2000

Concurrency Issues in Programmable Brick Languages

Gilliad E. Munden

CONCURRENCY ISSUES IN PROGRAMMABLE BRICK LANGUAGES

By

Gilliad E. Munden

B.A. University of Maine, 1998

A THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

(in Computer Science)

The Graduate School

The University of Maine

August, 2000

Advisory Committee:

Laurence Latour, Associate Professor of Computer Science, Advisor

Seymour Papert, Distinguished Visiting Professor of Computer Science

Thomas Wheeler, Assistant Professor of Computer Science

George Markowsky, Professor and Chair of Computer Science Department

CONCURRENCY ISSUES IN PROGRAMMABLE BRICK LANGUAGES

By Gilliad E. Munden

Thesis Advisor: Dr. Laurence Latour

An Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Master of Science
(in Computer Science)
August, 2000

The programmable brick constitutes a domain for simple, autonomous robotics geared towards novice programmers in a constructionist setting. A large number of languages have been adapted from other domains to serve as a programming platform for this brick. However, there has yet to be an in-depth analysis of how these languages fit *this* domain. This work provides such an analysis of the existing brick languages in order to identify how they deal with the issue of concurrency as it relates to the brick. First, the brick domain is characterized and the languages involved are described. Second, the different approaches to concurrency are analyzed and a new approach is introduced (mode-based programming) that was specifically designed taking into account the features of concurrency being analyzed.

ACKNOWLEDGEMENTS

The supporting research work behind this thesis was a group effort. As such, there are several people to whom this work is indebted. This paper also represents only the first step in a continuing research project. Much of the work done this year was laying the ground for future work.

First, thanks are in order to Dr. Seymour Papert and the people of the Learning Barn for funding as well as assistance in the design and implementation of this research.

Second, thanks to the research team that participated in the robotics language research project in the Software Engineering Lab at the University of Maine Computer Science Department. Much of the language analysis was based on not only surveys of exiting brick robots, but also the implementation of many new robots. Erik Albert and Nathaniel Bates helped significantly in learning of the various brick languages and building a body of case studies to compare them. The other important factor of this research was gaining direct interaction with young programmers. It is important to gain an intuitive feel for how novices reason, and that can only be gained through experience. Katherine Comeau and Kevin Pelkey have been of great assistance in designing and executing both after school "Robotics Clubs" as well as summer camp sessions. These provided an opportunity to work directly with the children and programming.

Third, thanks to the Housing Authority for assistance in making time and space for these clubs and the America Counts program for partial funding of the club. In addition, thanks to Bonnie Blagojevic who did a great deal of "behind the scenes" work to make the clubs happen.

Finally, thanks to the committee members for a great deal of assistance in helping develop the concepts that went into this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter Descriptions

Chapter 1: Introduction

The goals of this thesis are outlined as well as what contributions are made. The domain of the programmable brick is characterized and the concurrency features, to be used in the later language analysis, are defined.

Chapter 2: Previous Work

The structure of a language is the result of conceptual development, real-world constraints and other related work. This chapter describes the evolution of the hardware and software of the LEGO programmable brick. The languages that exist for the programmable brick originate from a variety of sources; from academic research, to commercial development, to hobbyist exploration. The history of these languages, as well as the syntax, is presented in order to give a context for understanding of why these languages work the way they do.

Chapter 3: Concurrency Analysis

There has been relatively little work to date comparing the different languages that exist for this brick. At this point there does not even exist a clearly defined set of features that are desirable in a brick language. This chapter focuses on features of concurrency that relate to programming the brick. A set of concurrency features is defined, and the different approaches to concurrency are analyzed in terms of these features.

Chapter 4: Mode-Based Programming

An alternative approach to brick programming is introduced, mode-based programming. In this language, the user defines a program in terms of modes of operation. This chapter describes the concept of modes as well as their relationship to existing programming languages in the AI/robotics research field. A couple in-depth examples of mode-based programming are presented along with an analysis of how it compares to existing methodologies.

Chapter 5: Conclusions & Future Directions

This chapter first summarizes the findings of the paper in the context of the languages and concepts presented. The rest of this chapter serves as a guideline to several interesting topics of future exploration. These topics include future directions for mode-based languages to possible changes in the hardware of the programmable brick.

Chapter 1: Introduction

This thesis is focused on evaluating the treatment of concurrency within the domain of the programmable brick. In order to do so, the following steps are taken:

1. Characterize the domain of the brick in terms of adjacent domains (robotics, programming languages and constructionism) as well as the history influencing its development.

2. Identify a key concept (concurrency) and define a series of features that characterize how well it is implemented.

3. Group the existing languages into major categories based on their approach to concurrency and perform an evaluation on these features to identify benefits and shortcomings.

4. Synthesize an adaptation of the existing methods that capitalizes on the benefits and addresses the shortcomings (mode-based programming).

The major contributions of this thesis are the following: the identification and categorization of existing brick languages, a thorough discussion of concurrency issues related to the brick and the introduction of the mode-based approach to programming. Mode-based programming is introduced as one possible language approach that could be derived by applying the concurrency features. Ideally, the concurrency features described here will serve as a metric for analyzing other new languages in this domain as well.

1.1 Domain of the Brick

The programmable brick is a hardware device meant for novices to use in building simple, autonomous robotics. The device was developed as one of several “toys to think with” - a learning tool for children to used in *constructionist* classrooms [28][33][34]. Constructionism

is the philosophy that people learn best by building artifacts that have some personal value to them [28]. With this model, the learner develops their own knowledge through building these artifacts (such as robots). The brick physically consists of a processor embedded in a plastic brick with an infrared (IR) port, for communication, and ports for several sensors and actuators. The brick was designed to work with standard LEGO™ parts to create the physical structure of the robot; hence the name *programmable brick*. [17]

Most of the languages examined in this thesis were developed for either the MIT brick or the LEGO® RCX™. However, the discussions here are not limited to these implementations. The concept of the programmable brick can be implemented in many ways. The general view of the brick has the following components:

1. An embedded microcontroller with a firmware that supports concurrency
2. Ports for taking input from analog sensors
3. Ports for controlling analog actuators
4. Inter-brick communication capabilities

This is the most general description of the brick, and the language issues discussed here are pertinent to any brick implementation fitting these parameters. This description of the brick is purposely being kept general so that this work can fit to a wider range of hardware than just the MIT brick and the RCX.

The second significant characteristic for the domain of the brick is the user body. In this case, it is the body of novice programmers. It is being assumed, for the purposes of this study, that young, novice programmers do not reason in an inherently different manner than adult novice programmers. As a novice language, it should have relatively low barriers to entry. Necessary robotics concepts (such as concurrency and control) must be available, but in a manner that leverages the intuitive knowledge of the novice. It is also important to remember that this is a language for learning. Therefore, a language that only allows the user to create absolutely trivial programs is not useful. The language should be flexible enough to

permit the programmer to make relatively complex robot behavior. It should be a tool that the user can grow with. These two goals (low barrier of entry and high ceiling) are often directly in conflict with each other.

The third major factor for characterizing the domain of the brick is the performance necessities of the robots created with the brick. Optimization is not a serious constraint. First, programs for the brick tend to not be incredibly complex. Second, the robots generally do not have to fit serious real-time constraints. If a process is only able to page a sensor every 10^{th} of a second, instead of every 100^{th}, it will not have a serious affect on most projects. Keep in mind that these are generally small, plastic robots using inexpensive analog motors and sensors, which introduce their own inaccuracy. In fact, the benefits of having an optimized language would most likely be mitigated by the physical components used for the robot.

This thesis is focused on the programming language concerns for the domain of the brick. There have not only been many physical instantiations of the programmable brick, but there have also been many programming languages developed for this device [12]. These languages have been, for the most part, adaptations of existing languages. These existing languages come from other domains varying from animation to process control and provide a range of different approaches to thinking about brick algorithms. To establish a context for the analysis, chapter 2 will describe the history behind the programmable brick languages.

1.2 Concurrency Analysis

What has not been done in this domain is a thorough analysis of the existing tools, from a programming language perspective. Very little work has been done evaluating whether these languages are even suitable for the brick. This paper provides such an evaluation of these languages. Note that this study is focusing on programming languages as opposed to environments. There are many aspects of the programming process that are affected by

features of the development environment. This paper purposely ignores those features and concentrates on language features as independent of specific development environments.

Of the different programming language issues, that of concurrency is identified as being of significant importance. Concurrency is a necessary feature in robotics in general due to the need to react to a constantly changing real world environment. What makes expressing concurrency challenging with this tool is that the brick is intended as a learning tool. Children and other novices encountering programming for the first time will need to be able to deal with concurrency in order to make effective robots.

1.2.1 Relevance of Concurrency in Domain

In order to be robust, even simple robotic algorithms need to perform multiple, concurrent actions. This often involves performing internal processing while monitoring external events. This type of check and act concurrency [32] can become quite cumbersome when using explicit time slicing. Because of this, it is much more beneficial to have a language that inherently supports some type of multi-tasking.

To illustrate the type of problems that can happen with explicit time-slicing, take the following example. This example involves a robot car that drives until it hits a wall, then stops. With a purely sequential language the code would look something like the following (using a Logo-like syntax):

```
to run
  ab, on      ; turn on motors A and B
  loop [
      if switch1 [ ab, off ]      ; If button pressed,
  ]                               ; motors off.
end
```

Now add to this robot a light that flashes continuously while the car is driving forwards. A sequential program to do this action using time-slicing would look something like this:

```
to run
  ab, on
  loop [
      if switch1 [ ab, off ]
      flash_light
      ]
  ]
end
```

The problem that occurs is that if the switch is activated while the light is flashing, the robot will not recognize that activation until after the light has finished the *flash_light* routine (which may last a long time).

This is not a good option for a couple of reasons. First, time slicing is not an intuitive way to think about handling concurrent tasks. People walk and chew gum concurrently. They do not take a step, bite down, take another step, bite again, etc. Second, time slicing is inefficient in a high-level language [32]. Optimizing such tasks should be relegated to lower-level architecture.

For these reasons, most programming languages for the brick have some form of built-in concurrency. The tools for accessing this underlying concurrency provide a layer of abstraction over the sequential processor of the brick. The purpose of this abstraction is to allow users to access the power of multi-tasking in a manner that is intuitive, flexible and appropriate to the problems encountered in the brick domain. The focus of chapter 3 will be examining the existing abstractions.

1.2.2 Concurrency Features

In order to make concurrency accessible to the novice programming community several different abstractions were developed for the existing brick languages. The different approaches to concurrency are identified and described. A common set of *concurrency*

features is used to compare these different abstractions. The purpose of this is to determine the tradeoffs of control and understanding between these different approaches.

This thesis proposes a simple set of features that describe how concurrency is presented in a programming language. These features are not presented as the only means of description, but as a starting point for future discussion. These six features are: articulation, process creation, syntax, conflict resolution, visibility and naturalness.

1.2.2.1 Articulation

The articulation of a language relates the amount of control that users have over processes. All of the languages evaluated here allow users to define operations that are executed concurrently. Articulation can be described in terms of the operations on concurrent processes made available: start, stop, restart, define and redefine.

It is possible, in any of the options discussed in this thesis, for the programmer to use flags to implement more articulate control. This is not as powerful as built in commands, however, because flags are dependent on how often they are checked. In addition, it makes the programmer's work more complicated.

1.2.2.2 Process Creation

There are two types of process creation for brick languages; implicit and explicit. Explicit process creation is where the user makes a direct call to start a process. Implicit process creation is where a new process is created as a side effect of a control structure or function call. The user is not made explicitly aware of the implementation of concurrency. Note, implicit instantiation does not necessarily mean that the articulation of the language is low.

1.2.2.3 Syntax

Where these concurrency tools fit into the syntax of the language can have a strong affect on the understanding associated with them. Concurrent processes can be represented as separate

code blocks, control structures or even data types. In choosing to represent a process in such a manner, the user will associate characteristics of other tools with related syntax.

Because of this, language elements that have very different semantic meaning should have a distinct syntactic appearance. It should be clear to the reader of a language's source code what constructs are related to concurrency and what are related to sequential operations. An evaluation of syntax involves identifying possible points of confusion between concurrent and sequential constructs. Assessment for this is qualitative.

An important idea in syntax is being called *axis of control*. This is based on the concept that that there are different distinct types of control that the programmer can define over the flow of a program. The three types of control identified for this thesis are sequence, control-flow and concurrency. Written code in any language should clearly express these different types of control in a distinct manner, and the syntax of the language can have a significant affect on how clear this distinction is.

1.2.2.4 Conflict Resolution

Conflict resolution can be an important issue even in simple robotics. Even for languages that do not support global variables, the actuators of the robot represent a global resource where it is very easy to generate inter-process conflicts. Unfortunately only one of the languages presented here, LegoSheets, currently addresses this problem.

1.2.2.5 Visibility

Visibility characterizes how easily a user can determine the active processes at a given point in the program's code. Due to unforeseen interactions between concurrent processes, debugging these languages can often be difficult. In order to understand the interactions between processes, it is necessary to be able to see what processes are active. A language that better supports visibility allows the user to more easily determine these interactions.

Visibility is considered “high” if a human reader can determine the processes that would be active at a given point in the source code without stepping through the program. Visibility is considered “low” if a reader must step through the code from the beginning of the program in order to determine what processes may be active at a given point in the source.

1.2.2.6 Naturalness

The idea of concurrency is not one that is unique to computer science. In the physical world events constantly occur in parallel. People who have never been exposed to concurrent processes have already acquired models for dealing with parallelism. *Naturalness* characterizes how well the language constructs for concurrency fit the novice’s intuitive model for parallelism.

Natural Programming involves the study of natural language in order to distill a formal programming language [25][26][27]. This study has identified certain statistical patterns in natural language descriptions to programming problems. This paper will use these studies as well as intuitive reasoning about the naturalness of programming concepts.

1.3 To the Reader

The programmable brick was designed as a tool for learning. Languages like Logo and Smalltalk began a trend of looking at programming as a medium for teaching. Because of this, language findings here relate to more than just those in the computer science field. However, this particular paper is written as a computer science thesis. In order to convey programming language concepts, terminology and analogies from this field are used. An effort was made to define terms as they were used, but it may be necessary to skim over more technical sections.

This document also refers to terms and concepts that lie outside of the field of computer science. As a learning tool, the brick can be described in terms of the fields of

psychology and education. Some reference to these fields is made, but only at a very superficial level. Assessment from these perspectives is considered outside of the range of this thesis. Discussions concerning subjects like naturalness and visibility of a language are made from the perspective of computer science understanding, not epistemological.

Chapter 2: Previous Work

The purpose of this chapter is to establish a context for the language analysis in the rest of the paper. The programmable brick was developed as a tool to be used in constructionist learning. The premises of constructionism and their relation to the brick are discussed here. In addition, the culture of programmers that has risen around the brick is briefly discussed in order to understand the myriad of languages that have been developed for brick programming.

Figure 2.1 illustrates a family tree of brick languages studied during the course of this thesis. This set of relationships can be somewhat complex, so it is recommended that the reader use this figure to orient themselves during this chapter.

2.1 Constructionism

The programmable brick was designed to be a toy for children to use in building their own inventions - a constructionist activity. This section explores some of the background behind constructionism and how it affects the brick.

2.1.1 History

Dr. Seymour Papert formed the Epistemology and Learning research group at the MIT Media Lab in order to explore the interactions between learning and digital technology. The primary focus of this group has been on Dr. Papert's philosophy of constructionism. As a result, much of the research been concerning the development of "toys to think with" [18][33] [34]. These are toys children can use to construct their own artifacts.

Integrating Dr. Papert's interest in artificial intelligence, this group became the first to explore programming as a constructionist activity. These explorations lead to one of his most famous contributions, the Logo programming language.

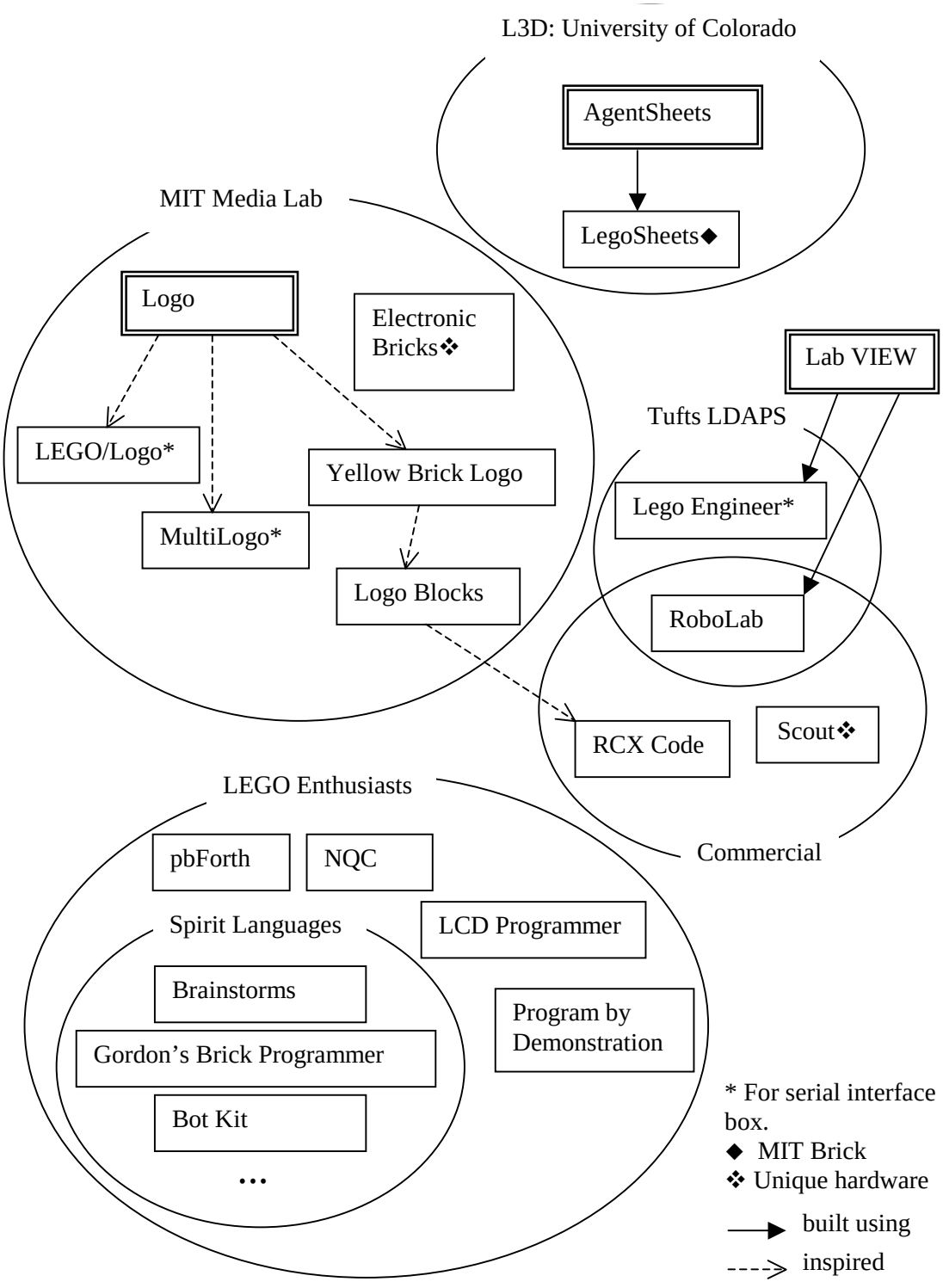

Figure 2.1
Relationships of Programmable Brick languages

The earliest incarnation of Logo was in the form of a specialized robotics language. This particular robot was designed to drive on a large sheet of paper with a pen that it could lift and push down on the surface. By dragging the pen as the robot moved, it could draw geometric patterns on the paper. Children could program this robot by pressing switches, instructing it to move forward, turn a certain number of degrees, etc. This robot became nicknamed the "Turtle", because of the shape of the protective plastic dome covering the circuitry (Figure 2.2).

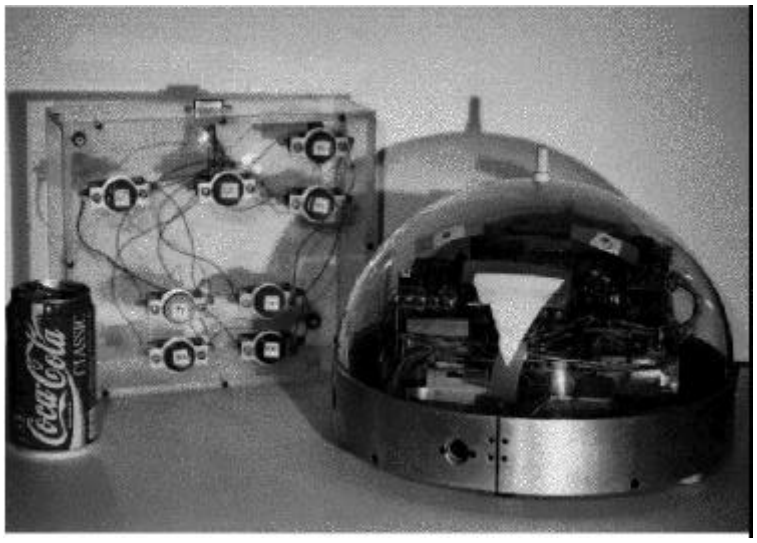

Figure 2.2
The Robotic Logo Turtle (reprinted from [15])

It was, however, Logo's second incarnation as a graphical language that caused its widespread acceptance. Without the need for robotics, which were quite expensive, schools were able to provide children with an explorative, programming language that existed entirely on the computer (Figure 2.3).

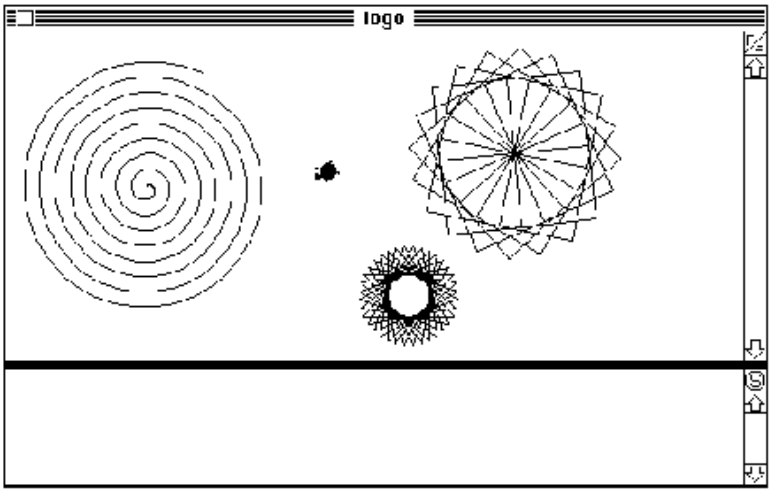

Figure 2.3
The Graphic Logo Turtle (reprinted from [15])

2.1.2 Collaboration with LEGO

A shared interest in the constructionist approach to learning between LEGO® and the MIT Media Lab was recognized in 1986 [37]. As a result, the MIT Media Lab entered into a research partnership with LEGO® in order to develop an integration of physical construction with programming. It is this research partnership that eventually led to the development of the programmable brick.

LEGO® Technic™ parts provided a very flexible architecture for building the mechanical aspects of the robot, and the Media Lab developed hardware and software that would interface with these parts. The final product was a micro controller embedded within a plastic brick with several ports for attaching sensors and actuators. This programmable brick, along with the sensors and actuators, were designed to interface with LEGO components. In addition, this brick was designed such that the user could develop their own programs.

2.1.3 Constructionism & Programming Languages

Since the beginning of the E & L research group, constructionism has been associated with programming. From this long association, there are principles of what a programming

language needs to be in order to be a valuable constructionist tool. First is that the tool should be relatively easy to learn. For instance, the Logo language supports implicit type declaration and a very simple syntax to support ease of learning. Second, the tool should have a very high ceiling. This means that the user should be able to implement a wide variety of solutions with the language. Logo's functional model has been expanded over the years and used for many different applications. It is the interaction between these two principles that becomes challenging.

An assumption that has gone into the development of constructionist programming languages like Logo is that children do not reason about programming in an inherently different manner than adults. What affects how people reason about programming, is experience with programming. If a language is developed such that it is intuitive and consistent to the novice adult, it should be the same for a child.

2.2 Hardware Evolution

The brick has gone through several cycles of evolution. Though the focus of this thesis is on language issues, it is relevant to give a history of how the hardware has evolved since these changes have had, in some cases, a direct affect on the languages.

2.2.1 Serial Interface Box

Before it was feasible to have a microprocessor resident on the robots, an interface "box" was made to allow the user's desktop PC act as the brain of the robot. The box acted as an interface between the robot and the computer. [18] This allowed analog motors and sensors to be operated from a desktop PC (Figure 2.4).

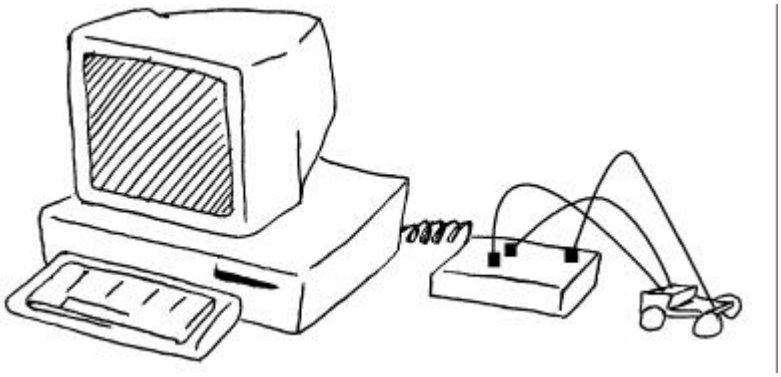

Figure 2.4
Serial Interface Box

Children would built a robot using LEGO® Technic™ motors, gears, blocks and sensors. The sensors and motors would be connected to the serial interface box. Then, the children would write a program on a desktop PC to take input from the sensors and control the motors.

Several versions of the serial interface box have been developed. However, all of these followed the same basic model. The important feature is that the robot was tethered. This limited the possible tasks that could be solved using the robot. Most of the robotics projects developed under this model involved stationary robots, since mobile robots can get easily tangled.

2.2.2 Braitenburg Creatures

The limitations imposed by the serial interface box model made it evident that a model for autonomous robotics was needed. Two such models were pursued. One model involved a single, intelligent brick with a powerful processor and several peripheral motors and sensors, the programmable brick. The other model involved a set of smaller bricks, each representing a logical statement. These smaller bricks would be connected to physically form the program. This second approach became known as Electronic Bricks [11] [18].

The robots created with these bricks became known as Braitenburg Creatures, named after Valentino Braitenburg. These vehicles were in a large part inspired from a book that he wrote, Vehicles: Experiments in Synthetic Psychology [4]. In this book, Braitenburg introduces a series of theoretical robots, composed of logical circuits. These robots where theoretical in that they were not implemented; their purpose was more to serve as a medium for thought experiments. Starting with simple, reactive algorithms, he builds in complexity until achieving a robot that displays "intelligent" behavior.

Though the goal of the electronic brick project at MIT was not to achieve robot consciousness, Braitenburg's work still bore relevance. Braitenburg demonstrated that interesting behavior could be accomplished with analog circuits using principles like positive and negative feedback.

For instance, Braitenburg introduced a robot that has an affinity to light. This robot has two photocells in the front and two drive wheels in the back, each drive wheel connected to a separate motor. The photocells are connected to the opposing motors (right photocell to left rear motor, and vice versa). The circuitry is set such that the more light a photocell receives, the more power is sent to the corresponding motor -- positive feedback (Figure 2.5).

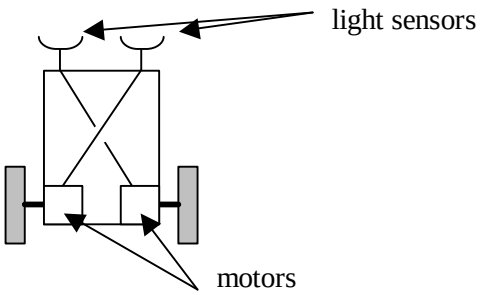

Figure 2.5
Positive-feedback light follower

This resulting robot is one that will head towards the brightest light source in the room. It will continually weave back and forth, turning towards the light, unless both photocells have the

same value; in other words, the vehicle is pointed towards the light source. At this point, both motors turn at the same speed making the car go forward in a straight path.

It is this approach to robotics programming that inspired the electronic bricks. The robot is programmed in terms of positive and negative feedback systems set up with sensors and actuators. For instance, the above robot would be implemented with four electronic bricks: two photocell bricks and two motor bricks. In fact, the physical robot would look much like Braitenburg's theoretical robot.

The electronic brick model is unique in several ways. First, it provides a manner of programming totally independent of a desktop PC; electronic brick programs are not composed or debugged on a PC. Second, it is the only analog programming language for this domain; instead of composing a program of commands, a program is described in terms of logical relationships between sensors and actuators. Third, it is the only physical programming language; the user defines a program by connecting physical bricks together in patterns that describe the logical circuitry.

This line of research was eventually dropped because it was more cost-effective to push all of the electronic controls to one, large brick. With this move, analog brick programming was dropped by the wayside.

2.2.3 MIT Programmable Brick

The hardware for the programmable brick has gone through many instantiations -- both in the academic and commercial fields. A discussion of all of these versions is outside of the range of this thesis. Instead, the hardware advancements pioneered at the Media Lab will be discussed collectively as the MIT Programmable Brick. Discourse on commercial versions of the brick will be primarily limited to the RCX model (see next section).

There were many individuals who contributed to the development of the hardware for the programmable brick. However, much of the technological move, from a serial interface

box to an on-board microprocessor, is attributed to Dr. Fred Martin. As part of his Ph.D. dissertation, Fred Martin taught a series of winter-term courses, called the *Robot Design Competitions*, at MIT [15] [16]. The purpose of the course was to give undergraduate engineering students experience in working with a problem in the physical world. This was a college-level approach to constructionism.

In addition to giving insight into the problem solving techniques of these undergraduate students, these competitions allowed Dr. Martin to refine the development of the Handy Board; a very close relative of the brick. The first year, the board was comprised of an on board controller, but no processor. The controller still had to be tethered to a PC that ran the program. At this stage, the students were writing programs for the robot in an assembler. Subsequently, the brick became the processor for the robot and merely connected to the computer to download programs, upload data or run in an interactive, command line mode (Figure 2.6).

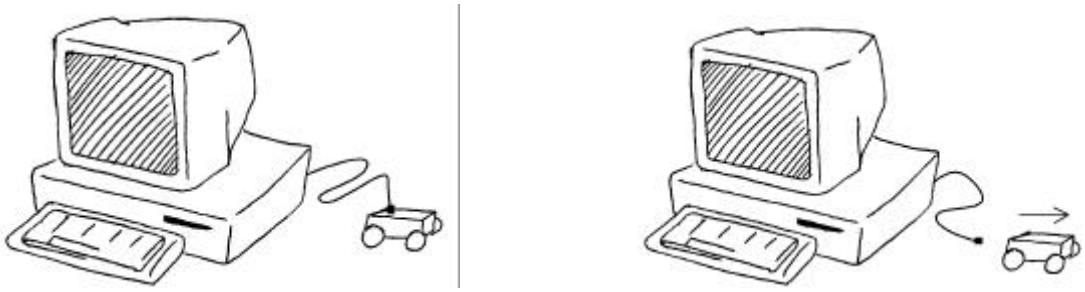

Figure 2.6
Temporary physical link to computer

Dr. Martin then went on to lead a series of outreach programs to local schools with an adapted version of the Handy Board. This was the programmable brick. As the brick moved to schools, various adaptations of the Logo language were used to program the brick. [17]

However, the primary mode of use remained the same. A single brick operates as the CPU as well as digital to analog controllers for the motors and sensors of the robot. The program is composed on the computer, compiled and downloaded through a temporary connection to the brick. The connection is removed and the brick can run the program autonomously.

2.2.4 RCX

There have been several commercial instantiations of the programmable brick released by LEGO. However, for the purposes of this study, the focus will be on the LEGO® RCX™ Programmable Brick. Most of the languages examined in depth in this paper were developed for the RCX.

The RCX is much like the MIT prototype bricks. It has an embedded microprocessor along with a series of input ports and output ports. To these ports a number of different sensors and actuators can be attached. Unlike the MIT bricks, which use a physical connection to communicate with a desktop PC, the RCX uses an infrared (IR) transmitter/receiver (Figure 2.7).

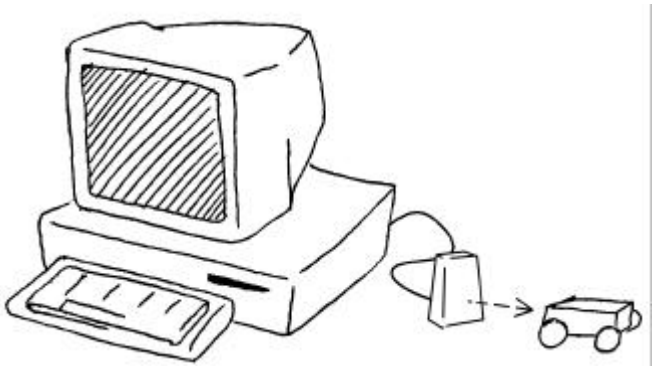

Figure 2.7
IR tower and programmable brick

The IR port allows the RCX bricks to communicate with each other. With most programming languages, the communication protocol is very primitive and it only allows for one byte of data to be sent with each packet. However, this is still enough for some interesting operations.

The programs for the RCX are compiled to a byte code before they are downloaded onto the brick. These byte codes, or op codes, are run by the brick's *firmware*. It acts as the operating system of the brick as well as the implementation of the virtual machine that runs the downloaded byte codes. This firmware is maintained in RAM memory of the brick; therefor it is volatile and can be erased by cutting the power source.

2.3 Brick Languages

Given the short amount of time that the programmable brick has been in existence, a significant number of languages have been developed. The history of these languages is relevant in that it establishes a context for understanding implementation decisions of the existing languages.

2.3.1 Logo Dialects

The Logo programming language was originally designed for use with a specialized robot. This language then was moved to a GUI environment where the user controlled the movement of on-screen sprites. As the LEGO brick research came about, Logo was moved back to robotics. This time Logo was not used for a specific robot, but for an open-ended class of simple, extensible robots.

Logo is a functional language with a very minimalist syntax. It was designed to be easy to learn, but allow programmers to solve complex problems.

From the MIT Media Lab there have been several generations of brick programming languages. Most of these languages are variations of Logo. This section is an outline of some of these languages.

2.3.1.1 LEGO/Logo

LEGO/Logo was developed for use with the serial interface box. Essentially, this language had the basic Logo syntax with special commands added for checking sensors and operating actuators [18]. The user is given a single thread of control, so concurrent monitoring and action must be performed using time slicing.

The environment had a simple, textual interface but was quite appropriate for many of the types of models that could be constructed with the serial interface box. However, there was still an inherent limitation that arose out of the lack of a concurrent model.

2.3.1.2 MultiLogo

The limitations of languages like LEGO/Logo motivated explorations into models of concurrent languages. MultiLogo was developed as a tool to explore conceptual difficulties children have with concurrent, *agent-based* programming [32].

The user writes a program as a series of interacting agents. Each agent has its own name, state and thread of execution. Communication between the agents is accomplished through asynchronous message passing.

The message passing in MultiLogo is unique in that data is not passed between agents, but commands. For example, agent X sends the message 'off' to agent Y. Upon processing the message, agent Y executes the command 'off' (turning off the motor). Each agent has a queue of commands. When a message is sent, it can be sent to either the head of the queue or the back of the queue, by the choice of the sender. The `ask` command sends the instruction to the end of the agent's queue, while `demand` sends the instructions to the head of the agent's queue (Figure 2.8).

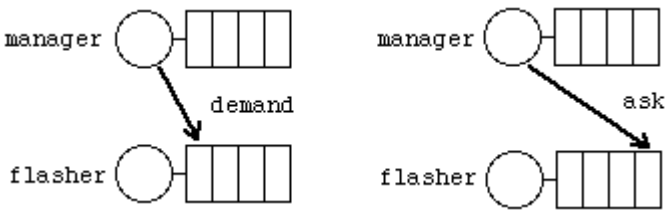

Figure 2.8
Instruction queuing in MultiLogo
(reprinted from 32])

Figure 2.9 is a MultiLogo program for a robot that moves back and forth while flashing a light.

```
walker ==>
to walk
  talkto :motor-port
  repeat 6 [onfor 30 rd]
end

flasher ==>
to flash
  talkto :light-port
  repeat 20 [onfor 4 wait 2]
end

manager ==>
to walk-and-flash
  ask :walker [walk]
  ask :flasher [flash]
end

manager ==>
walk-and-flash       ; initiate the program
```

Figure 2.9
Walker/flasher program in MultiLogo

The program consists of three agents, the walker, the flasher and the manager. The manager's only job is to start the walker and flasher on their closed-loop routines. This is a common formula for MultiLogo programs. In fact, it is the only use of message passing that is demonstrated in [32]. This suggests that there is more of a need for the ability to spawn statically defined processes. This will be discussed further in the next chapter.

Case studies revealed significant conceptual problems in children working with these languages. Specifically, asynchronous message passing and internal agents were difficult to use [32].

2.3.1.3 Yellow Brick Logo

Yellow Brick Logo introduced another adaptation of Logo that handled concurrency in a different manner. Yellow Brick Logo (YBL) was based on Microworld's (MW) Logo; a commercial product developed by LSCI®. MW Logo adds concurrency features in order to allow users to make multiple, interactive turtles (sprites). Turtles that can run around on the screen following some pattern, yet checking for collisions with other sprites and/or mouse clicks. These concurrency features, originally intended for interactive animations, were ported to the brick.

Like its parent language, Logo, YBL is a functional language. In order to accomplish this, replacement firmware was developed for the RCX. The existing firmware that LEGO provided did not have a stack, so could not support a functional language. YBL has not been made available to the general audience of programmable brick users. At the time of this thesis, it has only been distributed to education and research groups.

2.3.1.4 Logo Blocks

Logo Blocks is a visual counterpart to the Yellow Brick language. The user programs by connecting icons representing commands to create a control-flow diagram. These icons are obtained from a multi-sectioned "bin" (Figure 2.10). With a visual language, such as Logo Blocks, the dividing line between the language and the development environment becomes unclear.

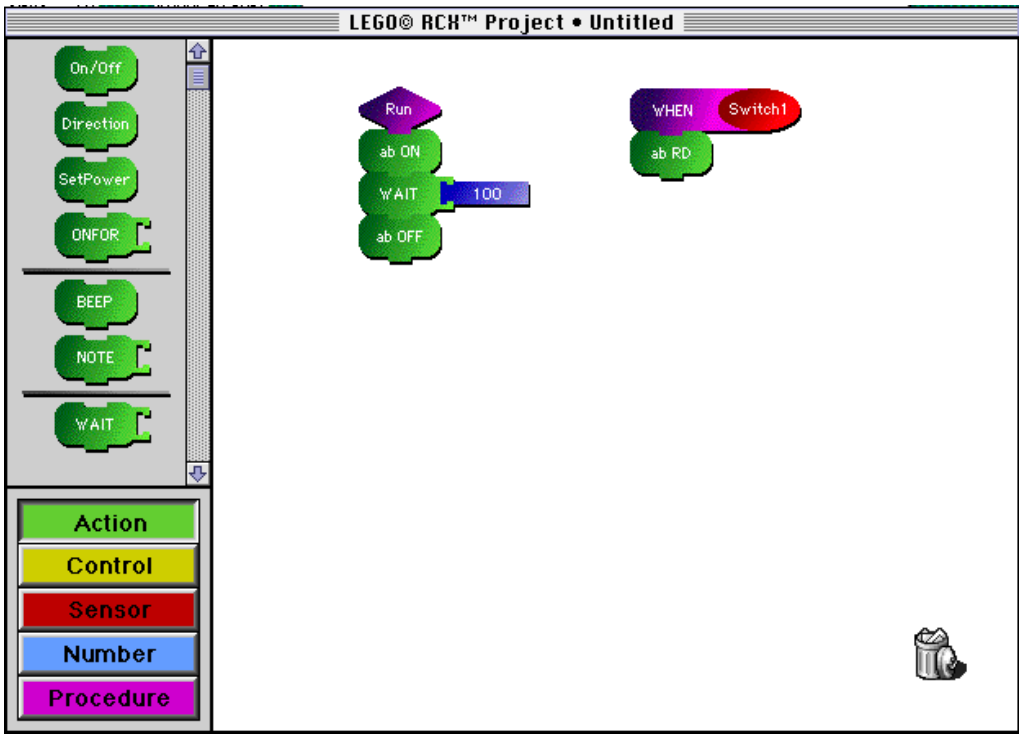

Figure 2.10
Logo Blocks environment

The syntax of the language is expressed through the shape and color of the command blocks. The user is unable to make syntactically incorrect programs, because syntactically incorrect combinations simply do not fit together. In addition, when the user encounters an unknown command, the shape will indicate how the command works. For instance, observe Figure 2.11.

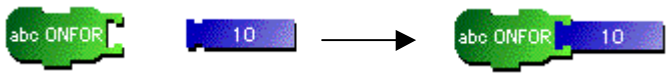

Figure 2.11
Visual linking syntax demonstrated with a parameterized function

The language does not, however, provide any assistance with understanding semantics. The abstract colors and shapes portray nothing concerning their underlying meaning. This is in

contrast to Robolab (presented later in this chapter) which uses icons to convey semantic meaning.

The language does not have quite as much flexibility as Yellow Brick Logo. All of the variables are global, user-defined procedures that cannot accept parameters. In addition, the control over concurrent processes is not as great as with YBL (see next chapter).

To express a flexible syntax requires a fairly complex set of tokens. There are commands, variables, control structures, logical statements and operators, etc. This results a large number of icons to sort through to find the correct one for a specific task (Figure 2.12).

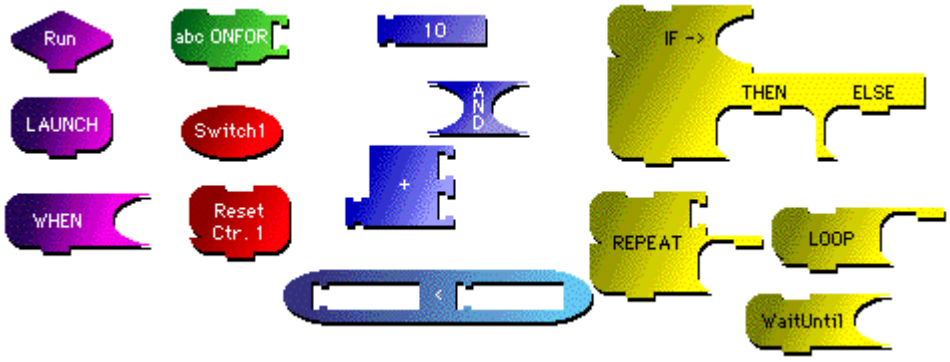

Figure 2.12
Some of the variety of syntactic shapes in Logo Blocks

However, Logo Blocks is a wonderful stepping stone to a textual language because each block has the equivalent textual command within it. Essentially, extract the text from the color-coded blocks and the user has Yellow Brick code. Alleviating syntactic guesswork, thereby lessening the barriers to entry, is the most significant contribution of this language.

2.3.2 LDAPS Research

The LEGO Design and Programming System (LDAPS) research at Tufts, like the work at the Media Lab, took an approach of looking at how technology can affect education [8]. This research initiative was also actively collaborating with LEGO. In addition, LDAPS was

collaborating with National Instruments – a software development company focusing on process control systems.

The result of this collaboration was the development of two LEGO robotics-control languages using National Instrument's LabVIEW. Because of the significant affect that LabVIEW had on Tufts' languages (Lego Engineer and RoboLab), it is relevant to first discuss the LabVIEW environment before the languages that were built on top of it.

2.3.2.1 LabVIEW

LabVIEW supports a visual, extensible programming environment developed primarily for use with process control systems. A program is defined by creating directed graph. Nodes represent functions to be performed on data. The arcs connecting the nodes represent flow of data or control; depending on the type of arc.

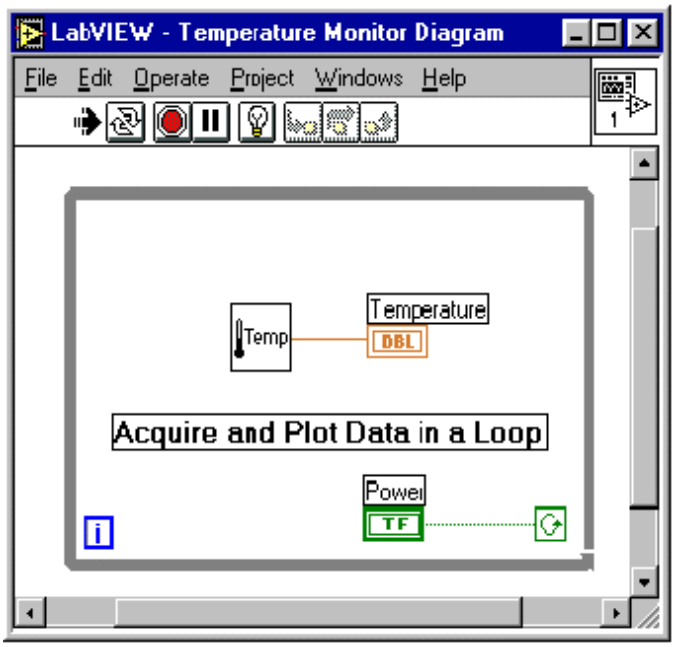

Figure 2.13
LabVIEW visual syntax

Figure 2.13 depicts a sample LabVIEW program that samples temperature data. Notice that there are actions depicted by squares within a looping control structure. Shape, color and spatial metaphors are used to express a program.

Since LabVIEW is meant for process control, the environment assumes a certain mode of use. A PC is running the actual program, connected through an interface card to the external sensors and actuators of the physical system.

As mentioned earlier, this environment is extensible. National Instruments provides an API for defining new types of nodes (functions) to exist within LabVIEW. It was this API that was used by Tufts' LDAPS group to develop LEGO Engineer and RoboLab.

2.3.2.2 LEGO Engineer

LEGO Engineer is a programming language for the serial interface box. The model of use with the serial interface box is very similar to the model of use that LabVIEW's software was developed for. For this reason, LEGO Engineer was developed within LabVIEW; by adding subroutines that communicate with LEGO's hardware. This was done making a two-stage development environment.

In the first stage, there are two windows; a front panel window and a diagram window (see Figure 2.14). The front panel window is a direct interface to the serial box. This allows users to affect the actuators. Motors can be turned on and off for periods of time defined by the users. It was observed that children did not make much use of this window[8]. The diagram window interacts with the output window, allowing users to define programs. Each node represents some type of operation. For instance, in Figure 2.13 the operations are wait for button press, call outputs and wait 5 seconds. The call to outputs refers to the output window. The motor states reflected in the output window are activated when the flow of control executes the outputs node.

The second stage of LEGO Engineer discards the front panel and focuses strictly on the diagram window (see Figure 2.15). This stage introduced a wider array of control structures, such as loops, forking and *if else* statements.

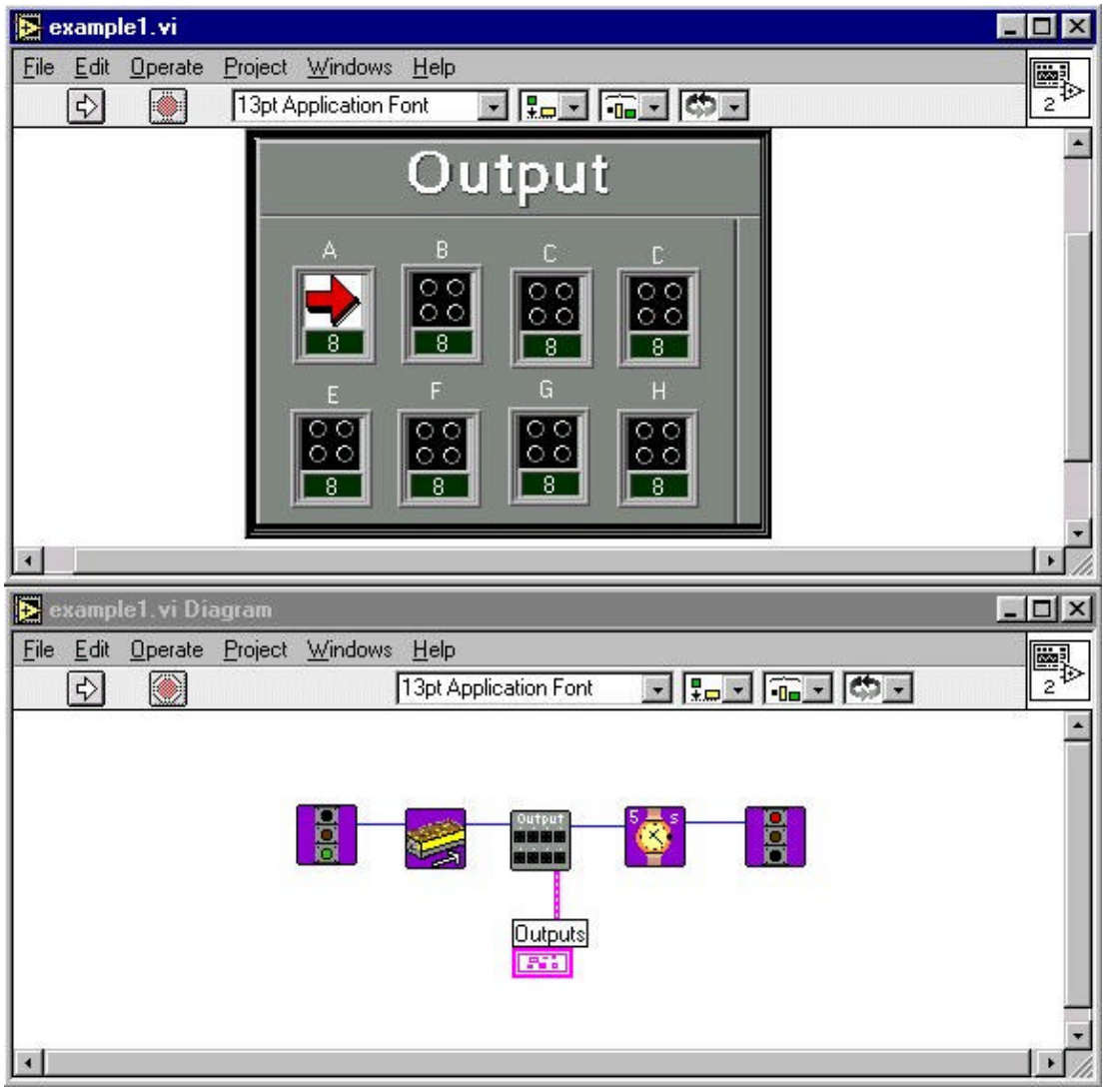

Figure 2.14
LEGO Engineer interface; 1st stage

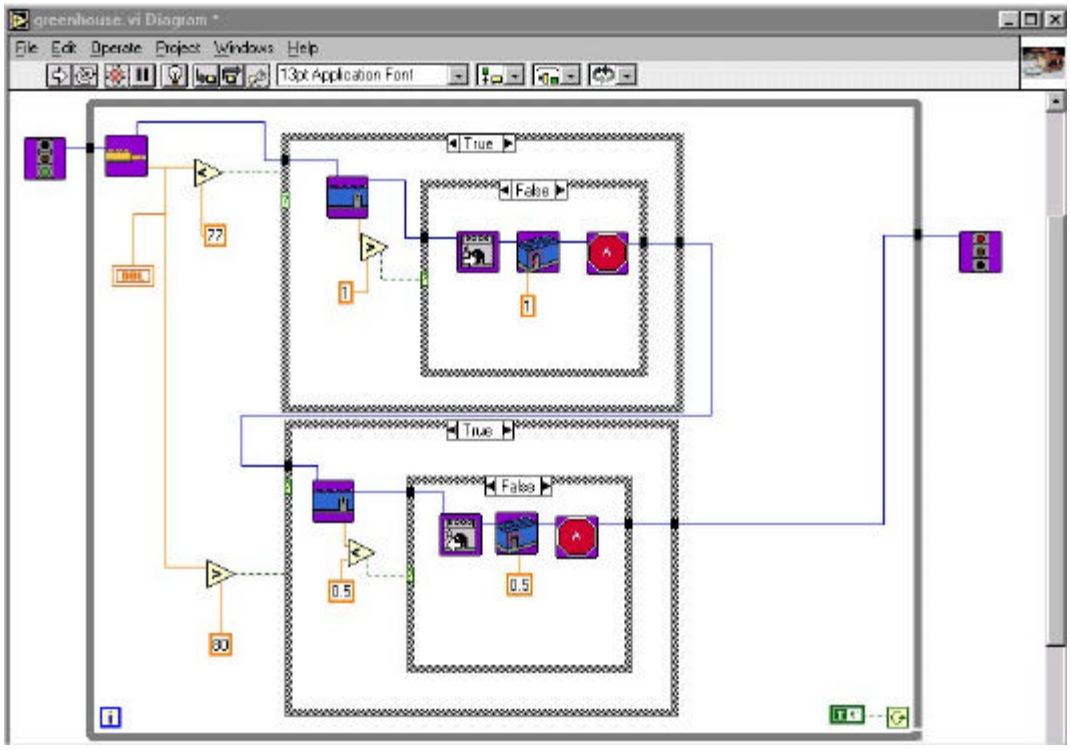

Figure 2.15
LEGO Engineer; 2nd stage

The control structures used are LabVIEW control structures. Notice that the program is described in terms of both control flow and data flow. The dark, thin lines indicating the ordering of events and the light, thin lines representing the flow of data. For instance, the temperature is sampled immediately after the start of the program; since it is the first node directly after the start signal (green light on top left). Out of this temperature sensor comes a control flow arc indicating the next operation to perform, and also a data flow arc that forks and goes to two separate tests. The tests are executed in the order that the control flow indicates, but the data being tested is from the source that the data flow indicates. The other interesting feature to note is how control structures (such as the *ifelse* and *loop* above) encapsulate the nodes that represent isolated commands, such as sample sensor data or turn on motor. These are syntactic features of the LabVIEW language itself.

2.3.2.3 RoboLab

The RoboLab language was designed by the LDAPS group to work with the RCX programmable brick. Remember that the brick, unlike the serial box, has an embedded processor so it can download the program and run autonomously from the computer. The RoboLab language is one of two currently available commercial languages.

RoboLab, like LEGO Engineer, also has two development levels (in this case, the Pilot and Inventor). It was also built on the LabVIEW system. However, this language further distanced itself from the visual syntax of the original LabVIEW diagram language. This was not a deliberate design choice, however. As will be explained further on in this section, the differences between the syntax of RoboLab and LEGO Engineer were more the result of technical constraints.

The Pilot language provides users with a restricted interface for creating programs. Only certain commands are available and there are no control structures. In addition, the Pilot

view provides a more restricted interface for creating programs than the diagram window of LEGO Engineer (Figure 2.16).

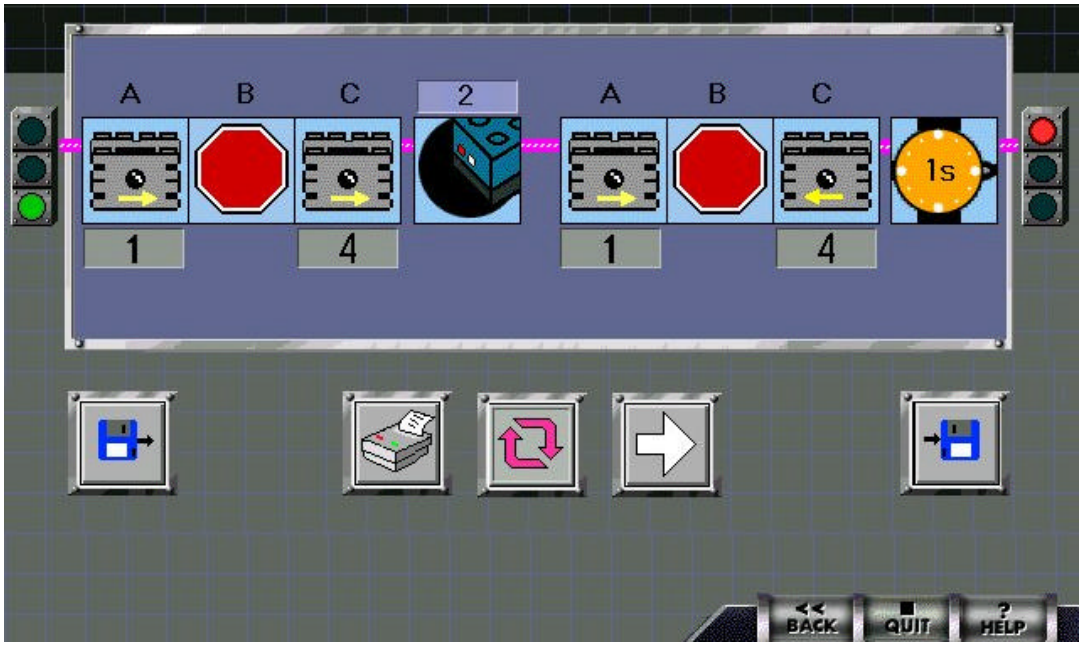

Figure 2.16
RoboLab: Pilot Level

Instead of a free form layout, commands are snapped together in a list. This provides for only single process, linear programs. The main idea behind this interface is that language is easier to learn since the child is presented with fewer options up front. Once the user becomes comfortable enough with the Pilot interface, they move on to the Inventor.

The Inventor interface more closely resembles the diagram window of LEGO Engineer. The user is provided with an open field for placing nodes. These nodes are connected with arcs to define control flow. Unlike Engineer, data flow cannot be defined. Sampled sensor data must be explicitly stored in a variable to be used at a later point in the program.

In addition, encapsulating control structures (such as the *loop* and *if else* statements illustrated in LEGO Engineer) do not exist. All programs are completely defined in terms of

nodes and control flow arcs. Control structures still exist, but they are defined in terms of these nodes. For instance, a loop is defined by placing a node denoting the beginning of the loop and a node denoting the end of the loop (Figure 2.17).

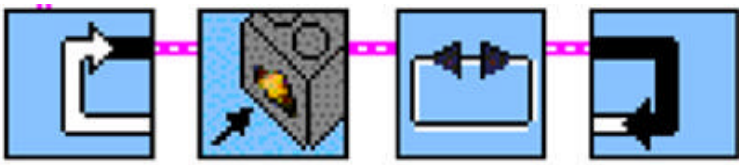

Figure 2.17
Loop with operations in RoboLab

The loops are not iterative, they are equivalent to a labeled *goto* statement. This is an unfortunate step back from the syntax of LEGO Engineer. The *loop* control structure of Engineer clearly encapsulated the operations that were within the loop (see Figure 2.15). Because the looping and other control structures of RoboLab do not encapsulate the code they affect, they do not show as clearly the overall structure of the program. This becomes especially evident with more complex problems (see Appendix A).

This change in control structures was made because of a technical mismatch between LabVIEW and the RCX brick. Since LabVIEW is designed for constant interaction between the PC and the external devices, the built-in control structures run on the PC. The PC does the processing, and the external devices are merely for I/O. The LEGO Dacta serial interface box fit this model of computation, so the traditional LabVIEW control structures were usable. The RCX brick, however, does not fit this model. With the brick, the program is downloaded from the PC to the robot, the communication link with the PC is severed, and the robot operates autonomously, internally processing inputs and operating actuators.

The square nodes in Figure 2.17 are user-defined subroutines (this feature is part of the extensible nature of LabVIEW). These subroutines are used to generate byte codes for the

RCX. The RoboLab "control structures" are simply LabVIEW commands that concatenate byte codes on to a program byte string that is sent to the RCX. In many ways, RoboLab is like an assembler with a visual syntax. Altering the actual control structures of LabVIEW to interface with the brick would have been a significant change to the underlying architecture. Though RoboLab was developed in an academic setting, it was distributed in a commercial market. Due to this, there was a push to get the language finished in a short time. Adding user-defined subroutines to act as control structures was an alternative that didn't require changing LabVIEW itself, and hence, was a faster option.

2.3.3 L^{3}D Research Group

Center for Life Long Learning and Design at the University of Colorado has been doing research and ongoing development with an application called AgentSheets [31]. This application is a visual, agent-based programming language for developing interactive languages. Like Logo, this language is meant for constructionist learning

One of the research projects under the L^{3}D group was creating a brick programming language in AgentSheets. This language, developed for one of the MIT prototype bricks, is LegoSheets.

2.3.3.1 LegoSheets

LegoSheets is a very unique programming language for the brick, so a little more time will be spent on this language to explain it fully. The language integrates use of iconic and forms-based programming in the only declarative language for the brick encountered in this study [9] [13]. This language, like MultiLogo, uses agency. The running program is the result of the interactions between the agents in the system.

In LegoSheets, the agents that are added to the program are chosen from a predefined selection of agents. These predefined agents have particular roles. The types of predefined

agents are as follows: motors, sensors, timers, global variables and “power user” (complex relationship) agents.

Each agent in LEGO Sheets has a single integer representing its state. The state of an agent is globally visible, but can only be altered by the agent itself. This single integer has a different meaning for different agents. For instance, with a motor the number represents the speed and direction (8 for full speed forward, 0 for stop, -8 for full speed backwards, etc), with a sensor, the state is the current sensor reading, and so on.

Each agent is programmed with a series of *declarative rules* (non-declarative rules are introduced in the next chapter). The format of a declarative rule is as follows:

```
if <condition> then <state change>
```

The condition of the rule is typically defined in terms of the state of some other agent in the language. For instance, take the following motor rule:

```
if TOUCH2 = 1 Then –5
```

This rule states that if the touch sensor on port 2 is equal to the value 1 (it is depressed), then set the motor to go backwards at power level 5. The rules within each agent are mutually exclusive and have a precedence, defined by the user. This means that if more than one rule within an agent is true at a given point in time, only one of the rules is executed. The order that the rules are listed in defines the precedence, therefore the first rule listed would be executed whenever there was a potential conflict. In addition to defining rules based on external conditions, there is an initial and default case.

The programming interface has two interacting components: the Worksheet and the Rule Editor. The user defines a program using both of these components.

The Worksheet is where the user defines all of the agents in the program by placing them on a grid (see Figure 2.18). The large block in the middle of the worksheet window represents the programmable brick. On the graphic brick is a series of motor and sensor ports. Motor agents and sensor agents must be placed next to the ports that they are to communicate

to. Timers, global variables, and power user agents can be placed arbitrarily anywhere else on the grid (except for on the brick).

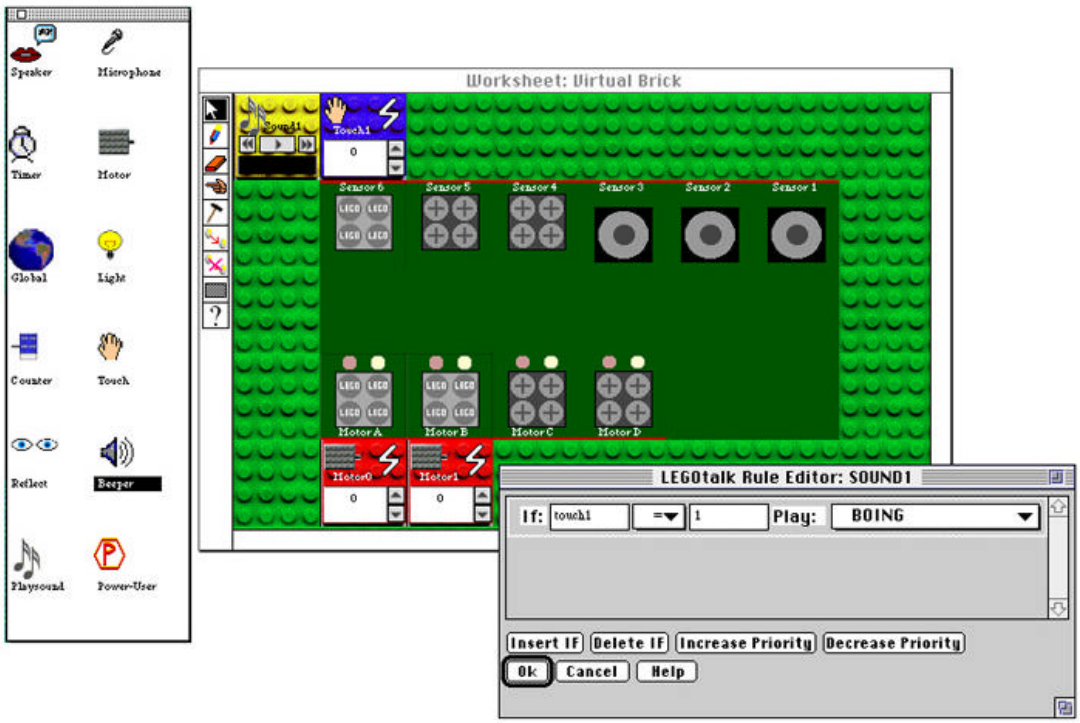

Figure 2.18
LegoSheets environment

For each agent, a rule editor can be opened (Figure 2.19). This is a *forms-based* interface for adding the rules that the agent must follow. Forms-based languages are ones where the user is presented with a series of edit boxes, pull-down menus, and other standard interface components in order to define the program.

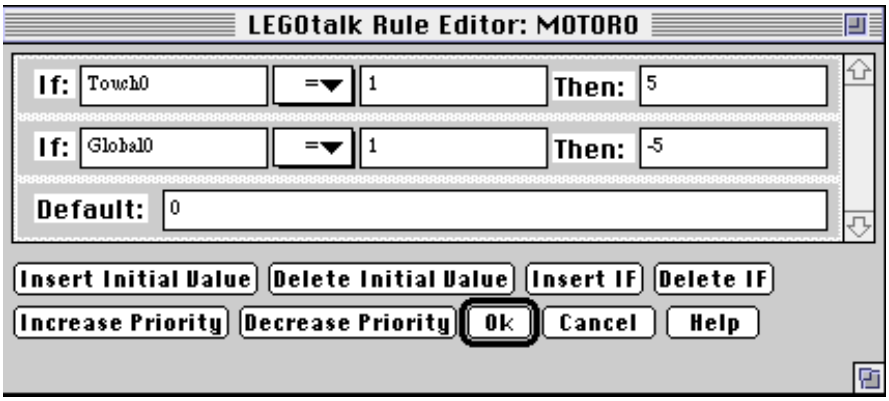

Figure 2.19
LegoSheets rule editor

2.3.4 Commercial Languages

For the RCX, there have been two base kits released (each with a series of extension kits). The home use kit is known as *Mindstorms*. The product is named after Dr. Seymour Papert's book on computing and education [28]. The other language, *RoboLab*, is intended for use with schools. The programming language with the school package was developed by Tufts LDAPS group. The Mindstorms kit uses a different programming language that is described below.

2.3.4.1 RCX Code

The RCX Code is the language distributed with the commercial LEGO™ Mindstorms® kit. There are many features of the language used here that closely resemble Logo Blocks. The overall language is a visual programming language using a control flow model. Most differences between RCX Code and Logo Blocks were implemented in order to make the language easier to learn.

This language is composed of iconic commands that are connected together to form multiple stacks, representing a program. The language provides standard control structures familiar to a procedural language.

The RCX Code language, like Logo Blocks, uses shape and color to represent syntax. However, the RCX Code syntax is simplified from LB; parameters for commands and control structures are entered through interface widgets built into the block, versus snapping on external blocks (Figure 2.20).

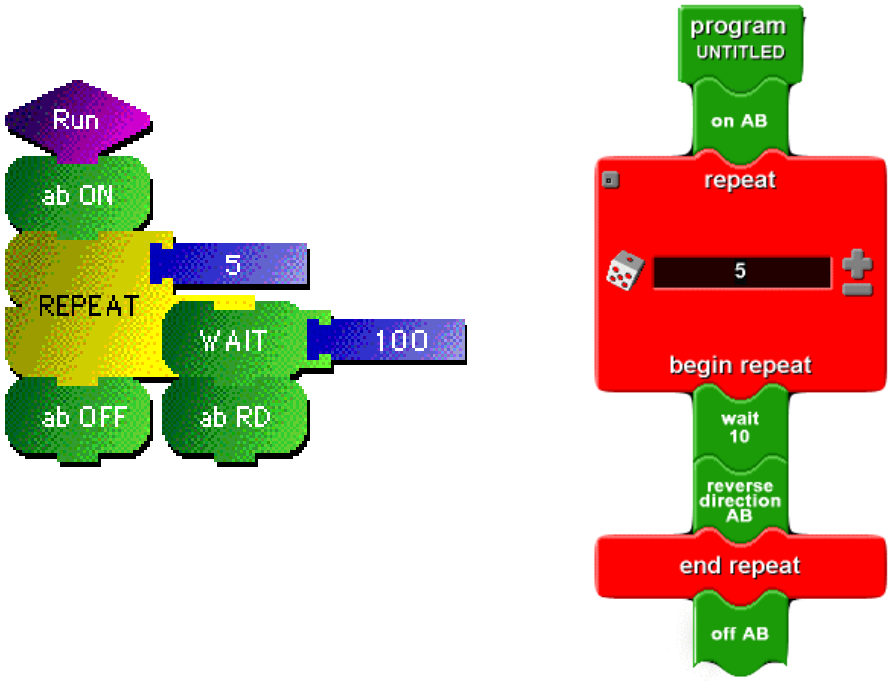

Figure 2.20
Repeat statement in Logo Blocks (left) and RCX Code (right)

Probably one of the most serious limitations of the language is that there are no variables, only a single counter. The counter can only be incremented and reset to its initial value, zero. The primary focus of RCX Code was to get the fastest "out-of-box" response. The language needed to be non-threatening and give the user immediate success. In addition, since there is not necessarily going to be a teacher around to instruct the user on programming, the language had to be pretty self-explanatory. However, because of the lack of variables, making robots to

solve dynamic problems, such as ones involving calibration, becomes difficult and sometimes impossible.

2.3.5 Unofficial Languages

It did not take long after the commercial release of the programmable brick for a community of enthusiasts comprised equally of adults and children to arise. The community itself has pushed the bounds of the brick and created several innovations.

Though the physical tool quickly proved popular, the RCX Code programming language was found to be too limiting by much of the programmer community [12]. Advanced users wanted variables, complex expressions, etc -- none of which was available in RCX Code. RoboLab, which had provided more capabilities in this aspect, saw little exposure to the market of hobbyists since the language was only advertised to schools. The other academic languages saw little or no exposure to the greater audience at all. As a result, there was a need in the hobbyist community for a more powerful language.

There were two resources that made further languages available. First, LEGO published the API to an Active X control for interfacing with the brick. This API (called Spirit) handled the IR port communication protocols and provided a wrapper around the standard firmware's op codes along with some simple control structures.

However, hobbyist development did not stop there. A reverse-engineering effort of the hobbyist community removed the need for the Spirit API. Kekoa Proudfoot, of Stanford University, published the op codes for the firmware of the RCX brick on the web [29]. The availability of the op codes, along with the reverse-engineering of the IR communication protocols, made available still more languages; such as Not Quite C. In particular, removing dependency on the Active X component allowed development environments on other platforms, such as Mac and Linux to arise.

Kekoa Proudfoot also published pictures of the inside of the brick on the web, which allowed people to determine the circuitry used and totally circumvent LEGO's firmware. Two firmwares that replaced the brick's operating system were LegOS [23] and pbForth [10].

Given this brief history concerning the LEGO brick community, the following sections describe with more detail some of the particular brick languages that were developed as a result.

2.3.5.1 Spirit Languages

As a result of the Spirit interface, several languages have been developed [3] [19] [36] using Visual Basic and Visual C++. Like NQC, these languages are all wrappers around the LEGO firmware with little abstraction provided to the programmer. Each of these languages simply introduces a different syntax to access the same operations provided by the underlying firmware. It is worth noting that some of these languages come with environments that demonstrate interesting interface principles. However, since these languages do not demonstrate any unique approaches to concurrency, they will not be further explored in this paper.

2.3.5.2 Not Quite C

Not Quite C (NQC) is, as its name indicates, a C-like programming language for the RCX brick [1][2]. It is not a true C language, but it mimics enough of the C syntax to be comfortable to programmers. The user can define subroutines, macros, compiler directives, etc.

The primary goal of this language was to expose as much of the capabilities of the firmware to the user as possible. The language includes a library of functions that have direct mappings to the op codes of the LEGO firmware. Other than some control structures and the

ability to name variables, subroutines and tasks, NQC allows pretty much direct access to the tools provided by the firmware.

It was decided that this language will receive some further exploration. It serves as a metric for other such "bare bones" languages that provide relatively little abstraction from the firmware of the brick. Most of the analysis in the next chapter concerning tasks can be applied to all of the Spirit languages.

2.3.5.3 pbForth

Forth is a stack-based, embedded-languages language. The pbForth language is a port of this language for the programmable brick [10].

With the pbForth development environment, the desktop PC is essentially a dumb terminal for the brick. Input typed in at the keyboard is sent to the brick as input, the brick processes this input, and sends back a response.

Another unique feature is that this is the only language that does not use pre-emptive multitasking. The pbForth language uses cooperative multitasking. With this model, the user must explicitly define points in code where a process will yield processor control to another process.

2.3.5.4 Program by Demonstration

With program by demonstration, there is no entry of code, via a personal computer or otherwise. The user "demonstrates" what the robot has to do, and the robot will extrapolate an algorithm to follow [20].

The robot runs in two modes, a learning mode and an executing mode. While the robot is learning, the user physically manipulates the robot, moving it through a series of actions. The actions are recorded and generalized into a set of instructions. When the robot is set to execution mode, the robot executes those instructions.

In addition to recording sequences of actions, this language also associates sensor activation with actions. For example, if the robot's touch sensor is activated during the learning mode, the movements performed in the next few seconds become associated with the touch sensor being activated.

A simple, obstacle-avoiding robot could be programmed as follows. Take a robot with a front-mounted touch sensor and a couple steering motors. Activate the learning mode. Push the robot directly forward; setting the rule that the robot should normally drive straight. Bump the robot into a wall activating the touch sensor. The robot emits a beep indicating that it is associating the next few seconds with the touch sensor activation. Back the robot away from the wall while turning it. Turn off the learning mode. (Figure 2.21)

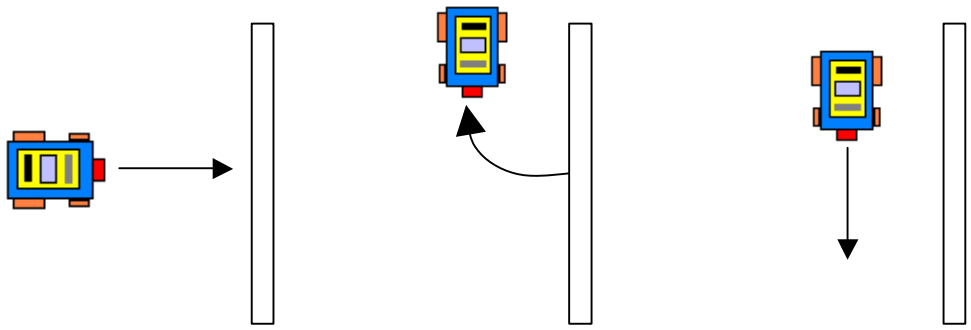

Figure 2.21
Program by example steps for obstacle avoidance

When the robot is activated it will drive forward under normal circumstances and, when the touch sensor is activated, it will back up and turn around. The entire program is by default set in a loop so the robot will return to driving straight afterwards.

This is a very fascinating method for programming simple algorithms. Removing the dependency on a desktop PC allows the user rapidly iterate through many designs. However, this comes at a price. The user cannot return to an algorithm and change just one operation. The entire algorithm must be changed at once. In addition, the robot must be constructed in such a way that the actuators can be physically manipulated. With many robots, the

mechanical structure prohibits manipulating actuators in this manner. Therefore, this method of programming is not necessarily a good fit for generalized robotics.

2.3.5.5 LCD Programming

A separate attempt to remove the need for a desktop PC is the LCD Programmer [35]. The LCD interface of the RCX brick is turned into a programming interface. The user is able to view one command at a time with the screen. Due to the severe limitations of the LCD display, this language is fairly cryptic. This language closely resembles programming in language op codes. Programming is done in terms of a single line of process using sequential commands which are entered by pressing combinations on the button interface of the brick. This much the same process as programming an alarm time on a digital watch.

Interestingly enough, the idea of LCD programming has seen another incarnation in the commercial realm. In the line of LEGO programmable bricks is a brick called the Scout®. This brick has a larger LCD display that is made for programming the robot. The display has a small, predefined set of options for programming the robot with relatively high-level operations; such as “drive in a zig-zag pattern”.

The difference here is that instead of a sequence of instructions, the user defines a series of overlapping operations that the robot follows. For instance, the robot is told to drive in a zig-zag pattern, but it is also given an affinity for light. As a result, the robot may break for the zig-zag pattern when it encounters a bright light source. It becomes difficult to determine, through mere observation, what the precedence of the operations is or, in some cases, which is active at a given point in time.

2.4 Summary

The existing languages for the programmable brick span a wide range of methodologies and come from a wide variety of sources (refer again to Figure 2.1). However, most of these

languages hold in common the fact that they are adaptations of languages that were developed for domains other than the brick. There has also been little cross-comparison of these languages. What does exist is more along the lines of listing features of the development language, not how suitable the languages are for the domain. The next chapter will introduce such a discussion.

3. Concurrency Analysis

This chapter takes the languages presented in the previous chapter and groups them into three major categories in terms of the concurrency tools they provide. These categories are tasks, splits and rules. For each category, the characterizing features of concurrency introduced in chapter 1 are analyzed.

3.1 Languages Not Covered

Not all of the languages discussed in the previous chapter will be included in this discussion of concurrency. Techniques such as program by demonstration and LCD programming either do not involve concurrency, or do not present it in a manner that can be controlled by the user. MultiLogo's agent-based model with asynchronous message passing is very conceptually complex. Case studies with children revealed many misconceptions about how concurrency worked in this language [32]. Due to these previously documented difficulties, MultiLogo was not examined here.

The pbForth programming language was also viewed as too complex to be practical for children's programming. This is primarily because of the cooperative multitasking model used. With cooperative multitasking, the programmer must explicitly define where control over the processor is yielded by one process and given to another. Concurrency is a difficult model as is; explicitly dealing with context switching makes the pbForth model too complex to be useful for novices. Instead, all of the models examined in this chapter use preemptive multitasking.

3.2 Tasks

For the purposes of this thesis, tasks are statically defined processes that can be started and stopped by other tasks at arbitrary points in time during the running of a program. The underlying operating system that comes standard with the RCX brick provides a concurrent model based on tasks. The user can define up to ten tasks for each program. One task is by default the starting task; much like the main routine in the C language.

Tasks are a special type of process. They cannot be dynamically allocated and they are permanently associated with a particular code block. Because of this static allocation, tasks have a static process id that can be determined before run time. Many task-based languages, such as NQC, provide an abstraction on top of this by giving tasks names.

The syntax for controlling tasks is based on procedure calls. For instance, in NQC the user will call the *start* procedure and give it the name of which task to start as a parameter. Likewise, there is a *stop* procedure that halts the specified process. Other languages, that do not provide the abstraction of naming, use the index of the task. (Figure 3.1)

```
task main ( )           // default beginning task
{ start buttonCheck;    // begin 2nd process
  OnFwd(OUT_A);
  Wait(1000);
  Off(OUT_A);
  stop buttonCheck;     // end 2nd process
}

task buttonCheck ( )
{ while(true)        // Loop forever, checking touch sensor
  {   if(SENSOR_1 == 1)
          Rev(OUT_A);
  }
}
```

Figure 3.1
Multiple task algorithm in NQC

The program above sets a robot to drive for ten seconds, then stop. While the robot is driving, whenever the touch sensor on port one is activated, the robot changes direction. One could imagine this as a robot that bounces off walls. Even in this simple example, there are

interactions between the processes. Even though the main task starts the motor running forward, this does not necessarily indicate the motor's state for the duration of the ten seconds before the motor is turned off. At that point, the motor could be forward or backwards, depending on how many times the direction has been reversed.

Articulation - Tasks provide the highest degree of articulation of any of the concurrency tools in this study. The user can start, stop and restart tasks at any point of the program. This is because the user is explicitly creating processes. It is true that the user cannot dynamically define new processes, but this is the case with all of the languages. The reason for this is that tasks are what the underlying Spirit firmware provides. Since most languages are based on this firmware, they cannot provide a more flexible model.

Process Creation - The process creation of tasks is explicit. This is accomplished by using the start command.

Syntax - Tasks are code blocks much like subroutines. Instead of being called, tasks are started and stopped. A problem with tasks is that the calls to control them are embedded in sequential code. Calls to control these parallel processes look very much like normal sequential calls.

Conflict Resolution - There are no conflict resolution tools provided by NQC version 1. During the writing of this thesis, a new firmware for the RCX was released that does provide some tools for managing concurrent access of the motors. These tools were made available in a new version of NQC, but an analysis of these tools is not included in this thesis.

Visibility - Due to the function-call like syntax of tasks, the visibility is very low. It is difficult to determine, at any point in the code, what processes are active. To do so necessitates the user deliberately stepping through the program to determine what tasks have been activated and deactivated.

Naturalness - The primary difficulty with the naturalness of tasks comes from the common method of use for parallel processes in brick robots. It is most common to use these

processes to monitor external events (Appendix A). Therefore, the user implements their own event model within a task by creating an *if* statement nested within a loop. Intuitively, this is not a natural manner in which to express reactions to the environment. We do not describe our own actions in terms of looping and constantly checking for events.

3.3 Split

A split is similar in some respects to the Unix-style fork. The current process breaks into two separate processes at a particular point of the code. However, there are also significant differences.

First, with the Unix fork both processes continue executing on the same segment of code. For a programmer to define two different operations to take place concurrently, the typical method is to save the process ID before the fork, then test that value after the fork to decide which operation to do. With a split, each process is assigned a segment of code to operate on. This way there is no necessity to test which process is active, it is inherent by which code block is being executed. For instance, YBL performs a split using the *launch* function. This function creates a new process and assigns it to operate on a list of commands.

```
…
launch [repeat 100 [ beep wait 50 ] ]
someFunction         ; repeat loop concurrent to this.
…
```

The above function creates a process that will perform the repeat loop while the current thread of process goes on to the next function. Notice that no process ID is necessary here. Incidentally, neither of the languages that provide splits, Yellow Brick Logo or RoboLab, provide a means of obtaining the current process ID.

The second major difference involves state. When a process forks in Unix, each resulting process has a separate copy of the state of the program. All of the examined languages make use of global variables that are shared across all processes. So for these

values, there is no notion of separate copies of state. YBL is the one language that uses splits and has local variables as well as global. Even with local variables, though, there is no separate copy of state. The local variables of the function that calls *launch* are not accessible by the generated process. For instance, take the following code block.

```
to main
  let [x 5]
  launch [ repeat x [beep wait 50 ] ]  ; error
end
```

This segment of code would not compile because `x` is not recognized in the process generated by the launch command. The only way to share values would be to use a global variable, and then all processes would be affecting the same state.

Articulation - The split looses much of the articulation of tasks. The user cannot restart a given process, nor can they stop a process that is running. The user can control when a process is initially started, but each process is responsible for ending itself.

Process Creation - Like tasks, process creation with splits is explicit. The user, however, is not given any handle to manipulate that process. For instance, with a task, the task name is used to start the process, so it serves as a handle to do further manipulation. With a Unix fork, a process ID number is available so that the programmer can affect other processes by that ID.

Syntax - The syntactic expression of splits is an interesting issue; not so much with the YBL *launch* command because this is simply a function, but more so with the RoboLab implementation of the split. Where YBL's use of launch provides no more assistance in visualization than tasks, RoboLab's directed graph syntax complements the concept of the split. Recall from the previous chapter that RoboLab is a visual language that uses a control flow model with nodes and arcs. The split is particularly well suited to this language because it can be represented visually by a fork in the graph (Figure 3.2)

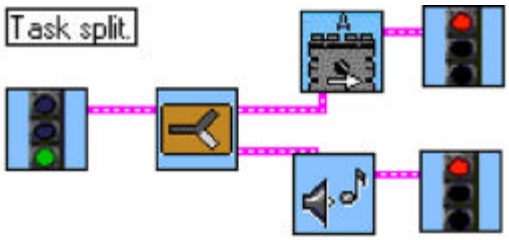

Figure 3.2
RoboLab split

The split node accepts one incoming arc and two outgoing arcs, creating a very nice visualization of concurrency. Both threads are executed to completion. Unfortunately, there is no indication of how long the threads execute. In the above example, the top thread will run for approximately 10 seconds before turning off the motor and ending. The bottom thread will loop forever.

As can be seen in this example, some of the sequential control structures in this language are represented in a sequential manner, one incoming arc and one outgoing arc. The arcs between the nodes could be a visual representation for the process. This model breaks down with RoboLab's implementation of *if* statements. This language provides a choice node where the graph forks and the thread of process follows only one of the possible paths.

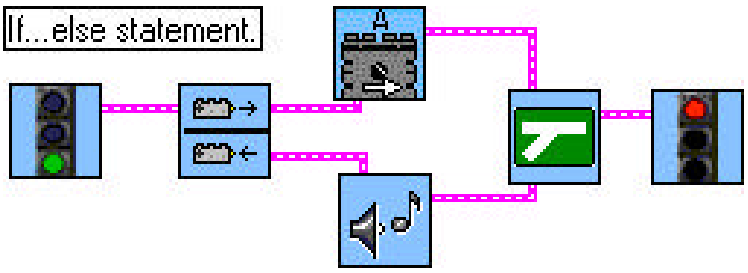

Figure 3.3
RoboLab *if…else* statement

The code in Figure 3.3 either turns the motor on or plays a tone, based on the state of the touch sensor. The key difference syntactically between the *if* statement and the split is that the graph

joins once again with the if statement. With complicated programs, though, this can become obscured and splits and ifs can become confused.

The above reflects a more general problem of expressing concurrency in two-dimensional languages. One dimension is used for expressing an ordering, or sequence of operations. The second dimension typically shares responsibility of expressing concurrency as well as control structures. There has been research into the possibility of three-dimensional programming languages for this specific reason [21]. This way sequence, control-flow and concurrency all have their own dimension to be expressed. However, a 2D computer screen is not a sufficient language for working with a 3D language [21].

Given this current hardware limitation, it is necessary to look at how two dimensions can express three axis of control. Inevitably, two of the axis of control will have to share the same dimension. Typically these two are control-flow and concurrency. It is left to additional syntax to clearly delineate when that second dimension is being used to express control-flow and when it is being used to express concurrency.

For the purposes of the current discussion concerning axis of control a correlation between visual and textual languages is relevant. Textual languages are often regarded as one-dimensional, but white space (such as the tab and carriage return) is often used as a second dimension of expression. This is typically for the sake of the human readers, as most compilers ignore this white space. Regardless, if by constraint or by convention, two dimensions of expression are allowed to textual languages as well, leaving the same problem of expression three axis of control in a 2D space.

Conflict Resolution - Conflict resolution is not an inherent feature in splits.

Visibility - The *launch* implementation offers no more visibility of concurrent process than tasks does. However, the visual split of RoboLab does a good job of using the graph

syntax to express parallelism. As discussed in the Syntax section, this is somewhat mitigated by the *if* statement having a similar syntax.

Naturalness - There are many metaphors that can be used to reinforce the idea of a split. Roads and rivers split to allow flow down two parallel paths. It is the issue of syntax described above that confuses the split.

3.4 Rules

There are two types of rules discussed in this thesis, imperative and declarative. The subsections 3.5.1 and 3.5.2 will characterize the differences between these two types of rules. First, however, the common properties of rules will be discussed.

Both of these types of rules are similar in that they are an association of a condition to a response. The condition is monitored, and when it tests true, the response is executed (or fired).

```
<condition> → <response>
```

When a rule is *active*, the condition of the rule is continuously polled, so that the response is executed as soon as the condition becomes true. When a rule is *inactive*, the condition is not being monitored, so the response will not be executed whether the condition is true or not.

Naturalness - The concept of rules is a very powerful one when interacting with real-world languages. For instance, there is the correlation to user interface programming. Interrupt handling takes on very much a rule-like methodology. A mouse click on a certain button is responded with a certain sequence of actions. The monitoring is handled implicitly by the operating system that throws interrupts for a standard set of conditions. Process control systems are another example of this. The system is defined in terms of conditions and responses to maintain a certain state. If the temperature gets to high, initiate a cooling process as a response.

Rules also have particular relevance to a field of research known as *natural programming*. Natural programming is based on the concept that by studying how people use natural language to describe programming tasks, patterns will be discovered that can be used in a formal programming language. Case studies in this field have revealed a tendency of novices to use rule-based descriptions for instructions [26][27]. "When the robot hits the wall, it turns around." We don't naturally think in terms of polling and context switching. However, it necessary in order to implement the above natural language algorithm in sequential code.

In fact, this tendency of understanding in terms of rules has presented difficulties for novices in understanding sequential programming. Different studies have seen this manifest with misunderstanding the *if* statement [25][32]. The *if* statement is often treated as though it launched a separate, daemon process to monitor the condition and executes the associated operations *whenever* the condition becomes true. This is not presented as an argument against *if* statements, but rather as an argument for the inherent *naturalness* of rules. If the concurrency of rules is naturally assumed, then it will be a valuable tool to use.

Both types of rules share a similar naturalness. However, in the other characteristics of concurrency, they differ. These differences will be described below.

3.4.1 Imperative Rules

An imperative rule is a structure that associates a condition with a response that is composed of a series of operations. When the condition is true, the response is acted upon by sequentially executing each operation in the series.

```
<condition> → { <op1>, <op2>, … <op n> }
```

There is a subtle implementation decision to how the examined languages use this construct that is not inherently implied by imperative rules. This is that while the actions of the response are being performed, the condition is not being monitored. Therefore, the

sequence of actions cannot be restarted to execute again before it has finished. In addition, the condition will not test true again until it has become false. For instance, take a rule that plays a melody when a touch sensor is pressed. If the touch sensor is pressed and held down, the melody will only play once. The touch sensor must be released and pressed once more for the melody to play once again.

The imperative rule is an interesting blend between sequential and declarative mindsets. The series of operations making up the response is executed sequentially, but the initiation of that sequence is performed in a more declarative manner. This allows languages that are primarily sequential to integrate some reactive programming constructs.

Three of the languages examined made use of imperative rules: Yellow Brick Logo, Logo Blocks and RCX Code. Within these languages, there are variations in how rules are implemented. These differences are based on whether the rules are explicitly activated by the user at some point in the source code or are implicitly activated without the user specifying when to start.

3.4.1.1 Yellow Brick Logo (Explicitly Activated)

In YBL, rules are stated in the form of two different functions; *when* and *every*. The *when* statement accepts two lists as parameters, one defining a condition and one defining a sequence of operations. From the point that the *when* function is called, a separate process is started that constantly monitors the condition and executes the sequence of operations whenever the condition becomes true. The *every* statement is a slight variation on this. It accepts two parameters as well, a number and a list of operations. The list of operations is executed regularly with a period determined by the first parameter, which indicates the size of the period in ticks. As functions, these two types of rules are embedded within the sequential code.

```
to main
  when [switch1] [ab, rd]    // rule
  ab, on
  wait 100
  off
end
```

This is the same program that has the robot driving for ten seconds, reversing direction when it collides with an obstacle. An important point is that the program would have a very different meaning if the *when* statement were the last instruction in the list. The rule would not be activated until the robot has stopped moving. This is the problem that can happen when rules are mixed within sequential code. They give a deceiving view of parallelism. The *when* statement defines a rule and activates that rule in the same step. In this case, the task approach gives a more straight-forward view of what is happening. The definition and the activation are separate.

Articulation - The articulation of the YBL imperative rule is limited to activation. The rule cannot be deactivated when it has started.

Process Creation - Process creation is explicit. Interestingly enough, the process is defined at the same point that it is created. A side effect of this particular implementation is that there is no name or id number left as a handle to further manipulate the process.

Syntax - As a descendant of Logo, YBL is a functional language. The use of concurrency does not violate this. These rules are actually defined and created by using functions. For instance, the *when* function accepts two parameters – a condition and a list of operations to do when that condition proves true.

Visibility - Visibility of active processes is nearly as complicated as with tasks. The call to activate a rule is nested within sequential code; therefore determining the processes active at a certain point necessitates stepping through the code.

3.4.1.2 Logo Blocks and RCX Code (Implicitly Activated)

Logo Blocks and RCX Code provide a slightly different approach to imperative rules. The programmer defines rules as code blocks separate from any sequential routines. These rules are implicitly activated when the program begins running, and remain active for the duration of the program (Figure 3.4).

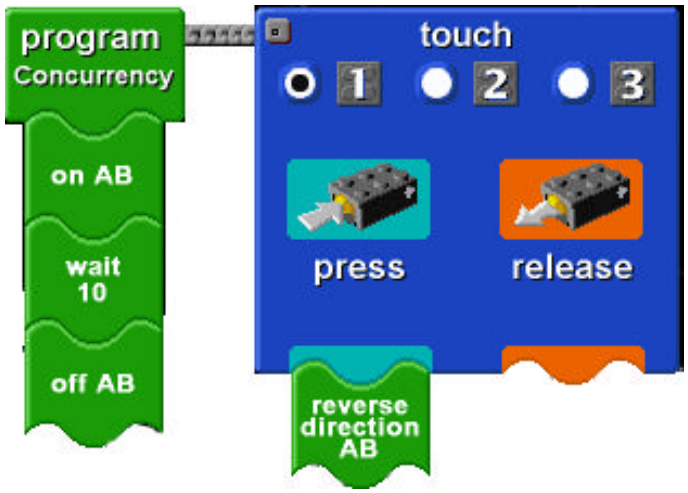

Figure 3.4
Rule in RCX Code

Articulation - The articulation of processes is less than that of explicitly activated rules because the user is not in control of when the rules are activated. This is not necessarily a trait that has to be associated with implicitly activated rules. This lower articulation is caused by the particular design of Logo Blocks and RCX Code.

Syntax - Rules are arranged as code blocks independent of the main program routine or any subroutines defined. This makes rules stand out clearly giving them an axis of control unique from sequence and control structures.

How these code blocks are arranged spatially brings up some interesting issues. Logo Block's rules are placed arbitrarily on a plane (Figure 2.10). All of the concurrent processes

on the plane are assumed to be running at once. Incidentally, *launch* exists in LB but is treated more as a rule. It defines a sequence of code that is to be run as soon as the program starts. References to concurrent processes are totally removed from the sequential code. Within each code block, the vertical axis is used to represent sequence and the horizontal is used for control. Separate code blocks represent either procedures or concurrent processes. Shape of the leading block indicates the difference between subroutines and rules.

RCX Code also uses code blocks to distinguish rules, however it makes more use of spatial reasoning. Rules are lined up along the horizontal axis with the main thread of the program. Subroutines, on the other hand, are placed arbitrarily on the workspace. The vertical and horizontal axis are used to express sequence and control while grouping on the horizontal axis is used to express parallelism. Also notice that there is only one type of rule in this language, it is the equivalent of the when statement. The forms-based interfaces of the rule blocks allow the user to chose from a statically defined set of rule conditions.

Conflict Resolution - Like most of the languages discussed in this chapter, there is no inherent conflict resolution.

Visibility - Visibility is very high with these languages because all of the rules of the system are active from the start of the program to the end, period. There is no ambiguity of what processes are active.

3.4.2 Declarative Rules

Declarative rules remove the mixing of sequential code with parallelism. The response of a declarative rule is not an action, but a state change. This type of rule currently exists only in the LegoSheets language. In fact, in LegoSheets the only construct available is the declarative rule. Because the language is so irrevocably connected to the concept of declarative rules, a fair amount of discussion of this construct was introduced in the Previous

Work chapter (Section 2.3.3.1). This discussion will focus on how the concurrency issues relate to LegoSheets.

All of the rules for each agent are active during the entire duration of the running program. The user's articulation over these rules comes in the form of setting the precedence of the rules within each agent. For anything beyond a simple, reactive robot, such as obstacle avoidance or line following, using precedence becomes quite tricky. This is primarily because any type of sequencing of actions requires special techniques (explained below).

The LegoSheets language has a limitation that does not allow the user to view more than one agent's rule editor at the same time. This limitation is not inherent to the language, though, so it will not be counted in evaluating the visibility of concurrency. LegoSheets is only a prototype and it will be assumed that a more developed environment would allow the user to view arbitrary sets of rules at any time.

Given this assumption, visibility is still an issue. The user must track, for each agent, what rule will be fired based on precedence and the current state of the machine. Under any situation, every agent will fire one of its rules, even if that rule is the default.

3.4.2.1 Building in Sequence

Though declarative rules work well for reactive languages, performing a sequence of actions can be difficult. To do this, the programmer must create an artificial program counter that the rest of the agents in the system refer to. Instead of reacting to the sensor or timers, the agents must react to the current instruction.

In LegoSheets, this can be accomplished using two agents in conjunction, a *power user* agent and a global variable agent. The power user is essentially a global variable that can evaluate complex conditions, whereas all of the other agents can only evaluate simple conditions in their rules. The power user tests the state of the system by looking at the other agents, and sets its own state to reflect the next step. The global variable looks at the state of

the power user and sets its own state based on this. The actuator, timer and other variable agents of the system look at this program counter variable to determine what their state should be (Figure 3.5).

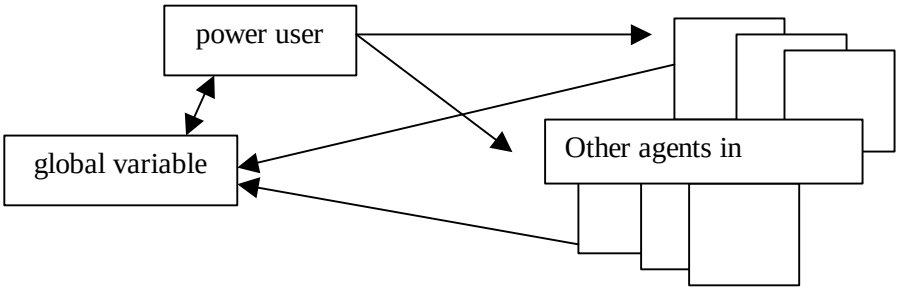

Figure 3.5
Model for building sequential structure in LegoSheets

Notice that the global variable is referenced as the program counter and not the power user. This is because the condition initiating a state change of the counter to *n* may itself change before the counter should update to *n+1*. The global variable provides a sort of buffer.

For instance, the robot collides against a wall and goes into a backup routine, updating its program counter. The condition initiating the change, the touch sensor activation, will cease to be true as soon as the robot is no longer in contact with the wall. However, it may be desirable for the robot to stay on this operation even after the touch sensor is no longer activated. The global variable changes itself when the state of the power user first changes. After that initial change, there may be times when none of the power user's rules are sufficed, letting it change to a default value. The global variable ignores these default values and only changes its own state when the power user is a value other than zero.

Aside from the difficulty of imposing sequence on the LegoSheets language, there is another important point to notice. The program counter here is not quite like a normal program counter in that it indicates a set of actions that are taken in parallel. When the counter is in a given state, the robot's overall behavior is a certain response. This starts to form a higher-level grouping of rules, an idea that will be explored further in the next chapter.

3.4.2.2 Concurrency Features

Articulation - All of the rules are active all of the time so articulation does not seem that great at first. However, the rules within each agent have a set precedence that is used to determine which rules fire when there is a conflict. This increases the amount of articulation, but not to the same degree of being able to shut rules completely off.

Process Creation - Process creation here is implicit. All of the rules are active from the beginning of the program until the end. In fact, how the rules are actually implemented in terms of processes is quite hidden from the user.

Syntax - With LegoSheets, imperative rules are the only type of syntactic construct available. There is no possibility of confusion, because there is nothing possible to confuse rules with.

There is a great benefit here in that learning the language is extremely simple. One basic construct performs everything. However, there is also a deficit to the way this is set up. The axis of concurrency is totally sacrificed. There is nothing inherent in the language that allows users to sequence a series of actions. Because of this, the elaborate work-around described in section 3.5.2.1 is needed.

Conflict Resolution - LegoSheets is the one system analyzed in this study that has implicit conflict resolution. This is because of three principles.

1. The only state information in the system is the state of the agents.
2. Only the agent can effect its state.
3. The rules within an agent, governing its state changes, are ordered by precedence.

Based on the written work concerning LegoSheets it is difficult to tell whether this was an intended feature, or just a side effect [9][13]. This benefit is not mentioned specifically anywhere.

Visibility - Due to nature of agency, the behavior of a robot is difficult to determine. Its actions are not captured by the code, but by the dynamic interactions between the different agents. Even though all of the rules can be assumed to be in parallel, the actions of the robot are much more difficult to determine. Therefor, the visibility is quite low.

3.5 Summary & Evaluation

Task based languages provide the highest degree of articulation of languages currently in the brick domain. However, the visibility of active processes is very poor.

Split operations provide no more visibility in textual languages than tasks do. Within a control flow visual language, such as RoboLab, it would provide better visibility if it weren't that visual forks also indicate decisions. This brings up the issue of being constrained to two dimensions for addressing three axis of control (sequence, control and concurrency).

Rules offer a more intuitive approach to reactive programming by making implicit use of concurrency. Imperative rules are used within more sequential languages and are either implicitly or explicitly activated. Explicit activation creates a similar problem to that with tasks, where active processes become less visible. Implicit activation causes rules to be active during the entire duration of the program; decreasing articulation, but creating a better visibility of the state of the processes. Declarative rules make reactive programming very simple, but any ordering of sequence requires an artificial program counter. Given this program counter that gives a central view of the language's state, all other agents within the language can react based on this global state.

The rule metaphor is quite powerful and the implicit activation of rules makes the concurrency of a language highly visible. This is not a factor of visual languages per se, but of having sets of code blocks that can be assumed to be active at the same time. Unfortunately, this configuration limits the user to a single, statically defined set of processes. The next chapter suggests a language construct that groups *implicitly activated imperative rules* into sets that can be activated and deactivated. This construct (called a *Mode*) leverages the intuitive nature of rules while providing the user a higher degree of articulation.

4. Mode-Based Programming

It has been observed that reactive techniques for programming help create autonomous robots that are more resistant to failure [15][16]. In addition, reactive algorithms are often used to describe the actions of languages with interacting agents [25][27]. Several existing languages for the brick integrate rule-based elements in order to allow reactive algorithms. However, the degree of articulation concerning these rules limits their usefulness. This chapter proposes a new language construct to increase the articulation of rules, thus making reactive algorithms more feasible for solving simple robotics problems.

4.1 Definition of a Mode

Modes offer a higher level structure to control rules. Rules in YBL and RCX Code are simply created, but they cannot be deactivated once instantiated. Users must resort to semaphores to mask out rules. However, a mode offers a way to group a set of rules together and activate and deactivate those rules.

A mode is a grouping of imperative rules that are to be active at the same time. All of the rules that are within a given mode are activated when the mode is started. When the mode is exited, all of the rules for that mode are deactivated.

```
mode <modeName> {
    <condition 1> → <operation list 1>    // rule 1
    <condition 2> → <operation list 2>    // rule 2
    …
    <condition n> → <operation list n>    // rule n
}
```

A *modal* program is defined in terms of modes, not procedures or processes. During the execution of the program, control is passed from one mode to the next (like a finite-state machine). Only one mode is active at any given time.

The primary goal of modes is to create scope constructs for rules in the simplest manner possible. Creating scope for rules is inherently complex, so the range of articulation with rules is limited for the sake of maintaining as much simplicity as possible.

4.1.1 Modes and Concurrency

Modes clearly define all of the rules that are active when the program is in a particular state. The advantage of modes is that there is no ambiguity concerning the concurrent processes that are active at any stage. The programmer can cleanly switch between different sets of rules with a single command.

The limitation imposed by modes is that the user is limited to statically defined sets of rules. Modes cannot be nested within each other. Once one mode calls another, the calling mode is no longer active. The newly called mode has complete control. In addition, modes cannot be dynamically instantiated during run time. All modes are statically defined in much the same sense as procedures are in C.

To make either nested modes, or dynamic mode creation available would work against the purpose of modes. One of the most important contributions of modes is visibility. A programmer can look at the source code and easily see what concurrent processes are active at any point.

It is recognized that the limitation on how modes can be used keeps modes from being as general purpose as tasks. However, robotics projects created with the brick tend to be ones that can fit into the modal class of problems (Appendix A). In other words, brick robotics programs typically can be described in terms of static sets of concurrent processes. This thesis proposes that, though some types of programs cannot be implemented with modes, this approach makes accessible the field of programs that apply to the brick.

4.1.2 Modes and Procedures

Modes do not preclude sequential program decomposition. A mode-based language should support functions, procedures, or some other type of sequential code block. These operations are then called from the rules of the mode language. The context of an operation is determined by the mode it is called from and the operation is executed in the thread that is dedicated to its calling rule. If a mode is exited while a function is being executed, the function is immediately halted. It is very significant that no remnants of the mode being exited are left when the next mode is started.

4.1.3 Other Options for Controlling Rules

The reason why programmers cannot activate and deactivate rules in languages such as RCX Code and YBL is that there is no point of reference for accessing the process created. For instance, in NQC, processes are defined using named code blocks called tasks. That name provides a point of reference to act on the process. Using its name, a process can be started and stopped.

As an interesting side note, YBL syntax does not inherently preclude control processes. In fact, MicroWorld's Logo, the language YBL is descended from did have a similar feature. MW Logo added concurrency to the original Logo with the *launch* statement and later on the *when* statement. The way this was done is relevant because YBL's treatment of concurrency was derived from MWL.

In MW Logo, to stop a process that has been begun with a launch, the user calls *cancel* passing it the same command list that was passed to the *launch* command. The language finds the process that is executing that list of commands and stops it. To stop a *when* statement, the programmer calls cancel on the condition the *when* statement is monitoring.

```
; Micro World's Logo syntax for controlling processes.
launch [ repeat 100 [fd wait 50 bk] ]
```

```
cancel [ repeat 100 [fd wait 50 bk] ]  ; cancels
launch

when [ color = red ] [fd]
cancel [ color = red ]      ; cancels when statement
```

Though these processes can be referenced without an explicit name or process number, the implementation is far from elegant.

There could also be a language where definition and instantiation of a rule were separated. Rules would be defined like tasks with some type of identifier, such as a name or index. These definitions would exist as separate code blocks, like subroutines. The rules could then be explicitly started and stopped by the programmer.

However, this syntax becomes very bulky and loses much of the elegance of rules. The code becomes nearly as complex as equivalent code written in NQC, or some other task-based language.

4.2 A Mode-Based Language

During this thesis work two prototype compilers have been developed for implementing mode-based languages. Though neither of these compilers are in a distributable form, they did provide a means of exploring different options for compiling a mode-based language.

4.2.1 pbProgrammer

The pbProgrammer (programmable brick programmer) was the first modal compiler. It did not implement a full language; there are no functions, complex expressions or complex conditions. Rather, it was developed as an approximation of modes to use for pilot case studies. This compiler was developed in Java and compiles to byte codes for the standard LEGO firmware version 1.0, based on Kekoa Proudfoot's documentation [29].

The syntax of the language was primarily based on that of Yellow Brick Logo. Modes in pbProgrammer consist of two parts. First, a set of sequential statements and control

structures. Second, a list of rules (equivalent of *when* statements). When the mode is entered, the sequential statements are executed to completion, then all of the rules are activated. That, and the fact that the rules are all mutually exclusive, means that there is no chance of conflict over resources. Aside from those implementation decisions, modes follow the same principles as described earlier in this chapter.

The pbProgrammer environment consists of a tabbed window where each tab is dedicated to a mode of the program (Figure 4.1). At the point of this compiler implementation, modes were being called *behaviors*. It was later discovered that the term was already coined in the AI field (see section 4.6.2).

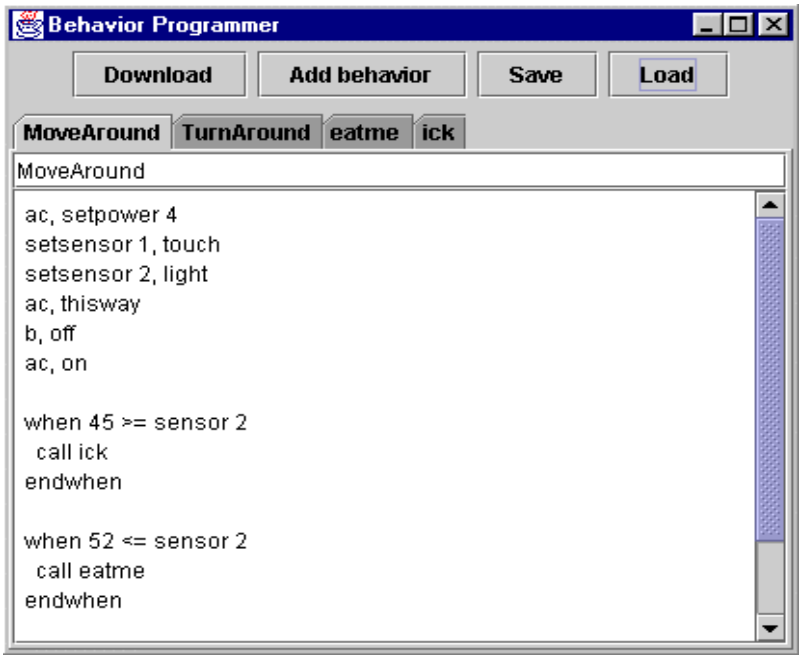

Figure 4.1
pbProgrammer environment

The case studies in Appendix A that make use of modes were implemented using this compiler. For more information on the compiling techniques behind pbProgrammer see Appendix C.

4.2.2 Modal

The second compiler was developed as a tool to explore different methods for compiling modes. This second compiler, written in Prolog, does not download code to the brick. It was designed purely to explore compiler theory. It is not a distributable compiler in and of itself. It is more of a guideline for building a full compiler.

The language that the compiler implements, Modal, is hence more of an abstract language. All examples of mode-based programming in the rest of this paper will be in this abstract Modal language. To learn more about the compiling techniques behind Modal, consult Appendix B.

In this implementation, all of the rules are activated upon calling the mode. The rules do not wait for a sequential series of statements to finish before activating. In addition, the rules within a mode are not mutually exclusive. This does open up the possibility for inter-process conflicts. This issue has not been resolved and is left for future work.

The types of rules available in the Modal language are taken from the Logo Blocks language. There is *when*, *every* and *start*. The *start* rule is the equivalent of the *launch* in Logo Blocks, the rule is fired immediately when the mode is first entered. The *when* and *every* rules operate exactly the same as in YBL or Logo Blocks, with the exception that the rules are only active while their mode is active.

```
when <condition> <operation list>
every <ticks> <operation list>
start <operation list>
```

There has not been much discussion in this paper on general syntax rules, such as choice of grammar symbols, delimiters, declarations, etc. These issues are secondary to the concept of modes itself. The two trial implementations of the mode language have a slightly different syntax, but their significant differences are more in the underlying implementations.

Syntax for existing implementations, as well as the sample code in this text, is relatively simple. It is primarily a mix between Logo and C syntax. The language is

procedural in nature with a very minimal number of syntactic elements (no end of line delimiters, implicit variable declaration, etc.). Modes have a different syntax for encapsulation than rules or procedures. Modes start use a Logo-style *to…end* while rules and procedures use C-style { }. These choices are based on the author's preferences and in no way determine the manner in which the syntax for a mode-based language must be implemented.

4.3 Example 1: Sentry

The concept of modes can be best explained using a simple example. This example will be composed of two problems, each of which can be effectively solved in terms of reactive rules. However, the combination of these two problems causes simple, rule-based approaches to break down.

The problem is as follows. A simple car that follows a line and, when it encounters an obstacle on the path, turns around and follows the line in the direction which it came. One could imagine setting up a course where the robot walks a sentry along a path between two points. It is an assumption that there are no other lines on the floor other than the one being followed. The robot has a single light sensor to view the ground and a single touch sensor that acts as a bumper for the robot.

4.2.1 Problem Decomposition

This problem breaks down rather easily into two reactive programs: following the line and turning around upon collision. The difficulty comes in getting these two sub programs to interface.

4.2.1.1 Following a Line

There are several reactive algorithms for following a line. The algorithm I am choosing to use in this case involves a single infra-red light sensor on the front of the robot pointed towards the ground.

The robot monitors for three states, in the middle of the line (assuming the line is sufficiently thick), off the line, and on the edge of the line. These three states can be easily detected using a simple IR sensor. We will say that the line is black, so the three states register as black, white and gray, respectively. The algorithm for reacting to these three states is as follows:

```
when sensor = black     // if too far on line
  turn right
when sensor = white     // if off line
  turn left
when sensor = gray      // if on edge
  go forward
```

An interesting side note is that the edge of the line that the robot is on determines the direction it is heading.

4.2.1.2 Turn-Around on Collision

Naively, it could be thought that turning completely would be as simple as making a timed turn. Another equally naïve assumption would be to measure the revolutions of the wheels and base the turning on a set number of revolutions. Both of these solutions introduce a fair amount of unreliability. The first is an entirely open loop algorithm that has no reference to the outside world to correct its model. The second is making the dangerous assumption that the revolutions of the wheel exactly measure the movement of the car.

A much more safe algorithm would monitor the line beneath the robot. Let us consider the physical structure of the robot. Since the steering wheels are in the back, turning the robot will swing the light sensors on the front off the line, then back onto it (Figure 4.2).

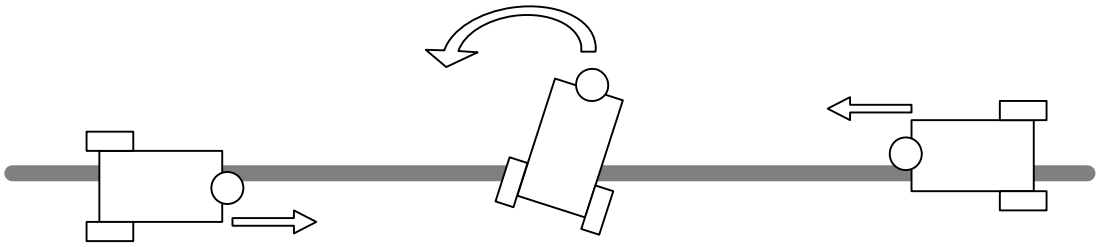

Figure 4.2
Sentry line follower (note, during middle step sees white)

The algorithm would be as follows:

```
turn left until sensor = white
continue turning left until sensor = grey
```

4.2.1.3 Combining the Algorithms

The problem comes when these two scenarios are combined. The algorithm for turning around conflicts with the rules for following a line. Once the robot begins to turn to the left over the black line looking for white space, the line following rule tells the robot that it should turn right because it is over a black line (Figure 4.3).

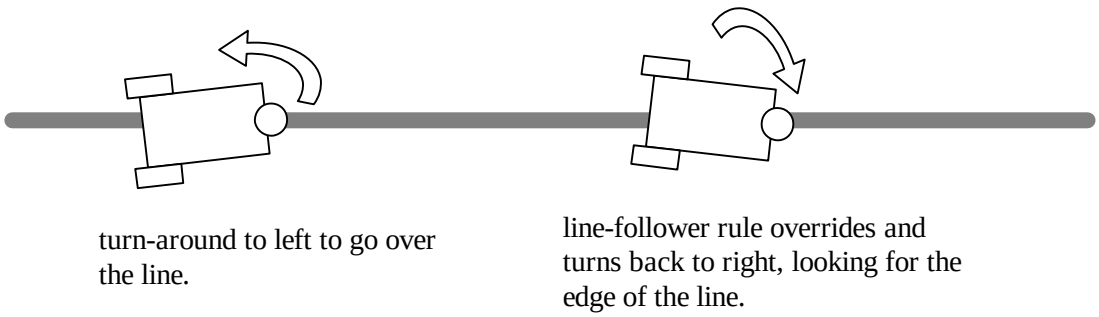

Figure 4.3
Conflicts of turning around & line following

The problem is that this rule language provides no manner to apply scope to the rules. All of the rules are active from the point that they are declared and cannot be affected.

Both RCX Code and Yellow Brick Logo provide this model of rules. In order to "deactivate" a rule in either language, users must program their own mechanisms, such as using a semaphore. In this example, a semaphore would be set to indicate that the robot is turning around. The rules for the line follower algorithm would not only check the light condition, but also check the semaphore to ensure that the robot is not in a turning *mode* of operation.

4.2.2 Mode-Based Approach

The very nature of this solution suggests that there may be a more appropriate syntax for expressing this idea of modality. The semaphore is used to indicate which rules should be ignored and which rules should be followed. Let us say that instead of using semaphores to group these rules, we use a control structure to serve this purpose. All of the rules that are active at a given point are grouped together as a *mode*.

Using this logic, the algorithm for the line follower would be as displayed in Figure 4.4.

```
mode line_follow
  when ( sensor == black ) {
      turn_right()
  }
  when ( sensor == white ) {
      turn_left()
  }
  when ( sensor == grey ) {
      go_forward()
  }
  when ( touch == 1 ) {
      call turn_around    // mode change
  }
end

mode turn_around
  start   {
      start_turn_left()
      until ( sensor == white )
      until ( sensor == grey )
      stop_turn_left()
      call line_follow
  }
end
```

Figure 4.4
Modal program for sentry robot

The *start* rule in the second mode activates when the mode is first entered. Notice that this manner of expressing the solution clearly groups all of the rules that are active at any given point. All of the rules of mode *lineFollow* are active during the entire life of that mode. Once the mode is changed to *turnAround* all of the previous rules are no longer active, allowing the new set of rules to operate without unforeseen interference.

4.2.3 Other Possible Solutions

In this simple example there is another possible language construct that could be used. That is mutual exclusivity. If the rules are all set to be mutually exclusive, then there would be no need for modes. Once the robot collides with an obstacle it can turn around without worrying about the line follower rules interfering. However, this is relying on two factors:

1. The line follower rules can execute their associated operations in a short enough time as to not interfere with the checking for collision.

2. That the second mode only requires one rule.

This is also totally ignoring the possibility of more than two modes. The second mode may call the third, the third calls the fourth and so on. A more elaborate example that makes use of chains of modes will be given later in this chapter.

4.4 Example 2: Can Collector

This example is another conceptually simple robot that becomes complex to program in existing robotics languages. The algorithm is as follows:

1. Perform a random walk: drive forward a set time, turn a random amount, then start again.
2. When the robot encounters a can (signaled by a touch sensor), activate a claw that grabs the can.
3. After grabbing the can, bring the can to the goal by following a gradient. Light sensors can be used to follow a gradient of color on the floor or an IR light source.

This seemingly simple algorithm is rather difficult to implement with existing models of concurrency, however is decomposes rather neatly into modes.

This algorithm will be implemented in parts. First, we will implement the random walk and can grabbing algorithm. Then, we will add the return to goal algorithm to the program.

4.4.1 Search and Grab

Let us start with the implementation of this robot in YBL. This can be created with a *when* statement to monitor the touch sensor for a can and a loop to perform the random walk. Though the when statement can halt itself using the *stop* command, it cannot tell the other process performing the loop to halt. This means a flag has to be used to signal the end of the program (Figure 4.5).

```
global [grabbed]

to hunt
  setgrabbed 0
  watch_can
  random_walk
end

to watch_can
  when [switch1] [
      setgrabbed 1
      grab_ball
      stop
  ]
end

to random_walk
  loop [
      forward
      turn_random
      if grabbed = 1 [stop]       ; jump out of function
  ]
end
```

Figure 4.5
YBL program, for can collection

The problem that happens with this algorithm is that the random walk can interfere with the grabbing mechanism. It will either cause the robot to random walk while the grab is working, or do it once more afterwards. This is because the random walk can only check the *grabbed* variable once for every iteration of moving and turning. The programmer could put tests between every statement, but that would be cumbersome. There is no clean solution to this rather simple problem in YBL.

Logo Blocks as well as RCX code use rules in a similar manner, so they would encounter the same problem as Yellow Brick Logo. The split mechanism lacks the power to shut off other processes, so RoboLab would have the same problem as well. Due to the sequencing of actions, LegoSheets would be dependent on using the global program counter method, which is difficult to read.

The only other viable method is to use tasks, so this next implementation is in NQC. Because of the articulation afforded by tasks, the user is able immediately turn off the random walk (Figure 4.6).

```
task main() {
  SetSensor(SENSOR_1, SENSOR_TOUCH);
  start random_walk;
  start watch_can;
}

task random_walk() {
  while(true) {
    forward();
    turn_random();
  }
}

task watch_can() {
  until(SENSOR_1 == 1);
  stop random_walk;
  grab();
  stop watch_can;
}
```

Figure 4.6
NQC program, for can collection

Proper control of processes is regained here, but only by sacrificing rules as a method of process control. The user is forced to move to a task-based language, which does not support good visibility of concurrent processes.

Now, we will try the same program with a mode-based approach. The robot acts in two modes. First is wandering around randomly while looking for a can. The second is grabbing the can and stopping (Figure 4.7).

```
mode hunting
  start {
      loop {
          forward()
          turn_random()
      }
  }
  when(sensor1 > threshold) {
      call grab_and_stop
  }
end

mode grab_and_stop
  start {
      grab()
      stop()
  }
end
```

Figure 4.7
Modal program, for can collector

Since all of the rules associated with the *hunting* mode are deactivated at the call, there is no opportunity for other actions to interrupt the grab_and_stop.

4.4.2 Return to Goal

An additional benefit of the mode-based approach is that it is easy to add further actions to the robot. After the robot has grabbed a can, it drives back to a home base by following the gradient. With the above Modal program, it would be as simple as creating another mode (Figure 4.8).

```
mode hunting
  start {
      loop {
          forward()
          turn_random()
      }
  }
  when(switch1 == 1) {
      call grab
  }
end

mode grab
```

```
  start {
      grab()
      call return_home
  }
end

mode return_home
  when(left > right) {
      veer_left()
  }
  when(right > left) {
      veer_right()
  }
  when(left >= source or right >= source) {
      stop()      // end program
  }
end
```

Figure 4.8
Modal collect and retrieve program

The same program would become significantly more complex using NQC. One option would be to create a function that is called after the ball is fired to perform the return home. This function would have to use explicit time slicing in order to follow the gradient. However, if the user has to resort to explicit time slicing in order to solve the problem, then the concurrency tools have failed. The other option would be to create more tasks to perform the gradient following (Figure 4.9).

```
task main() {
  SetSensor(SENSOR_1, SENSOR_LIGHT);
  start randomWalk;
  start watch_fired;
}

task random_walk() {
  while(true) {
    forward();
    turn_random();
  }
}

task watch_fired() {
  until(SENSOR_1 > threshold);
  stop randomWalk;
  fire_ball();
  back_up();
  start gradient_left;
```

```
  start gradient_right;
  start find_goal;
  stop watch_fired;
}

task gradient_left() {
  while(true) {
      if(LEFT > RIGHT)
          veer_left();
  }
}

task gradient_right() {
  while(true) {
      if(RIGHT < LEFT)
          veer_right();
  }
}

task find_goal() {
  while(true) {
      if(LEFT >= source or RIGHT >= source)
          StopAllTasks();
  }
}
```

Figure 4.9
Task-based collect and retrieve in NQC

If we continue to try to use concurrency in this case, we end up with a program where the visibility of processes becomes continually obscured. In this case, the user is encouraged by the language to perform explicit time slicing. Explicit time slicing moves users away from reactive programming techniques and encourages open loop algorithms.

The example here is not an amazingly complex robot. A simple tricycle-design robot with a couple light sensors and a touch sensor could perform this algorithm. However, with a very simple design, user can program solutions to perform very interesting problems using the correct language.

4.4.3 Adding to the Algorithm

Given this working algorithm, it would also be easy to add further steps. For instance, after collecting a can, the robot may drop the can, turn around and head back to collect another.

With a task-based language, this would involve more activation and deactivation of multiple tasks. With Modal, this would involve the addition of only one more mode. This new mode could then call the starting mode to form a loop (Figure 4.10).

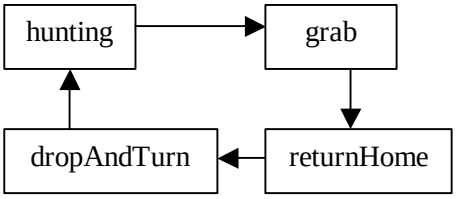

Figure 4.10
Mode-based algorithm for continuous collection

4.5 Evaluation of Modes

Modes are very good for problems that can be described as series of reactive algorithms. This construct allows the user to string reactive algorithms together into a higher-level, sequential structure. Modes were designed for the brick because a large portion of interesting programming problems for simple robotics can be described in terms of sequences of reactive algorithms. Given this domain, modes provide a very nice balance of visibility and articulation.

4.5.1 Articulation

Using imperative rules in a mode-based language allows users to define, start, stop and restart processes. There is, however, a constraint that is added to how users can control these processes. The user is not able to stop just one process at a time, or any subset of the active processes other than the complete set. This means that rules cannot be active across modes.

The same rule can be implemented in more than one mode, but this does not give the same effect as having one rule persistent across modes. For instance, a rule that fires every 30 seconds implemented separately in modes A and B would not give the same effect as one

persistent rule. Since the rules reactivate for each mode change, moving between the modes would result in a stutter in the rule firing.

This sacrifice in articulation was a deliberate decision made in order to provide high visibility. If rules were persistent across modes then it would be more difficult to determine, but reading source code, when they were active.

4.5.2 Process Creation

Process creation here is implicit. This does not mean that the user is unaware of concurrency, but it does mean they do not have the excess baggage of naming processes. The user is allowed to think in terms of rules.

4.5.3 Syntax

The specific tokens used to express modes and rules have been different in the various implementations made during this thesis research. In fact, at this point in time there isn't even a recommended set of tokens to use for future mode-based languages. However, the general concept exists that modes and rules are distinct from sequential constructs such as commands, control structures and functions.

A mode-based program will consist of a series of modes defining the overall structure. Each mode is composed of a set of rules, and the code within each rule is sequential. Rules cannot be within sequential control structures, functions or other rules. Therefore, all of the concurrent constructs (modes and rules) are at the top-level organization of the code. Functions exist outside of modes, but can only be called from within the sequential body of a rule. Again, the syntax is distinct between concurrent structures (modes) and sequential structures (functions).

4.5.4 Conflict Resolution

Though modes provide a syntax that allows programmers to more easily detect points of conflict between multiple processes, they do not inherently resolve conflict over global resources (such as actuators). Currently, the rules for each mode operate in an asynchronous, concurrent manner without conflict resolution. There are possible solutions for the implementation of the mode compiler to allow some primitive conflict resolution.

In the first mode compiler prototype developed, all rules within a mode were mutually exclusive. The user could define precedence of the rules by the order in which they were listed. Though this does provide a handy resolution over conflict, it is a rather limited form of concurrency. This implementation option was abandoned for future mode research. Another solution would be only having rules that affect the same actuators be mutually exclusive. This would require a compiler that can determine the context of every actuator operation. A third solution would be a subsumption type architecture such as that introduced by Rodney Brooks [5].

Further study of modes is necessary to determine whether mutual exclusivity scenarios unnecessarily complicate the model. Are ideas of precedence and subsumption natural to novice programmers? Also, do these methods solve a significant range of problems that cannot already be solved by effectively using modes?

4.5.5 Visibility

Visibility is one of the strongest features of the mode-based languages. The structure of a program is described in terms of modes. At any point of a running program, the robot is in one and only one mode. All of the concurrent processes active when a particular mode is active can be easily seen in the source code by looking at the rules listed within that mode.

4.5.6 Naturalness

Mode-based programming is designed to leverage the naturalness of rules while adding more articulation than afforded in existing rule-based languages. The naturalness of rules is substantiated by statistical studies [27] as well as observation [32]. However there is, as of yet, no such equivalent substantiation for modes.

What can be offered in this thesis is an intuitive argument for how we interact with our environment. We operate differently under the same stimulus depending on the context of the situation. Modes categorize that context. For instance, a person will respond to the same stimulus differently when they are in school as opposed to on a playground, or in a church.

4.6 Related Language Research

The concept of using rules within language constructs that limit their scope is not a new one. There has already been related language development within the artificial intelligence field. Two such languages will be presented in this section: Teleo-Reactive Programming and the Behavior Language. These languages are more flexible than the mode-based language, but are also more complex. It is valuable to look at the similarities.

4.6.1 Teleo-Reactive Programming

Teleo-reactive programs, like mode-based programs, are established on the concept of continuous feedback from the surrounding environment [22]. TR programs, however, compile down to circuitry. In fact, this is one of several languages that is designed to compile to literal circuitry.

In TR languages, a program is written as an ordered set of production rules. These can be thought of as equivalent to *when* rules in the Modal language. Whenever a condition is true, it's correlating operations are performed. These operations can be primitives (like motor controls) or they can call other sets of production rules. An interesting aspect of this language

is that if set of rules B is called from set of rules A, the rules in A are still in effect while B is running. This allows for a hierarchy of operations.

It is at this point that teleo-reactive programming departs from mode-based programming. The visibility of concurrent processes becomes obscured when sets of production rules can be in nested calls. One set of production rules no longer revealed all of the concurrent processes at a given point in time. This does allow for more types of solutions, but the visibility is obscured.

4.6.2 The Behavior Language

Rodney Brooks' Behavior language also allows rules to be group into higher-order structures; in this language they are called *behaviors* [5][6]. Multiple behaviors can be active simultaneously, though one behavior cannot directly activate another. Behaviors are actually activated by a *hormonal* model. Under this model certain amounts of appropriate feedback will automatically activate or deactivate particular behaviors. Behaviors also can communicate with each other using asynchronous message passing.

Even within each behavior the model is more complex. Rules can be nested within each other to enforce an ordering of events. For instance, take the following code sample of rules nested three layers deep:

```
(whenever (received? mess1)
  (whenever (received? mess2)
      (whenever (received? mess3)
          (print "Got 1, 2 and 3 sequentially")
          (done-whenever 1))))
```

The above code-segment only prints the message after receiving messages 1, 2 and 3 in sequence. Since they are stated as rules, there are no restrictions on how long a period occurs between these messages. This could be accomplished using a series of *wait until* statements, but this language uses only one type of control structure, the rule, so the rule must be flexible enough to express many types of control. Notice also the *done-whenever* statement at the end.

This is a break command that is given a zero-indexed integer telling it how far back out to jump. In this example, the program jumps back to the first level of *whenever* statement.

The other level of syntactic control added to rules in the behavior language is that of user controlled mutual exclusivity. Normally, all of the rules at the same level in a behavior are running in parallel and can all be fired at once. However, the user can define sub groups of rules that are mutually exclusive to each other.

```
(exclusive
  (whenever (received? bar) (print "Isolated BAR"))
  (whenever (received? foo)
      (exclusive
          (whenever (received? bar)
              (print "BAR within 2 seconds of FOO"))
          (whenever (delay 2.0) (done-whenever 0)))))
```

The above example emphasizes that this language is well acclimated to programming for events that are partially ordered. However, much of the same effect gained by *exclusive* can be gained by effective use of modes.

The underlying architecture supporting the Behavior language should also be mentioned. This language is built on top of Brooks' *subsumption architecture*. With this architecture, certain processes have precedence with respect to control over global resources. This allows processes of higher precedence to subsume control from other processes.

Some of the complication of this language is due to the fact that is meant for more complex robotics problems. However, some of it is due to having no sequential control structures. The idea of defining mutual exclusivity for subgroups of rules within a mode does seem interesting, but it is likely that it would be used in situations where it would be better to use separate modes. At this point, it is better to keep a simple model for the mode language, than try to achieve the power and flexibility of languages like the Behavior language.

5. Conclusions

A brick language's abstraction of concurrency should maximize the visibility and articulation over processes using a syntax that clearly delineates sequential control flow from concurrency. Tasks and splits do this to an extent, but do not provide a very easy to use model. The rules abstraction provides a powerful metaphor that leverages novice's natural language understanding. Code that makes use of rules with implicit process creation are very easy to understand because concurrency is highly visible.

Modes are introduced as a method of increasing the utility of these rules by allowing the user to activate and deactivate groups of rules. The rules are still implicitly activated and deactivated, and the high visibility of processes is maintained. This is accomplished by programming languages in terms of groups of statically defined groups of rules that are mutually exclusive to each other. With this approach certain types of programs become difficult; such as using rules that are persistent across multiple modes. Though it would be possible to remedy this by removing the mutual exclusivity of modes, this would be at the sacrifice of visibility.

The primary goal of this thesis was not to create a mode-based language. The focus was on the analysis of concurrency in the brick domain. The concept of modes is one that was identified during the case studies as key to brick programming. This thesis merely introduces the concept of modes. There is a great deal of work left to be done in order to determine the validity of mode-based programming (see Future Work).

The primary contribution of this thesis was the introduction of the concurrency features as a metric for evaluating brick languages. This metric was applied to a subset of the existing brick languages, and hopefully it can be used as an evaluation tool for future languages.

6. Future Work

This thesis represents only one step in what will be a series of explorations into the issues of user programming with simple, embedded technologies; such as the brick. During the course of this study, many significant areas of future research have been identified. This chapter will outline these areas of research as well as provide current observations and suggestions of approach.

6.1 Mode-Based Language

The concept of a mode-based language has been introduced in this thesis and some simple case studies have been performed using test compilers. However, to properly validate the usability of modes, extensive case studies with novices will need to be done. The existing compilers demonstrate the concept of modes, but lack debugging, error checking and many other features of a full development language that would allow extensive case studies.

In order to perform more extensive case studies, a more usable compiler and development language will need to be created. This section describes requirements and possible implementation options for implementing a mode-based language.

6.1.1 The Compiler

Considering the amount of work required in order to develop a compiler, it may be wise to construct a modal *precompiler* for the purposes of case studies. This precompiler would generate some other high-level code, such as NQC or YBL instead of op codes. This high-level code would then be sent to the target language's compiler, which would generate op codes and send them to the brick (Figure 6.1).

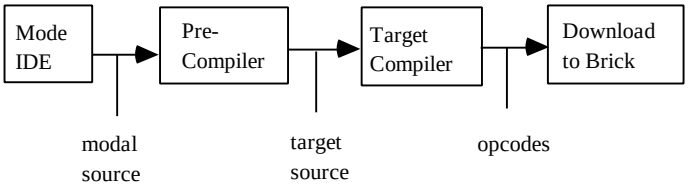

Figure 6.1
Structure for Modal precompiler

This would relieve a great many of the common compiling tasks; such as resolving complex assignments and conditions as well as implementing sequential control structures. In fact, the sequential code within each rule could be the target language. This way, the only work of the precompiler would be to convert modes to their equivalent tasks and provide a mechanism for changing modes.

```
mode  modeName {
  rule 1  {  NQC code  }
  rule 2 { NQC code }
  ...
}
...
```

In one sense, this could be viewed as an extension of the target language (NQC, YBL, etc.).

NQC would be a good candidate for a target language because it is freely distributed and provides full access to the services provided by the LEGO firmware. YBL comes with a different firmware that provides more variable space as well as a larger stack (allowing nested, and even recursive, function calls). However, YBL is dependent on a commercial product Microworld's Logo in order to run. NQC is only dependent on software that comes with the brick.

It would be possible to develop a full compiler, or even a new firmware that better supports modes. However, the investment of time required for such development would not be recommendable until the concept of modes is validated by more extensive case studies.

6.1.2 Development Environment

To the subject of development environments for this domain in general, there is a fair amount of exploration to be done. For this reason, I have pushed this general discussion to the next section. However, a mode-based IDE sufficient for further case studies can be developed independently of the research described in section 6.2.

It would be more beneficial at this point to create a simple environment that provides basic editing and debugging features. More advanced features can be added later on. Ideally, this simple environment would provide the following basic tools:

- An edit window with syntax color-coding.
- A console window for entering immediate commands.
- A watch functionality to monitor the current state of motors, sensors, variables and the active mode.
- Optionally, a template window providing code templates for code blocks, control structures and common commands.

Above all else, this IDE should be simple to use and minimalist. The purpose is to provide a environment that will allow the exploration of the concept of modes.

6.1.3 Extending Modes

However, modes do have obvious limitations as well. First, the user cannot create a program where combinations of rules are dynamically created. Such a language would cause the visibility of modes to break down. Second, rules cannot be persistent across modes.

It could be easily accomplished by allowing multiple modes to be active at the same time. This limitation exists because modes are mutually exclusive; a decision made to keep modes simple. There is a danger with trying to generalize modes to cover all case. In order to gain the extra articulation of rules being active across multiple modes; we could loose the

visibility and ease afforded by modes. This is why these initial explorations into mode-based programming have avoided concurrent modes.

If, in future research, this work were to be taken in a direction allowing concurrent modes, it is recommended that it be done in a manner that still allows visibility of processes by virtue of the structure of the code. It would not be recommendable to go the route of tasks and simply allow the equivalent of a *start* command. This would place the user back at the point of having to trace through code to find the possible permutations of active processes. A more readable language would enforce a hierarchical structure of modes. Modes on the same "level" would be mutually exclusive, but sub-modes could be called and run concurrently with the present mode (Figure 6.2).

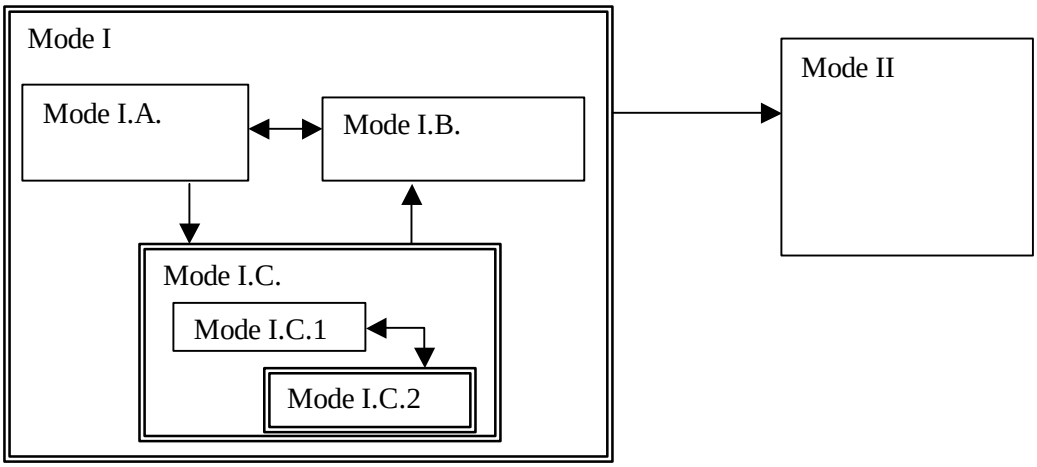

Figure 6.2
An example of hierarchies of mutually exclusive modes
(the double outlines represent active modes)

With statically defined hierarchies of modes, there are only set groups of modes that can be active at the same time. In Figure 4.4, modes I, I.C and I.C.2 are all active at once. This means that all of the rules associated with these modes are currently active. What is more, when mode I.C.2 is active, the user knows modes I and I.C must also be active. All of the sub-modes within a given mode are mutually exclusive (for instance, modes I.A, I.B and I.C).

In addition, when a mode is exited, the sub modes associated with it are exited as well. In regards to Figure 4.4, if Mode II is activated, Modes I, I.C and I.C.2 are all deactivated.

An example of such a language would involve a robot that had an override. For instance, add a time limit to the can collector, and it will shut itself off after 2 minutes regardless of what state it is in. The current mode-based model would not be able to handle this addition very elegantly.

Though a hierarchy of active modes presents some interesting possibilities, it does become more complex and it is not clearly evident how useful it will be. It is possible that some of the complexities of programming in this manner could be mitigated with proper visual syntax, but such exploration is beyond the scope of this work.

6.2 User Interface Investigation

Though this thesis did not evaluate development environments, this is a very important issue to the usability of the brick. In many cases the development environment has as great an effect on the usability as the language itself. This section introduces possible avenues of future research involved with user interface development related to brick IDEs.

The first topic is that of visual, or iconic, languages. The second is the topic of immediate feedback of operating the brick.

6.2.1 Visual Languages

Several of the languages studied in this thesis are at least partially visual in nature; in that they use icons or other such visual queues as part of their syntax. However, this use has been at a fairly low level, and there is the possibility for visual syntax to play an important, high-level role.

6.2.1.1 Current Use

For the most part, the existing iconic languages for the brick do little more than provide a one to one mapping of icons to equivalent textual commands. For a given program, there are typically just as many steps. The conceptual complexity of the program is not reduced.

Languages like RCX Code and Logo Blocks make use of shape to enforce syntactic meaning, therefore keeping the user from making syntactic errors. This is only useful, though at the early stages of learning to program and soon becomes cumbersome. Code templates provide much the same functionality and do not interfere with direct coding.

6.2.1.2 High Level Visualization

The most significant benefit that could be provided by visual programming has not been explored yet in this domain. That is to use visual techniques for maintaining high level structures of code. The most useful visual metaphors provided in RCX Code and Robolab are where concurrency is represented spatially. Coding of individual rules, functions, etc. can be done effectively with textual code, but the parallelism between rules and the changes between modes might be expressed well using visual metaphors.

For instance, the contents of each rule could be expressed as sequential text. These rules are then combined visually to express modes. These modes are connected to each other with transitions in a diagram that looks much like a finite state machine.

Figure 6.3 demonstrates a possible appearance for such a language. Note that code blocks, such as rules are collapsible. This is known as recursive containment and was pioneered by the Boxer programming language [7].

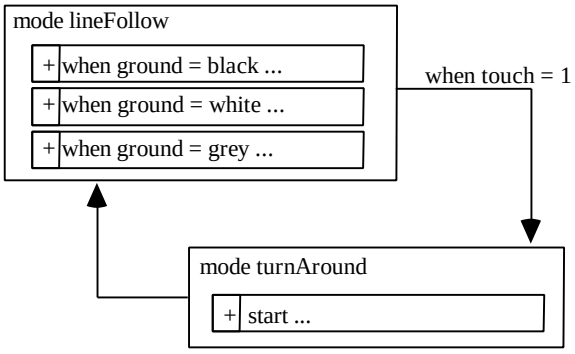

Figure 6.3
Graphic mode-based language

6.2.2 Immediate Feedback

One of the most challenging conceptual impediments to programming the brick is the separation of the physical artifact (the robot) from the code on the desktop PC that dictates its behavior. A program must be written and compiled on a PC, then downloaded and finally run on the brick. This model does not allow for immediate response, and therefor, makes experimentation difficult.

Several of the development environments allow for some form of direct brick manipulation in order for users to test physical design concepts and, in the case of YBL, sub-components of algorithms. As long as the brick is within a direct line of communication with the IR tower, the programmer can give the robot direct commands. Most of these environments allow the user to check sensor values, and affect the state of the actuators. YBL allows the user to call downloaded subroutines. There are several general types of direct user manipulation available in brick IDEs.

6.2.2.1 Console

The idea of a console window originally came from Logo. The user is provided with an area to define procedures as well as a console window that allow them to type in individual

commands and procedure calls. These commands are executed immediately after entering a line.

For instance, with Logo the user would type in "forward 20" and the graphic turtle would instantly move ahead 20 pixels from its current location and orientation. In fact, the user can type in an arbitrary line of code with control structures, variable operations etc. Hitting the carriage return triggers the entered text to be executed. In addition, this console window can be used to page current values of variables. The user can request the current value of a variable, and it will be printed on the next line of the console window.

This idea was carried into Micro World's Logo and, hence, into Yellow Brick Logo. The user enters a command into the console window and a message is sent to the brick, through the IR tower, to perform that instruction. This provides a wonderful environment for testing and interacting with code. The state of sensors can be paged in the same way as variables.

The console is the most flexible of all of the direct user manipulation environments provided. However, the other approaches need to be examined, because they may be better for the types of debugging necessary for working with the brick.

6.2.2.2 Activated Code

This second model removes the necessity for an extra window. However, it trades this for an extra modality of use. The area where the program is entered is the same area where code can be interactively tested. The user selects an instruction, or series of instructions, and tells the system to execute this piece of code. This method is used in Bot-Kit, a Smalltalk environment for developing brick. The user selects a segment of code, then strikes a command-key to cause the code to execute.

A slight variation of this model exists in RCX Code 1.5. A tool menu is used to change the mode of the cursor to "run mode". The icon for the cursor changes to indicate the

mode. When the cursor is in this mode, the user can click on a single instruction and it is executed on the brick. RCX Code does not allow the user to select subroutines or multiple instructions, however this type of instruction activation does not necessarily preclude these options.

6.2.2.3 Brick Mirror

Several development environments have provided some type of a mirror of the current state of the brick ports. The interface consists of a graphic representation of the brick along with digital values for all of the current sensor and actuator port states as well as well as controls for manipulating the actuator port states. This type of interface has been implemented in LegoSheets, LEGO Engineer, and RCX Code 1.0.

The benefit of this type of interface is that it allows the user to view the state of all of the sensors at once. Whereas the console window requires the user to page the sensor values. It also allows for more interactive work with the motor ports. However, it provides little utility for debugging code.

6.2.2.4 High-Level Controls

Certain interfaces, like the RCX Control Center (an IDE for NQC) provide more abstract, remote-control interfaces that assume certain configurations for the robot. These interfaces require that the user defines which motors are playing which roles.

For instance, say the user builds a tricycle-design robot with two rear motors and a pivot wheel in front. In RCX Control Center (RCC), the user specifies the generic type of vehicle as well as which motors are operating the left and right hand side of the car. Given this information, the IDE provides a remote control-like interface where the user can enter high level commands like "turn left" or "go forward". These high level commands are interpreted by the system as activating the specified motors in predefined patterns.

These controls could be used to debug the physical designs of the robots or for recording a sequence of timed movements. The difficulty is that they are very specific to the design of the robot. A series of brick-robotic configurations would have to be identified and provided for the user to make use of this type of mechanism.

This type of interface would provide little use for debugging algorithms and would inherently only be applicable to a narrow range of robotics applications. It could be an augmentation of other immediate feedback debugging tools, but not a replacement.

6.3 Further Language Analysis

In order to explore the issue of concurrency other important language issues were not discussed in this work. During the course of conducting case studies two other significant programming language issues were identified. These are sensor monitoring and actuator control.

6.3.1 Sensor Interaction

There are two significant issues related to sensor interaction that have been identified. First, is whether sensor-related control structures are designed to monitor for sensor values in a differential versus discrete manner. Second, is the syntactic expression for expressing and testing ranges of sensor values.

6.3.1.1 Differential versus Discrete

Most of the existing brick programming languages monitor sensor values in a discrete manner. That is, the commands are designed to test the sensor value as sampled at some point in time. However, Robolab provides unique commands that monitor for changes (differentials) in the sensor values. For instance, one such command is "wait until the light sensor reading has dropped by X percent".

The benefit of this approach is that the robots are more robust to changes in light than ones programmed using discrete control structures and sampled constants. This is because there is an implicit sampling, and calibration, that occurs with the control structures that monitor differentials.

It is possible this technique could be very useful for any sensors that deal with ranges of values such as light, temperature and rotation sensors.

6.3.1.2 Expressing Ranges

When monitoring sensor values, ranges are often times more important than discrete values. Part of this is due to the variations in the readings of the physical sensors themselves. For instance, even provided steady lighting conditions a given IR sensor will oscillate within a range of values. It is useful to be able to regard a certain range as an abstract value (i.e. "green") and be able to determine whether the current sensor reading is within that range.

RCX Code' visual programming language provides double-ended sliders for selecting ranges of values for most of its sensor tests. In addition, the LEGO Scripting Language provides a textual notation for expressing ranges of values. In both these cases, ranges are expressed within control structures. This does little than provide a shorthand for complex conditions. A possible further exploration would be to allow ranges to be defined and named. These ranges could then be used in control structures much like variables. For instance,

```
green = 54 to 58
...
if sensor1 = green then  ...
```

6.3.2 Actuator Control

One of the most common conceptual difficulties of the children working with robotics noted during our work with children was that of actuator control. This difficulty was with understanding the *persistent state* of the motors. When the motors of the robot were set to a

specific state (on/off, forwards/backwards, etc.) they would remain in that state indefinitely until acted up on again. In addition, the state of the motor is the cumulative result of many different function calls affecting different aspects of the state.

Oftentimes, children would become disoriented as to what the state of the motors should be at a given point in their code. A possible solution would be a single motor control that defined the total state of a motor, or motors, at a given point (power, direction, etc.). This would be a language construct specifically for younger, novice programmers being introduced to robotics.

6.4 Radio Brick

It is evident that the range of possible uses of the brick could be greatly increased by creating a stronger communication mechanism. Due to the reliance on line-of-sight that is required by IR communication, a radio-based protocol would be much better option. Such a protocol would allow for the following:

- A full debugging environment.
- The brick to act as a proxy for a desktop PC.
- Robust inter-brick communication.

Each of these possible uses are discussed in more detail below, as well as possible implementation options for the radio brick.

6.4.1 Debugging Environment

All of the immediate feedback interfaces discussed earlier are reliant on the brick having an open line of communication with the IR tower. Therefor, immediate feedback typically involves the user holding the robot in front of the IR tower while entering commands.

Since many brick projects are for mobile robots, this provides only a very superficial testing environment. Ideally, the robot should be tested in the conditions it was meant to run

within. A radio communication protocol would allow the robot to operate away from the computer while the programmer can monitor the internal state from the PC.

In fact, it would not be unreasonable for the user to be able to change the code of his/her program while the robot is running. Code traces, variable watches, sensor and actuator monitoring all become much more useful with such an environment. These can all act to close the gap between the physical robot and the abstract computer code.

6.4.2 Radio Brick as a Proxy

Due to the limited memory space of the brick, complex programs such as artificial intelligence planning algorithms cannot be used. However, radio communication would allow the brick to acts as a proxy for a desktop PC, which would do all of the processing. The brick simply sends updated sensor information to the PC, and the PC sends actuator commands to the brick. With this model, complex AI algorithms can be run using brick robots for *physical simulation* of more complex robots.

Currently, a great deal of the simulation work for robotics is done completely on the computer. This removes much of the unreliability that comes with physical systems. By allowing physical simulations of systems, researchers and undergraduate students would have the opportunity to work with these issues much more readily.

In addition, this would make multiple-agent systems much more feasible. Since setting up multiple agent systems with physical robots is often cost-prohibitive, an inexpensive (though simplistic) option would be a good alternative. Most robots commercially available are of a relatively static design. The brick is already designed to accommodate a wide variety of physical designs. Using the flexibility of LEGO Technic components along with the processing power of desktop PCs, the radio brick has the potential to be an incredible tool for prototyping advanced robotics projects.

6.4.3 Inter-Brick Communication

Inter-brick communication is limited by the line of site required by the IR port of the brick. With mobile robots, maintaining an open line of communication becomes incredibly difficult. As a result, examples of communicating robots are very simplistic. Typically these are limited to algorithms involving a "handshake" between the two agents that are set pointing to each other.

Radio communication would remove this limitation. In doing so, it becomes possible to create systems with groups of interacting robots. This would open up a range of planning, strategizing and self-organizing systems that could be implemented with the brick. The system could provide both broadcast and point to point communication; allowing for hierarchies of control as well as democratic models.

In addition, models involving localized communication could be created. By controlling the power of the radio signal, robots could communicate on long range or short range. Users could create models dependent on only robots within a certain distance of each other communicating. This would allow models of robots using aggregation.

6.4.4 Implementation of Radio Brick

The possible benefits of the radio brick for both K-12 audiences as well as research has made it an immediate subject of research interest. It is too early at this point in time to state the exact specifications of a radio brick implementation. However, an ideal radio brick would:

- provide networking with brick identification and error checking
- allow a large number of bricks within the same network
- accommodate new bricks entered to the network at arbitrary points
- allow multiple PCs communicating with multiple bricks within the system

The radio brick does *not* have to provide:

- High bandwidth communication

- Very large networks
- Transmission over distances greater than 30 or so feet

It is important to take these considerations into account in future design of the radio brick. The greatest utility of the brick is that it is a very inexpensive platform for robot development. It is not meant to be a tool for arbitrarily complex robotics. That ceiling of use keeps the brick inexpensive and still very useful for many types of work.

REFERENCES

1. Baum, D. http://www.enteract.com/~dbaum/nqc/

2. Baum, D. NQC Programmer's Guide version 2.0.

3. Bower, A. http://www.object-arts.com/Bower/Bot-Kit/

4. Braitenberg, V. Vehicles: Experiments in Synthetic Psychology, MIT Press 1984.

5. Brooks, R.A. "The Behavior Language; User's Guide" A.I. Memo 1227, MIT, April, 1990.

6. Brooks, R.A. "Integrated Languages Based on Behaviors" SIGART Bulletin, Vol. 2, No. 4, August 1991, pp. 46--50.

7. diSessa, A.A. and Abelson. "Boxer: A Reconstructable Computational Medium". Communications of the ACM 29, 9, 1986, pp. 859-868.

8. Erwin, B. et. al. "Middle School Engineering with LEGO and LabVIEW" Presented at the LabVIEW in Education workshop, June 1998, MIT, and NIWeek August 1998.

9. Gindling, J. et. al. "LegoSheets: A Rule-Based Programming, Simulation and Manipulation Language for the LEGO Programmable Brick" Proceeding of Visual Languages, Darmstadt, Germany, IEEE Computer Society Press, 1995, pp. 172-179.

10. Hempel, R. http://www.hempeldesigngroup.com/lego/pbFORTH/

11. Hogg, D.W., Martin, F., Resnick, M. "Braitenberg Creatures" Epistemology and Learning Memo #13, MIT Media Laboratory, Cambridge, MA, 1991.

12. Knudson, J. B. The Unofficial Guide to LEGO® Mindstorms™ Robots O'Reilly & Associates, Inc. 1999.

13. "LegoSheets User Reference Manual". University of Colorado, Boulder. April 28, 1995.

14. Lewis, C., et. al. "Adapting User Interface Design Methods to the Design of Educational Activities", SIGCHI Proceedings, 1998.

15. Martin, F. Circuits to Control: Learning Engineering by Designing LEGO Robots, PhD dissertation, MIT Media Laboratory, 1994.

16. Martin, F. "Ideal and Real Languages: A Study of Notions of Control in Undergraduates Who Design Robots", Constructionism in Practice, National Educational Computing Conference, 1994.

17. Martin, F. "Kids learning Engineering Science Using LEGO and the Programmable Brick", AERA 1996 Annual Meeting. April 8-12. New York, NY.

18. Martin, F., Resnick, M. "LEGO/Logo and Electronic Bricks: Creating Scienceland for Children" Advanced Educational Technologies for Mathematics and Science (David L. Ferguson, ed.), Springer-Verlag, Berlin Heidelberg, 1993.

19. Maynard, R. C. http://www.netway.com/~rmaynard/html/brainstorm.htm

20. Munafo, R. http://www.mrob.com/robot/

21. Najork, M.A., Programming In Three Dimensions. Doctoral Thesis. University of Illinois.

22. Nilsson, N.J. "Teleo-Reactive Programs for Agent Control" Journal of Artificial Intelligence Research, vol. 1, 1994, 139-158.

23. Noga, M. http://www.noga.de/legOS/

24. Ostroff, J.S. "Composition and Refinement of Discrete Real-Time Languages" ACM Transactions on Software Engineering and Methodology, vol. 8, no. 1, January 1999, pp. 1-48.

25. Pane, J.F. "A Programming Language for Children that is Designed for Usability". presented at the 7th Workshop on Empirical Studies of Programmers: Graduate Student Workshop, Alexandria, VA, October 24, 1997.

26. Pane, J.F., Myers, B. A., "Usability Issues in the Design of Novice Programming Languages" Carnegie Mellon University, School of Computer Science Technical Report CMU-CS-96-132, Pittsburgh, PA, August 1996.

27. Pane, J.F., Ratnamahatana, C., Myers, B. A. "Studying the Language and Structure in Non-Programmer's Solutions to Programming Problems" *International Journal of Human-Computer Studies*, to appear, 2000.

28. Papert, S. Minstorms: Children, Computers and Powerful Ideas, Basic Books, NY, 1980.

29. Proudfoot, K. http://graphics.stanford.edu/~kekoa/rcx/

30. Rader, C., Brand, C., Lewis, C. "Degrees of Comprehension: Children's Understanding of a Visual Programming Language" SIGCHI, March 22-27, 1997.

31. Repenning, A. Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Languages. PhD dissertation, University of Colorado, 1993.

32. Resnick, M. "MultiLogo: a Study of Children and Concurrent Programming" Interactive Learning Languages, vol. 1, no. 3. 1990.

33. Resnick, M., Bruckman, A., and Martin, F. "Pianos Not Stereos: Creating Computational Construction Kits" Interactions, vol. 3, no. 6 (September/October 1996).

34. Resnick, M., Martin, F., Sargent, R., and Silverman, B. "Programmable Bricks: Toys to Think With" IBM Languages Journal, vol. 35, no. 3-4, pp. 443-452.

35. Van Wagner, D. R. http://alumni.cse.ucsc.edu/~davevw/onscreen/

36. Wick, A., Kilpsch, Wagner, M. http://www.cs.indiana.edu/~mtwagner/legoscheme/

37. http://el.www.media.mit.edu/groups/el/projects/programmable-brick

APPENDIX A: ROBOT DOCUMENTS

During the course of this thesis, undergraduate students of the Software Engineering Lab implemented case studies for the brick. For each case study, a robot was constructed to perform a simple task. Then, the robot was programmed in a series of different languages to accomplish the same task. Finally, a document was written for each robot describing the different program implementations. A full listing of these "Robot Documents" is available at the following URL:

http://www.umcs.maine.edu/~pbrick

Of particular pertinence to this thesis are three case studies, where one of the languages used was a mode-based language called pbProgrammer (scooper, tram and line follower). This language was not used in all of the case studies because it was in the process of being developed during some of the case study work. Note, at the time that these papers were being written, *modes* were being referred to as *behaviors*. When it was discovered that the term *behavior* was already coined by Rodney Brooks [5] the name was changed to modes.

APPENDIX B: COMPILING MODES

This appendix briefly discusses two mode-based compilers that were implemented during the course of this thesis. The discussion here is strictly concerned with the issue of compiling a modes-based language to a target code that is task-based. Though general compiler issues, such as compiling expressions, are ignored here this appendix does make use of concepts from compiler theory.

The most significant difficulty in implementing a full compiler for a mode-based language lies in the limitations of the currently available firmware, which does not count on users making extensive use of concurrency. The LEGO® firmware only permits 10 tasks to the user. If the compiler allocated on rule per task, users of the language would become very quickly limited in the types of programs they could make. Therefor, the primary compiling concern was overcoming this limitation in number of tasks.

B.1 pbProgrammer

The pbProgrammer represents a "quick and dirty" method to implementing modes. In fact, it does not implement a true mode-based language in that not all of the rules within a mode become active at the point that the mode is entered. First, a startup set of commands is executed, then the rules are activated. This compromise was made for two reasons.

1. To easily achieve conflict resolution by having rules separated from the initial, sequential algorithm.
2. To simplify the compiler implementation.

Once the initial set of commands is executed, all of the rules are activated. The rules are mutually exclusive, creating an inherent conflict resolution.

The implementation of the compiler is fairly simple. A single task is dedicated to each mode. The sequential part of the mode is interpreted directly to byte codes. The rules, are implemented essentially as a case statement within a loop that comes after the sequential part of the mode (Figure B.1).

```
mode thisMode
     ab, on
     c, on

     when sensor1 > 30
         a, off
     endWhen

     when sensor2 = 1
         b, setpower 2
     endWhen

     when sensor2 < 20
         c, off
         wait 20
         c, on
     endWhen

endMode
```

translates to

```
task thisMode {
     ab, on
     c, on

     loop {
         if sensor1 > 30
         {   a, off
         }
         if sensor2 = 1
         {   b, setpower 2
         }
         if sensor2 < 20
         {   c, off
             wait 20
             c, on
         }
     }
}
```

Figure B.1
Mode to task conversion
(with mutually-exclusive rules)

The actual target is a byte code, but the pseudo code on the right gets the point across better. Changing to a different mode is accomplished by simply starting the associated task and ending the current task. There is nothing particular about tasks in this case that makes this necessary. The entire program could be implemented in one task. All the modes could be compiled back to back in one block. Changing a mode would be accomplished by performing a jump to the appropriate line.

B.2 Modal

There were a few issues with the pbProgrammer that necessitated further compiler exploration. First, there were many deficiencies periphery to modes: lack of complex expressions and conditions, function support, etc. Second, it was questionable whether the rules within a mode should be limited as they were by the mutual exclusivity.

For this reason, a new mode-based language was implemented that made a more full implementation of modes. This language was called Modal.

The following is an excerpt from the documentation of the Modal compiler. Much of the original documentation was concerned with details of either the target language or Prolog – the language the compiler was written in. As much of that was left out as possible in order to give a generic view for compiling a modes-based language to a task-based language.

B.2.1. Compiler Structure

The compiler structure is composed of five components. Most of these are familiar to from the general description of compilers above. The reader gets a string of characters from a text file, the tokenizer creates a symbol list, the parser creates an abstract syntax tree, and the translator generates code based on that Abstract Syntax Tree (Figure B.2).

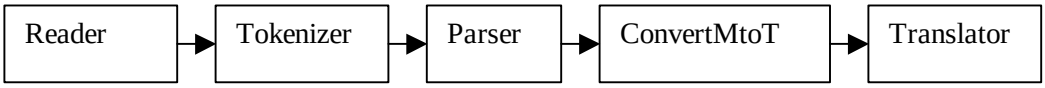

Figure B.2
Structure of Modal compiler

An extra step was added to the process because of limitations in the target byte code interpreter that the translator is generating code for. Due to these limitations, there must be two passes on the AST. One to build it in terms of the logical structure of modes, and one to restructure the tree in terms of the implementation structure of tasks.

The target byte code interpreter knows nothing of modes, but does have allow a static set of up to 10 predefined tasks. These tasks are run concurrently and can be started and stopped by control of the programmer.

When a mode is active, all of its rules are monitored concurrently. One way to implement this would be to allocate a task to each rule. When the mode is changed, all the tasks associated with rules of the previous mode are turned off. In addition, all tasks associated with the new mode are started. The problem with this implementation is that the number of modes and rules becomes very limited. The total number of rules across all modes is limited to 10. This method is wasteful in that, with a multi-mode system, it would be impossible for all of the tasks to be used at a given time. They are simply sitting idle waiting for their mode to be activated.

A second way to implement modes involves using a many-to-one mapping of rules to tasks. Each task contains one rule from each of the possible modes. That way, when the mode is changed, each task simply switches which rule that it is implementing. With this technique compiling, tasks cross-cut the different modes (Figure B.3).

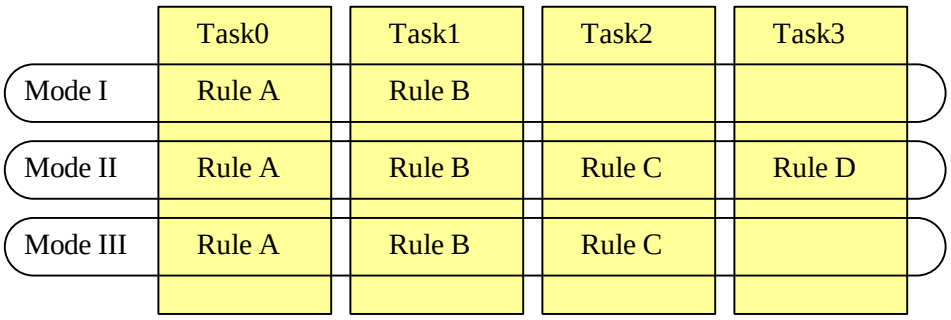

Figure B.3
Relationship of modes, rules and tasks

The difficulty with this technique is that it cannot be done with a single pass compiler. The parser must parse all of the modes before even the first task can be generated. For this reason, a second pass on the abstract syntax tree was made. This pass was called "Convert

Modes to Tasks", or "convertMtoT". The rules needed to be changed from being grouped by modes to be grouped by tasks (Figure B.4).

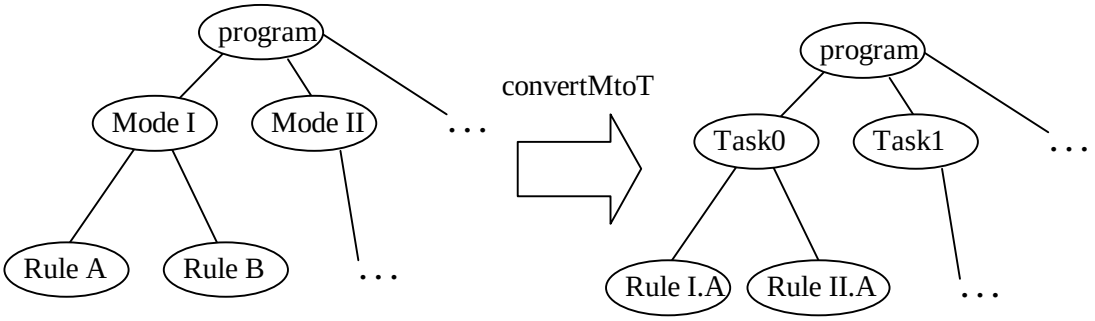

Figure B.4
AST manipulation for mode-based to task-based code

BIOGRAPHY OF THE AUTHOR

Gilliad E. Munden was born in Elizabeth City, North Carolina on October 25, 1975. He was raised in Rockland, Maine and graduated Valedictorian of his class from Rockland District High School in 1994. After high school, Gilliad attended the University of Maine earning his Bachelor of Arts in Computer Science and a minor in Business Administration. During his undergraduate degree he worked with ASAP Media Services, a student-run multimedia group, for four years and as a result, he joined the New Media Committee. Gilliad remained an active member of this committee for the remainder of his time at the University of Maine, helping to develop curriculum for the emerging New Media major. He graduated head of his class in the Computer Science Department in 1998.

He then entered the Master's program of the University of Maine Computer Science department. During his first year he developed software for the Instructional Technologies department on campus. In the second year, Gilliad lead the programmable brick research project at the University of Maine. It was the first year of this project, which was formed as a collaboration with Distinguished Visiting Professor Dr. Seymour Papert. After receiving his degree, Gilliad will be joining Stroudwater Technologies, a custom software development firm, to begin his career in software design. Gilliad Munden is a candidate for the Master of Science degree in Computer Science from The University of Maine in August, 2000.