging track sections from the palette, and place the robot as shown in Figure 10.

In this window we define a behaviour for our robot as a finite state machine (FSM), following the subsumption architecture [3]. First we select the behaviour we are going to edit using the popup menu labelled Behaviour. Since we are about to define a new behaviour, we select the item New behaviour… which we can use to create and name a new behaviour. To name the starting state for the behaviour, we type straight ahead into the Current text box in the State control group. This text box appends the character • to the beginning of the name to indicate that this is the start state for the behaviour.

When the robot is placed in an environment in SDM, its effectors are all set to "don't care," and its sensors display whatever icons they compute from the environment. The final preparatory task is setting effectors to whatever values they should have as long as the robot is in the straight ahead state, and selecting the sensors significant in implementing the behaviour. We set the two motors to “forwards at speed 5,” and set to “don’t care” any sensors we are not interested in. Effectors not involved in implementing the behaviour remain in the “don’t care” state. We refer to the sensors and effectors that we are interested in as *active*. SDM provides appropriate tools for setting sensors and effectors. For example, the representation of a drive wheel is toggled by clicking it, and clicking a motor pops up a slider for choosing a speed value.

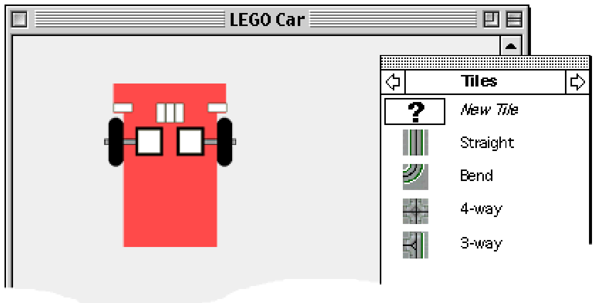
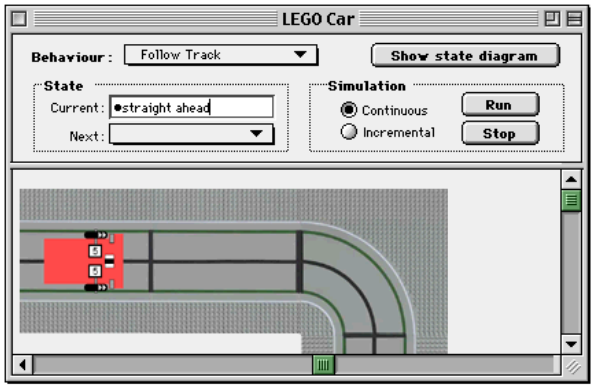


Figure 9: Representation of the LEGO car in HDM

To begin building the FSM for the behaviour, we click the Run button. Simulation begins and the robot moves forwards at a constant speed 5 until the value of an active sensor changes, halting the simulation. This will occur when the robot reaches the curved track section as in Figure 11(a). The robot should now turn right, so we create a new state veer right by selecting New state... from the Next popup. The popup is set to this new state. For this state we set the left and right drives to “forwards at speed 5” and “stopped,” respectively.



Behaviour : Follow Track
Current: ●straight ahead
Next:
[✓ Continuous] Incremental

Figure 10: Environment for programming the LEGO car.

Now we click Run to resume the simulation, which stops again as soon as the sensors change, as shown in Figure 11(b), at which time we select the existing state straight ahead from the Next popup, which sets the drive values defined for that state, causing the robot to continue straight.

The next change in sensor values is to the configuration ▯▯▮ (left to right). Because this configuration was previously encountered in the state straight ahead, there is already a transition for it in the FSM. Therefore, since the simulation mode is set to “continuous,” the simulation will not halt, but continue with a transition to the state veer right.

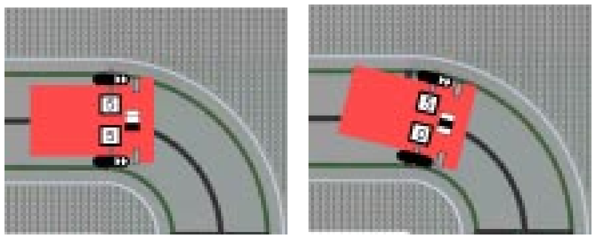


Figure 11: Programming a right turn.

Programming proceeds in this way until all necessary states and transitions have been defined, at which time the simulation runs without stopping. At any time during the programming process, clicking the button Show state diagram opens a window depicting the FSM for the behaviour under construction. For example, if this button is clicked at the point in the above description when the robot halts in the state veer right, the window shown in Figure 12 appears. The start state of the FSM is outlined. The label on a transition is the combination of sensor values that triggers the transition.

In the subsumption architecture, behaviours are autonomous and concurrent, making it possible to build complex reactive systems in an incremental manner. For example, once the Follow Track behaviour has been defined for the LEGO car, we can expand the control program’s capability by adding a Back Up behaviour that reverses the motors if the touch sensors at the front of the car are activated. Clearly motor settings produced by Back Up should take precedence over those from Follow Track. The subsumption architecture provides message-flow diagrams to specify the precedence relationships between behaviours. We have yet to address this aspect of subsumption.

For ease of presentation we have used a simple robot which has a rather limited effect on its environment. It is important to note, however, that the only limitations of our approach to control programming are those inherent in the subsumption architecture. In fact there may be potential for some “direct manipulation” editors for defining the effect that effectors have on the environment.

In the example, we built an idealised robot and environment where sensors and effectors behave perfectly. It would be important in a practical HDM to make the simulations more realistic by introducing imperfect performance, possibly by calibrating the model against the hardware.

4 Concluding remarks

A recent series of experiments in applying visual programming to the task of controlling robots began with simply using a general-purpose visual language, and progressed through more domain specific implementation techniques. Although this work produced VBBL, a general visual language for robot control, we felt that the result was disappointing in that it did not in any way exploit the fact that robots are physical objects with an obvious visualisation, providing instead concrete representations of abstract control constructs. This result is not really surprising, of course, since maintaining generality while providing direct representation of robots are rather contradictory aims.

The proposal presented here is an attempt to reconcile these two aims by taking an approach analogous to compiler compilers for programming languages. The compiler compiler supplies full generality, and specific features are provided by language definitions that the compiler compiler processes. Accordingly, we have proposed a robot programming system consisting of two modules; a Hardware Definition Module (HDM) and a Software Definition Module (SDM). HDM consists of a set of editors for specifying the structure and function of a robot and its environment, while SDM uses the model built by HDM to enable the programmer to build control programs by directly manipulating a visual representation of the physical robot.

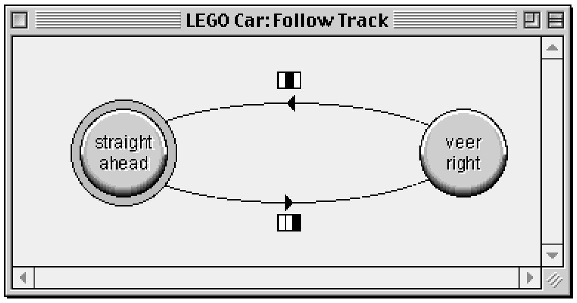


Figure 12: FSM for turning right.

In HDM the programmer describes a robot as a hierarchical structure of components, capturing the way constituent components are constrained together, and the function of the sensors and effectors with which the robot observes and alters its environment. Thus HDM is similar to CAD editors, but its purpose is quite different. To ensure that a device can be manufactured, a CAD editor is concerned with every intricate structural detail, whereas we are interested in capturing only those features of a robot's mechanics relevant to programming it.

The output of HDM is a model that captures the syntax and semantics of a particular robot. This model can be used by SDM for various purposes. We have described how, by directly interacting with a simulation of robot activity, a programmer can build the finite state machines that implement behaviours in the subsumption model of robot control. Clearly, SDM could also incorporate a simulator for observing the performance of control programs developed in other systems.

The modules described here have not yet been implemented. Our intention is to build them as an extension to the application framework of Prograph CPX and use them to test the practical viability of this approach. In the process we will formalise our example-based description presented here, addressing issues such as robustness and generalisability.

Issues yet to be investigated include dealing with environmental properties that are not well expressed by tiles; how to generate the subsumption model message-flow graphs via direct manipulation of the robot model; what other models for robot control might be used as a basis for programming in SDM; how to deal with situations where the relationship between sensors and effectors is described by a continuous function, rather than discrete as in our examples here; and how to generalise these concepts to three dimensions.

5 References

[1] C. Balkenius, *Natural Intelligence for Autonomous Agents*, LUCS 29, Lund University Cognitive Studies (1994).

[2] Borland, *Delphi Reference Manual* (1995).

[3] R.A. Brooks. A Robust Layered Control System for a Mobile Robot. in *IEEE Journal of Robotics and Automation*, (1986).

[4] M. Burnett and A. Ambler, Interactive Visual Data Abstraction in a Declarative Visual Programming Language, *Journal of Visual Languages and Computing*, (1994), pp. 29-60.

[5] P.E. Caines and S. Wang, COCOLOG: A conditional observer and controller logic for finite machines, *SIAM Journal of Control* (1995).

[6] P.T. Cox, C.C. Risley, T. Smedley, Toward Concrete Representation in Visual Languages for Robot Control, *Journal of Visual Languages and Computing*,9(2), (1998) to appear.

[7] P.T. Cox, T. Smedley, Using visual programming to extend the power of spreadsheet computation, *Proc. of Advanced Visual Interfaces Workshop*, Bari (1994), 153-161.

[8] CRS Robotics, *RAPL-3 Language Reference Manual*, (1997).

[9] R.E. Fikes and N.J. Nilsson, STRIPS: A new approach to the application of theorem-proving to problem solving, *Artificial Intelligence*, 2(4), (1971), pp. 189-208.

[10] T.R.G. Green, M. Petre, Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework, *Journal of Visual Languages and Computing*. 7(2), (1996), pp. 131-174.

[11] IBM, *Visual Age for Smalltalk*, 1995

[12] E. Jackson and D. Eddy, *Design and Implementation Methodology for Autonomous Robot Control Systems*, International Submarine Engineering Ltd., http: //www.ise.bc.ca/robot1001.html, (1997).

[13] E. Knapp, Wisconsin Power & Light, *Private Communication* (1997).

[14] K. Konolige, K. Myers, E. Ruspini and A. Saffioti, The Saphira Architecture: A Design for Autonomy, *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, 1997,pp. 215-235.

[15] R. Kowalski and M. Sergot, A logic-based calculus of events, *New Generation Computing*, vol. 4, no. 1, (1986), pp 67-95.

[16] Y. Lespérance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R.B Scherl, A logical approach to high-level robot programming - a progress report, in B. Kiupers (Ed), *Control of the Physical World by Intelligent Systems, Papers from the 1994 AAAI Symposium*, New Orleans (1994), pp. 79-85.

[17] Microsoft, *Visual C++*

[18] Pictorius Incorporated. *Prograph CPX User's Guide*. (1993).

[19] D. Poole, Logic Programming for Robot Control, *Proc. 14th International Conference on Artificial Intelligence*, Montreal, (1995), pp. 150-157.

[20] M. Resnick. Behavior Construction Kits. *Communications of the ACM*, (1994), pp.65-71.

[21] K. Schmucker, Apple Computer, *Private Communication* (1997).

[22] D.C. Smith, A. Cypher and J. Spohrer, KidSim: Programming Agents Without a Programming Language. *Communications of the ACM*, vol. 37, (1994) pp. 54-68.

[23] H. W. Stone, Mars Pathfinder Microrover: A Low-Cost, Low-Power Spacecraft, *Proceedings of the 1996 AIAA Forum on Advanced Developments in Space Robotics*, Madison, WI, (1996).

[24] Symantec, *Visual Café*