A Visual Programming Environment for Autonomous Robots

by

Omid Banyasad

A Thesis Submitted to the
Faculty of Computer Science
in Partial Fulfillment of the Requirements
for the Degree of

Master of Computer Science

Approved:

Dr. P.Cox, Supervisor

Dr.T.Smedley, Faculty of Computer Science

Dr. D.Thomas

DalTech, Dalhousie University

Halifax, Nova Scotia

2000

DalTech Library

“AUTHORITY TO DISTRIBUTE THE MANUSCRIPT”

Title:
A VISUAL PROGRAMMING ENVIRONMENT FOR AUTONOMOUS ROBOTS

The above library may make available or authorize another library to make available photo/microfilm copies of this thesis without restrictions.

Full Name of Author: Omid Banyasad

Signature of Author: ______________________

Date: ______________________

Abstract

Until relatively recently computer programmers had to use machine-oriented programming languages owing to the fact that memory and processing power were insufficient for both computation and complex human-computer interaction. Computers today are equipped with larger memory capacity and offer much higher processing power. Visual programming is one of the many research fields that has emerged and become active as a result of this technological advancement. Recently, there has been considerable interest in applying visual programming languages to robot control. In this work, we will introduce the reader to our proposed visual language for programming autonomous robots. In this system the physical characteristics of a specific robot can be incorporated with the general syntax and semantics of the underlying control architecture to provide both generality and domain specificity. This has been accomplished by dividing the programming system into two major modules; Hardware Definition Module or HDM and Software Definition Module or SDM. This work focuses on the second module SDM and describes in detail how a control program is created by using the models previously generated in HDM. We also present a complete and precise reformulation of the subsumption architecture for robot control on which our system is based.

Acknowledgments

This work might never have been completed without the generous help and support of my supervisor, Dr. Phil Cox, who was always eager to help and his door was always open for discussion. I would like to thank him for being my mentor during the past eighteen months and for all those long discussions that we had. I thank him for introducing me to the field of visual languages and most importantly for teaching me how to convey ideas in writing.

I would also like to thank Dr. Trever Smedley and Dr. Dave Thomas for agreeing to be a member for my guiding committee.

Last, and not the least, I would like to thank my parents who have always supported me. Without their support, encouragement and love this work would have never been completed.

Table of Contents

List of Figures

List of Abbreviations

DFSM	Deterministic Finite State Machine
EMM	Extended Moore Machine
FSM	Finite State Machine
HDM	Hardware Definition Module
SDM	Software Definition Module
VPE	Visual Programming Environment
VPL	Visual Programming Language

1 Introduction

Until relatively recently Computer programmers had to use machine-oriented programming languages owing to the fact that memory and processing power were insufficient for supporting both computation and complex human-computer interaction. Programming in machine-oriented programming languages requires programmers to highly abstract the original problem in textual machine code procedures. A programmer who maps the original problem into a machine-oriented language should not only have a strong knowledge of the problem domain, but also a solid understanding of the programming language in use.

Computers today are equipped with larger memory capacity and offer much higher processing power. Visual programming is one of the many research fields in computer science that has emerged and become active as a result of this technological advancement. Visual Programming Languages (VPLs) enable programmers to directly represent the structure of algorithms and data. As a result, a programmer is able to build programs that are more comprehensible, robust, and maintainable. While VPLs in general are aimed at visually representing the structure of algorithms and data, they do not specifically benefit from the visual representation of physical entities in the problem domain. Programming a robot using VPLs is a good example.

Recently, there has been considerable interest in applying VPLs to robot control. This interest can be traced back to the work of Gindling *et al.* [11] that inspired Allen Ambler to propose a competition in which visual languages were to be used to control a simple robot to achieve a specific task. The competition was judged at the 1996 IEEE Symposium on Visual Languages and was repeated the following year. Although research in applying visual languages to the problem of robot control is a recent activity, many different models and approaches have been employed for this purpose. Altaira [9] is a visual language that implements a rule-based model for robot control. Cocoa [1] is another visual language that implements a rule-based model but is not specifically designed for robot control; however, with minor changes it can effectively be used for programming simple robots in a 2D environment. Such a modified version was submitted to the competition held at the 1997 Symposium on Visual Languages. VBBL [15] is a message flow domain-specific visual language based on Brooks' subsumption architecture [20] designed as an extension to the application framework of Prograph CPX [17].

Robots physically exist and therefore have an obvious visual representation. In addition, the common feature of robots when operating, is that they make physical changes in their surrounding environment, including themselves, at least some of which are physically observable, such as changes in position or colour. Existing VPLs are no more useful for programming robots than they are for any other problem domain since they do not directly represent domain entities or observable changes.

1.1 Hypothesis

Considering the increasing interest in applying VPLs to the problem of controlling robots and the aim of VPLs to simplify programming tasks, it should be possible to design a domain-specific visual language for robot control that incorporates a good editor environment in which programming is performed by direct manipulation of programming constructs that reflect the physical characteristics of the robot and its environment.

1.2 Research Objectives

The overall objective of this research is to investigate the feasibility of a domain-specific VPL for programming robots. To this end we will:

- Provide a summary of the concepts of VPLs on which this research relies.
- Introduce robot control systems design in general and Brooks' subsumption architecture [20] in detail as a basis for our control system design.
- Give a precise reformulation of the subsumption architecture to provide a control system on which to base our VPL.
- Design a robot programming system in which the physical characteristics of a specific robot can be incorporated with the general syntax and semantics of the underlying control architecture, thereby providing both generality and domain specificity.
- Evaluate the language and its tools and proposed environment.

This research does not address the applicability of VPLs to high level tasks such as planning and problem solving which may arise during the process of programming a robot, although such investigations would provide an interesting extension to our work.

1.3 Overview

The remaining chapters of this work provide the background, design and evaluation of a visual programming environment for robot control. In Chapter 2 visual languages are discussed in general with an emphasis on those aspects of VPLs that seem to be most relevant to our work. A well known set of criteria for VPL evaluation, the *Cognitive Dimensions* due to Green and Petre [7] is also discussed in Chapter 2 and will be used in Chapter 6 for assessing our results. Since we have not implemented the proposed model, there can be no additional practical assessment via experiment.

In Chapter 3 we provide a general introduction to robot control systems. A traditional control system design for robot control is presented and its drawbacks discussed. We then present the subsumption architecture due to Brooks [20] intended to address the shortcomings of the traditional approach, and explain the ways in which this model is inadequate as a basis for a VPL. A new subsumption model is proposed, followed by an illustrated example.

In Chapter 4 we present the overall structure of our proposed system, and discuss the necessity for dividing the robot programming process into two distinct consecuitive tasks, performed in two seperate modules. In this chapter we will also explain the first of these, the Hardware Definition Module (HDM), its design environment and the data structure used for defining robots and objects. The Software Definition Module (SDM) is discussed in Chapter 5 illustrated by an example. This chapter explains how the robot and objects previously defined in HDM are used to build a simulated environment in which to program a robot by direct manipulation.

In Chapter 6 we apply Green and Petre's cognitive dimensions to our work as a preliminary assessmentt of our VPL for robot control.

Finally, in Chapter 7 , we summarize our results and suggest directions for further research.

2 Visual Programming

Although the many definitions of the term “ visual programming” do not agree in detail, a notion common to them all is that visual programming is “the use of meaningful graphical representation in the process of programming” [22]. Visual programming has been an active research area in recent years resulting in many visual programming languages (VPLs) and visualization systems. Some of these visual languages are designed specifically to facilitate the process of programming in a certain field, while others aim for a wider range of problem domains. Regardless of the problem domain, the philosophy behind VPLs is that programs created with images and graphics are easier to understand than those presented in text. There is empirical evidence for and against the usability of visual languages in certain fields [18][23]. There are many factors that should be considered when the usability of a visual language or a visualization system is studied, such as the nature of the problem domain, the targeted user’s programming skills and cognitive abilities, and the environment in which the languages are implemented.

Visual programming languages aim to directly represent the structure of algorithms and data, thereby enhancing the programmer’s ability to build and comprehend programs. In traditional textual languages the structure of algorithms and data is encoded in strings of text. Visual languages remove this layer of abstraction and allow programmers to directly manipulate the structure of the program. This direct representation is cited by Green and Petre [7] as one of the most important factors in enhancing the programmer's ability to build and understand the structures of a program.

There are two distinct aspects to visual programming; visual programming environments and visual programming languages.

2.1 Visual Programming Environments

Visual Programming Environments or VPEs, assist programmers to build, edit, debug and execute programs by incorporating appropriate graphical representations. A VPE consists of a set of tools and a user interface for accessing them. A system is said to be a VPE when some of these tools are graphical [13]. Programs written in VPEs are typically still created in a textual programming language so that both the structure of the program and the relationships between the programming constructs are defined in some textual format. Although VPEs for textual programming languages are not VPLs [12], they can effectively improve the process of creating, editing and debugging textual programming language programs. Java Workshop and Microsoft Visual Basic are good examples of VPEs, providing graphical representation of interface elements which can be arranged and manipulated directly.

Some visual programming implementations provide VPEs. For example, the Prograph CPX [17] is a VPE which provides the VPL Prograph as the programming language.

2.2 Visual Programming Languages

In a VPL, the semantics of the language are defined by "meaningful graphical representations" [22] intended to help programmers to more easily comprehend the structure of programs and data. In a visual programming language the relationships between the programming constructs are represented visually, using devices such as lines, graphs and trees. These "meaningful graphical representations" are intended to reduce the abstraction levels of the program's parts, thereby enhancing the “directness” of the program. Directness of a program is a measure of how concretely it displays code and data structure, a property also refered to by Green and Petre as “closeness of mapping”. We will further discuss closeness of mapping in section 2.3. A good example of directness is a rule-based visual language in which the rule set directly represents how the output of a process or action is implied by detecting a condition in the rule domain, as in AgentSheets [2] or Cocoa [1]. Another good example of directness is the graphical representation of class hierarchies in Prograph [17].

Classifying a language as visual or not is controversial. The phrase “meaningful graphical representation” is an important factor in making this classification. In the literature a number of definitions for a VPL can be found, for example:

> “When at least some of the terminals of the language are graphical, such as pictures or forms or animations, we say that the language has a visual syntax........ We use the term visual programming language (VPL) to mean a language with a visual syntax.” [13]

and

> “[A visual language is one in which] a pictorial, iconic or graphical syntax (as apposed to a textual syntax) is used as the primary means of expressing the logic of the program being written.” [21]

For example, Prograph is a VPL according to both definitions, while Visual C^{++} [14] is a VPL according to the first definition but is not a VPL according to the second.

Visual programming languages may effectively employ text to enhance their directness. For example, when a conditional branch in a flow of a program is desired, a simple textual “if”, framed in a box, may express the concept of conditional branch more effectively than a not very expressive icon. Text can also be used in tables and charts or even as comments. The fact is that the evidence supporting the expressiveness of graphics does not disprove the benefits of text in certain cases.

2.2.1 Common Characteristics of VPLs

VPLs are usually integrated with visual environments. This suggests that a VPL is to be characterized by attributes of both its syntax and environment [13]. Some of the common characteristics of VPLs are [4]:

1.Simplicity
2.Concreteness
3.Explicit depiction of relationships
4.Immediate visual feedback
5.Closeness of mapping

VPLs aim to provide simplicity in the process of creating and editing programs by reducing the number of concepts used to build a program. VPLs achieve this by removing many of the complex and difficult abstractions that usually are used in textual programming languages. For example, most VPLs relieve programmers from defining variables for the implicit dataflow in textual languages.

Concreteness means “use of specific values, rather than a description of possible values” [13].

VPLs use explicit visualization for expressing the relationships between program elements. For example, flowcharts in a VPL show the control flow of the program clearly, and data flow diagrams depict the data dependencies between operations.

Immediate visual feedback provides instant response to the changes made to a program. Whenever a programmer makes a change, the system automatically updates any related display information. For example, in spreadsheets changing a value immediately propagates changes to dependent cells.

VPLs aim to improve the way in which programmers express data and algorithms in a program and the way they edit and modify programs. VPLs help programmers to understand the logic of their programs and algorithms by explicit depiction of program relationships. This is achieved through the visual nature of the VPLs and “closeness of mapping” that allows the gap between the problem world and the programming world to be narrowed. This should be particularly applicable to and attainable in a concrete domain such as robot control.

One of the primary goals of domain-specific VPLs is to enable experts and scientists to solve complex problems in their domains of expertise through programming without necessarily being expert programmers. In this case the programmer can concentrate on the solution to the problem rather than fitting the solution to the language. According to Green and Petre “The closer the programming world is to the problem world, the easier the problem-solving ought to be” [7]. While some textual languages have effectively relieved programmers from fitting the solutions into the machine-oriented procedures of code, they are a long way from mapping between a problem world and a program world (at least insofar as the problem world can have a graphical representation). Closeness of mapping between a problem domain and a programming language and its environment, dictates how much mental effort a programmer must expend to create, edit, debug, examine and maintain a program.

2.3 Evaluation Techniques for Visual Programming Languages

Every programming system is composed of its notational structure and its support environment. Notations are those symbols that the user sees and manipulates, while the environment provides the necessary techniques to manipulate those symbols. Evaluation of a programming system must include both the notation and environment, because a user’s interaction with the system relies on both.

The Cognitive Dimensions introduced by Green and Petre [7] provide a framework for a comprehensive assessment of a programming system. This framework is aimed at assessing a programming system based on a number of cognitively important aspects of the system’s notation and interaction style. Green and Petre list 13 cognitive dimensions of a system. Table 2-1 provides a brief definition of each.

Dimension	**Informal Definition**
Abstraction Gradient	Availability and scope of abstraction mechanism
Closeness of mapping	Closeness of programming structures to problem domain
Consistency	Similar semantics are expressed in similar syntactic forms, so that when some of the language is learnt the rest can be inferred
Diffuseness	Number of symbols required to express a single concept
Error-proneness	Possibility of making mistakes because of the poor notational design of the language
Hard mental operations	Thought processes required to formulate an expression, made difficult by the notation
Hidden Dependencies	Important relationships between entities are not visible
Premature Commitment	Making decisions before the needed information is available
Progressive evaluation	The ability to execute the program partially before all of it is put together
Role-expressiveness	The purpose and role of each component is easily inferred
Secondary notation	Extra information other than program syntax that conveys extra meaning, above and beyond the semantics of the language
Viscosity	The effort needed to make a single change.
Visibility	Ability to view parts of a program simultaneously and easily

Table 2-1. Green and Petre’s Cognitive Dimensions.

There are important trade-offs between different dimensions for a programming system. Improving the system in one dimension is likely to affect other dimensions as well. For example, adding abstractions tends to impose hidden dependencies. Increasing the level of abstraction can reduce both visibility and viscosity. Programmers have to guess ahead and commit prematurely when abstractions are added to a system owing to the fact that they have to define the abstractions before anything else is done. In general it is important to note that changing a system in order to alter its rating along one of these cognitive dimensions can not be done arbitrarily, since many of these dimensions are coupled and changes in one will impose changes on the others.

2.4 Summary

Visual programming languages and visual environments are aimed at facilitating the process of programming by incorporating images and meaningful graphical representations of data, algorithms and tools. VPLs use graphics as part of their notations and to express relationships between programming constructs, while VPEs use graphical tools to manipulate the symbols and notations of a language.

Closeness of mapping between problem domain and program structure, regardless of being textual or visual, is an important factor in usability of a language in a specific field. However, for problem domains with concrete visual representations visual languages seem to have obvious advantages over textual ones.

3 Robot Control

3.1 Introduction

A robot, in most people's understanding, may be a programmable mobile machine or simply an articulated arm that can move objects in its reachable space. For the purpose of this work a *robot* is a programmable machine equipped with at least one *actuator*. An actuator is a device that can release non-mechanical energy in its surrounding environment, such as light or other electromagnetic waves, or can mechanically change the geometry of the objects in its environment, including itself, in a controlled fashion.

A robot can also be equipped with *sensors*. A sensor is a device that can sense the existence of or measure a specific kind of energy in its operating domain, such as an infrared sensor measuring infrared light or a touch sensor sensing mechanical energy. The sensors and actuators of a robot are connected by some structural parts collectively called the *body*. The body of a robot simply provides connecting structure and has no significance for programming purposes except for the geometric relationships it imposes on sensors and actuators.

Robots are divided into two major categories. The first consists of robots with actuators and no sensors, such as those used in assembling products. "Such robots perform repetitive and possibly quite complex sequences of actions and operate in a fully defined environment”[16]. Robots in this category do not require problem-solving capabilities and fail to adapt to a new environment unless reprogrammed. Programs controlling these robots essentially define a sequence of actions with no feedback processing.

Robots in the second category are equipped with sensors as well as actuators and are designed to operate in an environment which is dynamic in some respects. Programs controlling these robots contain feedback processing which computes values for the actuators based on the sensors’ measured values. Autonomous robots, a subcategory of robots with both sensors and actuators, can perform certain assigned tasks without the supervision of an operator. The level of autonomy is the degree to which a robot can respond to environmental changes in a logical fashion as it was being controlled by an operator. Mobile autonomous robots such as the Mars Rover [8] and Autonomous Underwater Vehicles [24] are two examples of robots in this subcategory.

There are two major distinct methodologies for controlling a practical autonomous robot: highly structured hierarchical [3] control systems and distributed layered control systems based on the concept of subsumption [20]. In a layered system, the control system is constructed from a set of layers, each of which implements a class of the robot’s behaviours, with the most basic behaviours at the bottom encompassed by layers of increasingly sophisticated behaviours. Subsumption means that behaviours on one layer can substitute for behaviours on another layer.

3.2 Hierarchical Structure

In the NASA Standard REference Model for telerobots (NASREM) [3] architecture, a complex control system is partitioned into several functional levels. Each level is divided horizontally into sensory processing, world modeling and task decomposition modules. Figure 3-1 illustrates the NASREM’s control systems block diagram. Each level receives commands from the level above and sensory information from the level below. In NASREM the layers from top to bottom are defined by the time frame in which they perform their tasks. The architecture is a hierarchy of control loops with response time constraints becoming increasingly tight as one progresses down the hierarchy. Higher levels in the hierarchy require slower update frequency and provide increased problem-solving capability.

Levels in NASREM represent levels of autonomy. For example, lower levels are concerned with robot geometry and dynamics, medium levels are concerned with the geometry of the world, and higher levels are concerned with the function and coordination of objects in the world, including the robot itself.

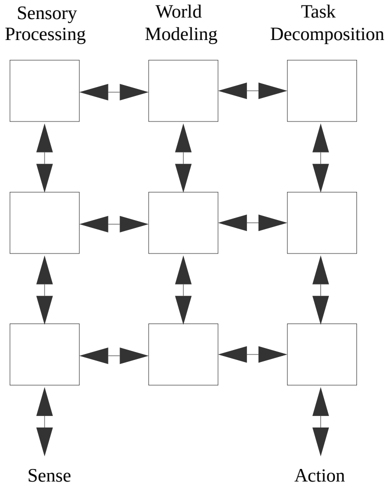


Figure 3-1. NASREM Control Systems Block Diagram

3.3 Subsumption

The traditional method for designing a control system for a robot is to decompose the problem into a series of subproblems. Each subproblem in this method is a functional unit as illustrated in Figure 3-2. In this traditional method the sequence of all functional units composes the overall control system. Input signals are received by the robot's sensors, and processed by a perception module which passes the results to the next functional unit of the control system

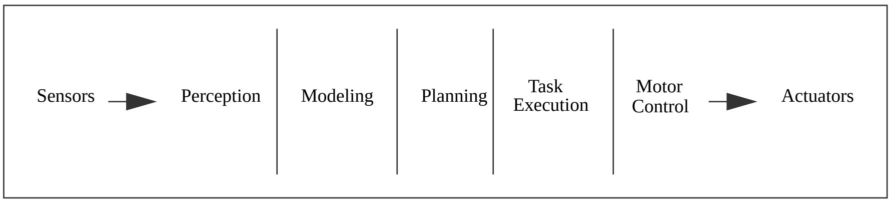


Figure 3-2. Traditional decomposition of a mobile robot control system into functional units (Brooks 1986)

Hence each slice of the control system in Figure 3-2 receives its inputs from the previous slice, processes the data and passes the results to the next functional unit (next slice). In general, the robot receives input signals through its various sensors, and processes them to obtain commands which are applied to the actuators to achieve the required response.

Brooks [20] mentions many drawbacks to this system. For example, the robot cannot effectively be tested unless all pieces in the control system are already built. This is because information from the sensors is transformed as data passes through the stages of the control model. Information received from sensors' output is meaningful only for the first stage of the model and meaningless for the rest. Either changes to a particular piece must be done in a way that avoids changes to the interfaces to adjacent pieces, or the effects of a change must be propagated to the adjacent pieces, changing their interfaces and functionality.

Another problem with the traditional control system is the time required for a signal to go through all the stages from the sensors to actuators. A problem may arise when changes in the robot's environment occur more quickly than the robot can process them.

Based on these difficulties, Brooks introduced a new control system model in which, instead of decomposing the problem into subproblems based on "internal working", the decomposition is based on “task achieving behaviours”. Figure 3-3 illustrates this horizontal control layer structure.

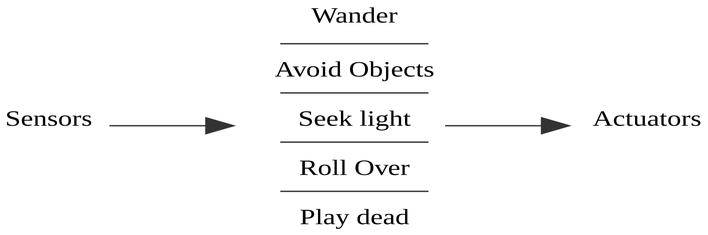


Figure 3-3. Decomposition of a mobile robot control system based on task achieving behaviours (Brooks 1986)

Behaviours illustrated in Figure 3-3 ideally can run in parallel. Each layer of behaviour can receive the sensors’ outputs, process them and send the commands to the connected actuators.

Unfortunately, this can lead to another problem. Behaviours running in parallel may battle each other for control of the robot. For example consider a mobile robot with two simple behaviours: the first behaviour causes the robot to transit to point A, while the other behaviour makes the robot avoid obstacles it approaches. If both behaviours have equal priority to control the robot and there is an obstacle on the way to point A, then two contradictory behaviours will battle each other to control the robot.

Brooks introduced a solution to this problem in the form of *subsumption* [20]. In general, subsumption is a way to resolve such conflicts between different behaviours. The key idea in subsumption is to prioritize those behaviours which have access to the same actuator or set of actuators. Behaviours with higher priority can overrule the behaviours with lower priority by subsuming or inhibiting them (explained shortly). In the subsumption architecture all behaviours run in parallel. Those behaviours that have actuators in common have to be ranked. In general each behaviour is not aware of the existence of the other behaviours and generates its outputs anyway, but at the end only the behaviour with the highest priority takes the responsibility for controlling the common actuator by overriding the outputs of other behaviours. In our previous example, to resolve the conflict, obstacle avoidance behaviour will subsume the other behaviour when an obstacle is detected to make sure that the robot does not hit any object on its way to point A.

One of the advantages of this subsumption architecture is that unlike the traditional control system, each layer of behaviours can be connected to sensors, actuators and any other behaviour layer. Since the layers of behaviours are not strongly tied to each other, new behaviours can be added to the previous ones to create a higher level of autonomy.

3.3.1 Levels of Competence

In the subsumption architecture a number of *levels of competence* for an autonomous robot are defined. A level of competence consists of a desired set of behaviours for a robot [6]. All lower levels of competence are subsets of a higher one and a higher level of competence implies a higher degree of autonomy.

As an example, four levels of competence can be defined for a mobile robot as introduced by Brooks[20]:

1. Avoid contact with stationary and moving objects.
2. Wander aimlessly around without hitting things.
3. “Explore” the world by seeing places in the distance that look reachable and heading for them.
4. Build a map of the environment and plan routes from one place to another.

3.3.2 Layers of Control

Each level of competence is made up of a series of control layers. A new layer can simply be added to the existing level to obtain the next higher level of overall competence [24].

The first step is to build a complete robot control system that achieves level-0 competence. This system will never be altered. Next, another control layer is added to the level-0 control system to obtain level-1. The new layer can examine the outputs from the level-0 control system and is also permitted to inhibit or suppress the normal flow of inputs and outputs of the lower level. This new layer, combined with level-0, achieves level-1 competence. To achieve higher levels of competence the same process can be repeated. Figure 3-4 illustrates this architecture.

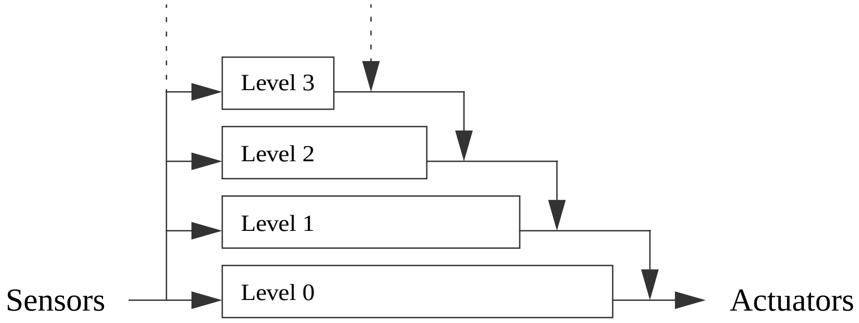


Figure 3-4. "Control is layered with higher level layers subsuming the roles of lower layers when they wish to take control. The system can be partitioned at any level, and the layers below form a complete operational control system." (Brooks 1986)

Figure 3-4 is taken directly from [20], however, we feel that boxes in this figure should be labeled with the name of layers rather than levels to follow the terminology. Then a collection of layers from bottom up represents a level.

3.3.3 Structure of Layers

In Brooks' subsumption architecture, layers are built from a set of small processors that send messages to each other [20]. Brooks defines each processor as a *Finite State Machine* (FSM), augmented with some variables, which can hold simple or structured data. Brooks' FSMs can hold some data, and send and receive messages over the connecting wires. They require no handshaking or acknowledgment of messages. Brooks defines each FSM as a module which has a number of input lines and a number of output lines. Input lines are buffered and the most recently arrived message on a line is always available for inspection [20]. Each state is named, and when the system first starts up each module starts in its distinguished state *NI*L. Each module has a distinguished input called *Reset*. When a message is received on the reset input, the module switches to state *NIL*. . A state in Brooks' FSMs can be specified as one of four types. *Output*, *Side Effect*, *Conditional Dispatch*, and *Event Dispatch*. When the FSM is in an Output state, a message is sent to an output line and a new state is entered. The output message is a function of the module's input buffers and instance variables. In a Side Effect state, one of the module's instance variables is set to a new value and a new state is entered. The new value of the instance variable is a function of the module's input buffers and variables. In a Conditional Dispatch state, one of two subsequent states is entered determined by a predicate on the module's instance variables and input buffers. In an Event Dispatch state, a sequence of pairs of conditions and states to branch to are monitored, and when condition found to be true, the corresponding state is entered.

In Brooks' subsumption architecture, a layer is a set of behaviours implemented as FSMs. An input line of a module is connected to an output line of another module from the same layer or a lower layer, or is directly connected to a sensor. An output line from one module is connected to input lines of one or more other modules or is directly connected to an actuator. Outputs may be inhibited, and inputs may be suppressed by output lines of other modules from higher layers. An inhibitor signal inhibits any output message from a module's output line for a predetermined time. Any message sent out that particular output line during that period is lost. A suppressor is very similar to an inhibitor. A suppressor signal replaces the input line signal.Figure 3-5 illustrates a module, suppression, and inhibition. For both suppression and inhibition, the time constants are written inside the suppression and inhibition circles.

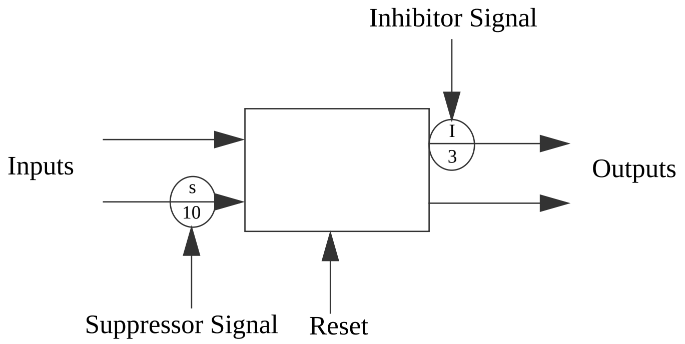


Figure 3-5. “A module has input and output lines. Input signals can be suppressed and replaced with the suppressing signal. Output signals can be inhibited. A module can also be reset to state NIL.” (Brooks 1986)

3.3.4 Notes on Brooks’ Subsumption Architecture

One of the major motivations behind subsumption architecture is to build control systems which are made up of independent layers that can run in parallel. Each layer can be considered as a transduction from sensors’ input values to actuators’ output values. Brooks mentions that in the subsumption architecture each higher layer can monitor the data from lower levels and suppress or inhibit the normal internal data flow of lower levels if required. This means that although the lower levels are unaware of the existence of the higher layers and can control the robot independently with some degree of autonomy, the higher layers are not independent of lower levels unless their inputs are limited to sensor values. In the subsumption architecture, whether or not a higher layer is independent of lower levels depends on whether it requires data from the internal data flow of lower levels. The form of data monitoring, suppression and inhibition in Brooks’ subsumption architecture prevents us from looking at each layer of behaviours as a black box. This means that although the degree of autonomy of the control system increases with the addition of higher layers, the overall control system must still be viewed as a distributed system made up of many small processing modules. The final block diagram of a control system constructed according to Brooks’ architecture must include the internal connections between layers, where one layer monitors or interferes with the internal data flow of data in another. Consequently, such a diagram cannot have the neatly layered structure suggested in figure 3-4. This in turn can impede the understanding of the control system by other designers, complicating the process of editing and maintaining such systems.

Another consequence of the internal connections between layers is that sometimes the designers of control systems must make premature commitments owing to the fact that they have to include unused output lines for some of the modules in their design that might be used by higher layers in future. The example presented in [20] provides a good illustrtation of such premature commitments.

Another characteristic of Brooks' subsumption architecture is that suppressions and inhibitions are limited to inputs and outputs, respectively. Considering that the output of each module is connected to the input of another module, except if directly communicating with the hardware, the inhibiting of an output is equivalent to inhibiting of the input of the succeeding module. The same argument holds true for suppression. This means that, practically, inhibition and suppression can affect both input and output of a module, although in cases that there are both suppressor and inhibitor signals on one single line from an output to an input, the order of them is significant.

In Brooks' architecture, although the overall control system is decomposed based on the desired behaviours of the robot, one might argue, quite justifiably, that each layer must still be decomposed in the traditional manner as illustrated in figure 3-2 and must therefore be completed before the expanded control system can be used.

In view of these observations and as a basis for a direct-manipulation visual programming system, we now propose a simpler, more streamlined subsumption model which is functionally equivalent to Brooks' architecture. In this model, each layer consists of a single FSM which is likely to be more complex than those that comprise the modules in Brooks' model. Layers can not monitor or interfere with the internal data flow of other layers since there is only one FSM at each layer, however a higher layer can accept the output of a lower layer as an input in order to monitor the functionality of the lower layer. A layer is also permitted to inhibit or suppress the inputs and the outputs of the layers below it. The succeeding sections explain this new view of subsumption architecture in more detail. Since FSMs are central to our discussion, we will now define them briefly but precisely. For a thorough presentation, the reader should consult one of the many books on automata and languages such as [10] from which our explanation is taken.

3.3.5 Finite State Machines

A Finite State Machine (FSM) is a simple, abstract computational device with a single input. A FSM consists of a finite set of internal configurations or *states* and a set of transitions between states. In Deterministic FSMs (DFSMs), for each input symbol there is exactly one transition out of each state. Each FSM has an initial state and some final or accepting states. A FSM processes a string of input symbols by consuming them in sequence, and making corresponding transitions between states. The machine accepts a string if this processing leaves the machine in a final state.

Definition: A Deterministic Finite State Machine is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

Q is a finite set of *states*

Σ is a finite *input alphabet*

q_0 in Q is the *initial state*

$F \subseteq Q$ is the set of *final states* and

δ is a function from $Q \times \Sigma$ to Q called the *transition function*.

A FSM is illustrated by a directed graph called a *state diagram*. The vertices of the graph correspond to the states and each arc labelled with an input symbol corresponds to a transition from one state to another. To be more precise: an arc from q_1 to q_2 labelled a indicates that

$\lambda(q_1, a) = q_2$. In a DFSM's state diagram there are exactly n transition arcs out of each state where n is the number of the elements in the input alphabet. Figure 3-6 illustrates a state diagram for a DFSM that accepts any string over the alphabet $\{a, b\}$ ending either in ab or ba. The input alphabet is $\Sigma = \{a, b\}$, the initial state, indicated by the arrow, is q_0, and $F = \{q_2, q_3\}$ is the set of final states each represented by two concentric circles.

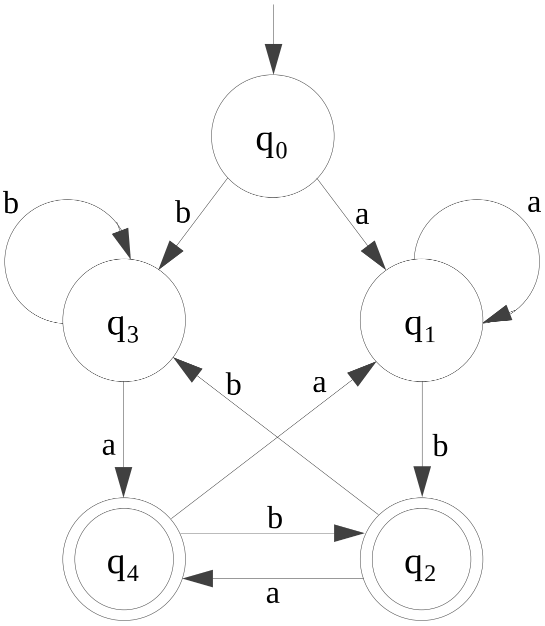


Figure 3-6. A sample State Diagram for a DFSM.

The FSMs that implement a robot's behaviours must produce outputs for actuators. DFSMs do not produce outputs; they just accept or reject a string of symbols. However there are two types of DFSM that generate outputs from a second alphabet. In *Moore machines* output is associated with the states and in *Mealy machines* output is associated with the transitions from one state to another. It can be proved that the two machine types produce the same input-output mappings.

Definition: A Moore machine is a six-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where Q, Σ, δ, and q_0 are as in the DFSM. Δ is a finite *output alphabet* and λ is a function from Q to Δ called the *output function*. $\lambda(q)$ is the output associated with state q.

The output of a Moore machine corresponding to the input string $a_1 a_2 \dots a_n, n \geq 0$, is $\lambda(q_0)\lambda(q_1)\dots\lambda(q_n)$ where $q_0, q_1, \dots, q_n$ is the sequence of states such that $\delta(q_{i-1}, a_i) = q_i$ for $1 \leq i \leq n$.

A layer implementing a robot behaviour must process values from several different input lines rather than just one, so in order to use Moore machines to implement behaviours, we must combine several input alphabets into one. Similarly, we must define an output alphabet which can split into several alphabets corresponding to the outputs of the behaviour. Consequently, to realise a behaviour with n inputs from alphabets $A_1, A_2, \dots, A_n$, we can define a Moore machine with input alphabet $\Sigma = A_1 \times A_2 \times \dots \times A_n$. Note that from the machine's point of view, elements of Σ are indivisible symbols. Similarly, If the layer has several outputs, the output alphabet of the machine will be the Cartesian product of output alphabets of the behaviour.

The state diagrams for Moore machines are similar to those of the DFSMs. The only difference is that in state diagrams representing Moore machines, each state is associated with a unique output. The name of the state is written in the upper half of the circle representing the state, and the output is written in the lower half.

A robot behaviour must respond to the changes in the environment by reading sensor values. Sometimes, the responses to the environmental changes must be a function of different sensor values. For example, when an autonomous mobile vehicle detects a pedestrian on the road the pressure applied to the brake pedal must be a function of speed. Each state of a Moore machine realising a robot behaviour must be able to generate different outputs according to the values of inputs that caused the transition to that state. We define *Extended Moore machines* to add this capability.

Definition: An Extended Moore machine is a six-tuple $M' = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where Q, Σ, Δ, δ, and q_0 are as in the Moore machine and λ is a function from Q to Γ, where Γ is the set of all functions from Σ to Δ and $\lambda(q_0)$ is a constant function.

We can show that each Extended Moore machine is equivalent to a Moore machine. Two machines are *equivalent* iff for every input string they generate exactly the same output string. Let $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ be an Extended Moore machine, then we define a Moore machine $M = (Q', \Sigma, \Delta, \delta', \lambda', q_0)$ as follows:

$Q' = \{(q,d) | q \in Q, d \in \Delta$ and $\exists p \in Q, s \in \Sigma$ such that $\delta(p, s) = q$ and $\lambda(q)(s) = d\}$

$\delta'((q, d), s) = (\delta(q, s), \lambda(q)(s))$

$\lambda'((q, d)) = d$

We leave it to the reader to show that M and M' are equivalent. Since each Extended Moore machine (EMM) has an equivalent Moore machine, here on we use the term EMM to indicate both.

3.3.6 Conventions

In order to simplify the examples we present in the remainder of this chapter, we introduce some notational conventions

- In a robot control system, an input or output line may or may not carry a signal. The way that a signal or lack of a signal is interpreted will depend on the hardware receiving it. For example, a signal value of zero on the input line of a motor may have the same effect as no signal at all. In order to support “no signal” values, we define a new *no-value* symbol η which is added to all the input and output alphabets by default.

The remaining conventions are introduced in order that Moore machines can be specified in a more compact form in which a single transition may be an abbreviation for a set of transitions.

- We define a transition arc labeled with set S as an abbreviation for n transition arcs labeled $x_1,\ldots,x_n$ where $S = \{x_1,\ldots,x_n\}$ as shown in figure 3-7.

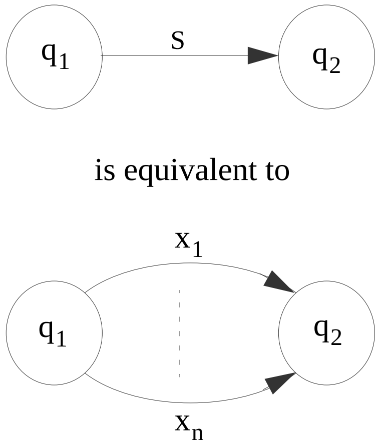


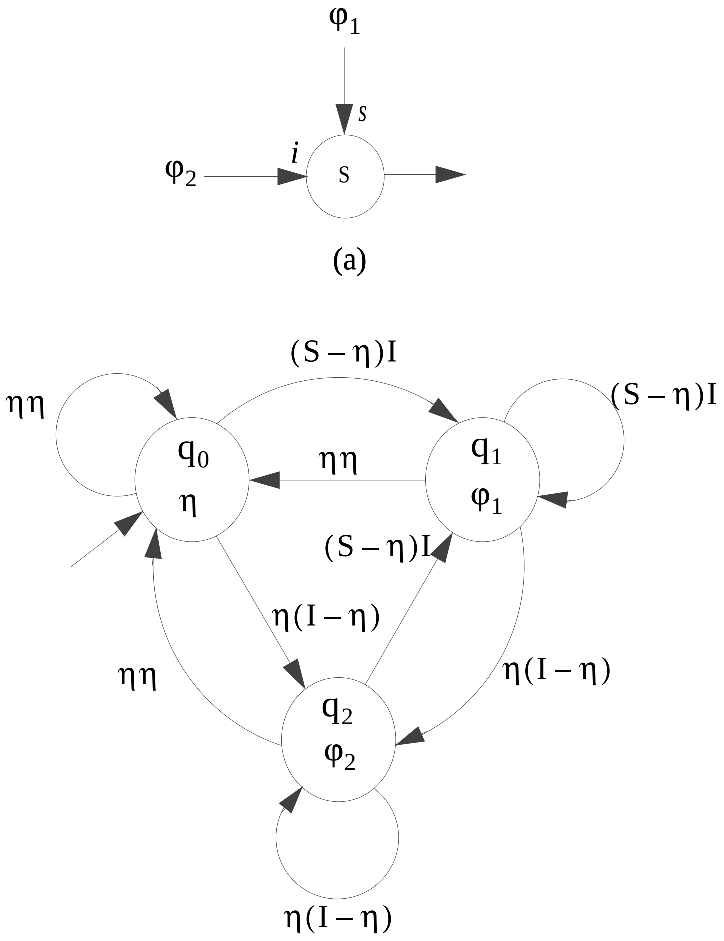
Figure 3-7. A transition labeled with set notation.

- A Cartesian product $X_1 \times X_2 \times \ldots \times X_k$ is abbreviated as $X_1 X_2 \ldots X_k$.
- Where x is an element in set A, we will also use x to mean $\{x\}$ when this meaning is clear in context.
- We indicate the complement of a subset by a bar over the subset. For example, if the input alphabet of a Moore machine is $A \times \{a, \ldots, z\} \times (B \cup \{\eta\})$, where $A = \{1, 2, \ldots, 10\}$ and $B = \{\uparrow, \downarrow, \circlearrowright, \circlearrowleft\}$ then $\overline{\{1, 2\}}\bar{a}\overline{\uparrow}$ denotes the set $\{3, \ldots, 10\} \times \{b, \ldots, z\} \times \{\downarrow, \circlearrowright, \circlearrowleft, \eta\}$.We extend this complement notation by applying it to consecutive components of a Cartesian product. For example, $x_1 x_2 \ldots x_{k-1}\overline{x_k \ldots x_l}x_{l+1} \ldots x_n$ denotes the set $\{x_1\} \times \{x_2\} \times \ldots \times \{x_{k-1}\} \times (X_k X_{k+1} \ldots X_l - \{x_k x_{k+1} \ldots x_l\}) \times \{x_{l+1}\} \times \ldots \times \{x_n\}$ where $x_i \in X_i$ for $k \le i \le l$. The usefulness or applicability of this notation in particular situations depends on how the components of a Cartesian product are ordered.

- If x is an n-tuple then we write x_i to denote the i*th* component of x for $1 \le i \le n$. In an EMM the output associated with each state is a function of the input value that caused the transition to the state. We use φ to denote the input value that caused this transition. For example when the output of a state in an EMM is the i^{th} component of the input value, we use φ_i to denote the output of that state. We use this convention extensively in the next section where we introduce the state diagrams for EMMs for suppression and inhibition.

3.3.7 Suppression and Inhibition

In the subsumption architecture the conflicts between different behaviours that wish to send messages to the same destination (an input to another behaviour or an input to an actuator) is resolved by using suppressors and inhibitors which can be defined as EMMs. Suppression and inhibition EMMs prioritize behaviours by selecting only one of the two components of their input symbols as their output. In the suppression EMM, the output is the first component of the input as long as this component is not η. When the first component is η, the output is the second component of the input. Figure 3-8(a) illustrates a suppressor and figure 3-8(b) illustrates the state diagram for the EMM that realises it. The input alphabet of the EMM in figure 3-8(b) is $S \times I$ where S and I are the alphabets of the suppressing and normal inputs respectively. Notice that both S and I contain η. The output alphabet for the suppression EMM is $(S \cup I)$.

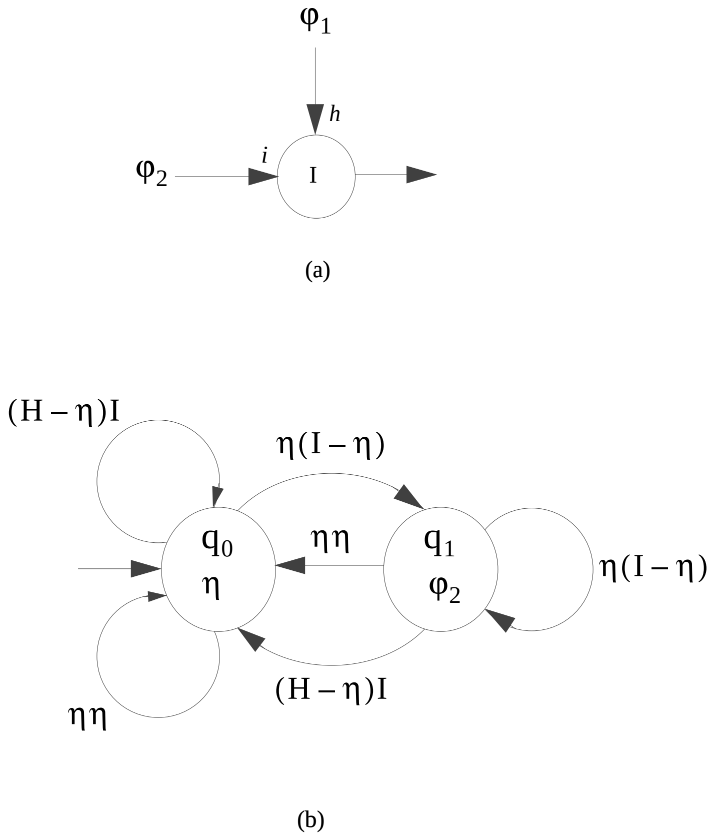


(a)

(b)

Figure 3-8. A suppression EMM state diagram.

In the inhibition EMM, the output is η as long as the first component of the input is not η. When the first component is η the output is the second component of the input. Figure 3-9(a) illustrates an inhibitor and figure 3-9(b) illustrates the state diagram for the inhibition EMM the input alphabet of which is $H \times I$. The output alphabet for the inhibition EMM is I.



(a)

(b)

Figure 3-9. An inhibition EMM state diagram.

3.3.8 Control System Block Diagram

In our model of the subsumption architecture, a layer consists of a single behaviour implemented as an EMM. The outputs of a higher priority behaviour can subsume or inhibit the inputs and outputs of a behaviour with lower priority. A behaviour with higher priority can also monitor the data generated by lower priority behaviours by accepting their outputs as part of its input. The outputs of lower layers used as inputs for higher layers can be used as simple inputs, but can not be used to suppress or inhibit other inputs: however they might be suppressed or inhibited by others. This restriction maintains the priority scheme between different layers. Figure 3-10 illustrates the control system block diagram for level-0.

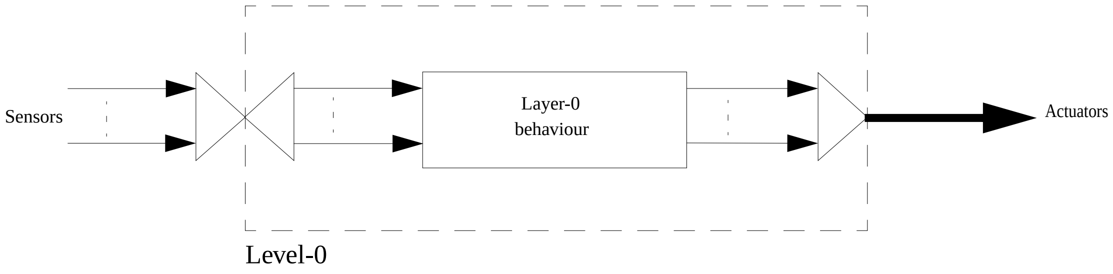


Figure 3-10. Control System Block diagram for level-0.

In figure 3-10 each thin line carries a single value while the thick output line carries a vector of values. The triangle preceding level-0 is a *collector*, which receives a number of individual and vector inputs and generates an output vector that contains all the individual elements of the inputs by concatenating its input values and vectors. The triangle following the collector is a *distributor.* A distributor receives a vector and, in general, generates a number of values or vectors each of which is composed of a subset of individual elements of the input vector. In this case, the output of the distributor are individual values, as indicated by the thin lines. In figure 3-10, the distributor is part of the internal structure of level-0 while the preceding collector is not.

A level-0 control system is a transduction from sensor values to actuator values with no feedback from outputs to inputs. Feedback in a control system is used to remember the previous state of the system; however, since a level-0 control system in this model is implemented as a single EMM, the state of the EMM completely represents the state of the system, so feedback is unnecessary.

Figure 3-11 illustrates the control system block diagram for level-N. Notice that there is no internal interconnection between layer-N, level-(N-1) and the subsumption function blocks within level-N. All the interconnection between different layers in the control system is achieved through a single external feedback.

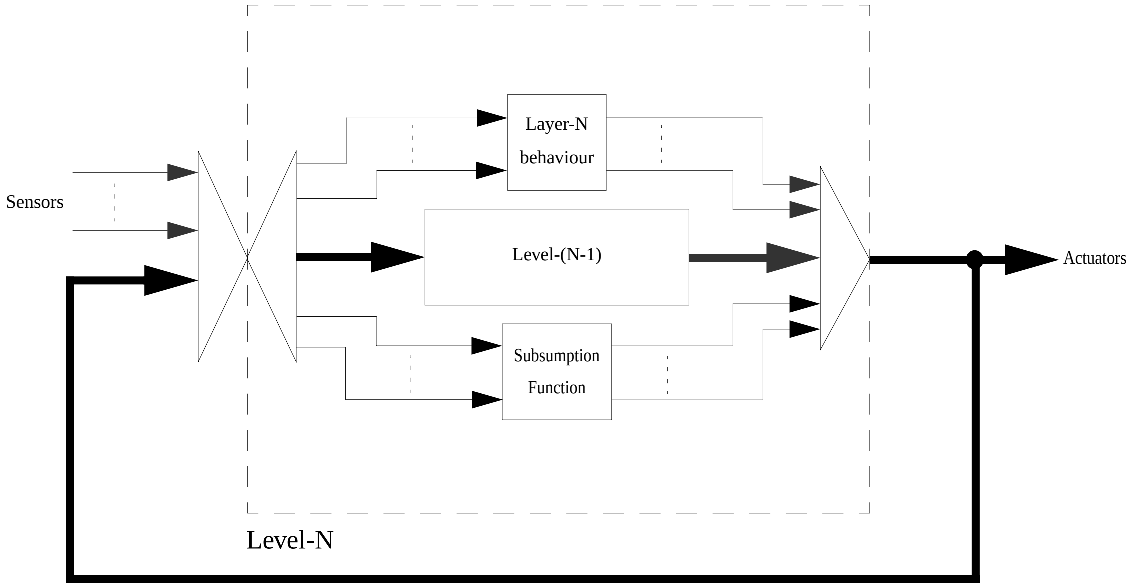


Figure 3-11. A recursive Control System Block Diagram for level-N.

In the subsumption function, the outputs of layer-N can be used to subsume or inhibit other inputs. The inputs to level-(N-1) are items selected from the external feedback vector and the sensor values. Inputs to layer-N are items selected from the external feedback vector corresponding to the outputs of level-(N-1) and the sensor values. The items selected from the external feedback vector corresponding to the outputs of level-(N-1) enable layer-N to monitor the outputs of lower layers if required.

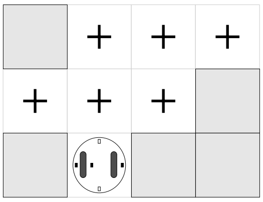


Figure 3-12. A robot and maze.

As an example of how the behaviours and subsumption functions can be used to control a system, consider a car in a square grid. Each cell is either empty or occupied with a block. Each empty cell in the grid is marked with a black cross in the centre. The car is equipped with four infrared sensors mounted at the front, right, left and the back of the car, and two motors at either side of the car. The sensors look horizontally, sensing the presence of objects. Blocks placed on the board create a maze as illustrated in figure 3-12. The robot is also equipped with an infrared sensor mounted underneath the robot slightly to the left side, detecting the black crosses. This sensor is used to identify when the car is in the middle of a cell. It can also detect when a 90° rotation to the right or left is complete because of its left offset.

The most trivial behaviour of the robot, called Command, is to move from one cell to another in either the forward or reverse direction, or to rotate in the same cell around its centre 90° clockwise or counterclockwise. A level-0 control system can implement such behaviour. The input alphabet for the Command behaviour is $C \times U$ where C = { ↑ , ↓ , ↻, ↺} and U={ █ , ▯ }. The symbols █ and ▯ respectively represent the presence or absence of a marking beneath the underneath infrared sensor. The symbols in C, from left to right, represent the commands move forward, move backward, turn right and turn left. The output alphabet is $M_L \times M_R$ where M_L = M_R = { ⬆ , ⬇ , η }. M_L and M_R represent the left and the right motors, respectively. ⬆ indicates a motor is in the forward direction and ⬇ represents a motor in reverse. Figure 3-13 illustrates the state diagram for the Command EMM.

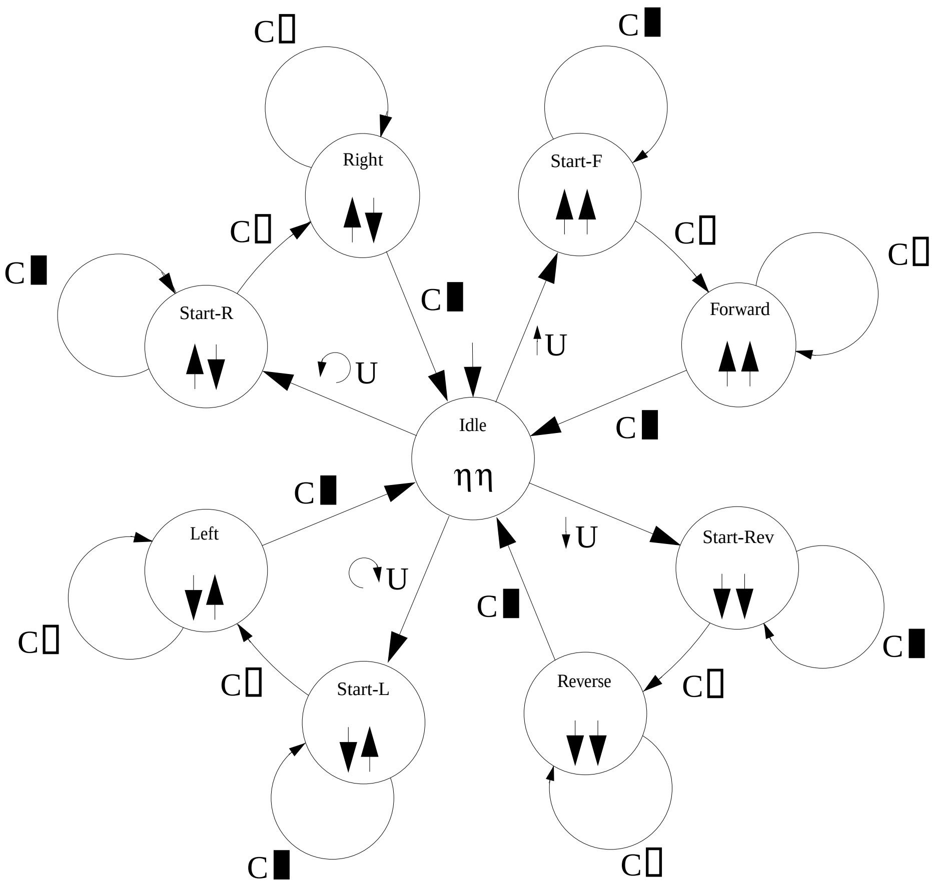


Figure 3-13. State diagram for the Command EMM.

In figure 3-13, the output ↑↑ will move the robot one cell forward when applied to the motors. A symbol with two arrows in different directions makes the robot rotate 90° clockwise or counterclockwise around its center in the same cell. Figure 3-14 shows the block diagram for the level-0 control system.

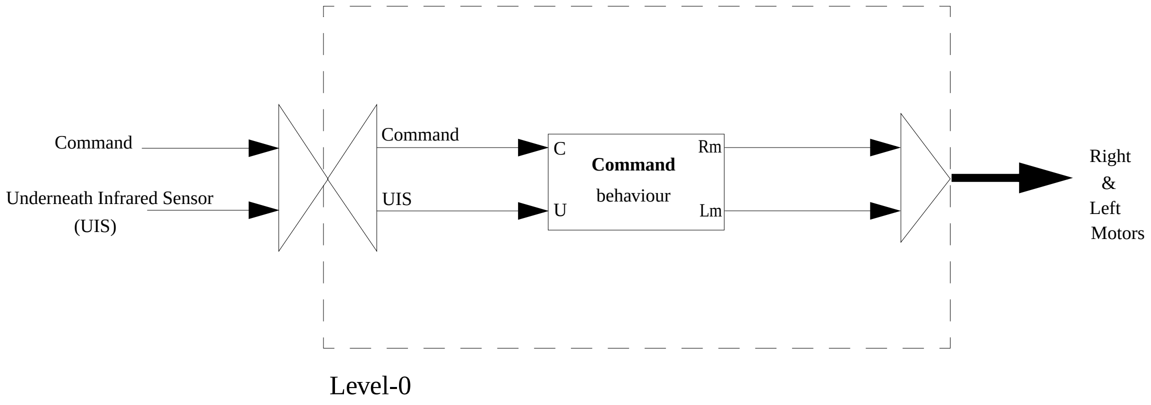


Figure 3-14. Level-0 control system block diagram.

A simple algorithm for traversing a maze and finding an exit is to keep a wall at the right side of the robot. We will call this the Forward behaviour of the car. A layer-1 control system can implement such a behaviour. First we define the input and output alphabets of the EMM that realises the Forward behaviour. The value of each horizontal infrared sensor is represented by rectangular symbols similar to underneath infrared sensor. An empty rectangular symbol (▯)represents an opening in the direction corresponding to that sensor. A solid rectangular symbol (▮) represents the presence of a block in that direction. The input alphabet for the Forward behaviour is $F \times R \times M_L \times M_R$ where F = R ={ ▮ , ▯ } and F and R represent the values of the front and the right sensors, respectively. The output alphabet for the Forward EMM is C. Figure 3-15 illustrates the state diagram for the Forward EMM.

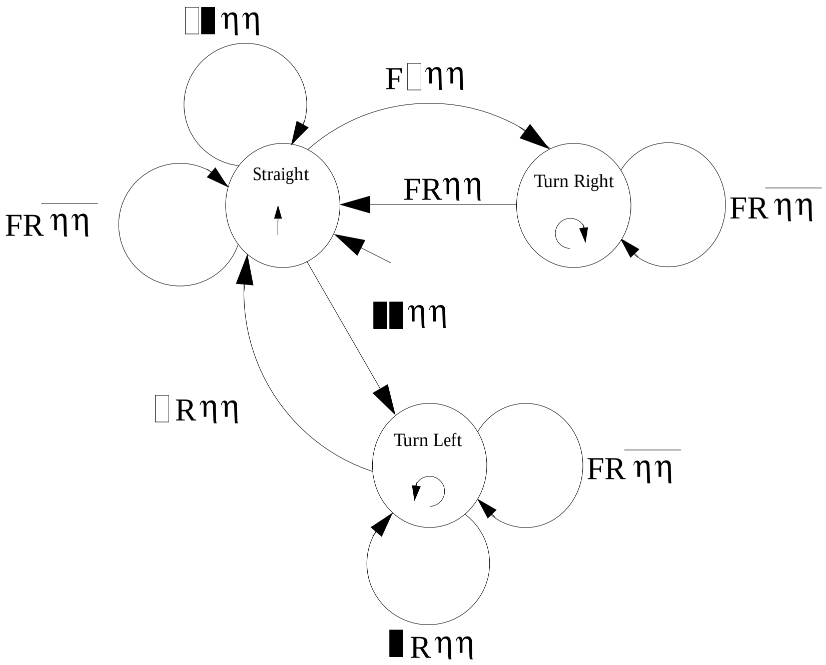


Figure 3-15. State diagram for the Forward EMM.

This example assumes that the cell the car starts in is not a trap or loose cell; that is, there is at least one opening from the start cell and at least one block in one of the four cells adjacent to the start cell.

Figure 3-16 illustrates the level-1 control system block diagram. In figure 3-16, the external feedback vector carries 4 elements.

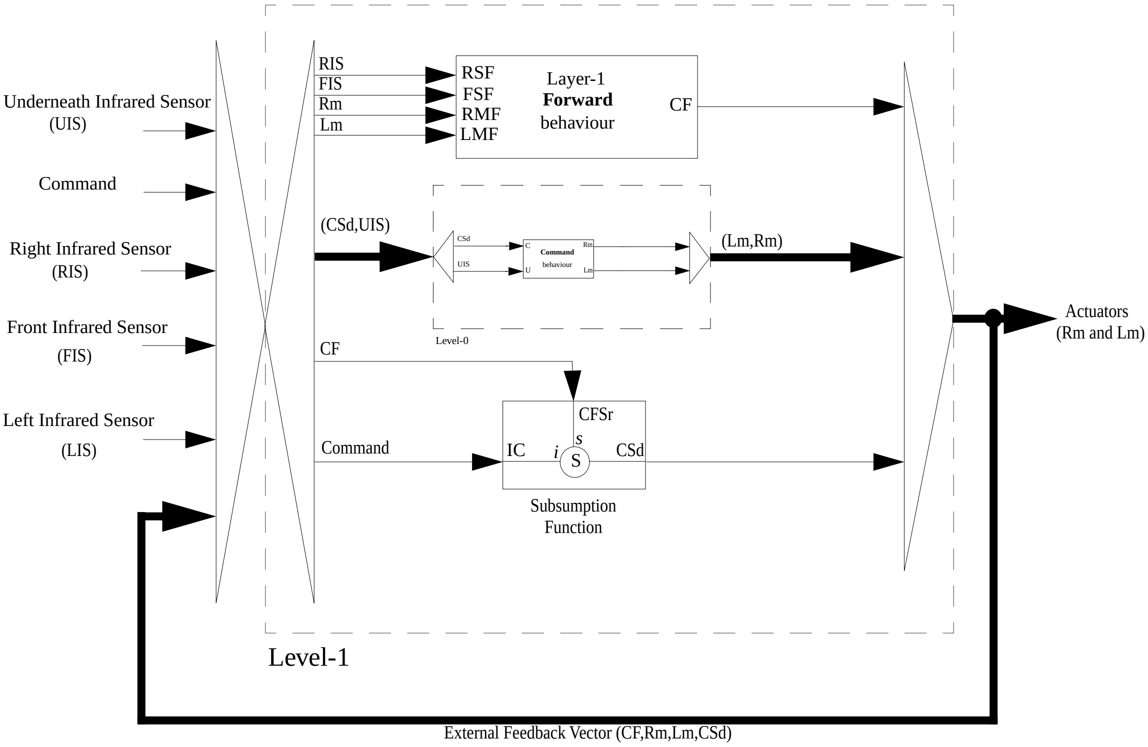


Figure 3-16. level-1 control system block diagram.

In the Forward behaviour the robot takes two left turns when it comes to dead ends. A new Backup behaviour can effectively improve the traverse time by avoiding unnecessary turns when a robot encounters a dead end. This new behaviour is added to the level-1 control system to obtain level-2. Figure 3-17 illustrates the state diagram for the Backup EMM.

The input alphabet for the Backup EMM is $L \times F \times R \times B \times M_L \times M_R$ where L = F = R = B ={ ▮ , □ }, and L, F, R and B represent the values of the left, front, right, and back sensors, respectively. For example, □▮□▮ indicates an opening at the left side, an obstacle in the front, an opening at the right side and an obstacle behind the robot. The output alphabet for the Backup EMM is C. In figure 3-17, the machine stays in the initial state Idle generating η as long as the robot does not encounter a dead end. In the Backup EMM, transitions occur only when the Command EMM is in the Idle state, that is when the robot has finished a move from one cell to another or has finished a turn to the left or right.

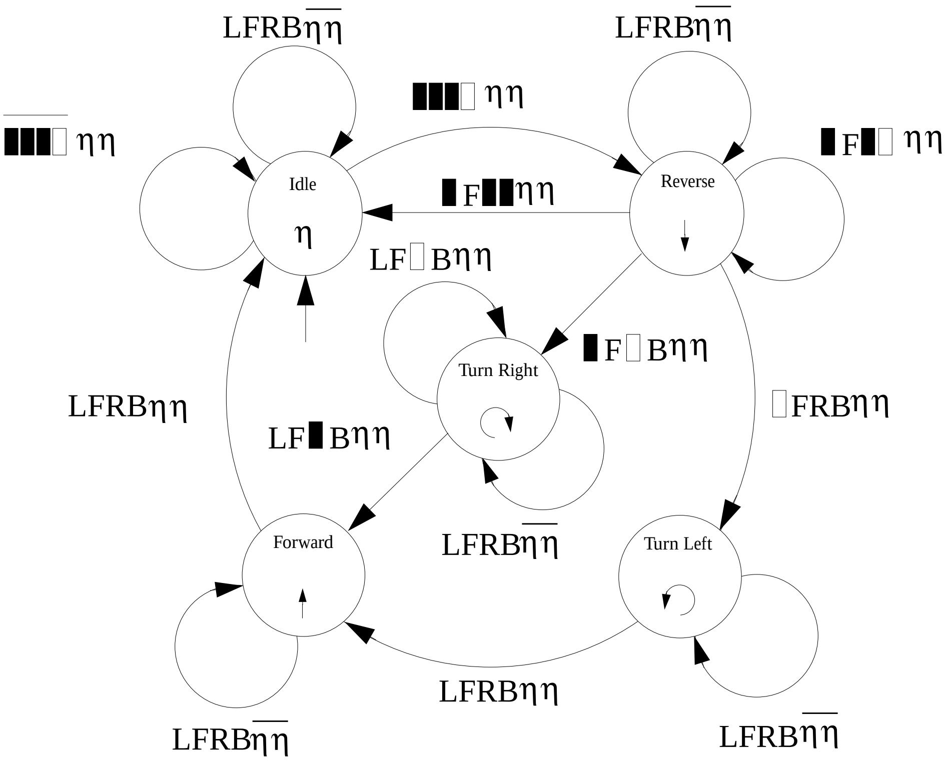


Figure 3-17. State diagram for the Backup EMM.

Figure 3-18 demonstrates the level-2 control system block diagram.

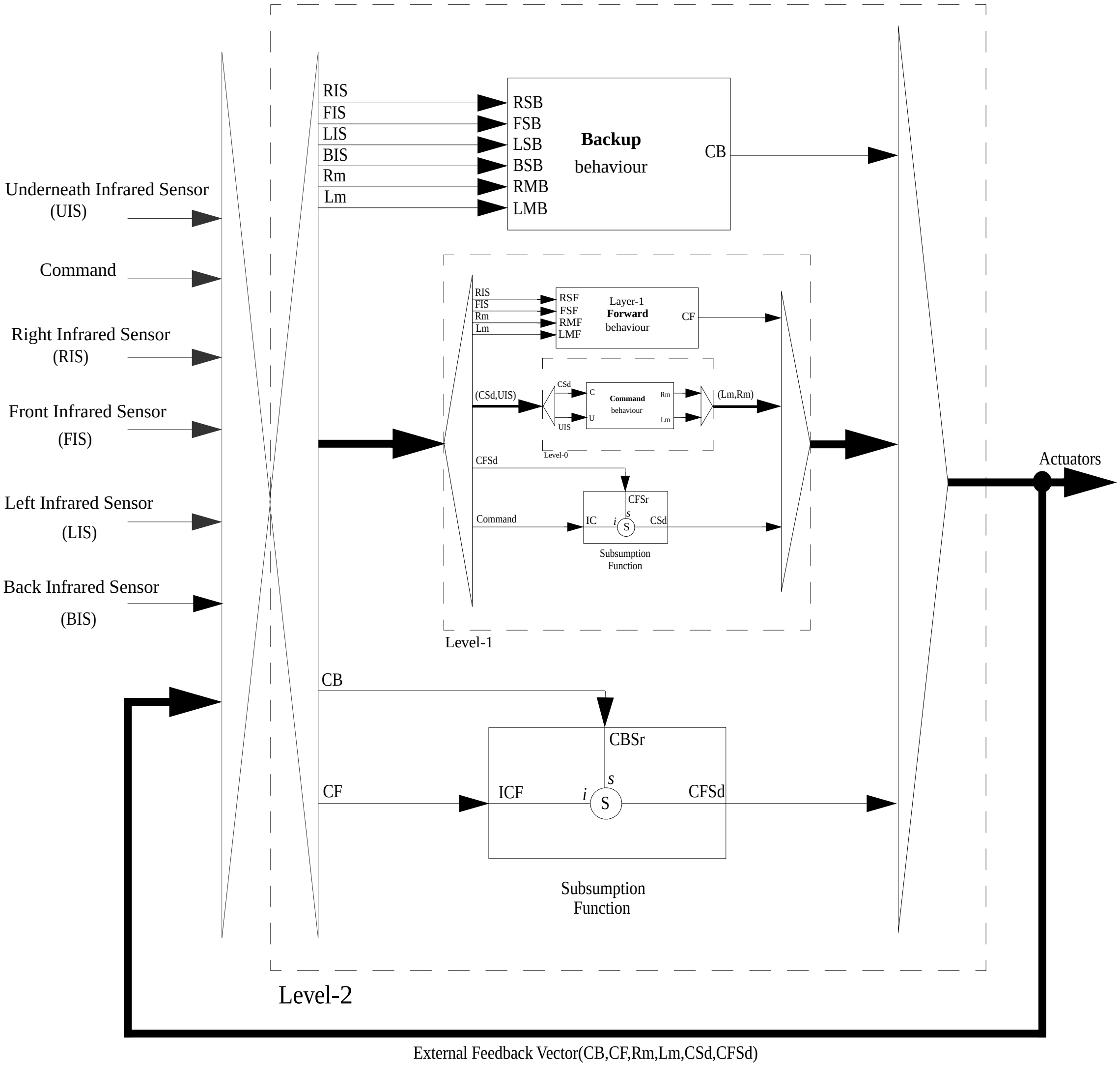


Figure 3-18. Level-2 control system block diagram for the Lego car.

4 Hardware Definition Module

Achieving a general purpose visual programming language for robot control which concretely represents the physical characteristics of a robot, while at the same time maintaining the generality of the language presents an interesting dilemma [16]. The programming constructs of such a language must be adequately specific to directly represent the physical characteristics and actions of a specific robot, while appropriately general to reflect the visual representations of a wide variety of robots. Although bringing these two characteristics together in a visual programming language is desirable their contradictory nature implies that conventional programming language design is unlikely to provide a solution.

4.1 Reconciliation of Generality and Concreteness

To achieve a concrete, general-purpose VPL for robot control, Cox and Smedley propose in [16] that a robot control programming systembe constructed from two major modules. The Hardware Definition Module (HDM), is the first part of the system in which the structure of the robot and its environment is defined. The HDM environment provides the necessary tools to customize the overall programming language for a specific robot. The Software Definition Module (SDM), is the second part of the system in which the control program is defined by using the control model created in Chapter 3 and the more concrete programming constructs previously defined in HDM. The SDM environment simulates the robot in action and provides the required tools to define the different levels of a robot's behaviours by direct manipulation of parts representing that robot as described in Chapter 5.

4.2 Hardware Definition Module

The HDM is a design environment similar to those used in Computer Aided Design (CAD) tools. The purpose of HDM is to model a robot and the objects in its environment. These models are used in SDM as the basis for simulation used to drive the programming process.

The concept of HDM and SDM was originally introduced in [16] with an emphasis on HDM. The work presented in that paper is an explanation of how objects and robots are designed in HDM by working through an extended examplewhich gives the flavour of how HDM should work. However, it does not describe in detail how the generated models are used in SDM to create a control system. The work described in [16] very briefly outlines the use of FSMs for implementing the behaviours of a robot with an example, but does not address the remainder of the subsumption model. The design environment suggested in [16] is a 2D environment but could easily apply to 3D given appropriate 3D rendering and manipulation technology.

Since the purpose of this work is to introduce a mechanism by which a control system is created in SDM, we will not discuss HDM in detail; however, we will briefly introduce the functions that HDM must perform in order to support SDM. Where we feel there is a need for minor modification in the HDM in [16] because of the 3D nature of this work, we will explain the modification. We invite the reader to consult [16] for a discussion of HDM.

4.2.1 Requirements

As we mentioned in Chapter 3, a robot consists of sensors, actuators and body. Sensors gather information from the environment; actuators transform the environment in some way, and the body provides the connecting structure but is logically is insignificant for programming purposes.

4.2.1.1 Objects

The most trivial component created in HDM is an object. The model for an object consists of some geometrical information and a set of *properties*.

4.2.1.2 Properties

Each object in the operating environment of a robot may have a set of properties. Properties are physical attributes of objects that can effect the sensors of a robot if the sensors are defined to be sensitive to those properties. Some properties can be effected by the actuators if those properties are not defined as *static properties*. A static property is one that is defined once and will be fixed in SDM. A *dynamic property* is one that can be manipulated by actuators. Since properties will be implemented as variables we need to define their types.

4.2.1.3 Types

As in most programming systems, we require the notion of type in both HDM and SDM. variables used as object properties must have types; so must sensor inputs and actuator outputs. In both HDM and SDM we need access to a set of *standard types* such as integer, float, character, string, and boolean. A programmer should also be able to create types. Thisrequires a *type editor* with which a programmer can create new types that are a subset of standard types, or create symbolic types, such as those we used in Chapter 3 in our Car example. The icons for a symbolic type might be created in a drawing window or pasted from elsewhere.

Clearly the exact nature of the facilities provided by such a type editoe will varydepending on the characteristics of the type being defined, for example, whether it is finite, countable, not countable, or consists of typeable characters.

4.2.1.4 Sensors

The model for a sensor must contain an *external function* for reading the values from the actual hardware. It must also include a mechanism by which the input values can be rerouted from the actual hardware to the values generated in the simulated environment. A sensor is a physical entity, therefore we need a model for its physical structure. In the SDM we will use different appearances of a sensor for programming purposes, therefore, HDM must provide a mechanism by which the appearance of a sensor changes to reflect the changes in the read value from the hardware or the simulated environment. In one extreme, a sensor can have as many appearances as the range of its input value where the input is a countable type.

4.2.1.5 Actuators

The model for an actuator, similar to that for sensors, must contain an *external function* for writing values to the actual hardware. Since SDM relies on providing accurate simulation of the robot, there is a need in HDM for a mechanism for defining the way the simulation environment is transformed in response to values fed to simulated actuator . Similar to sensors, actuators are physical entities and therefore we need a geometrical model for simulating their mechanics. Since most actuators can be categorized as mechanical actuators, the geometrical model for an actuator is expected to be more complex than those of sensors. For example, creating an articulated arm for a robot requires generating dynamic structural dependencies between components to reflect the physical characteristics of the real arm. Bilateral attachments, parent-child relationships and creating joints are some of the required structural dependencies that enable us to create complex actuators.

4.2.1.6 Attachment

Once we have built the required sensors, actuators and the body, we need to *attach* these components to create a robot as a single unit. The attachment process should closely investigate the mechanical effects of the actuators on the robot itself as a single unit. For instance, in our previous car example we specify the effects of the right and left motors on their geometrical models, however, when we attach the components together the combined effect of the motors on the geometrical model of the robot must follow the rules of mechanics.

5 Software Definition Module

The Software Definition Module (SDM) is a simulation environment in which control programs are created by directly manipulating the models generated in HDM for a robot and its environment. In SDM the subsumption model developed in Chapter 3 is adopted as the control system.

To show how a complete control system is created in SDM, we use our previous car example to demonstrate the programming steps. When SDM is started a menu bar containing File, Edit, View, Behaviour, Simulation and Library menus appear. To create a new file, the New item from the File menu is selected. A normal open file dialogue box appears from which the desired HDM library file can be selected. In this example, the Car.lib Library file is selected. This opens a Library Palette named Car Library containing names and miniaturised pictures of the Car, Tile and Obstacle previously defined in HDM. A workspace window consisting of four views, similar to those in HDM, named SDM:Car.sdm that contains one instance of the car is also opened. The four views are called Plan, Front, Left and Camera that provide the top, front, left and camera views of the simulation environment, respectively. There is also an empty pane for defining input and output registers at the top of the workspace window, called the *register pane*, that contains Add Reg. and Delete Reg. buttons for adding and deleting registers.

The Camera view is undefined at the beginning a camera can be added to the scene by selecting New Camera from the View menu. A camera provides a user-defined view of the design scene in which the programmer can move around, assemble the simulation environment, and manipulate objects freely. Since in the car example only 2 dimensions are significant, the Plan view is expanded to occupy the entire workspace window by selecting the Plan item from the View menu. To create a simulated environment, a set of tiles and obstacles are copied from the Car Library Palette by dragging and dropping them into the workspace window. Figure 5-1 illustrates the Car library palette, the SDM workspace and Menu bar after a set of obstacles and tiles are copied into the workspace.

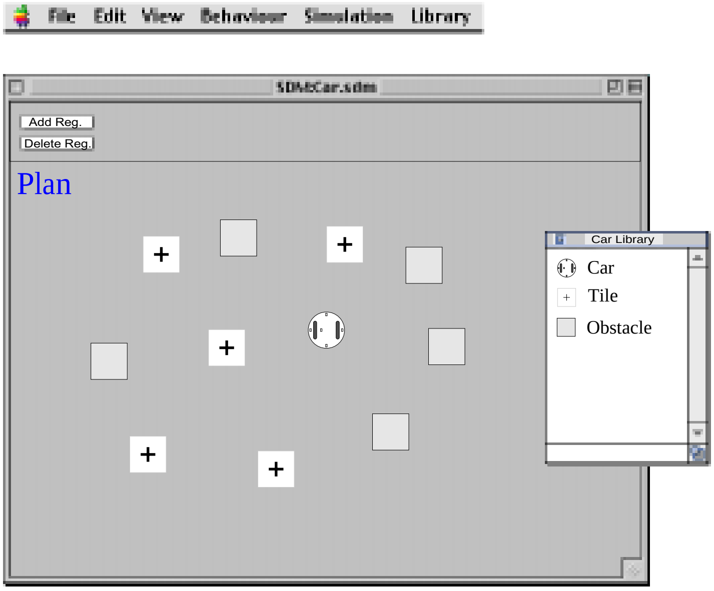


Figure 5-1. SDM workspace and the Car Library Palette.

Once a simulated environment is created in the workspace by arranging the objects and robot, the programming can be started by defining the most trivial behaviour of the robot. To define this first behaviour, we select the New Behaviour item from the Behaviour menu. An empty floating palette named Layer-0 appears, and will be used to define the inputs, outputs and the states of the Extended Moore machine (EMM) realising layer-0 behaviour. The Layer-0 palette contains an editable text box at the top that can be used to name the behaviour. The name entered in this box can be edited at any time if the user wishes to rename the behaviour. The name of the behaviour has no significance from the programming point of view, although an intuitive name can effectively add to the clarity of the program. We name the first behaviour Command as in section 3.3.8. Figure 5-2 illustrates a workspace created by arranging objects defined in HDM and an instance of the robot along with the Layer-0 floating palette.

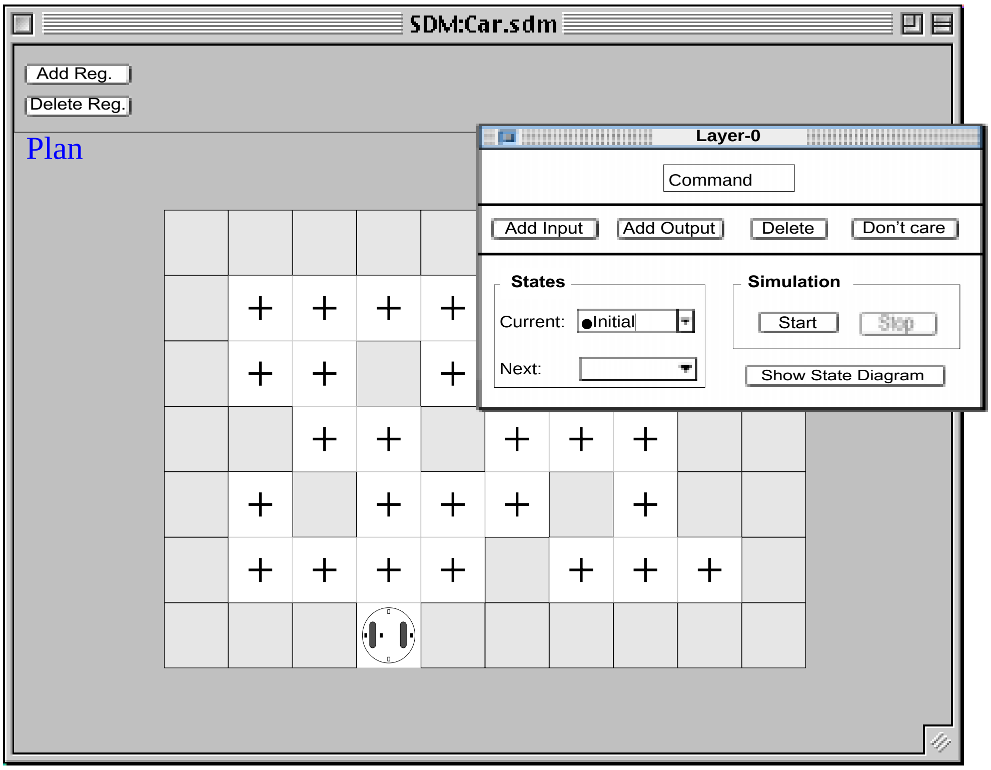


Figure 5-2. SDM workspace and the Layer-0 floating palette.

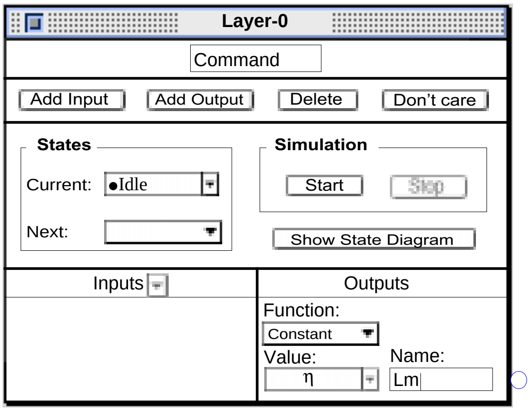
Although the aim of the programming process is to generate EMMs, the philosophy of SDM is to focus the programmer's attention on making the simulated robot behave properly rather than on the underlying abstractions.

The first step in programming the layer-0, Command, behaviour is to define the initial state. The default name for the initial state is Initial, however it can be changed by the user. As figure 5-3 illustrates, we rename the initial state in this example to Idle. The Character • before the name indicates that this is the initial state of the behaviour. This character can not be deleted from the initial state's name and every behaviour can have only one state the name of which starts with this character; that is each behaviour can have only one initial state. The next step in defining the initial state of the Command behaviour is to define and set the output(s) of the initial state.

The Command behaviour must generate appropriate outputs for both the left and right motors to move the car one cell forward or backward, or turn the car 90° clockwise or counterclockwise. To define an output we click the Add Output button. A new output is added to the Layer-0 floating palette. An output is represented by a panel containing the output name in an editable text box called Name, initially containing "untitled"; a Function pop-up that specifies how the output value is computed and initially set to Constant; a Value pop-up and box for setting the output value; and a terminal represented by a small circle attached to the right of the panel. The value pop-up and box are enabled and disabled depending on the item selected in the Function pop-up and the type of the connection made to the terminal. Since initially, the Function pop-up contains Constant and there is no connection made to the terminal, so the Value box is uneditable and set to η and the pop-up is disabled. When the item selected from the Function pop-up is not Constant, the Value box is uneditable and the pop-up is disabled indicating that the output is computed by a function. Where the Function pop-up is set to Constant and the output type is textual such as standard types or textual user defined types, the Value box is set to editable and the pop-up is disabled. The Value pop-up is enabled only when the Function pop-up is set to Constant and the output has a user defined iconic type. In this case the Value box is set to uneditable and the output value can be selected from the Value pop-up.

Like the behaviour name, the name of an output has no logical significance but can be chosen to improve the readability of the program. A new output is named “Untitled” by default. The name can be edited at any time. In our example, we rename the output Lm as figure 5-3 shows

.



Current: ●Idle

Next:

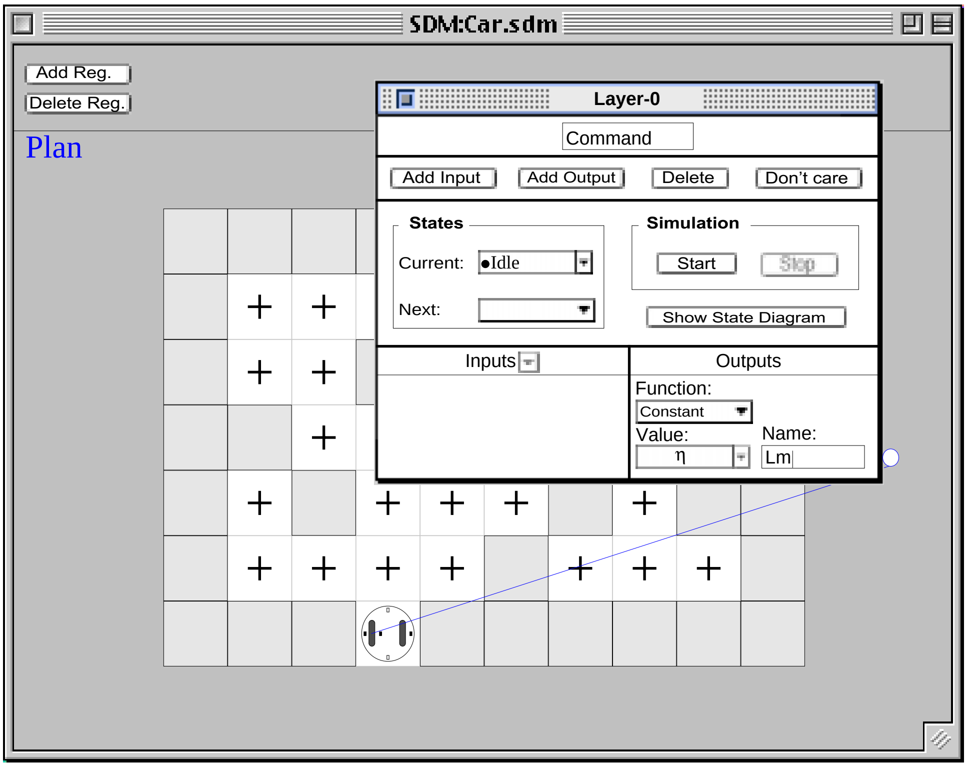
Function: Constant

Value: η

Name: Lm

Figure 5-3. Defining an output.

The Lm output will generate values for the left motor of the car, so we need to connect it to the left motor. Connecting the Lm output to the left motor is accomplished by clicking on the terminal and dragging, while the mouse button is depressed a rubber band connects the terminal and the cursor, and as the cursor passes over any item in the simulated environment to which the terminal can be connected, the item is highlighted. In our example, we choose the left motor of the car. When the mouse button is released, the terminal and left motor are connected as shown in figure 5-4. Note that a connection can be built in either direction. For example, we could have clicked down on the left motor and dragged a rubber band to the terminal.



Current: ●Idle

Next:

Function: Constant

Value: η

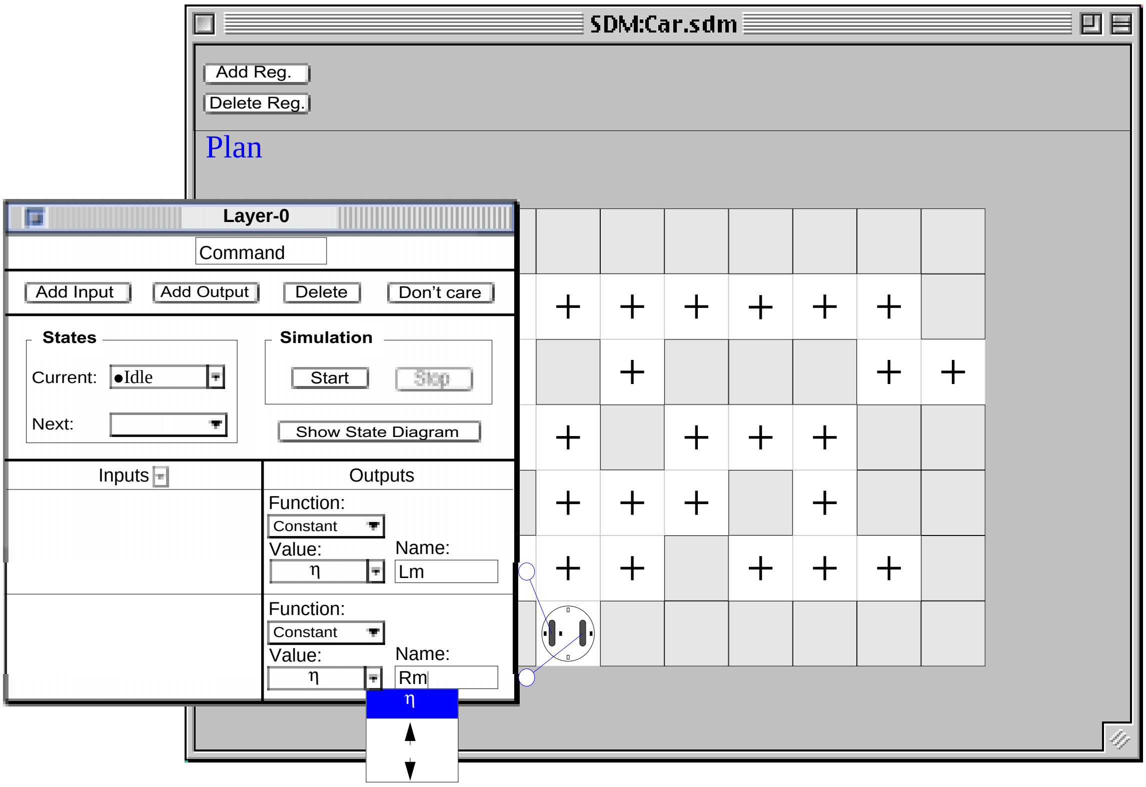
Name: Lm

Figure 5-4. Connecting an output to an actuator.

Once the Lm output is connected to left motor, the type of the left motor is assigned to the Lm output. Since the left motor has a user-defined iconic type and the Function pop-up contains Constant, the Value box is set to uneditable and the Value pop-up is enabled. We can change the output value by selecting a value from the pop-up. In our example we want the car to stay still in the Idle state, so we leave η as the output.

Where the output should be presupplied or user defined function, the desired function can be selected from the Function pop-up. The Function pop-up has also an Add new function option. When this option is selected, a dialogue appears in which the function is named. On closing the dialogue, the name of the new function is added to the Function pop-up, and a programming environment is opened in which to build the function. This programming environment can be a programming window of the underlying language, for example a Prograph method window where the underlying language is Prograph.

We define a second output corresponding to the right motor of the car, name it Rm and connect it to the right motor of the car in the same way. Figure 5-5 shows the simulated environment and the contents of the Rm Value pop-up.



Command

Current: ●Idle

Next:

Function: Constant

Value: η

Name: Lm

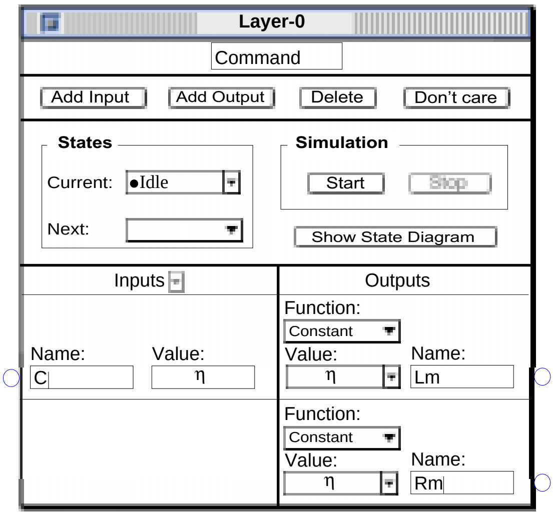
Function: Constant

Value: η

Name: Rm

Figure 5-5. Setting the output values.

In the Idle state of the Command behaviour we want the car to stay still waiting for a move or turn command. We define an input for the Idle state by clicking the Add Input button which adds a new input to the Layer-0 floating palette. An input is represented by a panel containing the input name in an editable text box called Name and initially containing “untitled”; and an uneditable Value box initially containing η; and a terminal as figure 5-6 shows. We rename the input C.



Current: ●Idle

Next:

Name: C Value: η

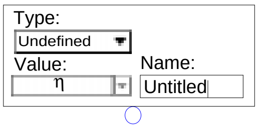
Function: Constant
Value: η Name: Lm

Function: Constant
Value: η Name: Rm

Figure 5-6. Defining an input.

Like behaviour and output names, the names of the inputs are insignificant and can be changed at any time. Appropriate names help programmers to differentiate between the inputs thereby increasing the readability of the program. Names of the inputs and outputs are internal to the behaviours and act like comments from the programmer's point of view.

Since the C input must receive its values from a source that does not correspond to a sensor on the car, we need to define a register where we can set the value of the C input. To define a register we click the Add Reg. button at the top of the workspace window which places a new register in the register pane. As figure 5-7 shows, each register is represented by a panel containing an editable Name text box initially "Untitled"; a Type pop-up for defining the type of the register contained Undefined initially; a Value box and pop-up for setting the value; and a terminal. The names of registers have no logical significance and can be



Type: Undefined

Value: η Name: Untitled

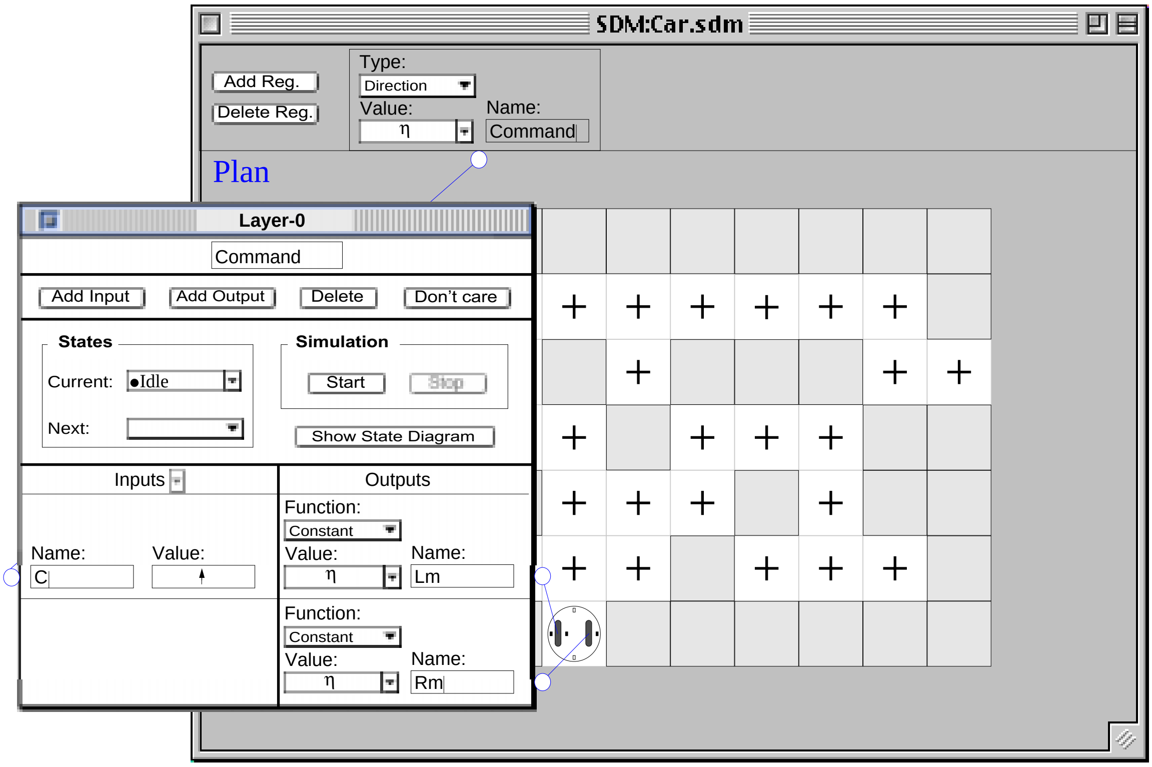
Figure 5-7. A register pane.

changed at any time. The Type pop-up contains an Undefined option which is the default type; a set of standard, built-in types; some user defined types imported from HDM, or already defined in SDM; and a New type option. Selecting the New type... option opens a dialogue in which the new type must be named. On dismissing the dialogue the name of the type is added to the user defined types of the pop-up and a type editor similar to that used in HDM for defining sensors' and actuators' types is opened. When the new type has been created and the editor window is closed, the new type is assigned to the register.

The Value box and pop-up are very similar to an output Value box and pop-up. Initially, the Value box is uneditable and contains η , and the pop-up is disabled. When a standard type or a textual user-defined type is selected from the Type pop-up, the Value box changes to editable and the pop-up remains disabled. The pop-up is enabled when an iconic user defined type is selected or created, in which case the Value box will be set uneditable.

In our example we have renamed the register Command and create a type that contains four symbols representing move forward, reverse, turn right, and turn left commands as described in section 3.3.8. We name the new type "Direction". Since we have defined an iconic type, the pop-up is enabled and the Value box remains uneditable.

Once the type of the Command register is defined, we connect the register to the C input of the Layer-0 floating palette as previously described. This also assigns the type of the Command register to the C input. Figure 5-8 illustrates the simulation environment after the Command register is defined and connected to the C input.



Type: Direction
Value: η
Name: Command

Command

Current: ●Idle
Next:

Name: C
Value: ↑

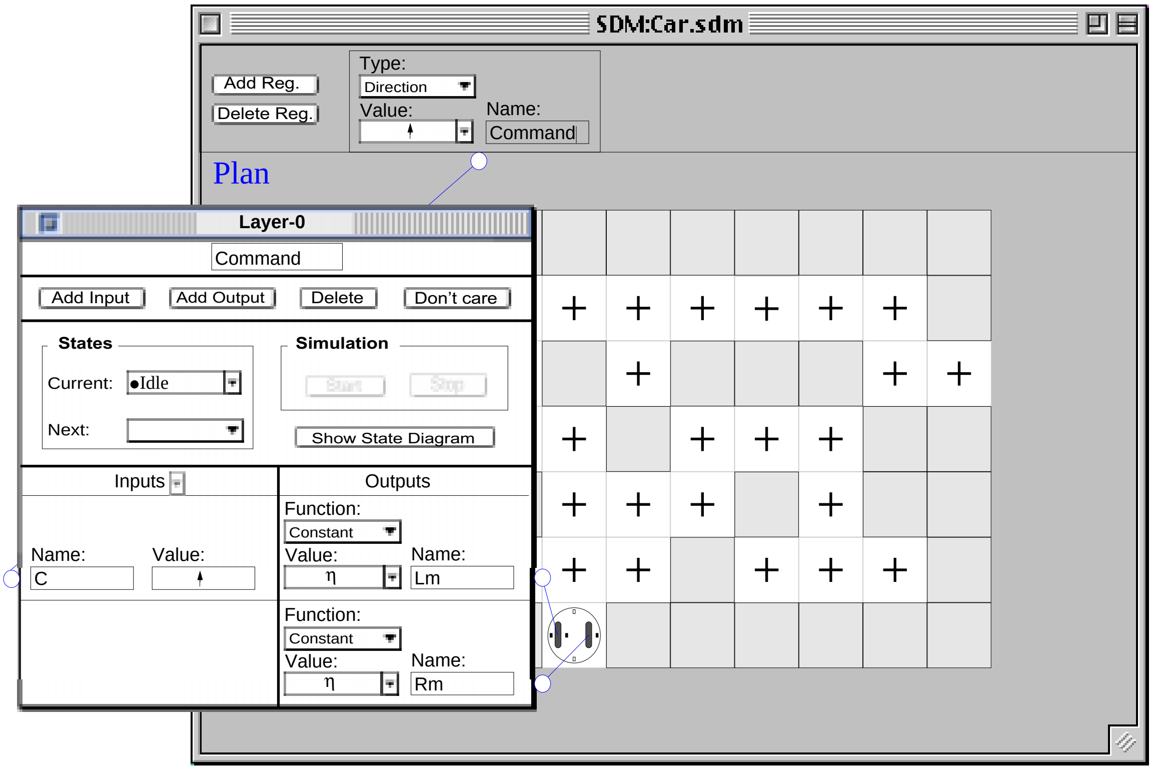
Function: Constant
Value: η
Name: Lm

Function: Constant
Value: η
Name: Rm

Figure 5-8. Defining a register.

In our example, after we created the Command register we immediately defined a type for the register before we connected it to the C input, however, we could easily change the order of the process by connecting the register to the C input and then defining a type for the register. In either case, the type of the register will be assigned to the C input.

To finish the definition of the Idle state we press the Start button. The simulation is started, the Start button changes to a disabled Resume button, the Stop button is enabled, and the values of inputs and outputs are shown in the Layer-0 floating palette. Since in the Idle state the outputs for both motors are set to η the car remains motionless and the simulation continues until we change the value of the Command register. To change the value of the Command register we select ↑ from the register's Value pop-up. ↑ appears in the input C Value box and the simulation stops immediately because any change in the values of the significant inputs of a state indicates a need to change state. When simulation is stopped both the Start and Stop buttons are disabled. Figure 5-9 illustrates the environment after the simulation is stopped.



SDM:Car.sdm

Type: Direction
Value: ↑
Name: Command

Layer-0

Command

Current: ●Idle
Next:

Inputs
Name: C
Value: ↑

Outputs
Function: Constant
Value: η
Name: Lm

Function: Constant
Value: η
Name: Rm

Figure 5-9. Idle state when the simulation is stopped.

The input value that caused the termination is displayed, as are the current output values. In our example, the output values are simply constant outputs defined for the Idle state. In general, however, the output values are computed by functions, and may be important to the programmer as he or she decides what the next state should be.

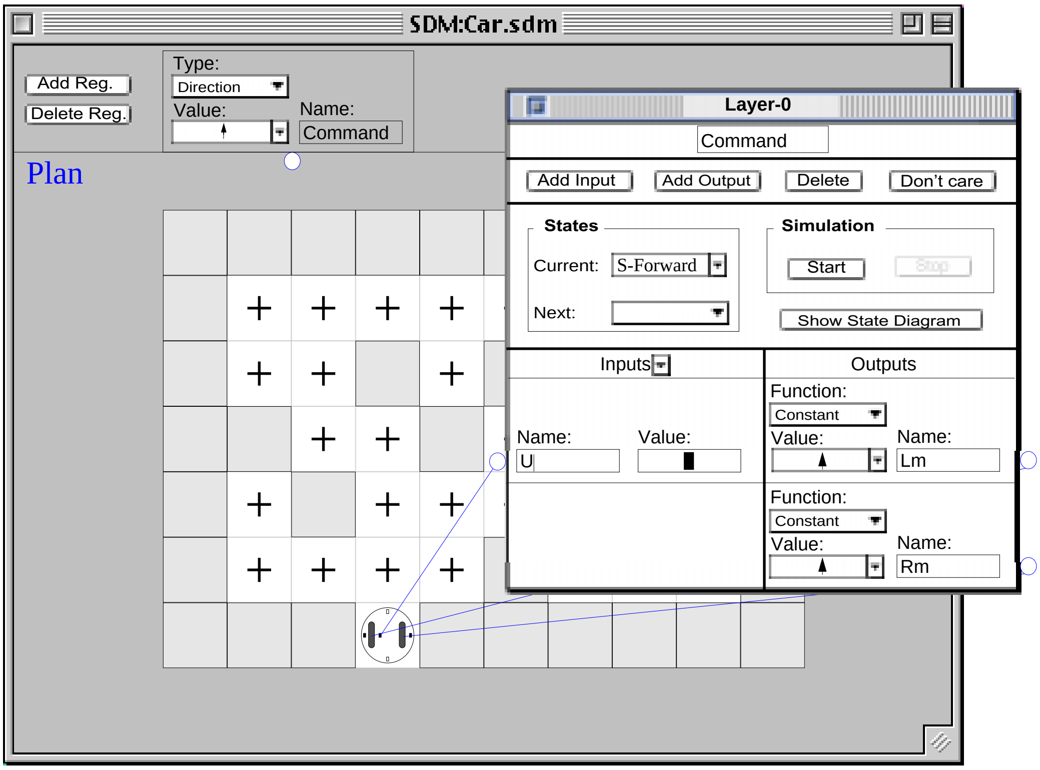
Since we now want the car to move forward, we need a new state which will drive both motors in the forward direction, so we choose to define a new state we choose the New state option from the Next pop-up. This creates a new state, transits the simulation to this new state and enables the Resume button.

The default name for a new state is State-n where n is the number of previously defined states for the current behaviour. However, we can rename them at any time. in our example, we rename the state S-Forward.

We want the S-Forward state to generate outputs for the Lm and Rm outputs to move the car forward until the UIS changes from ▮ to ▯ indicating that the car has moved off the centre of the square. We set both the Lm and Rm outputs to ↑ and define a new input for the state, name it U, and connect it to the UIS sensor of the car as previously described.

As soon as the connection between the U input and the UIS sensor is established, the value of the sensor appears in the Value box of the input U. When the Command behaviour receives ↑, it must not accept any other command until it finishes executing ↑; that is, the input C is insignificant for the S-Forward state. To indicate this, we select the input C by clicking on it and then click the Don’t Care button. This removes the input C pane from the inputs panel of the Layer-0 floating palette and adds it to the Inputs pop-up. This will define the input C as insignificant for the S-Forward state and from now on whenever this state is entered the input C will automatically be removed from the inputs panel. Figure 5-9 shows the Layer-0 palette after defining the S-Forward state and before resuming the simulation.

Note that before we resume the simulation, we can undo the performed actions one step at a time by selecting the Undo option of the Edit menu or undo the whole process by selecting the Undo all. This will take us back to the point where the simulation stopped for the last time.



Type: Direction
Value: ↑
Name: Command

Command

Current: S-Forward
Next:

Name: U
Value: ▮

Function: Constant
Value: ▲
Name: Lm

Function: Constant
Value: ▲
Name: Rm

Figure 5-10. setting an input to don't care.

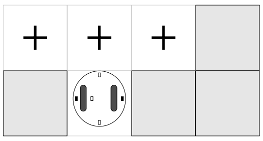
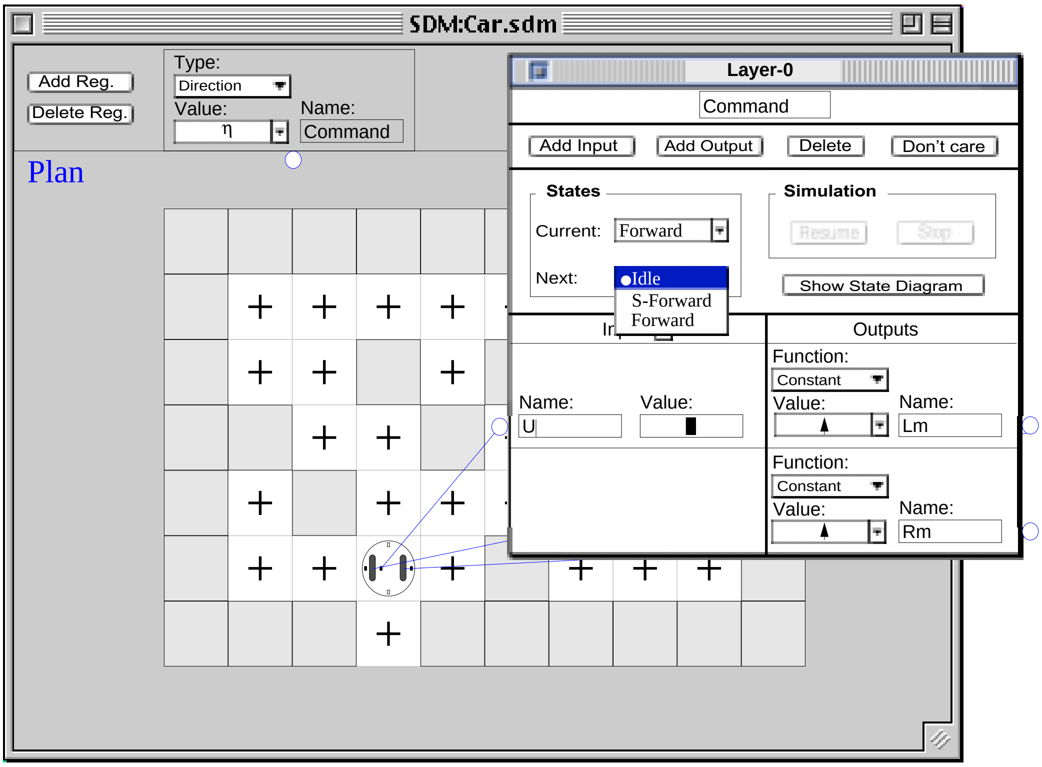


Figure 5-11. S-Forward state

When we resume the simulation by clicking the Resume button, this button is disabled, the Stop button is enabled and the car starts moving forward. As soon as the UIS changes from ▮ to ▯, as figure 5.11 shows, the simulation stops and the Stop button is disabled. Since the task of moving the car to the next grid is not complete, we create a new state, name it Forward, and set both the Lm and Rm output to ▲ .

When we resume the simulation the car moves forward until the UIS changes from ▯ to ▮ where the simulation stops. At this point the task of moving the car to the next grid is complete and the simulation should transit back to the Idle state. In the Idle state the value of the Command register will be monitored, so before we set the Idle state as the next state we set the Command register to η to ensure that the simulation will stay in the Idle state after we set it as the next state. To set the next state, we choose Idle from the Next pop-up indicating that when simulating is resumed, a transition back to the Idle state will be created and followed. Since the output values for the Idle state have already been defined, both Lm and Rm outputs are set to η by the system. Figure 5-12 illustrates the workspace window during the definition of this transition from the Forward state to the Idle state.



Type: Direction
Value: η
Name: Command

Command

Current: Forward
Next: ●Idle
S-Forward
Forward

Name: U
Value:

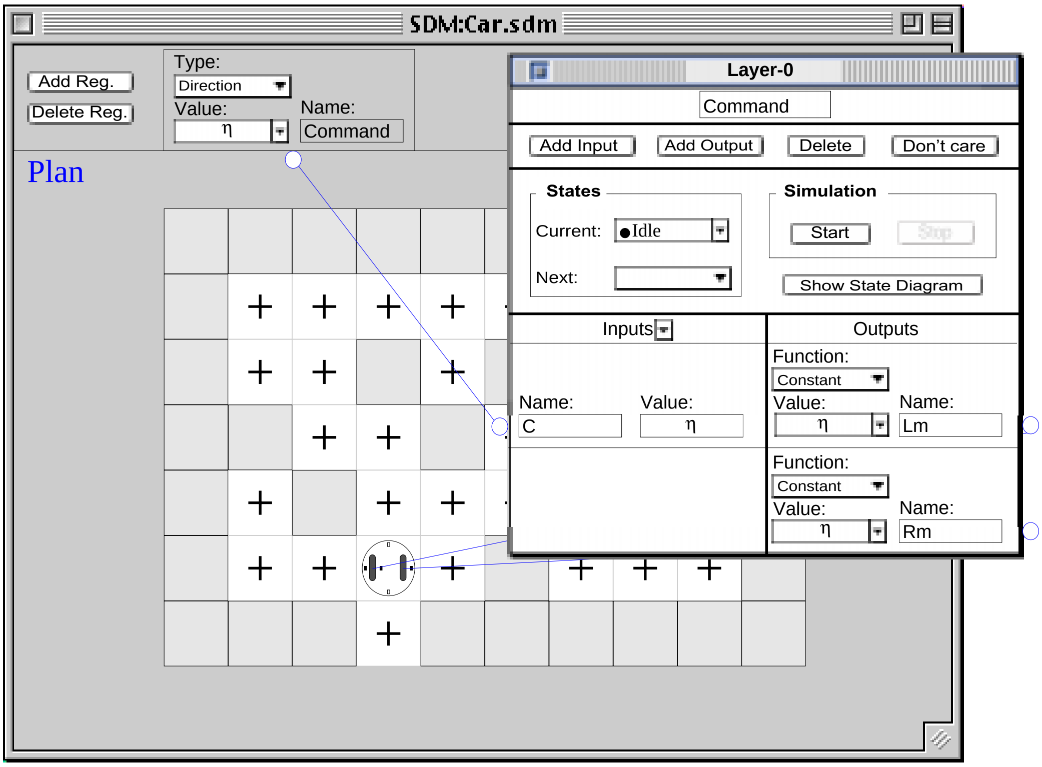
Function: Constant
Value:
Name: Lm

Function: Constant
Value:
Name: Rm

Figure 5-12. A transition back to the initial state.

After we choose Idle in the Next pop-up the Idle state is entered, the U input is removed from the inputs panel and added to the Inputs pop-up and the input C is restored from the Inputs pop-up to the inputs panel. Figure 5-13 shows the workspace after the Idle state is entered.

Notice that if we had not changed the value of the Command register, the S-Forward state would be immediately entered and the car would move to the next grid.



SDM:Car.sdm

Type: Direction

Value: η

Name: Command

Layer-0

Command

States

Current: ●Idle

Next:

Inputs

Name: C

Value: η

Outputs

Function: Constant

Value: η

Name: Lm

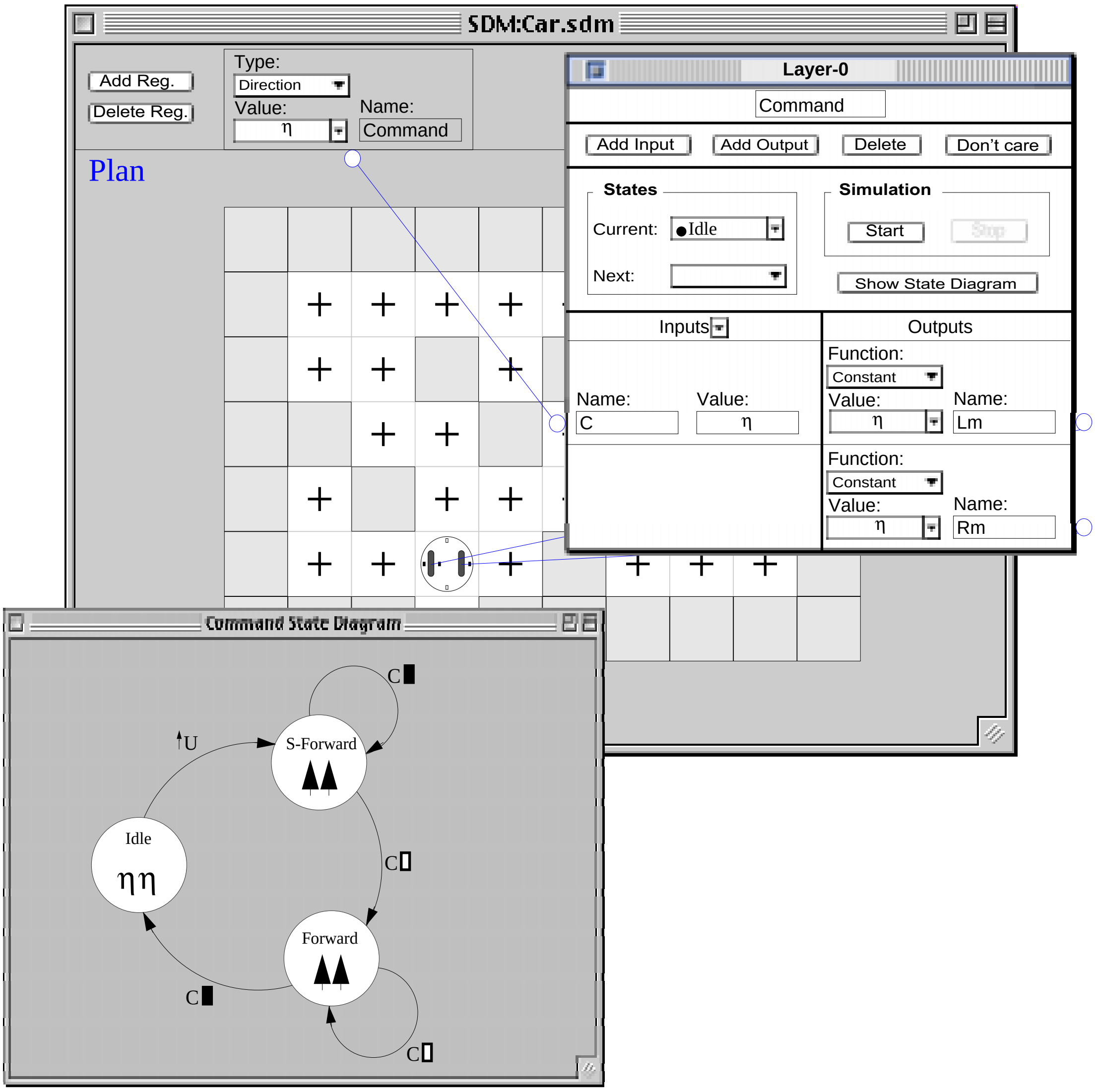
Function: Constant

Value: η

Name: Rm

Figure 5-13. Restoring inputs from the Input pop-up.

When we click the Resume button the simulation starts and stays in the Idle state generating η for both motors, causing the car to stay in the same position. At this moment we can change the value of the Command register to start the definition of a new state of the Layer-0 behaviour. At any time we can also view the state diagram of the behaviour generated so far by clicking the Show State Diagram button. When we click this button a window containing the partial state diagram generated is opened. Figure 5-14 illustrates the state diagram after the Idle, S-Forward, and Forward states have been defined.



Type: Direction

Value: η

Name: Command

Current: ●Idle

Next:

Inputs — Name: C, Value: η

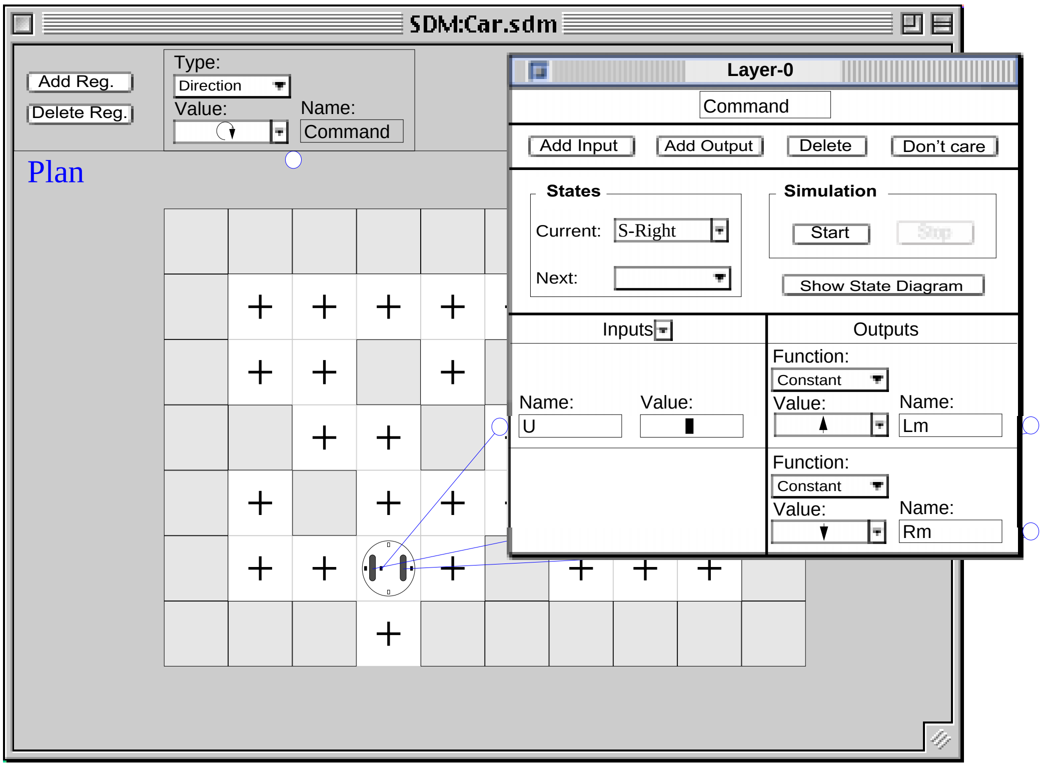
Outputs — Function: Constant, Value: η, Name: Lm

Function: Constant, Value: η, Name: Rm

Figure 5-14. Partial State Diagram.

We can define the remaining states of the **Command** behaviour in the same way. For example, when we change the Command register to ↻, the simulation stops. We define a new state and name it S-Right. Similar to the move forward command, when the execution of the turn right command starts there is no need to monitor the value of the input C. So we set the input C to don’t care and restore the input U by selecting it from the Inputs pop-up.We also set the Lm and Rm outputs to ↑ and ↓ , respectively. Figure 5-15 illustrates the simulation environment after the S-Right state is created and before the simulation is resumed.

Figure 5-15. Defining the Right state of the Command behaviour.



Type: Direction
Value:
Name: Command

Layer-0

Command

States
Current: S-Right
Next:

Simulation

Inputs
Name: U
Value: ▮

Outputs
Function: Constant
Value: ↑
Name: Lm

Function: Constant
Value: ↓
Name: Rm

When we click the Resume button the simulation is resumed and the car starts rotating to the right. As soon as the UIS changes from ▮ to ▯, as figure 5-16 shows, the simulation stops. We define a new state and name it Right. In the new state we leave Rm and Lm outputs unchanged and resume the simulation. The car turns to the right until the UIS changes from ▯ to ▮. At this point we select Idle from the Next pop-up to enter the Idle state.

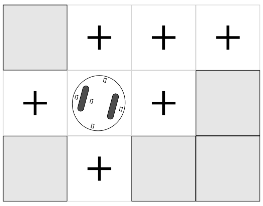


Figure 5-16. Turn right command.

We generate the states Reverse and Left in a similar way. After all the required states are defined the definition of the layer-0 behaviour is complete. At this time when the simulation is resumed, changing the value of the Command register to ↑ and ↓ will make the car to move one cell forward or backward, respectively. Setting the Command register to ↻ will make the car turn 90° to the right and setting it to ↺ will make it turn 90° to the left.

The layer-0 behaviour implements the level-0 control system. To add a new behaviour to level-0 to obtain level-1, we select the New Behaviour item from the Behaviour menu. This shrinks the Layer-0 palette into a small box labeled Level-0 with input and output terminals corresponding to the input and out put terminals of the Layer-0 palette. All the connections are redrawn as they were established in layer-0. An empty Layer-1 floating palette similar to the Layer-0 palette is also opened. We name the new behaviour Forward. Figure 5-14 illustrates the workspace at this point.

By shrinking the previous level into a box connected by wires to the robot, we encourage the programmer to view the box as a small computer or controlleradded to the basic robot, the two components together making up a more complex and competent robot.

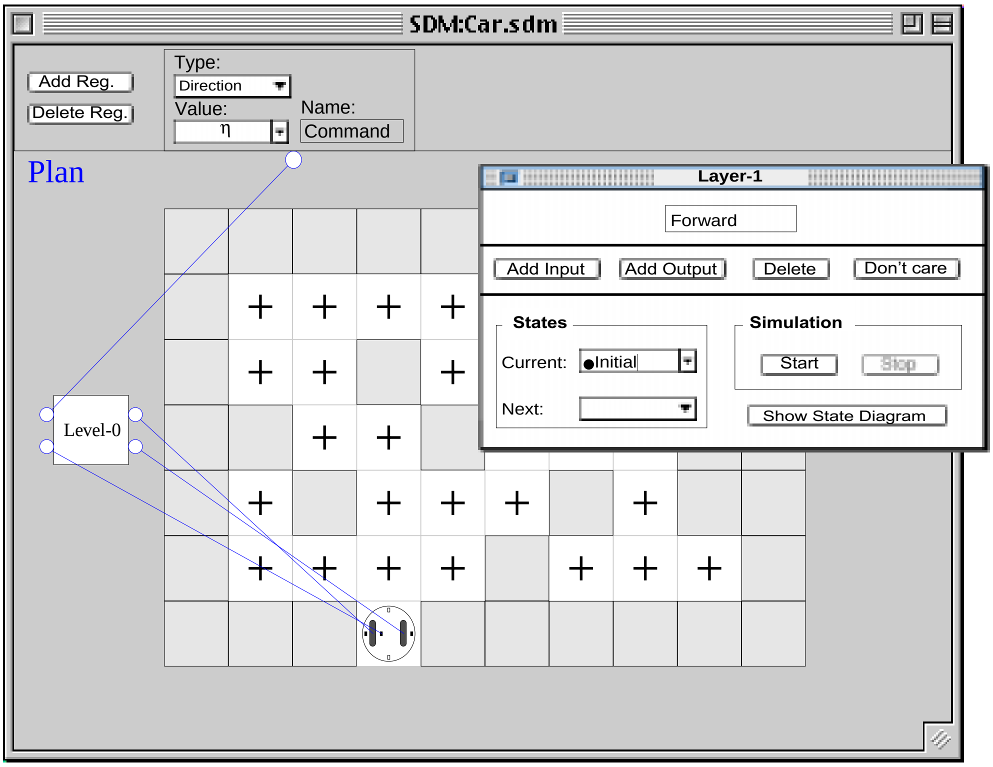


Figure 5-17. Creating the Forward behaviour

The first step in programming the Forward behaviour is to define the initial state. We start by renaming this state Straight. The Forward behaviour is intended to generate appropriate commands for the C input of the Command behaviour in response to changes in the values of the Front and Right Infrared Sensors. It also must monitor the outputs of the Command behaviour to decide when to generate a new command. That is, the Forward behaviour will not generate a new command until it makes sure that the Command behaviour has finished executing the previous command. To accomplish this we need to define one output for the Straight state and four inputs. We will use the output of the new behaviour to subsume the values generated by the Command register. Two of the inputs correspond to the Lm and Rm outputs of the

Command behaviour, the other two corresponds to the Front Infrared Sensor, and the Left Infrared Sensor.

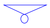
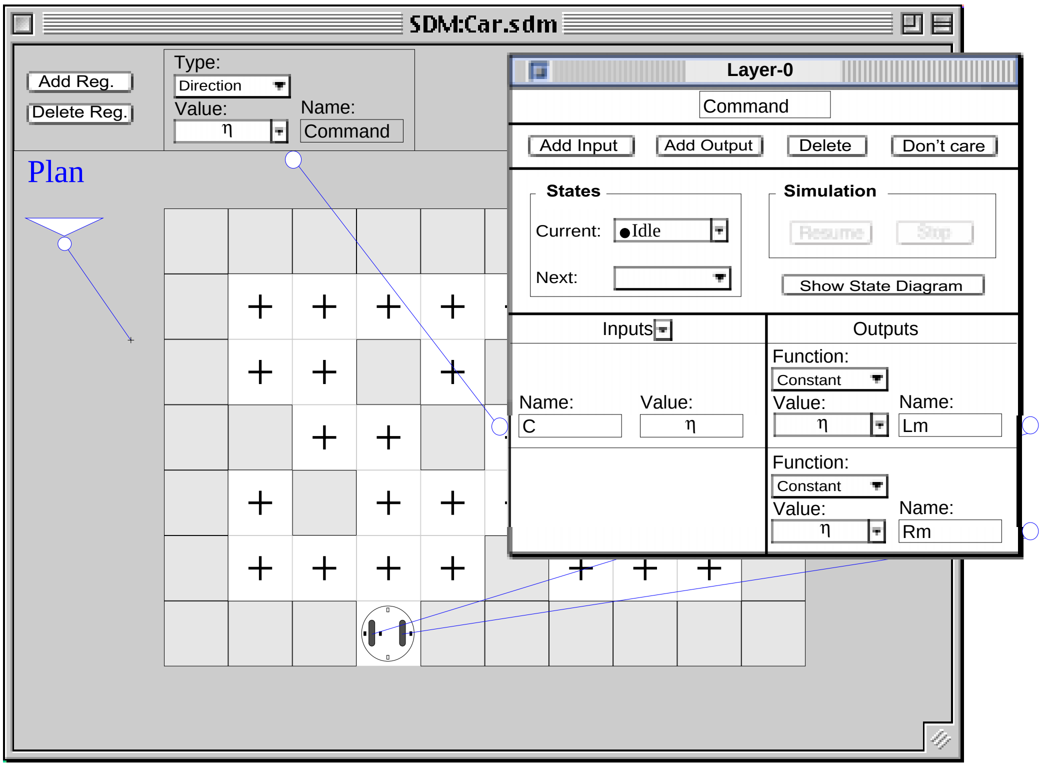
We create an output for the Straight state and name it CF. The CF output of the Forward behaviour will generate values for the C input of the Command behaviour, so we need to connect them. Connecting the output CF to the input C is accomplished by clicking on the CF terminal and dragging, when the cursor passes over the Level-0 box, the Level-0 box is expanded into the Layer-0 floating palette and the Layer-1 palette disappears. The source of the current rubber band changes to  indicating that the source of the wire is in a higher layer. Figure 5-18 illustrates the workspace at this point.

Figure 5-18. Expanding a level to make a connection.



Type: Direction
Value: η
Name: Command

Command

Current: ●Idle
Next:

Name: C
Value: η

Function: Constant
Value: η
Name: Lm

Function: Constant
Value: η
Name: Rm

When we move the cursor over the C input terminal, the terminal is highlighted indicating a valid connection. We release the mouse button to complete the connection, at which point the

Layer-0 floating palette shrinks back into the Level-0 box, all the connecting wires are redrawn and the Layer-1 palette reappears. Since the C input was already connected to the Command register a conflict is detected and to resolve it a small *junction node* appears.

Each junction node can be set to be a *suppressor* or *inhibitor*. However, by default the node is an inhibitor. A junction node has three terminals as shown in figure 5-19; An empty triangle terminal for the normal input, a solid triangle terminal for the controlling input and a circle terminal for the output. These terminals are drawn optimally on the boundary of the node; that is at the point closest to the other end of the attached wire.

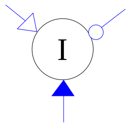


Figure 5-19. An inhibitor.

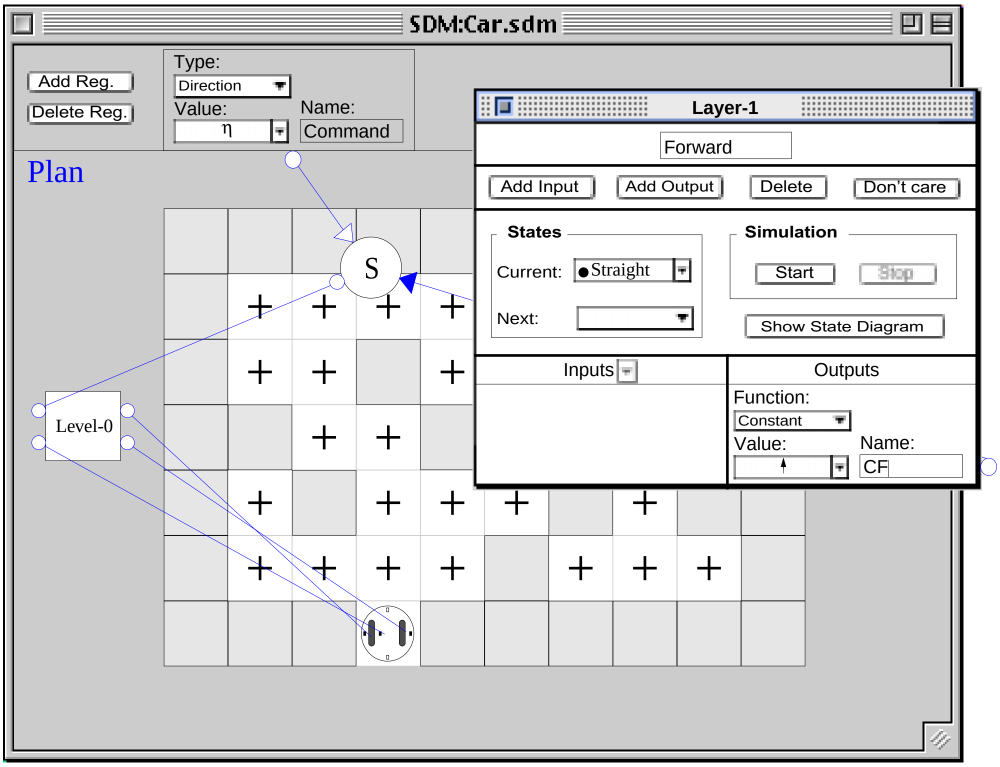
A junction can be toggled between suppressor and inhibitor by clicking the node with the option key pressed (option-click).

Since the Forward behaviour needs to subsume the value of the Command register in order to control layer-0, we toggle the junction node to a suppressor.

Normally, when one value is used to subsume another a check for type compatibility is performed. In our example, since CF had no type before the connection was made, it is assigned the type of the destination, the input C of layer-0.

In the Straight state we want to generate ↑ indicating that the car must move forward. To accomplish this, we leave the Function pop-up unchanged and set the output to ↑ by selecting it from the Value pop-up.

Figure 5-20 illustrates the workspace after the output CF of the Layer-1 is connected to the input C of the Layer-0 and the output of the state is set to ↑.



Type: Direction

Value: η

Name: Command

Forward

Current: ●Straight

Next:

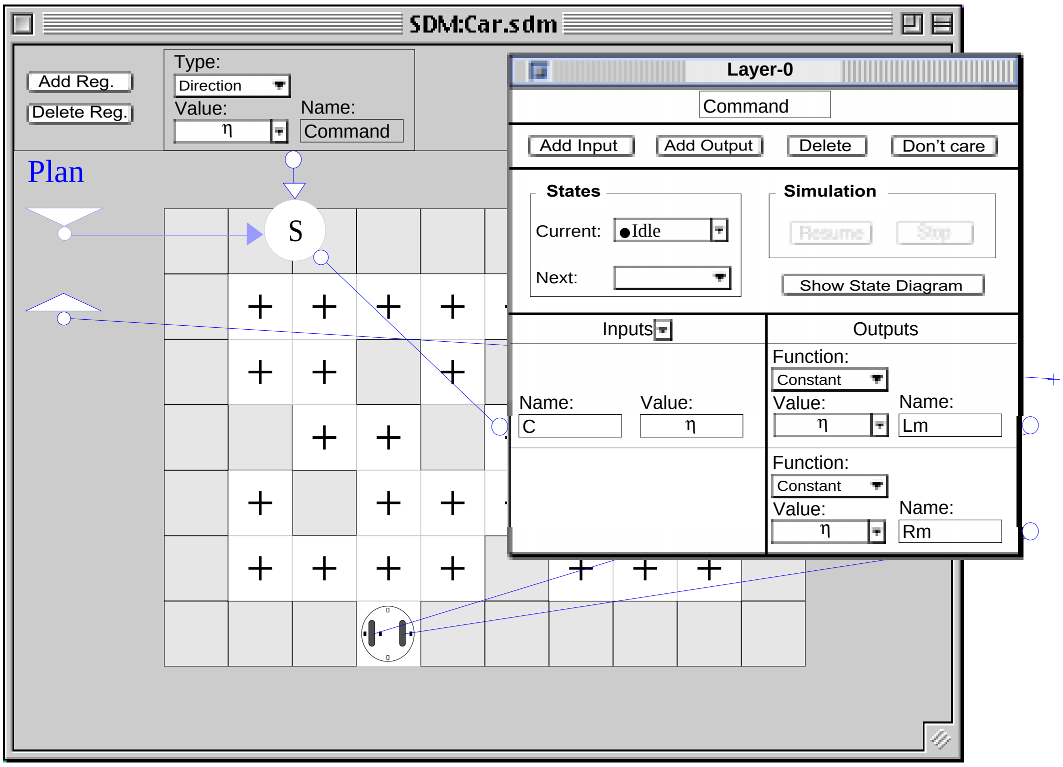
Function: Constant

Value:

Name: CF

Figure 5-20. A suppressor node.

We add two inputs to layer-1, name them RSF and FSF, and connect them to the Right and Left Infrared Sensors of the car, respectively. We define another input, name it LMF and connect it to the output Lm of layer-0 in the same way that we connected the output CF to the input C of layer-0. The only difference is that when layer-0 is expanded, the rubber band is connected to to indicate that the destination of this connection will be an input to a higher layer. Figure 5-21 illustrates the workspace just before we complete the connection to the output Lm.



Type: Direction
Value: η
Name: Command

Command

Current: ●Idle
Next:

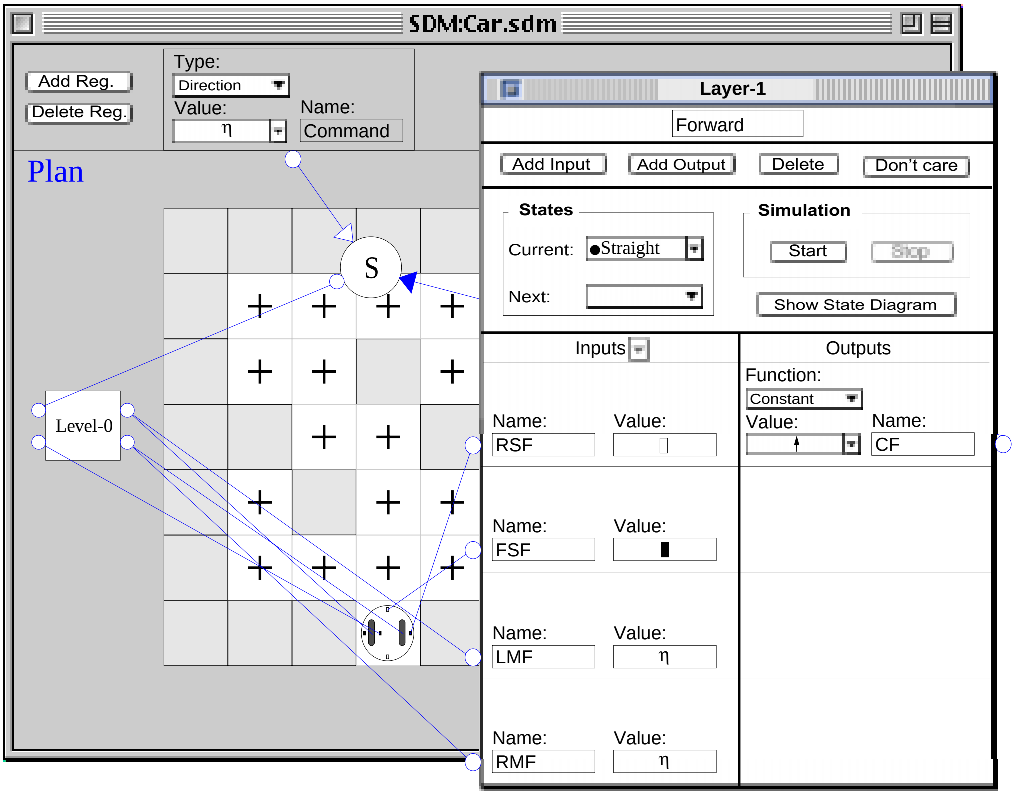
Name: C
Value: η

Function: Constant
Value: η
Name: Lm

Function: Constant
Value: η
Name: Rm

Figure 5-21. Using an output of the Command behaviour as an input to the Forward behaviour.

We add a forth input to layer-1, name it RMF and connect it to the output Rm of layer-1 in the same way. Figure 5-22 shows the workspace window after both LMF and RMF are defined and connected to Lm and Rm, respectively.



Type: Direction
Value: η
Name: Command

Forward

Current: ●Straight
Next:

Name:	Value:
RSF	▯
FSF	▮
LMF	η
RMF	η

Function: Constant
Value: ↑
Name: CF

Figure 5-22. Forward behaviour

Now the definition of the Straight state is complete and we start the simulation. ↑ output is generated subsuming the value of the Command register. The Forward state from the Command behaviour is entered, generating ⬆ for both Lm and Rm. The inputs LMF and RMF of the Forward behaviour are consequently changed causing the simulation to halt. Since the move to the next grid is not complete, we select Straight from the Next pop-up to be the next state, and resume the simulation.

The car moves forward until the Right Infrared Sensor changes from █ to ▯ as in figure 5-23. At this point since the move to the next grid is still not complete, we again select Straight from the Next pop-up and resume the simulation. The car moves forward until it is in the middle of the next grid at which point the Command behaviour finishes executing the ↑ command. The Command behaviour enters the Idle state and both Lm and Rm outputs of the Command behaviour and consequently LMF and RMF are set to η . This causes the simulation to stop.

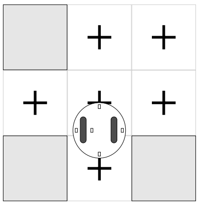
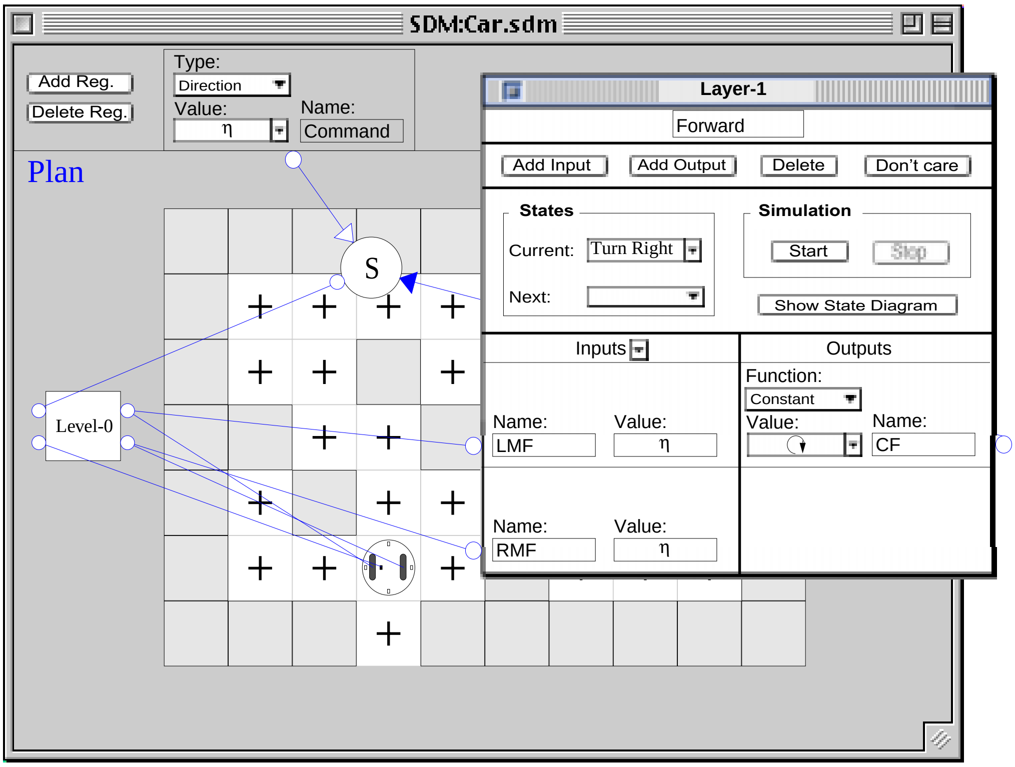


Figure 5-23. Passing an obstacle.

Since there is an opening to the right of the car, we need to create a new state in which the car turns to the right. When there is an opening to the right of the car, it must turn to the right no matter if there is an obstacle in front of the car or not; that is, the value of the Front Infrared Sensor is insignificant when deciding whether to turn to the right. So, we select FSF and click the Don’t Care button. Now we define a new state and rename it Turn Right. The Turn Right state must generate the output ↻. To accomplish this, we select ↻ from the CF Value pop-up. When the decision for turning to the right is made, the Right Infrared Sensor of the car is insignificant until the turn is complete; that is, the Right infrared sensor is insignificant while in state Turn Right. So, we set RSF to Don’t care before we resume the simulation. This means that in the Turn Right state only LMF and RMF are the significant inputs. Figure 5-24 shows the state of the environment after the Turn Right state is created and before the simulation is resumed



SDM:Car.sdm

Type: Direction

Value: η

Name: Command

Layer-1

Forward

States

Current: Turn Right

Next:

Inputs

Name: LMF Value: η

Name: RMF Value: η

Outputs

Function: Constant

Value: ↻

Name: CF

Figure 5-24. Creating the Turn Right state.

When we resume the simulation, the output value of the Command register is subsumed. The Right state from the **Command** behaviour is entered, generating ↑ and ↓ for the Lm and Rm outputs, respectively. The LMF and RMF inputs of the **Forward** behaviour are consequently changed causing the simulation to halt.

Since the turn to the right is not complete, we define Turn Right to be the next state and resume the simulation. The car starts turning to the right and continues until the Right state of the **Command** behaviour finishes executing the ↻ command. At this point the Idle state of the **Command** behaviour is entered causing the Lm and Rm outputs of the **Command** behaviour to both change to η and consequently both LMF and RMF become η causing the simulation to halt. Now since the turn to the right is complete we set the next state to Straight.

When we resume the simulation, the Straight state is entered, both RSF and FSF are restored from the Inputs pop-up, and the car moves forward until it reaches the middle of the next grid at which point the Forward state of the Command behaviour finishes executing the ↑ command. The state Idle of the Command behaviour is entered, generating η for both Lm and Rm outputs. As a result both LMF and RMF are changed to η and because of the position of the car in the environment FSF and RSF both generate █ causing the simulation to halt.Figure 5-25 shows this situation.

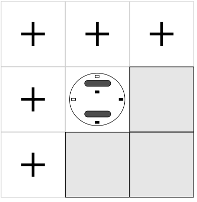


Figure 5-25. Start turn left

Now since there are obstacles in the front and to the right of the car we create a new state that generates a left turn command, name it Turn Left, set the output to ↶, and resume the simulation. The car starts turning to the left and as soon as the values of LMF and RMF are changed the simulation stops. Figure 5-26 shows this situation. We select Turn Left from the the Next pop-up and resume the simulation.

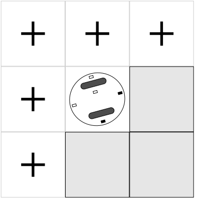


Figure 5-26. Turning to the left

The car turns to the left until the left turn is complete. When the Left state of the Command behaviour finishes executing the ↶ command it enters the Idle state and both LMF and RMF are set to η causing simulation to stop. Since the turn to the left is complete and there is no obstacle in front of the car, as figure 5-27 shows, we select Straight from the Next pop-up and resume the simulation.

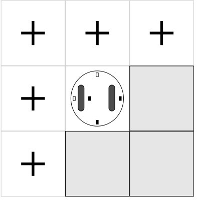


Figure 5-27. Completing left turn.

The Straight state is entered and the car moves to the next grid. When the car is in the middle of the next grid since there is an opening to the right, the Turn Right state is entered and the car starts turning to the right.

The simulation continues until the car faces an obstacle after finishing a turn to the left; that is, the car is in a dead end as in figure 5-28.

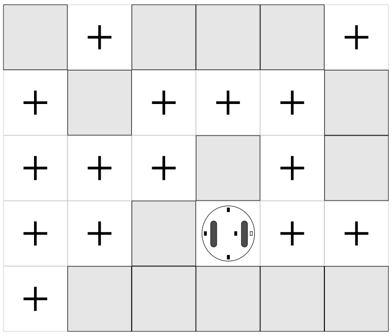


Figure 5-28. A dead end.

At this point, since the car faces an obstacle it must take another left turn to keep the wall (obstacle) to its right. To accomplish this, we select Turn Left from the pop-up as the next state and resume the simulation. The car starts turning to the left and when the left turn is complete since there is an opening in front of the car, the Straight state is entered and the simulation continues uninterrupted and the car traverses the maze.

We can view the state diagram generated for the Forward behaviour so far by clicking the Show State Diagram button. Figure 5-29 illustrates the state diagram window for the Forward behavior at this point.

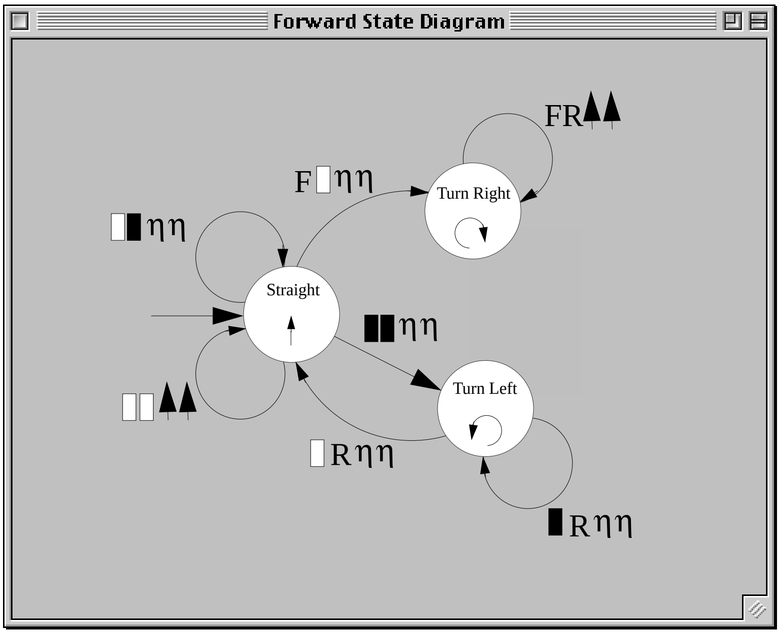
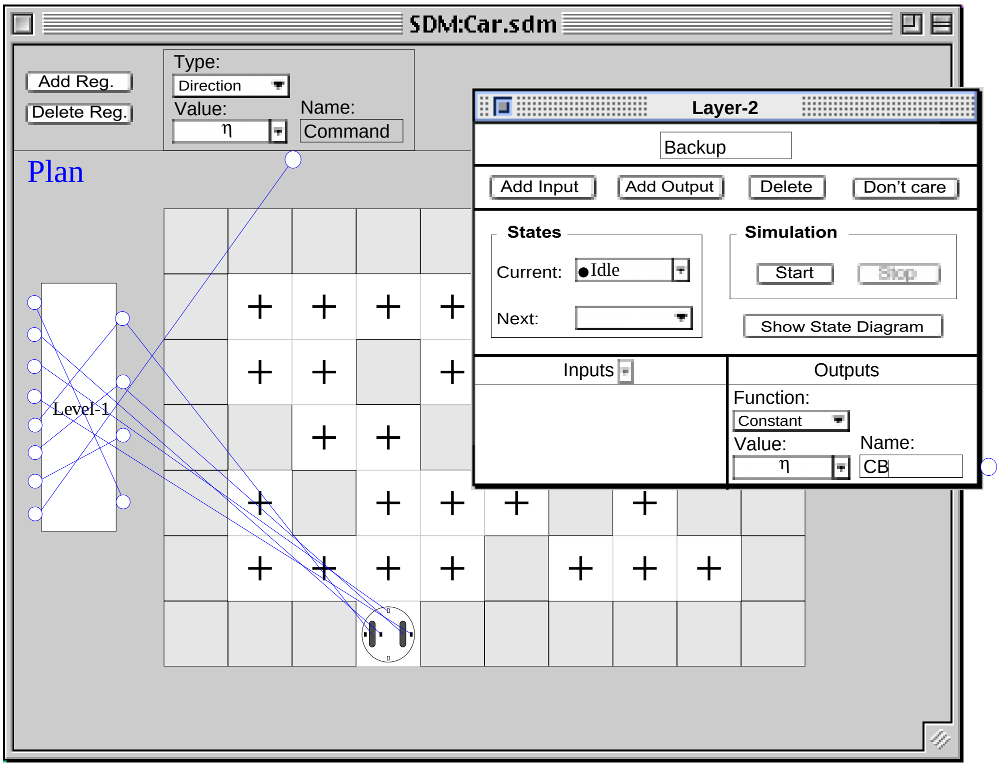


Figure 5-29. Forward State Diagram.

Notice that not all the possible transitions out of the states have been defined completely, for example there is no transition in this state diagram from the Straight state to itself when the inputs are ▮▮▲▲ . When the system wraps the Forward behaviour along with the Level-0 box to create the Level-1 box as a result of adding a new Backup behaviour (explained shortly), all the possible undefined transitions will be interpreted as transitions to the same state.

Now we place the car in the start cell and add a new behaviour to the control system to implement the Backup behaviour as described in section 3.3.8. When we select the New Behaviour from the Behaviour menu the Layer-1 floating palette and the Level-0 box are merged together in a box labeled Level-1 and an empty Layer-2 floating palette appears in the workspace. We name the new behaviour Backup.

We name the initial state of the backup behaviour Idle and add an output named CB. Figure 5-30 shows the state of the environment after the CB output is defined.



SDM:Car.sdm

Type: Direction
Value: η
Name: Command

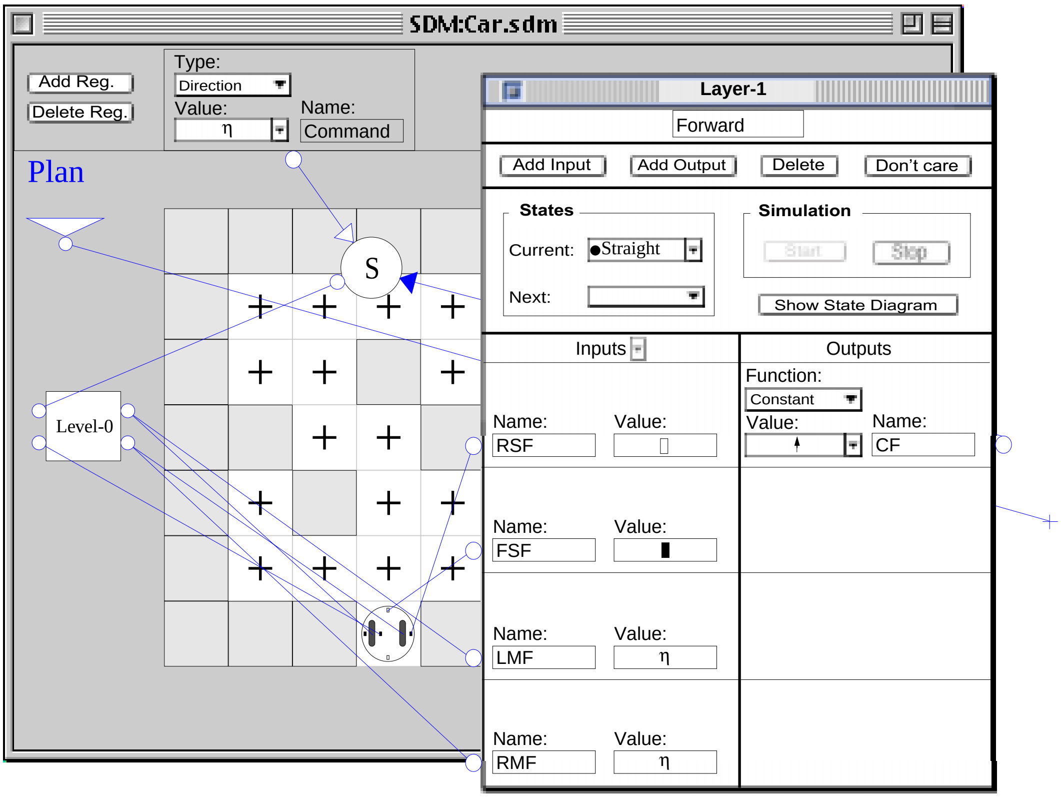
Layer-2
Backup

States
Current: ● Idle
Next:

Outputs
Function: Constant
Value: η
Name: CB

Figure 5-30. Creating Backup behaviour.

The Backup behaviour is to monitor the values of the horizontal sensors and generate a backup command when it detects a dead end. This will relieve the car from taking unnecessary left turns on dead ends. The Backup behaviour must stay in its initial state generating η as long as the car is not in a dead end cell. This implies that the output of the Idle state must be η. The Backup behaviour generates values that have the same effect as the output of the Forward behaviour, so we connect the CB output of layer-2 to the CF output of layer-1. This connection is established in the same way that we connected the CF output of layer-1 to the C input of layer-0.Figure 5-31 shows the state of the environment before we make the connection to CF.



Type: Direction
Value: η
Name: Command

Forward

Current: ●Straight
Next:

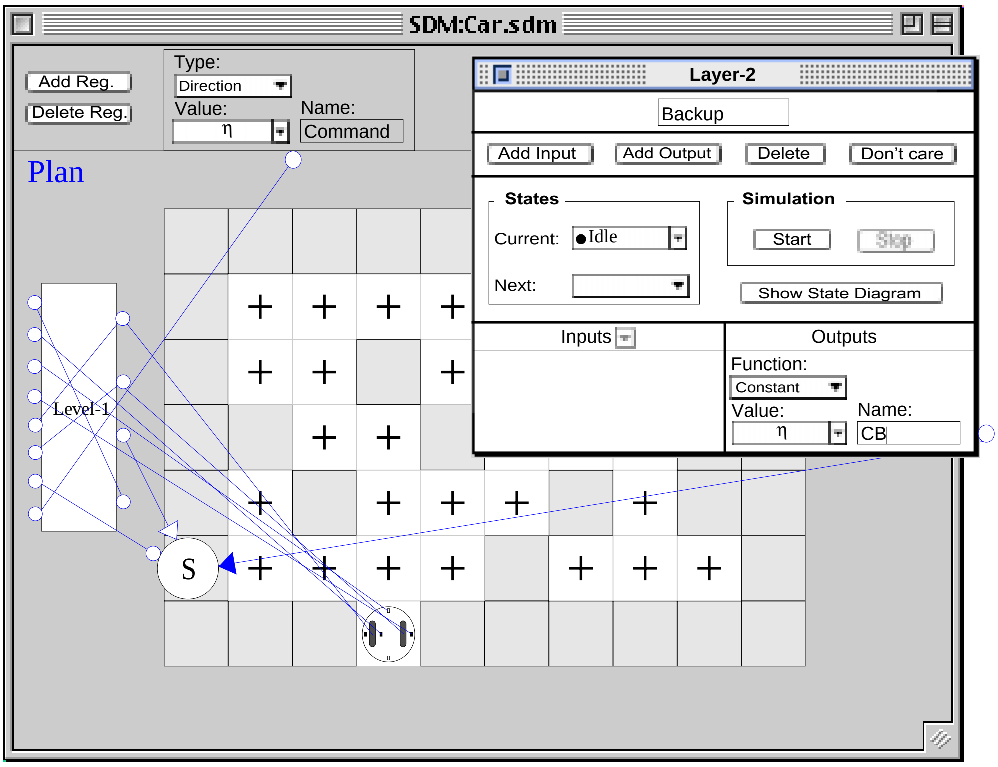
Function: Constant
Value:
Name: CF

Name: RSF Value:
Name: FSF Value: ▮
Name: LMF Value: η
Name: RMF Value: η

Figure 5-31. Expanding Level-1 to Layer-1 and Level-0.

When we make the connection to the CF output, the Layer-1 floating palette and the Level-0 box are shrunk back into the Level-1 box and the Layer-2 palette reappears along with an inhibition node. Since the Backup behaviour generates values that replaces the output of the Forward behaviour, we toggle the inhibitor node to a suppressor as previously described. This assigns the type of CB to CF and the inputs and outputs of the Level-1 box are rerouted accordingly. Figure 5-32 illustrates the workspace window at this point.

.



Type: Direction
Value: η
Name: Command

Layer-2: Backup

Current: ● Idle
Next:

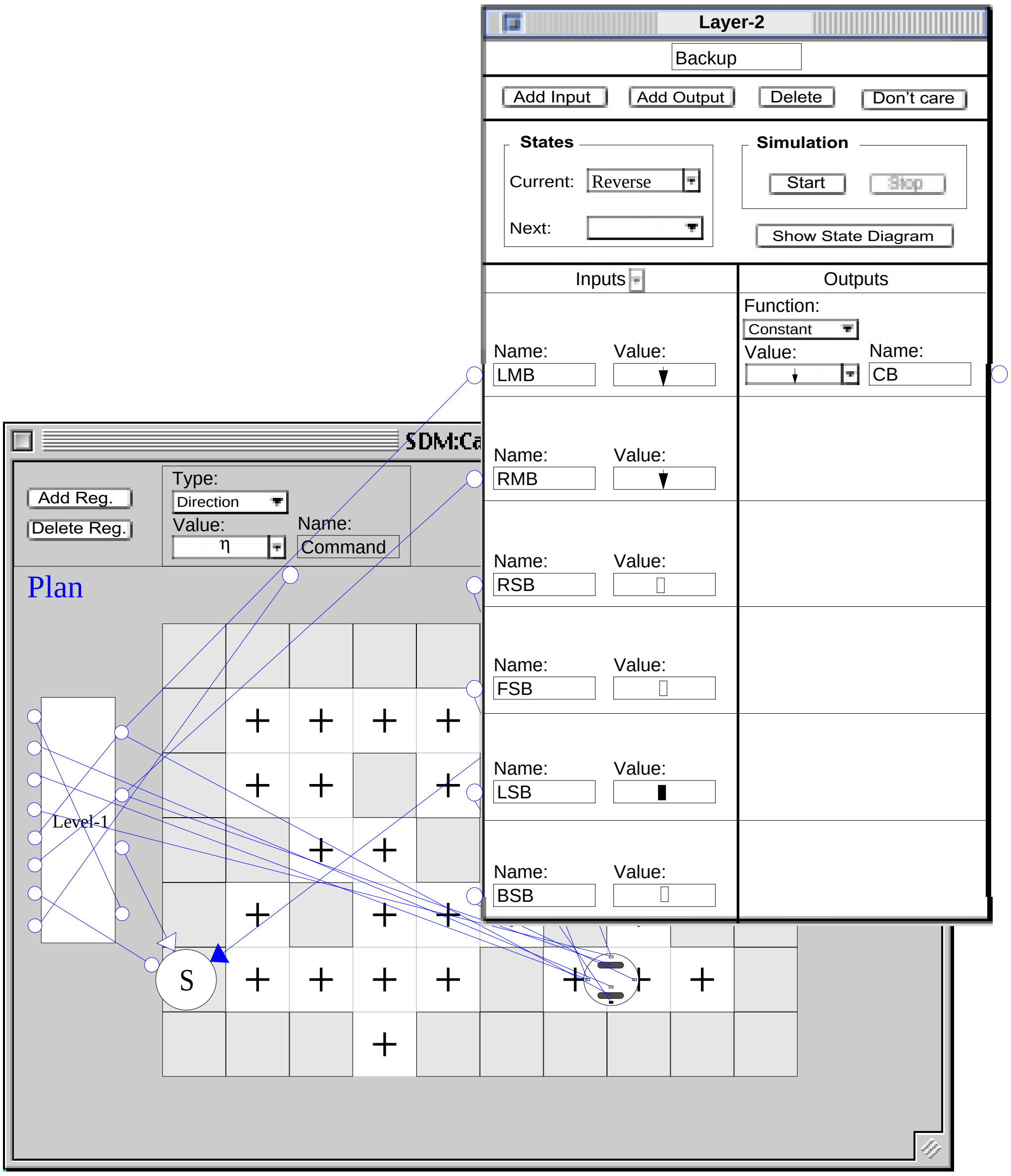
Function: Constant
Value: η
Name: CB

Figure 5-32. Rerouting of the inputs and outputs of Level-1 box.

The Idle state of the Backup behaviour monitors the values of the Left, Front, Right and Back Infrared Sensors in addition to the Lm and Rm outputs of the layer-0. we define FSB, RSB, LSB, and BSB inputs and connect them to FIS, RIS, LIS, and BIS sensors of the car, respectively. We also define RMB and LMB inputs and connect them to Lm and Rm outputs of layer-0.

We resume the simulation and whenever it is stopped because of a change in the horizontal sensors or the Lm and Rm outputs, we select Idle from the Next pop-up as the next state and resume the simulation until it stops when the car is in a dead end cell. At this point we define a new state, name it Reverse, set the output to ↓ and resume the simulation

The car starts reversing and as soon as the value of one of the horizontal sensors is changed (for example the front sensor is changed from ▮ to ▯) the simulation stops. Since execution of the reverse command is not complete, we select Reverse from the Next pop-up and resume the simulation. Figure 5-33 shows the state of the environment before the simulation is resumed.



Backup

Current: Reverse

Next:

Function: Constant

Value: ↓ Name: CB

Name: LMB Value: ▼

Name: RMB Value: ▼

Name: RSB Value: ▯

Name: FSB Value: ▯

Name: LSB Value: ▮

Name: BSB Value: ▯

Type: Direction

Value: η Name: Command

Figure 5-33. Reverse state.

When we resume the simulation, the car reverses to the previous grid and as it comes to the middle of the grid the simulation stops. At this point since there is an opening behind the car and an obstacle to the left, it must take a right turn. We create a new state, name it Turn Right, set the output to ↻, and resume the simulation. When the turn to the right is complete the simulation stops because of the changes in the input values.

We continue defining the remaining states of the Backup behaviour in the same way.

6 Evaluation of SDM

In this chapter we will use Green and Petre's Cognitive Dimensions, as described briefly in Chapter 2, as the criteria for evaluating our proposed SDM. Since the system is not yet implemented, there can be no additional practical assessment via experiment at this point.

6.1 Abstraction Gradient

Abstraction gradient is the availability and scope of abstraction mechanism and there are two parts to it; the amount of concepts that a programmer needs to master in order to get started; and the amount of new abstraction that he or she has to build to do anything significant. In the case of SDM, the initial level of abstraction is low. Programming concepts are few, i.e. direct representation of domain entities plus notions such as behaviour, state, inputs, and outputs. A programmer can create simple behaviours fairly directly with no new abstractions. To do anything more significant, requires building a sequence of abstractions (levels), each dependent on the last. This suggests that we can classify SDM as "abstraction-hungry". However, the usual criticism of abstraction-hungry systems does not seem to apply since each abstraction is built on the last, and because the process is dynamic, the programmer gets continues feedback on the success of failure of his or her program. In general, SDM scores well on the abstraction gradient criterion.

6.2 Closeness of Mapping

We believe that our proposed visual language scores well in this dimension. The closeness of mapping is achieved through two different aspects in this work; mapping the subjects (robot and objects) into the language and mapping the control system into the language. The first mapping is achieved by using models generated in HDM as programming constructs in SDM. The second mapping is accomplished by reformulating the subsumption architecture to provide a nested control system suitable for our visual language. Also, because of the interactive nature of the process, the semantics of EMMs is "brought to life", i.e. the whole process is a concrete demonstration of EMM semantics

6.3 Consistency

Consistency is achieved through simplicity. Once a programmer has created one behaviour there are no more concepts that must be learnt in order to create complex control programs. Similar concepts are expressed in similar syntactic forms which enables a programmer to predict the way that system works. For example, once a programmer has created a connection from an output of a behaviour to an actuator, he or she can easily expect the same mechanism be employed for connecting a sensor to an input of a behaviour.

6.4 Diffuseness

Diffuseness is the number of required "symbols" or "entities" to express a "single concept". In order to evaluate our visual language in this dimension we need to specify what an "entity" and a "single concept" are. For example, using a single wire to express the concept of connection between an actuator and an output of a behaviour can be interpreted as a low diffuseness. There are two kinds of symbols in SDM, those corresponding to objects and those representing concepts from the control model. Given that the former directly corresponds to the domain, the structures they are used to build probably strike the right balance between diffuseness and terseness. Representation of the control model using elements of the second type is probably more diffuse, but it's hard to say whether it's too far one way or the other. The direct view of the EMM is pretty appropriate though.

6.5 Error-Proneness

Mistyping is one of the major sources of errors in programs. In our language since the name of the inputs, outputs, registers, states and behaviours are logically insignificant, such typos will not have logical effects on programs. On the other hand, since the connections in the program are made through pointing and clicking, it can be a source of "slips" specially where the items are very close to each other and not very visible. This can be avoided by changing the resolution of the screen and zooming in and out when user wishes to create such connections. In general without an implementation it's pretty difficult to assess the error-proneness of this notation

6.6 Hard Mental Operations

The rating of a visual language in this dimension depends on the problem domain as well as the syntax and semantics of the language itself. Although our intention has been to provide a simple programming environment to facilitate the process of programming a robot, there is no need to mention that the problem of programming autonomous robots is a complex activity. Evaluating our language in this dimension requires empirical study and a comparison with other languages.

One part of the language that might give rise to hard mental operations is the construction of subsumption functions. The logic behind the connections in a network of inhibitors and suppressors could get quite complex. Nevertheless, since subsumption architecture provides a mechanism to solve a complex problem by breaking it into smaller and much simpler tasks, we expect our proposed visual language to perform fairly well in this dimension.

6.7 Hidden Dependencies

The dependencies between components at each level are depicted by using a concrete mechanism, i.e. using wires. The dependencies between components at different levels are also depicted by using wires but are not as obvious as the former ones, since the component of one end of such a wire will be buried, possibly several levels deep. The dependencies between outputs and inputs of a behaviour are not visible explicitly, however, programmers can consult the state diagrams at any time to understand the mapping from inputs to outputs.

6.8 Premature Commitment

Premature commitment occurs when the programmer must make a decision in advance of having the necessary information to make it. We have tried to relieve programmers from guessing ahead by postponing the definition of new concepts to the time that they are actually needed. A good example of this is defining registers or inputs for the behaviours after the definition of the behaviour has already been started.

6.9 Progressive Evaluation

This is the ability to execute the program partially before all of it is put together. In this case the program is being created as it is being run. Once the level-0 control program is completed it is treated as a complete control program and can easily be added to, in order to create more complex programs, so at each level the programming process relies on the execution of the level below. SDM therefore scores well in this dimension.

6.10 Role-expressiveness

Role expressiveness is the degree to which if the purpose and role of each component can easily be inferred. Obviously the role of those parts that map directly to the physical domain is clear. We have introduced a small number of components for programming purposes. For example, representing a completed control level as a box with input and output terminals together with the feedback wires suggests their function as controllers of some kind. This is further reinforced by the process which creates them. In general, concrete symbols are reasonably role-expressive; wires are pretty good too since they indicate the passage of some data; the box is also role-expressive. other more abstract things are less expressive, although the labels such as “name”, and “type” do give some hints about the functions of some of the smaller pieces. On balance, though, role-expressiveness is fairly good in comparison with other languages.

6.11 Secondary Notation

Secondary notation refers to the use of extra information for conveying extra meaning outside the formal syntax. The best example of secondary notations that can effectively help programmers to understand programs in our visual language is the use of names for registers, inputs, outputs, states and behaviours. We feel that for complex programs the use of intuitive names for these components will be critical. Another example of secondary notation in this work is the use of two different shades of blue for active and inactive wires and junction nodes. When a lower level control box is expanded all the wires coming from upper layers are drawn in a paler shade of blue. Like any other programming language, this one could benefit from some commenting mechanism.

7 Summary & Future Research Directions

In this work we have proposed a visual language and environment for programming autonomous robots. This system allows a programmer to create a complex control system by implementing a sequence of simple behaviours. As a basis for this language we also presented a precise reformulation of the subsumption architecture for robot control due to Brooks. In our model of the subsumption architecture a number of levels of competence are defined each consisting of a behaviour implemented as an Extended Moore machine, where a higher level of competence implies a higher degree of autonomy. Each level of competence is made up of a series of control layers. A new control layer can simply be added to the existing level to obtain the next level of overall competence. Once a control system is designed, it can be used as a black box, however, a programmer can simply view the internal structure of the control program at different levels to decide how to connect the new behaviour to the existing behaviours to obtain a higher level control system. We described the requirements for a design environment, similar to those used in Computer Aided Design tools, in which a programmer creates models for a robot and environmental objects. We also proposed a user-interface for the programming environment in which a programmer creates and modifies the control program by direct manipulation of models previously generated in HDM for a robot and the objects in the environment.

7.1 Future Work

This work presented the required ground work for implementing a visual language for robot control and leads to many new topics for further exploration which we will briefly summarize.

7.1.1 Implementation for Empirical Studies

One important aspect of this work that remains outstanding is to implement both modules of our proposed visual programming system. We have assumed that programs created in a visual environment by direct manipulation of programming constructs representing the robot and domain objects would be superior to textual languages for robot control because it increases “closeness of mapping” which should enhance the programming process, according to Green and Petre. This assumption can not be effectively be verified without empirical evidence.

7.1.2 Refinement of Hardware Definition Module

In this work we did not discuss HDM in detail. Our work mostly relied on the HDM proposed in [16]. However, in order to employ the proposed HDM we need to make necessary modifications to provide a design environment that supports three dimensions and as a consequence there will be changes in the way sensors and actuators are modeled in this design environment. There are also other issues that need to be resolved in the design of HDM, such as the definition of types, specification of the function of sensors, and the effects of actuators.

7.1.2.1 Sensors and Actuators

There are several different kinds of robots and although we have divided them into two major categories, each can be subcategorized further. The most important factor for classifying robots is the kinds of sensors and actuators they are equipped with. As a result, to provide a design environment in which a programmer can create models for a variety of robots we need a through investigation of different kinds of sensors and actuators presently used in robots. This will enable us to implement a design environment that is appropriately general for modeling components of a wide variety of robots.

7.1.3 Other Control Systems

We chose a reformulation of subsumption architecture as the basis for our control model because of its nested, easy-to-understand and incremental nature. The nested nature of this control system allows programmers to focus on one behaviour of the robot at a time rather than interacting with the whole control system at once. The incremental nature of this model enables programmers to build, execute, edit and refine programs in small steps.

There are many other control models for programming robots, such as hierarchical[3] and logic-based control systems [5]. A possible extension to this work could be investigating the feasibility of these control models as a basis for other visual languages for robot control. It might also be worthwhile to attempt to combine different control models in one visual programming system.

7.1.4 Planning and Problem Solving

Defining an autonomous robot's interactions with its surrounding environment is the basis of controlling the robot. There are many other problems that may arise during programming an autonomous robot, such as planning and problem solving. There is no doubt that a visual language cannot effectively be employed for the task of programming a robot until it addresses these issues. In our work, we suggested that for these high level tasks a programmer will have access to the underlying programming language; however, this means that the nature of the underlying language will bias the overall effectiveness of the system for the purpose of programming a robot. An interesting extension to this work would be an investigation on the feasibility of visual languages for solving such high level tasks especially logic-based languages which also serve as the basis for many planning and problem-solving systems.

Bibliography

[1] A. Cypher, and D. Smith (1995). KidSim: End User Programming of Simulations. In: *Proceedings of CHI'95*, Denver CO, 27-34.

[2] A. Repenning (1995). Bending the Rules: Steps Toward Semantically Enriched Graphical Rewrite Rules. In: *Proceedings of 1995 IEEE Symposium on Visual Languages*, 226-234.

[3] Albus J. S., R. Quintero, and R. Lumia (1994). Overview of NASREM: The NASA/NBS Standard Reference Model for Telerobot Control System Architecture.

[4] Burnett M. M. & Djang R. W. (1998). Introduction to Visual Programming Languages: Scaling-Up Issues. *Tutorial 1998 IEEE Symposium on Visual Languages.*

[5] David Poole (1995). Logic Programming for Robot Control". In: *Proceedings of 14th International Joint Conference on AI (IJCAI-9*5).

[6] Eric Jackson & David Eddy (1997). Design and Implementation Methodology for Autonomous Robots Control Systems. International Submarine Engineering Ltd.

[7] Green T. R. G. & Petre M. (1996). Usibility Analysis of Visual Programming Environments: A Cognitive Dimension's Framework. In: *Journal of Visual Languages and Computing*, v7, pp.131-174.

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

[9] J. J. Pfeiffer, Jr. . A Rule-Based Language for Small Mobile Robots. In: *Proceedings of the 1997 IEEE Symposium on Visual Languages*, pp. 162-163.

[10] John E. Hopcroft and Jeffrey D. Ullman (1979). Introduction to Automata Theory, Languages, and Computation. Addison Wesley, Boston.

[11] J. Gindling, A. Ionnidou, J. Loh, O. Lokkebo, A. Repenning (1995). LEGOSheets: A Rule-Based Programming, Simulation and Manipulation Environment for the LEGO Programmable Brick. In: *Proceedings of the 11th IEEE Symposium on Visual Languages*. 1995. pp. 172-179.

[12] Margaret M. Burnett and Marla J. Baker (1994). A Classification System For Visual Programming languages. *Technical Report* 93-60-14.

[13] Margaret M. Burnett, Adele Goldberg, Ted Lewis (1994). Visual Object -Oriented Programming. Printice-Hall/Manning.

[14] Microsoft, Visual C++

[15] Philip T. Cox, Christopher C. Risley and Trever J. Smedley (1998). Toward Concrete Representation in Visual Languages for Robot Control. In: *Journal of Visual Languages and Computing* (1998) 9, 211-239.

[16] Philip T. Cox, Trever J. Smedley (1998). Visual Programming for Robot Control. In: *Proceeding of the 1998 IEEE Symposium on Visual Languages*, pp. 217-224.

[17] Proghraph International (1993). Prograph CPX User Manuals.

[18] Rajeev K. Pandey, Margaret M. Burnett (1993). Is It Easier to Write Matrix Manipulation Programs Visually or Textually? An Empirical Study. In: *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pp. 344-351.

[19] Risley C. C. & Smedley T. J. (1998). Visualization of Compile Time Errors in a Java Compatible Visual language. In: *Proceedings of the 1998 IEEE Symposium on Visual Languages*, pp. 22-29.

[20] Rodney A. Brooks (1986). A Robost Layered Control System For a Mobile Robot. In: *IEEE Journal of Robotics and Automation*, RA-2, 14-23.

[21] Schmuker, K. J. (1996). Rapid Prototyping Using Visual Programming Tools. In: *Common Ground CHI 96 Tutorial Notes.*

[22] Shu N.C. (1988). *Visual Programming*, Van Nostrand Reinhold Company Inc., New York.

[23] Whitley K. N. . Visual Programming Languages and the Empirical Evidence For and Against. In: *Journal of Visual Languages and Computing*, v8, pp. 109-142.

[24] Xichi Zheng. Layered Control of A Practical AUV. International Submarine Engineering Ltd.