From SCADE to Lego Mindstorms

David White

Supervisor: Dr. Gerald Luettgen

3rd Year Project
Department of Computer Science
University Of York

Word Count: 22543, counted by WinEdt, all appendices excluded
70 Pages

March 18, 2004

Abstract

The synchronous approach to designing safety critical real-time systems allows the notion of physical time to be replaced with an ordering among events. One such language of this type is Lustre which allows control to be expressed through dataflow equations. This language is used as the basis for an industry-leading tool called SCADE. SCADE provides a graphical environment for expressing dataflow equations as well as the state based formalisms found in other synchronous languages such as Esterel. In addition, C code can be automatically generated from designs, and verification facilities exist for checking designs for both expected and unexpected behaviour.

An implementation platform for SCADE designs would be a valuable tool to aid in the teaching of synchronous reactive systems. The implementation platform this project will assess is the Lego Mindstorms architecture, which combines the capabilities of Lego as a construction set with an embedded computer system called the RCX. When the RCX is used with the open source firmware BrickOS, it can execute programs written in C. This project will demonstrate a way of translating a SCADE design into C code that can be executed on the RCX. Furthermore, two complex robots will be built and programmed highlighting important features of SCADE and exploiting Lego Mindstorms sensory and actuatory potential.

Contents

List of Figures

Chapter 1

Introduction

Today, and indeed for the last 25 years, we entrust computers to take control of safety critical systems. Safety critical systems are described as:

"A computer, electronic or electromechanical system whose failure may cause injury or death to human beings." - The Free On-line Dictionary of Computing

This definition entails the most important property of safety critical systems: they must function correctly. Nevertheless, there have been a number of examples of such systems failing and with predictably catastrophic results. Therac-25 was a machine for delivering radiation therapies and a software bug made it very easy for the machine to deliver 100 times more than the correct dose. Due to this, six patients received massive overdoses [25].

"In 1981, a software error caused a stationary robot to move suddenly and with impressive speed to the edge of its operational area. A nearby worker was crushed to death." - Real-Time Systems and Programming Languages, Burns and Wellings

With these grave accounts of such incidents, the liability falls on the engineers who designed the system. Therefore, it is their responsibility to ensure that the systems produced are safe. This can be addressed in two ways that are not mutually exclusive: either provisions can be put in place so that when a failure does occur it is handled gracefully or the system can undergo a rigourous mathematical testing which can prove the system will not display any unexpected behaviour. Obviously the latter is a more desirable solution and an approach known as the synchronous paradigm can provide this.

SCADE [15] is a state-of-the-art design tool for specifying reactive systems using the synchronous approach. It provides a way to express dataflow equations graphically using a well-defined formalism based on the tool's underlying language; Lustre [1]. Furthermore, designs produced using SCADE may be simulated and formally checked for desired and undesired behaviour. Code can then be automatically generated in a high level language (such as C or ADA) and compiled to an implementation architecture.

Lego Mindstorms [16] is the implementation architecture that will be used for this project. The Lego "brick" provides a small embedded system (called the RCX) that can take control of three sensors and three actuators. In addition,

using the open source firmware, BrickOS [20], C programs may be compiled to the RCX's architecture and executed. Hence, high-level designs produced in SCADE can be passed through a compilation chain and eventually run on the Lego Mindstorms platform.

This compilation chain has not been developed yet; however, there is some preliminary work that allows designs produced using Lustre, to be compiled for the BrickOS platform. Therefore, an aim of this project is to investigate compiling methods for Lustre and SCADE designs so they can be executed under BrickOS. The end goal of this is to develop the Lego Mindstorms platform as a teaching aid for reactive systems. For this reason, the interface to the compilation process must be simplified as much as possible. An investigation into communication between multiple RCX's will also be conducted since one RCX by itself is quite limited in its input/output capabilities. The further aims of the project are to build two robots: a brick sorter and a line follower with obstacle avoidance. These will help to illustrate synchronous reactive systems and further the research into using this system as a teaching aid.

The employed methodology for this project is the following: first the compilation options for SCADE will be investigated since only designs specified in Lustre can currently be compiled to the BrickOS platform. Once a SCADE compilation route has been designed and implemented, the two robots will be built, simulated (inside SCADE) and finally implemented using Lego Mindstorms.

The remaining report is structured accordingly. Chapter 2 contains a literature review exploring all the relevant topics that apply to this project. This includes: reactive systems, Lustre, SCADE, Lego Mindstorms and BrickOS. A tiny, but complete development will be shown at the end to clarify and consolidate the topics discussed in the literature review. Chapter 3 will analyse the previous work conducted on compiling a Lustre design to BrickOS compatible code before discussing the various options available for performing the same operation on a SCADE design. This chapter will also investigate the aforementioned communication between multiple bricks and the impact this has on the synchronous approach. Chapters 4 and 5 will describe the two robots constructed: a brick sorter and line follower with obstacle avoidance, respectively. The chapters detail the problem analysis, design, implementation, testing and evaluation for each robot. Chapter 6 concludes the project and discusses some further work that could be carried out in the field.

Chapter 2

Literature Review

2.1 Reactive Systems and the Synchronous Approach

Reactive systems were first introduced in [11, 12]. Benveniste and Berry [2] make the distinction between real-time and reactive systems by defining reactive as "a system that maintains a permanent interaction with its environment" where as a real-time system is in addition "subject to externally defined timing constraints". The definition given by Halbwachs [6] helps to further define reactive systems: "Reactive systems are computer systems that continuously react to their environment at a speed determined by this environment". Halbwachs introduces this definition to distinguish reactive systems from transformational systems (systems where the inputs are available at the beginning and an output is available on termination) and interactive systems (systems that continually interact with an environment, but at their own rate).

A reactive system has the following main characteristics [6]: firstly, they involve a large amount of concurrency. That is, a large amount of outputs need to be produced determined by a large amount of inputs, possibly all at the same time. Thus it is convenient to think of this in terms of interacting tasks. This characteristic points towards the systems often being implemented in terms of parallel components. Secondly, they are subject to strict time requirements, which include the frequency at which inputs should be sampled and the delay from a stimulus to a reaction being produced. Uses for reactive systems include: industrial process control systems, transportation control / supervision systems and signal processing systems. The areas of use imply the most important characteristic: reliability and dependability, since most of the systems implemented using this model will be highly safety critical.

Halbwachs [6] outlines some different approaches to building reactive systems: communicating finite state automata, petri net based models, task based models and communicating processes. However, none of these approaches will be explored any further since this project is based around the synchronous approach. The synchronous paradigm raises the level of abstraction to a state where a program can be considered as responding instantaneously from its inputs. That is, outputs are produced *synchronously* with their inputs [2]. This has a profound impact on the idea of time: physical time is replaced by the

notion of an order among events. In turn, this has implications for proving properties of the program, by removing the need to consider physical time the complexity of proofs are greatly reduced. However, for the synchronous paradigm to be used, the synchronous hypothesis must be upheld: that the program can react quickly enough to perceive all external events in suitable order - [6]. This means that the maximum time to compute outputs from inputs must be less than the granularity of abstract time being considered. Thus for the synchronous hypothesis to be correctly applied, its maximum possible reaction time must be less than the rate at which inputs have to be sampled.

2.2 Overview of Synchronous Formalisms and Languages

There are two main formalisms within the synchronous paradigm: dataflow and state based. The dataflow model [8, 9] (otherwise known as multiple clocked recurrent systems) works on the idea that outputs are defined as equations based on the inputs. It contains no implicit notion of state. Incoming data is simply transformed in a certain way and then provided at the outputs. Dataflow is normally thought of being asynchronous however, if each operator is considered to take zero time the system can be considered synchronous. This lends itself to the declarative programming style and the two languages based on it, Lustre (section 2.3) and Signal [6], are both declarative.

State based formalisms, on the other hand, use an automaton as a means of control with changes occurring when transitions are followed to different states. This is the idea behind the imperative language Esterel [10] which uses a front-end called SyncCharts to specify the behaviour of the program. Obviously, there is an implicit notion of state in this model. Hence the two models are very different and each is targeted to provide a solution to a different set of problems. Dataflow for when data must be continuously monitored, transformed and reacted to and state based when the system must follow some control flow pattern.

Recently, advances in combining the two formalisms have been made. SCADE dataflow models can have Safe State Machines [29] (an improvement on SyncCharts) embedded in them. A full discussion of this is left to the section on SCADE (2.4).

2.3 Lustre

2.3.1 Introduction

Lustre [1] is a language based on the dataflow model that adheres to the synchronous paradigm. Since dataflow modelling is normally in the domain of control theorists it is not a language designed for computer scientists. Providing the design problem fits well into the dataflow model, there are many advantages to be gained by using such a language for design. Firstly, the dataflow model is inherently parallel. Moreover, it is a fine grained parallelism, meaning as soon as an operator is provided with inputs its output can be computed. Thus, the only synchronisation constraints come from dependencies between the data.

Secondly, the language contains no side-effects meaning that it is mathematically cleaner. This in turn implies, once again, that the complexity of program proofs are significantly reduced. Furthermore, the dataflow model supports hierarchical decomposition. This results in solutions being more comprehensible if suitable abstraction is used to assemble operators into logical groups. Lastly, it allows the combination of textual and graphical notations (although there is no standard graphical notation). This is desirable since some subsystems are best described using a textual notation and some graphically.

2.3.2 Language Overview

A short overview of the language of Lustre will now be presented. This is adapted from [1, 3, 4, 5, 6, 7] - for a detailed explanation of the language one of these references should be consulted.

Lustre is a language based on clocks, the base clock is the finest division of time perceivable inside the program. Slower clocks can then be built on top using clock altering operators. Any variable or expression is defined by a flow, this is made up from a sequence of values (which may be infinite) and a clock, which identifies its value (a member of the sequence) at a certain instant. Constant values are flows on the basic clock with an infinite sequence where all elements are the same. Sequences can be of type integer, boolean or real and a tuple constructor is provided for making combinations of the elementary types.

A variable or output must be assigned to exactly once as this will determine its value for the cycle. This introduces the substitution principle which is a key idea in Lustre. For any assignment A = E where E is an expression, any use of A may be replaced with E with no change to the program semantics. The converse of this also applies. Furthermore, the definition principle states that the behaviour of A must be completely defined by the equation and the values of the variables contained within it. Problems with causality are avoided by insisting that a variable/output must not instantly depend on itself.

As in other languages, complex expressions are built up from operators. Operators can only be applied to operands on the same clock. This seemingly limiting requirement is resolved by providing operators that alter the clock of a flow, as described below.

First, two operators will be outlined that change the value of a flow produced on a particular clock instant. The `PRE` operator, when applied to a flow, will have the value that the flow had on the previous clock instant. If `A` is a sequence (a1, a2, a3, ...) then `PRE (A)` is the sequence (nil, a1, a2, a3, ...). Here nil is used to indicate an undefined value since there is no previous value for `A` at the first instance. To resolve this problem, `PRE` is often used with `->` operator (named the "followed by" operator) which is used to define initial values for a sequence. If `B` is the sequence (b1, b2, b3, ...) then `B -> A` is (b1, a2, a3, ...) more relevantly, `B -> PRE(A)` is (b1, a1, a2, a3, ...).

Now operators that change the clock of a flow will be discussed. The `when` operator is used to slow down the clock of its first argument according to the second argument. For this to make sense the second argument must be a clock (a boolean valued flow). The `current` operator is used to project a slower clocked flow onto a faster one. The operator gives the slower flow a value when its clock is false equal to its value the last time its clock was true. These operators are

B	true	false	true	false	true
(0, 1, false) when B	(0, 1, false)		(0, 1, false)		(0, 1, false)
COUNTER((0, 1, false) when B)	0		1		2
COUNTER(0, 1, false)	0	1	2	3	4
(COUNTER(0, 1, false)) when B	0		2		4

Table 2.1: The effect of when and current operators on node parameters - [4].

illustrated in Table 2.1 however, for this table to make sense the concept of a node must first be introduced.

Lustre program structure is provided by node declarations. One or more equations of the form A = E can be composed into a single node and used in higher level expressions. A node is declared using: an interface specification (lists inputs/outputs, their types and possibly their clocks) and a system of equations that defines node outputs and any local variables in terms of the inputs. An example `COUNTER` node from [4] is shown below. This node increases its output by `incr_value` on each cycle starting from `init_value` and returns to `init_value` when `reset` is true.

```
node COUNTER (init_value, incr_value: int; reset: bool)
   returns (N: int);
let
   N = init_value -> if reset then init_value
                     else PRE(N) + incr_value;
tel.
```

With this and the clock operators in mind, it becomes clear that solutions must be structured in terms of clocks, which is the other use for operators that change the clock of a flow. For example, if an output needs to be updated only at certain times, then the clock of the expression to determine the new output should only be true at these certain times. Taking this further, node computation can be controlled by clocks too. In accordance with the data flow point of view, a node's basic clock is defined by the clock of its input parameters. This means that the expression `COUNTER( (0, 1, false) when B)` only counts when `B` is true. Whereas for the expression `(COUNTER(0, 1, false)) when B` only the output is filtered (Table 2.1). Interestingly, a node can take parameters that are on several different clocks providing this is specified in the declaration and the different clocks are provided as parameters.

In the latest version of Lustre programs can make use of arrays and recursion for structuring [7]. It should be noted that these new constructs are just syntactic sugar - they do not actually provide any increase in the expressive power of the language. Before the program is compiled, arrays are expanded into as many variables as they have elements and recursive nodes are unfolded into regular nodes. Because of this arrays must be indexed by, and recursion bounded by, compile-time expressions. It is also possible to take slices of arrays and these are often used with polymorphism. This allows computations on arrays to be expressed in a concise manner.

2.3.3 Compilation

Static verification in Lustre consists of clock consistency checking. At compile time each expression is associated with a clock. It then uses this to check that every operators arguments are on a suitable clock. This is defined by [6]:

- any basic operator with more than one argument is applied to operands on the same clock
- the clocks associated with actual parameters of any node instantiation satisfy the constraints imposed by the node interface

Theoretically, two clocks are equivalent if they are defined by the same boolean flow. However, testing the equality of two boolean flows is undecidable so Lustre considers a more restrictive form of equality. It defines two flows to be on the same clock if they can be made identical by repeatedly applying syntactic substitutions. It should be noted that these rules satisfy the definition principle: the clock of a variable cannot be inferred from the use of the variable. This means that Lustre can sometimes claim two clocks different when in fact, if it took the context of the usage into account, they could be proved equal.

Code generation for a Lustre program is a translation to a high level language (C / ADA). First, the compiler recursively expands each node call in the source program. With suitable renaming of parameters, variables and clocks this leaves a "flat" program from which code generation can proceed.

Next, using a synthesis borrowed from Esterel compiling techniques, it is possible to compile Lustre code into an automaton where the control is produced by simply calling an automaton step procedure at each instance of the base clock. This yields the following overall structure of the main wrapping program [6]:

```
Initialisations
infinite loop
   input handling
   calls to input procedures
   call to the automaton step procedure
      (this will call some output procedures by itself)
end loop
```

Automaton generated by Lustre are encoded in a standard OC [24] format which it shares with Esterel and Argos. This enables a whole host of tools and utilities to work for most synchronous languages. These utilities include: code generators (to C/ADA), automaton minimises, verification tools, display tools, graphic interface generators (so the wrapping code shown above does not have to be produced until implementation) and distributed code generators.

2.3.4 Verification

For verification to be possible it should be noted that the correctness of a program does not depend on the program in its entirety but rather as a small set of properties that the program should always fulfill. Moreover, many of these properties are "safety properties" that describe states that the program should *not* be in. These are the opposite of liveness properties which state that some

condition should appear in the future. Obviously safety properties are far easier to prove than liveness properties and some of the concrete reasons for this will be discussed. Safety properties can be verified by checking the properties of all reachable states - this allows the use of already mature reachability algorithms. They can also be checked compositionally which reduces the complexity of proofs and since a Lustre program is built in a modular way, they combine well.

Specifying safety properties for Lustre programs is particularly easy because they can be specified in terms of the language itself. For example if the safety property P is expressed by a boolean expression B, then P holds if and only if B holds true during any execution of the program. This can be easily specified in terms of the Lustre language itself using the assertion mechanism. [5] provides more details on this.

After safety properties have been specified, verification proceeds by analysing the control automaton (state graph) built by the compiler and checking on this graph that each property is upheld. The only problem with this method is that the state graph may become very large but if this is the case then there are techniques to reduce its size while retaining the information needed to prove the properties. This problem is investigated in [6].

2.4 SCADE

SCADE is an industry-standard tool that provides an integrated development environment for designing synchronous reactive systems. The underlying formalism used in SCADE is Lustre's dataflow equations and node view. These nodes may be expressed either graphically or texturally and both formalisms may be interchanged.

Surprisingly, the other synchronous formalism mentioned in section 2.2 is available in SCADE: safe state machines. These have been used in the Esterel language for a long time however, only recently due to the work of Charles Andre [29] they have been successfully integrated into a dataflow design tool like SCADE. At the moment their mode of operation is somewhat limited since they can only be implanted in a dataflow design, the reverse is not possible meaning that a safe state machine cannot have a subpart expressing dataflow equations. All the expressive power of Esterel Studio's safe state machines remains intact in SCADE. Thus this new combination of the formalisms is a very powerful design tool. SCADE also offers a state machine formalism however, this is depreciated now by safe state machines and should only be used for expressing very simple machines (with less than 15 states - [21]).

Another feature of SCADE is its simulation mode. Here, a design may be simulated by letting the user enter input values and then stepping the cyclic function both back and forward. Furthermore, if the design was expressed using the graphical formalism then intermediate output values can be seen after every operator in the network. Also, if the design consists of sub nodes and these too were designed graphically, then all the input, output and intermediate values of these nodes are available to the user. Breakpoints and graphs of variables over time are also offered to further aid debugging.

SCADE supports automatic code generation to C and ADA. The code produced by SCADE is similar to that of the Lustre compiler, however it differs

with its dealings of constants, data types and the working of the cyclic function. It should be noted that a SCADE design is first translated into Lustre and then translated into a high-level language. However, this intermediate Lustre code is a sort of pseudo-Lustre and does not adhere to the v4 Lustre language [24]. This will be investigated further in chapter 3.

An operator will be used frequently in SCADE designs called the conditional activation operator. This is similar to a Lustre `current` operator followed by a `when` operator, i.e. it activates a node only when its clock is true, and it provides the last value output when the clock is false.

2.5 Lego Mindstorms and the RCX

Lego Mindstorms [16] is the latest product to be produced by Lego. It is radically different from all previous Lego products since it allows Lego models to be computer-controlled. At the heart of Lego Mindstorms is the RCX. This brick is a small battery-powered computer system capable of controlling three actuators and reading three sensors.

The RCX has been described in a hierarchy of four layers [13]: hardware, system ROM, firmware and user programs. At the hardware layer the CPU controlling its operation is a Hitachi H8 series microcontroller. This CPU provides serial I/O, ADC and built-in timers. It contains 16KB of internal ROM which stores the system ROM, and a further 32KB of static RAM which is used to hold the firmware, any user programs currently on the device and the runtime environment of a program when it is running. Aside from the three output ports already mentioned, which may be connected to any Lego output device (buzzer or motor), the RCX has built-in: a LCD, four buttons and a speaker. It also provides an IR (infrared) interface which is used to download user programs to the RCX and in more complicated situations, provide communication between multiple RCXs. The three input ports can each have a Lego sensor connected. There are two different types of Lego sensors available: passive and active. Passive sensors do not require power and these include touch and temperature sensors. On the other hand, active sensors must be powered to work correctly. Active sensors in the Lego range are light and rotation sensors. There has been much experimentation with so-called "homebrew" sensors. [13, 19] describe how to make some of these more elaborate sensors.

The system ROM layer provides convenient methods for accessing the RCX's hardware. Furthermore, it also allows a firmware to be run on top of it (as in the case of the standard firmware) but if desired, the firmware can take full control of the RCX and bypass the system ROM layer (this is what BrickOS (section 2.6) does). Thus the firmware layer provides the capability for custom firmwares to be loaded onto the RCX. Currently, the custom firmwares available are: BrickOS, pbForth [17] and leJOS [18]. BrickOS supports the C programming language, pbForth the forth language and leJOS supports a JVM allowing Java programs to be run.

2.6 BrickOS

BrickOS [20] (formerly known as LegOS) is an open source replacement operating system for the RCX. It boasts a number of features that make it considerably more complex than the standard Lego firmware. Foremost, it allows programs written in the C programming language to be executed on the RCX. Obviously this is especially important for this project since SCADE is capable of automatically generating C code as a target language. Furthermore, programs run under BrickOS are executed natively rather than interpreted byte codes which is the method of the standard firmware. In turn this allows faster execution of programs. Another feature BrickOS presents that keeps it in the operating system domain is priority based preemptive multitasking. Synchronisation between multiple processes is possible with built-in support for POSIX semaphores.

Control of the RCX's hardware using BrickOS is on a much lower level than in the standard firmware. BrickOS provides a much finer control of outputs, for example, allowing the speed of motors to be adjusted over 255 values. Furthermore, when controlling the LCD every segment can be switched on or off independently. BrickOS also provides access to the RCX's internal speaker. Although this output is not nearly as useful as the others, it can aid in debugging since the LCD is quite limited in what it can display.

The value of sensors can either be taken as raw or adjusted. Raw mode offers the greatest sensitivity however it is often unnecessary accurate and adjusted sensors already have their value translated into a meaningful reading. Apart from offering support for the four standard RCX sensors (touch, light, rotation and temperature) it also allows usage of buttons on the RCX itself. The buttons it provides access to are view and program. This effectively gives the RCX two more touch sensors.

Program control in BrickOS is provided by two features: sleep and `wait_event`. The sleep functions are simple - they tell a thread to stop executing for a certain amount of time and then resume. The `wait_event` function is more complicated. It takes a function which returns a boolean value and then repeatedly calls this function until it returns true. At this point execution of the thread continues.

The last feature of BrickOS to be discussed is the LegOS Network Protocol (LNP). This provides a means for two or more RCXs to communicate. A computer may also participate using the program download tower. This protocol has two layers, an integrity layer and an addressing layer. The integrity layer guarantees that if a message is received it will be the same message that was sent. This is much like the Internet protocol UDP. However, it differs from UDP in that it is a broadcast mode – any RCXs in the receiving area will pick up the message. To provide directed messages the addressing layer is placed on top of the network protocol stack. Each RCX has a unique identifier that is specified when the BrickOS firmware is downloaded. Then when a message is sent, the recipients identifier is included and the message will only be received by that RCX.

2.7 A small development

This section will describe a small but complete development of a synchronous reactive system combining all the topics addressed during this literature review.

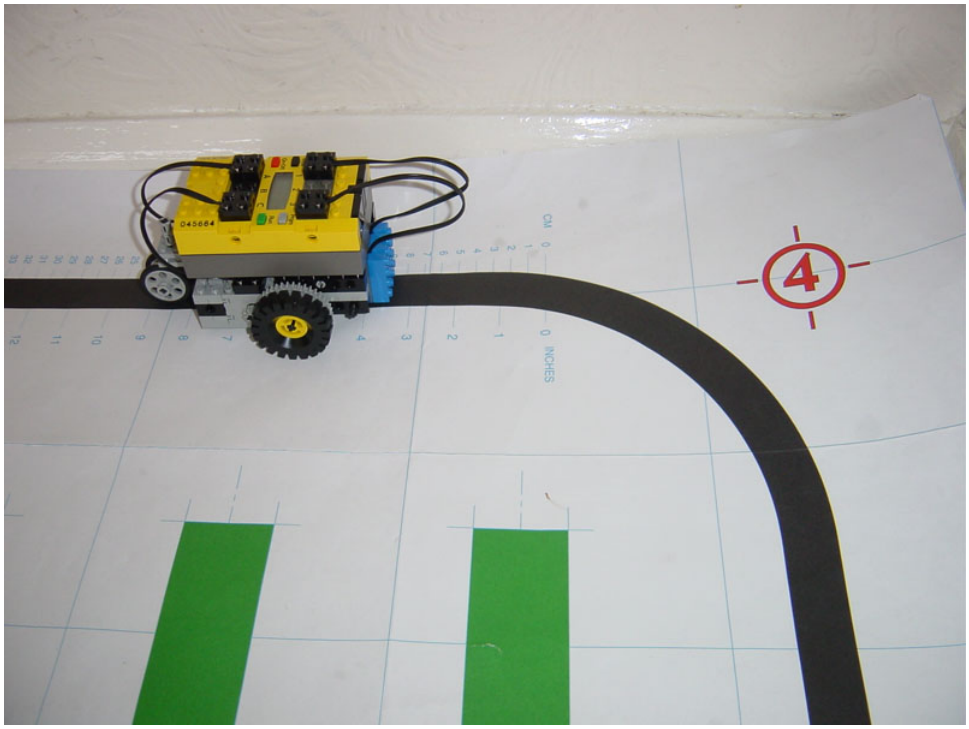

Figure 2.1: A robot following a line

The reactive system in this case will be a line following robot. The design will be expressed in Lustre data flow equations and, alternatively, as a SCADE graphical node.

2.7.1 Problem Analysis

Line following is a classic robotic challenge [14] (figure 2.1). There are many ways to accomplish this, varying in complexity as well as resources needed. The most simple in terms of resources is a robot with only one light-sensitive device. To follow a line that may turn in both directions using one light sensor, it is

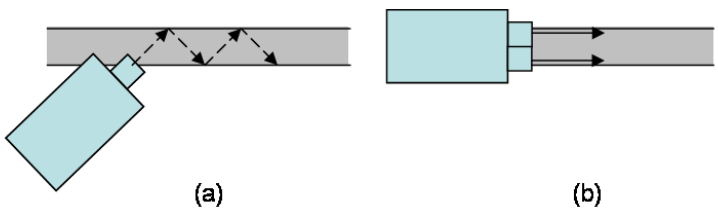

Figure 2.2: Methods for following a line

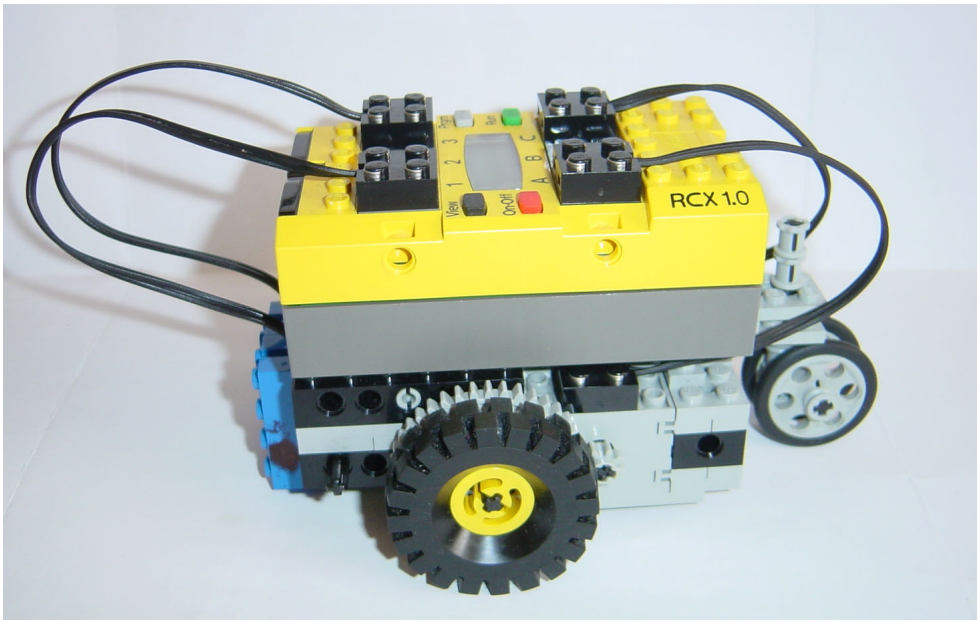

Figure 2.3: Improved Lego robot specifically for line following

necessary to zigzag along the line as shown in figure 2.2(a). However, if two light-sensitive devices are available then a much simpler method can be used (figure 2.2(b)). This relies on a light sensor being placed over each edge of the track. When a sensor detects that it is no longer over the track, the robot is instructed to turn in the relevant direction. This continues until the robot rejoins the track with the sensor over the edge again. For this to work correctly the track must uphold certain properties such as a constant thickness and a limit in the sharpness of its turns. Since only a simple robot is desired in this section for means of clarifying the methodology, the solution using two light sensors will be implemented.

2.7.2 Lego Design

The robot is based on a design from the Lego Mindstorms user guide [26]. This provides a robot with two sets of independently powered wheels, which allows it to move forward, back, turn left and turn right. Some alterations have been made from the original design. The first is to reduce the turning circle of the robot enabling it to follow tracks with tighter turns. This has been done by bringing the front wheels closer together and replacing the rear wheels with a caster (figure 2.3). Also, the original design provides room for only one light sensor. For this design it is necessary to have two light sensors and they must be positioned so that the photo sensitive parts are precisely over the edges of the tracks. For identical turns to be performed in both directions, the centre of the robot must pass directly over the centre of the line, therefore presenting no bias to either side. To obtain this precise positioning, the light sensors are mounted on an axle at the front of the robot. This way the light sensors can be

arranged without having to adhere to the Lego "bumps" positioning system.

2.7.3 Lustre Design

```
node followline (
    touch_1: bool;
    touch_2: bool;
    touch_3: bool;
    light_1: int;
    light_2: int;
    light_3: int;
    rotation_1: int;
    rotation_2: int;
    rotation_3: int
 )
 returns (
    Forward_A, Back_A :bool ;
    Speed_A : int;
    Forward_B, Back_B :bool ;
    Speed_B : int;
    LCD_INT : int;
);

let
    Forward_A = true
    Forward_B = true
    Back_A = if (light_1 > 40) then true else false
    Back_B = if (light_3 > 40) then true else false
    Speed_A = 255
    Speed_B = 255
    LCD_INT = (light_1 * 100) + light_3
tel.
```

The Lustre design has an unnecessarily complex interface. This is so it is compatible with the compilation process that will be described in section 3.1. With the exception of `Back_A` and `Back_B`, all motor outputs are fixed. Since `Forward_A` and `Forward_B` are fixed to true, the motors can only be in one of two states: forward or floating. These are purely determined by the value of the light sensor that corresponds to a motor (i.e. the left light sensor can turn off the right motor). When the light sensor reading rises above a certain threshold, the corresponding wheel is switched off allowing the robot to turn. As soon as the light sensor reading drops below the threshold, the motor is turned back on and the robot resumes a straight-line course.

2.7.4 SCADE Design

It can be seen from figure 2.4, that the SCADE design is identical to the Lustre design simply expressed using a graphical notation.

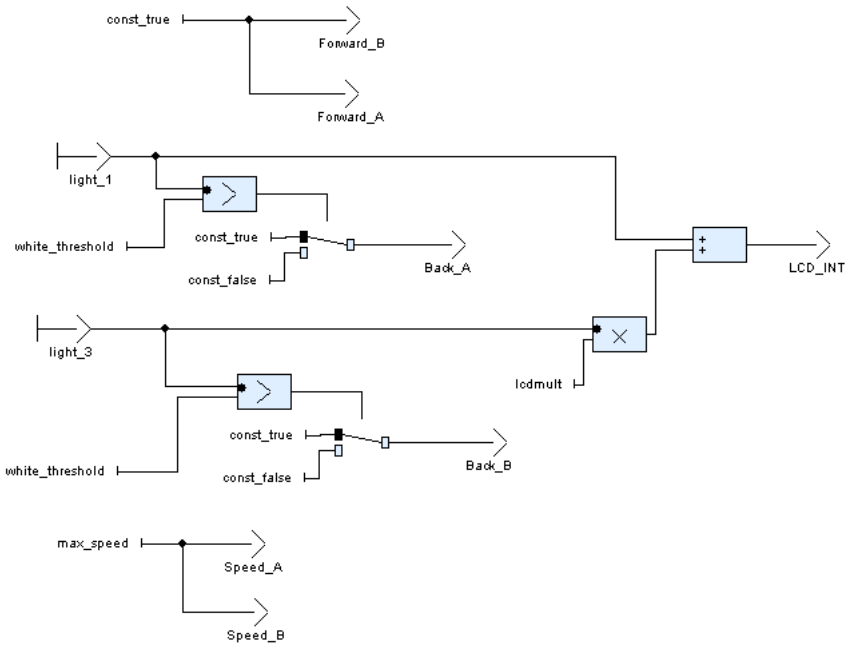

Figure 2.4: The SCADE node for following a line

2.7.5 Implementation

As briefly mentioned earlier, these designs were implemented using the compilation methods discussed in chapter 3. Therefore this part of the mini project will be omitted. After reading chapter 3, it will be clear how the tools developed in that section can be applied to this situation.

2.7.6 Evaluation

This section has provided the reader with a brief overview of the content of the project. It has shown how a reactive design is produced and although the compilation to executable code has been omitted, it is simply a case of executing a script. A simple Lego robot has also been described, highlighting the use of light sensors and motors as well as the mapping between the usage of motors in a reactive design and in BrickOS.

Moreover, this section has shown the benefits of using the synchronous paradigm to design reactive systems. It is undeniable that the designs presented above are exceptionally simple and as such this is a testament to the power of the synchronous paradigm as a design tool. The C program capable of performing the same task would undoubtedly be longer, harder to understand and it certainly could not be simulated or verified as easily.

The reader should also have a feel for the similarities between a Lustre textual design and a SCADE graphical design. Of course this is unsurprising since the underlying language of SCADE is Lustre. Therefore, this example shows that for simple problems, Lustre and SCADE designs will be identical.

It should be noted that this does not hold for complex designs, such as those involving conditional activation.

We will now proceed to investigate the compilation of synchronous languages to Lego Mindstorms; the first main contribution of this project.

Chapter 3

Converting Synchronous Language Designs to Code Executable on the RCX

Creating a robust methodology for translating SCADE and Lustre designs into code executable on the RCX is one of the main aims of this project. Previous work has already been conducted on Lustre to BrickOS translation [22] but SCADE to BrickOS translation has not been investigated yet. Creating a successful SCADE translation tool will greatly increase the RCX's usefulness as a teaching aid for synchronous reactive systems and enable the robots built later in the project to be programmed using SCADE.

3.1 Investigation into Lustre Compiling Options

To understand the compilation of Lustre to Lego Mindstorms, it is first necessary to examine the output produced by the Lustre compiler. Compilation of a Lustre program has already been briefly mentioned in section 2.3.3 but for further clarity it will briefly be repeated here. A Lustre design is implemented as a finite state automaton, the benefits of this being a small program with a bounded execution time. To create C code from a Lustre design, two programs from the Lustre suite are needed: lustre and poc. The design is first compiled into the OC intermediate language format [24] using the tool called lustre. Then poc or Pollux [24] is used on the OC language to generate the automaton environment. An overview of the code Pollux creates will now be presented.

3.1.1 The Automaton Environment Generated by Pollux

The environment of the automaton (or context) is stored using a structure. This contains the current state of the automaton as well as variables used to hold input and output values.

```
typedef struct  {
   void* client_data;
   int current_state;
```

```
   _boolean _V0;
   _boolean _V1;
   _boolean _V2;
} line2_ctx
```

As can be seen, Pollux uses self defined types for its variables such as `_boolean` to make its environment as unique as possible. In this example, `_V1` will hold the first input value of the system and `_V2` the first output value of the system. The variable `_V0` is used to keep track of when an input is updated, as will be seen in the next example.

```
void line2_I_touch_1(line2_ctx* ctx, _boolean __V){
   ctx->_V1 = __V;
   ctx->_V0 = _true;
}
```

The input procedures are generated automatically. This is so that the particular internal variable of the context that an input relates to only needs to be known to the compiler. Thus all the user must know are the names of these procedures and not the internal structure of the automaton context. For this reason the name of the input procedure is generated in a fixed form: the first part is the name of the main node, followed by an `_I_` (standing for input) and lastly the name of the input as specified in the design. The automaton context and value of the input are passed to the procedure which then updates the context with the value.

Before the automaton can be used it must first be initialised. This is accomplished using the following procedure:

```
void line2_reset(line2_ctx* ctx){
   ctx->current_state = 0;
   line2_reset_input(ctx);
}
```

This procedure is passed the context of the automaton and then proceeds to set the automaton to its initial state. After this it calls the input reset procedure.

```
static void line2_reset_input(line2_ctx* ctx){
   ctx->_V0 = _false;
}
```

This updates the context to reflect the fact that no inputs have been processed yet. Now that all the supporting code for the automaton has been examined, the transition function (or step procedure) will be explained:

```
void line2_step(line2_ctx* ctx){
   switch(ctx->current_state){
   case 0:
      ctx->_V10 = _true;
      line2_O_Forward_A(ctx->client_data, ctx->_V10);
      ctx->_V11 = _false;
```

```
      ...

      ...
      ctx->current_state = 0; break;
   break;
   }
   line2_reset_input(ctx);
}
```

Only one transition function is produced since, as stated in section 2.3.3, compilation starts from a "flat" program, containing only one node. It is implemented as a `case` statement, simply switching on the current state of the automaton. After enough computation has been performed to determine the value of an output, an output procedure for communicating this to the outside world is called. This then continues for all outputs. At the end, the input reset procedure is called to store the fact that the inputs must be updated before the step procedure is called again. This can be used to check that inputs are sampled frequently enough and therefore test the synchronous hypothesis. The output procedures are prototyped by the compiler although ultimately they must be implemented by hand. It is necessary for the automaton step procedure to call the output procedures since only it knows which variable in the context relates to an output. In the same style as the naming for input procedures, the output procedure names are generated automatically so that the step procedure knows how to call them.

```
extern void line2_O_LCD_INT(void*, _integer);

void line2_O_LCD_INT(void* cdata, _integer v)
      {lcd_int((unsigned) v);}
```

A further two functions are automatically generated by Pollux, the first for dynamic allocation of the context and the second for copying a context. Neither of these two functions is used in the implementation, so they will not be explained any further.

3.1.2 Previous Work on Lustre Compilation to Lego Mindstorms

Christophe Mauras [22] has already conducted some work involving the translation of a Lustre design to code executable on the RCX. Most of the compiling work conducted in this project stems from what Mauras has done and as such an overview of his work will now be presented.

The compilation procedure centres around a Perl script `legolus`, which first calls the necessary programs to compile Lustre to C (lustre followed by poc - section 3.1). It then selectively extracts parts of the C code and incorporates this into a new file containing the wrapping code necessary to support a Lustre reactive kernel. The parts extracted are listed above in section 3.1.1 and hence the wrapping code only consists of output procedures and the driving loop procedure. The full process is shown in figure 3.1.

The main node definition in the Lustre program must adhere to the following format:

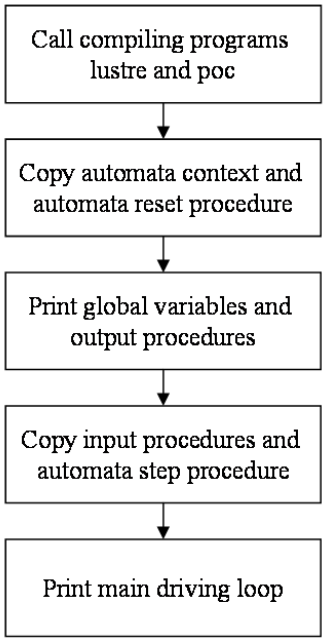

Figure 3.1: A Flow diagram describing the operation of legolus

```
node nodename (
    capteur_1: bool;    -- touch sensor connected to port 1
    capteur_2: bool;    -- touch sensor connected to port 2
    capteur_3: bool;    -- touch sensor connected to port 3
    lumiere_1: bool;    -- light sensor connected to port 1
    lumiere_2: bool;    -- light sensor connected to port 2
    lumiere_3: bool;    -- light sensor connected to port 3
    rotation_1: int;    -- rotation sensor connected to port 1
    rotation_2: int;    -- rotation sensor connected to port 2
    rotation_3: int     -- rotation sensor connected to port 3
)
returns (
Avant_A, Arriere_A :bool ;     -- Avant_A - Forward_A, Arriere_A - Back_A
Vitesse_A : int;               -- Vitesse_A - Speed_A (between 0 and 255)
Avant_B, Arriere_B :bool ;
Vitesse_B : int;
Avant_C, Arriere_C :bool ;
Vitesse_C : int;
Affichage_0 : int;      -- Control character 0 on LCD
Affichage_1 : int;      -- Control character 1 on LCD
Affichage_2 : int;      -- Control character 2 on LCD
Affichage_3 : int;      -- Control character 3 on LCD
Affichage_4 : int;      -- Control character 4 on LCD
Affichage_5 : bool;     -- Control character 5 on LCD (sign of number)
```

```
Seuil_1 : int;          -- Value above which lumiere_1 will report true (0..100)
Seuil_2 : int;          -- Value above which lumiere_2 will report true (0..100)
Seuil_3 : int           -- Value above which lumiere_3 will report true (0..100)
);
```

As can be seen, for simplicity the program considers each type of sensor available to each port whereas, obviously, only one sensor can be used with each port. This is so that the input handling part of the system can be done using the same piece of code no matter which sensors are connected to which ports.

The main driving loop will now be considered. This is responsible for initialising the system, timing the base clock, handling inputs and calling the automaton step procedure. Its functionality is shown in figure 3.2 and the code is detailed below.

```
main(){
  time_t temps_systeme = get_system_up_time();
  line1_reset(&the_ctx);
  dir_A = 3; dir_B = 3; dir_C = 3;
  first_A = 1; first_B = 1; first_C = 1;
  seuil_1 = 50; seuil_2 = 50; seuil_3 = 50;
  ds_active(&SENSOR_1);
  ds_active(&SENSOR_2);
  ds_active(&SENSOR_3);
  ds_rotation_set(&SENSOR_1, 0);
  ds_rotation_set(&SENSOR_2, 0);
  ds_rotation_set(&SENSOR_3, 0);
  ds_rotation_on(&SENSOR_1);
  ds_rotation_on(&SENSOR_2);
  ds_rotation_on(&SENSOR_3);
   while(1){
     if (SENSOR_1<0xf000)
       {line1_I_capteur_1(&the_ctx, _true);}
     else
       {line1_I_capteur_1(&the_ctx, _false);}
     if (SENSOR_2<0xf000)
       {line1_I_capteur_2(&the_ctx, _true);}
     else
       {line1_I_capteur_2(&the_ctx, _false);}
     if (SENSOR_3<0xf000)
       {line1_I_capteur_3(&the_ctx, _true);}
     else
       {line1_I_capteur_3(&the_ctx, _false);}
     if (LIGHT_1 > seuil_1)
       {line1_I_lumiere_1(&the_ctx, _true);}
     else
       {line1_I_lumiere_1(&the_ctx, _false);}
     if (LIGHT_2 > seuil_2)
       {line1_I_lumiere_2(&the_ctx, _true);}
     else
       {line1_I_lumiere_2(&the_ctx, _false);}
```

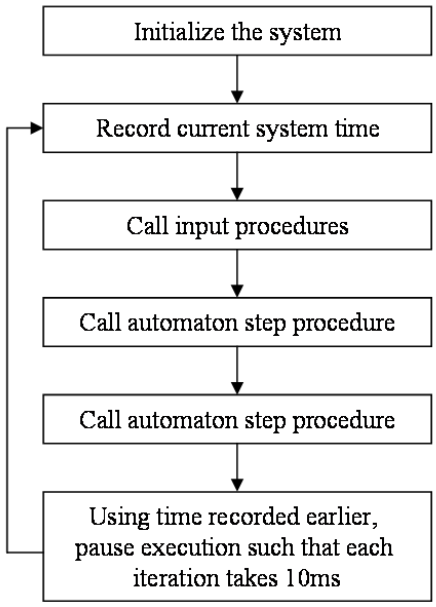

Figure 3.2: Flow diagram describing the behaviour of the main driving loop

```
    if (LIGHT_3 > seuil_3)
      {line1_I_lumiere_3(&the_ctx, _true);}
    else
      {line1_I_lumiere_3(&the_ctx, _false);}
    line1_I_rotation_1(&the_ctx, ROTATION_1);
    line1_I_rotation_2(&the_ctx, ROTATION_2);
    line1_I_rotation_3(&the_ctx, ROTATION_3);
    line1_step(&the_ctx);
    msleep((int) 10 - (get_system_up_time() - temps_systeme));
    temps_systeme = get_system_up_time();
  }
}
```

The procedure begins by resetting the automaton context. It then executes some BrickOS specific commands enabling the sensors to work in all different modes. This is to do with the port overloading discussed above. Since BrickOS operates at such a low level it enables the program to consider having three different sensors permanently attached to the same port. Therefore, for the correct sensor attached, a meaningful value will be generated and stored in the automaton context. Next, a non-terminating loop is begun which is responsible for stepping the automaton at the right time with the system inputs present when this occurs. It does this by calling the input procedures that were automatically generated by Pollux. It then steps the automaton, pauses execution such that each cycle of the loop will take 10 milliseconds and resumes for another iteration. Observe that the step procedure will call the output functions. The cyclic delay (default 10ms) can be changed by replacing the 10 in the line

Forward_A	Back_A	Resulting direction
F	F	Floating
F	T	Back
T	F	Forward
T	T	Brake

Table 3.1: Values of Forward_A and Back_A and the corresponding motor direction

below with the desired cyclic delay.

```
msleep((int) 10 - (get_system_up_time() - temps_systeme));
```

The touch and rotation sensors are handled in the expected way: the touch sensor returns a value of true or false, and the rotation sensor returns a signed integer indicating the revolution count. However, the light sensor is interpreted in a non-standard way, as a boolean. This is accomplished by using an output to set the threshold at which the light sensor input should provide a true reading, therefore determining the light sensor value in the input processing part of the driving loop.

Motor outputs are handled in a straightforward way. Since BrickOS allows motors to be in four states it makes sense that these should be handled by two boolean variables, `forward_<motor_letter>` and `back_<motor_letter>`. Table 3.1 shows the resulting motor state (forward, back, floating[1] and brake), determined from the value of the variables forward and back. It is in the implementation where the motor procedure calls become complex. This complexity arises since the order in which the Forward and Back output procedures are called changes depending on how Pollux implements the automaton step procedure. Furthermore, two output procedures are used where only one system call needs to be made. This is solved using the following piece of code:

```
void line1_0_Avant_A(void* cdata, _boolean v)
      {if (v) dir_A -= 2;
       if (first_A) first_A = 0;
       else {motor_a_dir(dir_A);dir_A = 3;first_A = 1;}}
void line1_0_Arriere_A(void* cdata, _boolean v)
      {if (v) dir_A -= 1;
       if (first_A) first_A = 0;
       else {motor_a_dir(dir_A);dir_A = 3;first_A = 1;}}
```

Using variables `first_<motor_letter>` and `dir_<motor_letter>` the state of the motor and the order in which the procedures are called is taken into account and as a result the system call for determining the motor direction is only made once. The speed of a motor is set using a simple integer output.

However, there are a few problems with the script. The first is a superficial one, the original script was written in French making the first task to translate

[1]Floating is a motor operation mode where no power is supplied, but the motor axle is free to turn from other forces. The opposite of this mode is brake, where the motor axle provides a resistance to forces trying to turn it.

it into English. Next, the script uses depreciated BrickOS commands such as `sys_time` (a variable that holds the time) instead of `get_system_up_time()` (a function that returns the time). However, the most limiting part of the script is the way it insists light sensors must be used. The method employed has already been explained above, hence only an example will be presented here to explain why this is a problem. Consider a light sensor that must differentiate between three different ranges of values. For this to be possible, after each reading the output threshold would need to be updated, since a boolean can only distinguish between two ranges. This means it would take two cycles to determine which one of the three ranges the value is in.

Lastly, LCD control is quite complicated, allowing each 7-segment section to be individually controlled. A more relevant function for the LCD to perform would be to display an integer.

3.1.3 New Perl Script for Lustre Compilation - legolus2

As part of the project an updated script was produced addressing the negative points of the original script raised above. The Perl code is available in Appendix A. The language translation and depreciated system commands were trivial to resolve. The light sensor problem was removed by making the light sensor input an integer, therefore any comparisons on its value can be made inside the Lustre design and an output threshold is no longer needed. The LCD outputs have been simplified to a single output capable of displaying a signed integer. A sample node interface to be used with the new script is shown below.

```
node nodename (
                touch_1: bool;
                touch_2: bool;
                touch_3: bool;
                light_1: int;
                light_2: int;
                light_3: int;
                rotation_1: int;
                rotation_2: int;
                rotation_3: int
 )
returns (
Forward_A, Back_A :bool ;
Speed_A : int;
Forward_B, Back_B :bool ;
Speed_B : int;
Forward_C, Back_C :bool ;
Speed_C : int;
LCD_INT : int;
);
```

3.2 Investigation into SCADE Compiling Options

There are two main routes available for compiling a SCADE design to C code compatible with BrickOS. The first seeks to manipulate the SCADE project files into a Lustre design and then complete the rest of the compilation process using the method described above (3.1.2). The other method hopes to make use of the code generation techniques already present in SCADE. There are advantages and disadvantages to both of these methods and either can be used with varying degrees of success. Both compilation methods will be explored in detail but first it is necessary to provide an overview of the way SCADE stores designs.

SCADE uses two types of files to keep track of the contents of a project: .saofd and .saofdm files. Saofdm files are used to store the general project settings, one for each project, and for each node in the project there is a corresponding saofd file containing its details. The layout of saofd files bears a resemblance to Lustre design files; however, there are some major differences that prevent it from being compatible with Lustre.

3.2.1 SCADE Design to Lustre v4 Format

A tool is available that claims to be capable of translating a SCADE project description file (.saofdm) to a Lustre design (method A in figure 3.3). The tool is called SCADE2lustre and is included as part of the SCADE suite [15]. However, the Lustre design produced by this tool does not adhere to the academic Lustre v4 language (as used in section 3.1). This means that it is not a compatible input to the script that has been discussed above. It is possible that this pseudo-Lustre design could be transformed into one that is compatible; however, each SCADE language construct would need to be checked for compatibility making this a time-consuming approach. As such it is doubtful that this approach would work correctly for all SCADE constructs.

A second tool is available that is able to perform the SCADE to Lustre transformation. The tool is a UNIX filter called s2l, which is produced by Verimag [23]. The tool works with the node description files (.saofd), which must first be concatenated together in a suitable order (method B in figure 3.3). It then transforms the concatenated saofd files into a Lustre design that strictly adheres to the v4 language. Thus, it would be possible to use this as an input to the Perl script. However, it does not support some of the more advanced SCADE language constructs, in particular conditional activation. Also, it would be difficult to automatically concatenate the saofd files since the order in which it must be done changes with each design.

3.2.2 Translating SCADE's C Code to Executable Code on the RCX

The other avenue to explore for compiling a SCADE design into a BrickOS C program involves using the internal code generation in SCADE (method C in figure 3.3). The first stage of this uses the tool SCADE2lustre which has already been described above. SCADE2lustre's output is used as input to the SCADE

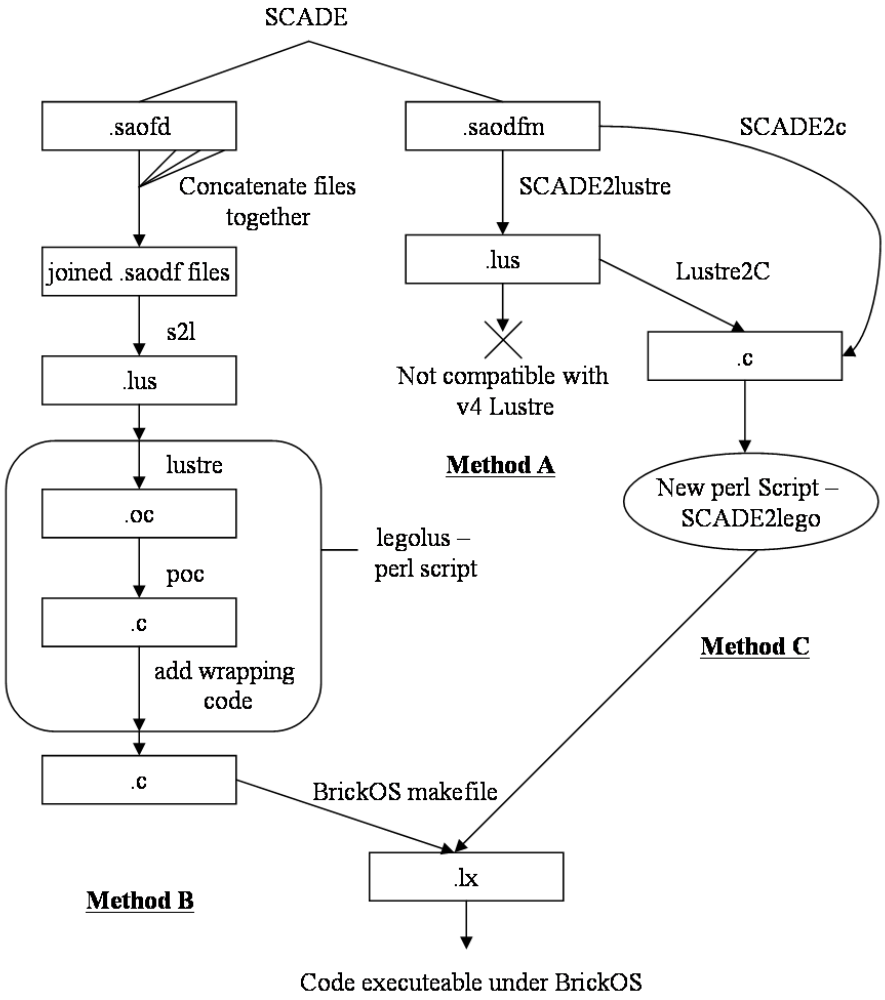

Figure 3.3: Different methods for translating a SCADE design to BrickOS code

suite program Lustre2C. This program serves a similar role as Pollux (section 3.1.1) which creates the automaton environment necessary for the reactive design to be executed. This process can be performed more easily using SCADE2c which calls both SCADE2lustre and Lustre2C.

However, the automaton environment produced by Lustre2c is not the same as the one produced by Pollux. It is a more "open" format, for example it does not support automated calling of output procedures. This has the disadvantage of being much worse at information hiding but does allow more flexibility. Nevertheless, the Perl script (with the Lustre compiling components removed) will not create correct code to interface with the environment produced by Lustre2c. Therefore, a new script (based on the previous work of Mauras [22]) will need to be created that can generate code executable on the RCX while supporting the Lustre2c automaton environment.

3.2.3 Evaluation of SCADE Compilation Methods

Of the three compilation methods analysed, the one using SCADE's own internal code generation is the most promising. This is because the internal code generator will always support any language constructs in SCADE since it is part of the SCADE suite. Whereas for the other two methods, it is conceivable that they may not work correctly for all language features; indeed it is already know that some features are incompatible (e.g. s2l and conditional activation). Also, if the input/output format should change, then the script can be updated to reflect this, unlike the other two, which are fixed in their functionality. For example, safe state machines require a different way of handling inputs. Initially, support will not be provided for safe state machines, however, it is an example of how the script can be updated, thus providing flexibility.

Before the new script can be explained it is necessary to examine the automaton environment that Lustre2c generates.

3.2.4 Environment Generated by Lustre2C

Although it is possible to generate "flat" programs (as in Pollux which only produces code for one combined node), lustre2c will be run with no expansion enabled so there will be a separate context and step procedure generated for each node. Having said this, the main node's step and initialisation procedures call all the sub-node's step and initialisation procedures respectively; hence we need only be concerned with the main node.

Lustre2c generates four files of relevance to this implementation: `<node_name>.h`, `<node_name>.c`, `<node_name>_global.c` and `<node_name>_const.c`. The header file (`<node_name >.h`) contains the contexts for all nodes in the system and prototypes for the initialisation and step procedures. For example,

```
typedef struct {
_int _I0_init_value;
_int _I1_incr_value;
bool _I2_reset;
_int _O0_count;
_int _L4_countgreen;
bool _M_init_0_countgreen;
```

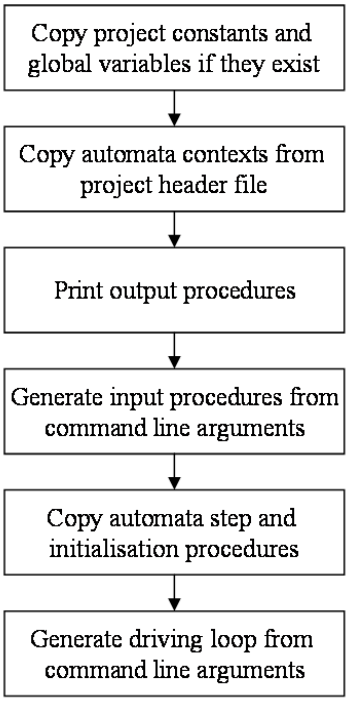

Figure 3.4: A flow diagram describing SCADE2Lego's operation

```
} _C_counter;

void counter_init (_C_counter *);

void counter (_C_counter *);
```

The associated .C file (`<node_name>.c`) contains the code of all the functions prototyped in the header file. Lastly, the files `<node_name>_const.c` and `<node_name>_global.c` c contain constant values and global variables, respectively, for the project. In a similar style to Pollux, Lustre2c uses unique types for variables in its environment such as `_int` and `bool`.

3.2.5 SCADE2Lego Script - Usage and Description

The SCADE2lego script (figure 3.4) is called as follows:

```
Scade2lego <node_name> -i <no of inputs> <list of inputs,
ordered as they appear in SCADE> -o <no of outputs> <list of
outputs, ordered as they appear in SCADE>
```

The script begins by writing a series of #include statements, for the BrickOS system calls. It then #defines true and false (since the SCADE2c code uses these) and then creates types for `_int` and `bool`. Next, it checks for the existence of the project constants file (`<node_name>_const.c`) and, if it exists,

it proceeds to copy the constant declarations (identified using regular expressions) to the output file. If there is no constants file then the script continues regardless, since this represents the situation where a project has no constants. Exactly the same operation is then performed on the global variables file (`<node_name>_global.c`), i.e. global variables are identified by regular expressions and copied to the output file. The header file is then opened and the context for each node is copied to the output file.

Next, the variables used for holding the automaton context and the motor states are generated (`first_A`, `dir_A` etc. - section 3.1.2), which is followed by a printing of all the possible output procedures. This is done so it is not necessary to selectively produce only output procedures that are used in the project, since any uncalled output procedures will not interfere with the program. The motor output procedures are the same as those designed by Mauras in the legolus Perl script, and their functioning has already been discussed in section 3.1.2.

Creating the input procedures is more complex since it is necessary to know which variable in the automaton context corresponds to a particular input. This information is derived from the arguments that are specified when the script is executed. Therefore, by using the order in which the inputs are listed and SCADE's well-defined naming scheme, input procedures can be automatically produced outside the SCADE compiling program. This is in contrast to the way that Pollux handles input procedures, as it automatically generates them while generating the automaton context. Thus, the user does not need to know the structure of the internal automaton context.

After this, the final piece of the automaton environment, the initialisation and step procedures from `<node_name>.c`, are then copied to the output file.

Lastly, the main procedure is built. First, if any of the sensors are active, then a BrickOS system call must be made to initialise them as such. Furthermore, if any of the sensors are rotation sensors, they must be initialised, turned on and a small delay provided to allow them to settle (three further system calls). The driving loop is then created starting with calls to the input procedures generated earlier in the script. A call is then made to the main node's step function before generating the output calls. The order specified by the command arguments (after the `-o`) is used to pass the relevant part of the automaton context to the output procedure. Lastly, a delay is inserted which defines the base clock for the reactive kernel.

The code for SCADE2lego is available in Appendix B.

3.2.6 Testing of SCADE2lego

Since the script performs many independent operations, i.e. the operations do not rely on the correct functioning of others, each section can be tested independently. The exceptions to this are requests for opening files because, if these fail, then so will other parts of the script. Therefore, the first stage was to test that all system file calls can correctly handle failures. This was accomplished by artificially causing the calls to fail one by one and noting the resulting behaviour.

The script uses four operations responsible for copying data between the input files and output file. Testing these involved checking that the regular expressions used to identify the parts to be copied were correct. Although it

was difficult to determine this for all cases, a variety of input files were used and in each case the regular expressions identified the correct part of the file.

What remained to be tested was that the input procedures and main driving loop were generated correctly. The rest of the script just writes predefined text to the output file and was therefore trivial to test.

Testing the generation of the input procedures was accomplished by enumerating seven different possibilities. These were, changing the number of inputs used between zero and three and checking that boolean input procedures were generated for touch sensors and integer input procedures were generated for light and rotation sensors. In each case the part of the automaton context that the procedure should update was matched up against the actual variable in the context, therefore showing them correct.

In the main driving loop, there are two parts to test. First, that the script generates correct system initialisation calls for the sensors and also that it calls the input and output procedures correctly. The initialisation calls were tested by checking the results produced for touch, light and rotation sensors. The correct calling of input and output procedures was tested by running the script with different combinations of inputs and outputs and then checking that the corresponding calls generated were correct.

3.2.7 Evaluation

The main aim of this project is to provide a way to use Lego Mindstorms as a teaching aid for synchronous reactive systems. The script, combined with SCADE2c, helps realise this by providing a simple process for creating code executable under BrickOS from a SCADE design. In its current state, the script supports all SCADE language constructs apart from safe state machines. Procedures are built-in to handle touch, light and rotation sensors, as well as motor outputs, control of the LCD and use of the RCX's internal speaker. If a designer using the script should require more input/output functionality then the relevant procedures can easily be written and added to the script.

Future work on the script should first involve providing support for safe state machines. The script interface could then be improved by reading the inputs and outputs directly from the SCADE2c generated header file. Currently they must be specified on the command line, which can be inconvenient if a project contains many inputs and outputs.

3.3 Communicating RCXs

A single RCX is quite limited in terms of its ability to interact with its environment. Three sensors and three actuators are enough for only simple designs. Therefore, it would be very useful if the RCX was capable of controlling more outputs and reading more inputs. Mindsensors [27] manufactures input multiplexors which increase the environment interaction an RCX can handle. However, none of these devices were available for this project so another method of expanding the RCX's I/O capabilities was implemented.

As described in section 2.6, BrickOS provides the ability for a RCX to send an arbitrary message to one or more other RCXs. Consequently, one RCX can

take control of another's I/O capabilities, thus providing it with at least six sensors and at least six actuators depending on the number of RCXs communicating. This protocol is developed without regard for the synchronous hypothesis and should not be considered a well-defined extension to the system, simply a method of extending the RCX's I/O capability. Now that the motivation has been introduced, an in depth description of the BrickOS features required will be given.

3.3.1 Description of BrickOS Features Required for Communication

Since only two RCXs will be communicating, BrickOS's integrity network layer will suffice. This is because messages will only be directed to one RCX; therefore, it is not necessary to identify individual RCXs. To send a message using the integrity layer, the function `lnp_integrity_write` is used. It takes two parameters, the address of the start of the packet and the length of the packet in bytes. Messages are received by setting up a message handling function. This function will be called every time an incoming packet is received and therefore it should be kept small since it will interrupt execution of other threads on a regular basis. Note that this interruption will not affect the synchronous hypothesis since the period at which the reactive kernel is called is kept constant by dynamically adjusting the pause time in the driving loop. The operating system is informed of the function used to handle messages by calling `lnp_integrity_set_handler` and passing the name of the message handling function.

The network protocol to be implemented is a master/slave communication with timeouts. Therefore, the RCX running the reactive kernel will be the master and the other the slave. The slave will simply process output instructions it receives and send sensor readings back to the master. Because communication between the two bricks will be happening asynchronously from the reactive kernel, a separate thread will be used on the master to adjudicate the communication. Threading in BrickOS is very easy to set up. Firstly, a variable must be defined of type `tid_t` to hold the process ID of the thread. Next, a call is made to the function `execi()` which starts executing a thread and returns a corresponding process ID. `execi()` takes as parameters, the function that will be run in the thread and its arguments, the priority of the thread and the stack size available to it.

3.3.2 Implementation

The main procedure (which encompasses the driving loop) requires very few changes to enable communication to take place. Firstly, it must call the procedure to set the incoming message handler and then start execution of the master communication thread. When an incoming message is received, the message handler is activated and copies the incoming data to a permanent location. It then sets a global variable recording the fact that a message has just been received.

The master communication thread consists of a non-terminating loop. The loop begins by recording the current system time before sending the outgoing actuator control packet. It then uses the `wait_event` feature of BrickOS to poll for a return message or a timeout. This is calculated using the global incoming

message variable and the time that was recorded before the message was sent. After it wakes up, it performs a check on the incoming message variable to determine whether a message has been received or if a timeout has occurred. If a timeout has occurred then this is recorded; otherwise, it resets the timeout variable representing the fact that normal communication has resumed. A short delay is then taken to help prevent message collisions before repeating the loop.

This method only provides a framework for communications - the main driving loop is responsible for building the packet that is sent to the slave. Similarly, it must also interpret the packet received from the slave.

An overview of the program running on the slave will now be given. The message handler is almost identical to the one running on the master brick; it stores the incoming data and records the fact that a message has just arrived. The main procedure, after setting up the message handler, consists of a non-terminating loop. It constantly polls the incoming message variable, and when a message arrives it continues executing. First, it sets up its outputs according to the packet received. Next, it builds an outgoing packet consisting of its sensor readings before sending this back to the master brick. It then waits for the next message to arrive.

A framework for the communication protocol is given in appendix C.

3.3.3 Evaluation

Using the implementation in practice proved to be quite reliable; however, theoretically it breaks the most important rule of designing synchronous reactive systems - the synchronous hypothesis. This is because inputs coming from the slave brick are not perceived synchronously with the local inputs on the master brick. The reverse of this is also true; actuator control signals are received on the slave brick a long time after they were issued by the reactive kernel. The order of this delay is about 10 times larger than the base clock of the reactive system (100ms). Fortunately, in the domain that is being dealt with in this project this is not a problem since a 100 milliseconds delay does not have a large effect on all but the fastest reacting models. Therefore, as stated in the introduction, this addition should not be viewed as a well-defined feature of the system but simply as a way of extending the RCX's interactive capabilities.

Having said this, it is possible for the synchronous design to be somewhat protected against the problem of communication loss. By simply adding a boolean input to the system that reflects the status of communication between the bricks, the system can enter a safe state when communication fails. This would also involve the slave brick's code being extended to set its actuators into a safe state when communication fails. Since the slave brick can also detect a timeout, this is a reasonable requirement.

The 100 millisecond delay mentioned earlier is based on the amount of data that needs be transmitted back and forward to read three sensors and control three actuators. This would be reduced if less sensors or actuators were being used. Also, the timing values being used at the moment are quite lax to ensure that message collisions do not occur frequently. Therefore, more aggressive timings may be possible but determining their correctness would require extensive experimentation.

Chapter 4

A Brick Sorting Robot

After developing the SCADE compilation script, it could then be put to use on the first robot; a brick sorter. The idea behind this robot is as follows; first new bricks are introduced to the system from an external supply. Next, a brick's colour is checked using a light-sensitive device and transported to a basket which reflects its colour. The basic system requirements are quite simplistic, so two additional requirements were specified to introduce some more complexity. One further requirement is that some system monitoring must be performed so mechanical jams etc. can be detected. Furthermore, if a jam is detected then the operator of the system should be alerted in some way before being able to resume the system.

4.1 Problem Analysis

The problem readily divides itself into two subsystems: a sorting subsystem and a monitoring subsystem. The sorting subsystem is the more complex of the two and will be investigated first.

There are numerous different ways to transport bricks around the system by, for example, using a robotic arm. However, for this application it makes sense to use a conveyor belt, since this will provide the required functionality with minimal resources. Bricks will be introduced to the system at one end of the conveyor belt and sorted at the other. In the middle, a light-sensitive device will be placed above the conveyor belt so that the colour of a brick can be measured as it passes along the belt.

There are also many methods available for implementing a sorting mechanism for the bricks. The method chosen is to have an actuator that pushes certain colour bricks off the belt before they reach the end. This method is the simplest since it makes use of the end of the conveyor belt as a natural sorting bin.

Monitoring in the system will take the form of checking the speed of the conveyor belt since this is likely to be where a jam will occur. This can easily be done using a rotation sensor. From the hardware point of view, it will be necessary that when the conveyor belt stops moving, the motor driving it will be physically disconnected in some way, thus preventing it from burning out. Eventually the jam will be detected and the motor switched off, but it is before

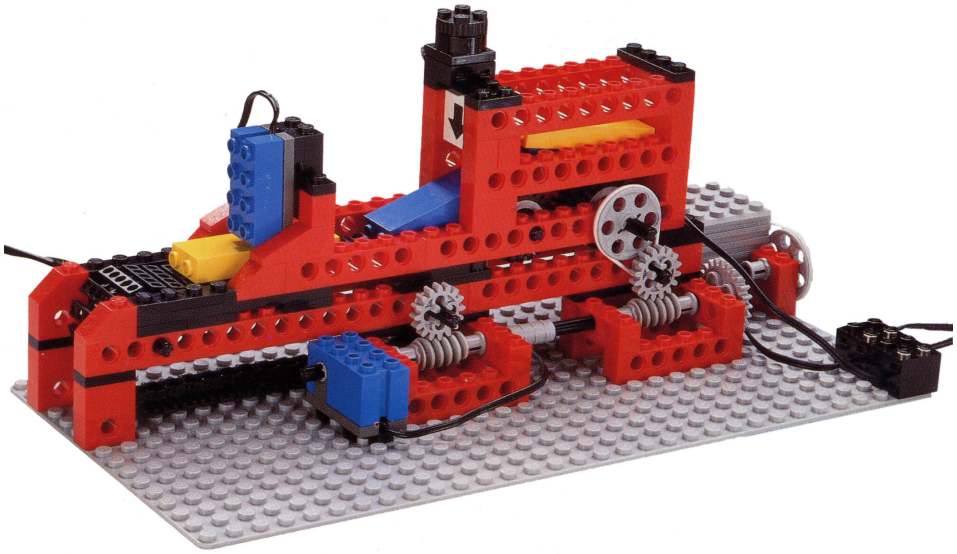

Figure 4.1: Original Lego Brick sorter - [26]

this event occurs that the hardware based protection must take effect.

4.2 Design

4.2.1 Lego Design

The basic design of the system has come from Lego model number 9701-6 [28]. A photo of the model is given in figure 4.1. This design provides a brick loading mechanism for introducing bricks to the system and a conveyor belt which moves bricks from the loader under a light sensor. However, the system has no sorting mechanism so this will be designed and implemented as an add-on at the end of the conveyor belt. The final model is shown in figure 4.2 and the sorter subsystem in figure 4.3. Each time a brick is to be pushed off the conveyor belt, the main driving cog completes a full rotation, thereby extending its arm over the conveyer belt and retracting it again so bricks can pass by it. The sorter arm detects when it has returned to the start position by using a rotation sensor. Due to the chosen gearing, and the granularity of the rotation sensor (16 divisions per rotation), the rotation sensor records 27 intervals per full rotation.

The same method can be used to calculate the update interval of the rotation sensor measuring the speed of the conveyor belt. Using the gearing ratios, it can be seen that under normal conditions the rotation sensor will update its reading every 300 milliseconds. The period at which the sensor is polled will be a bit more than this to ensure that false belt jams are not generated. This means that a motor could spend more than 300 milliseconds trying to turn a jammed axle, which may be enough to damage the motor. Hence, there must be special hardware present to deal with a jam. This is accomplished using a special clutch gear, which allows the cog driving it to turn independently of the

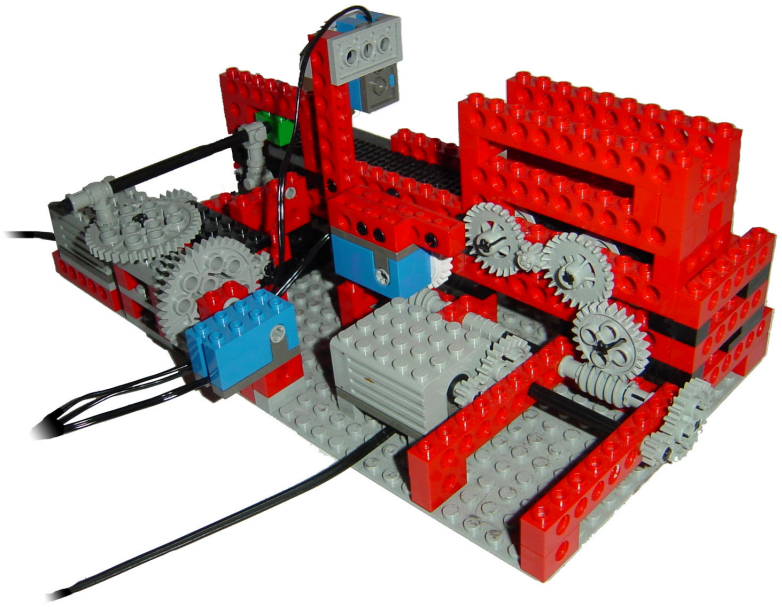

Figure 4.2: The Brick sorter with a sorting subsystem

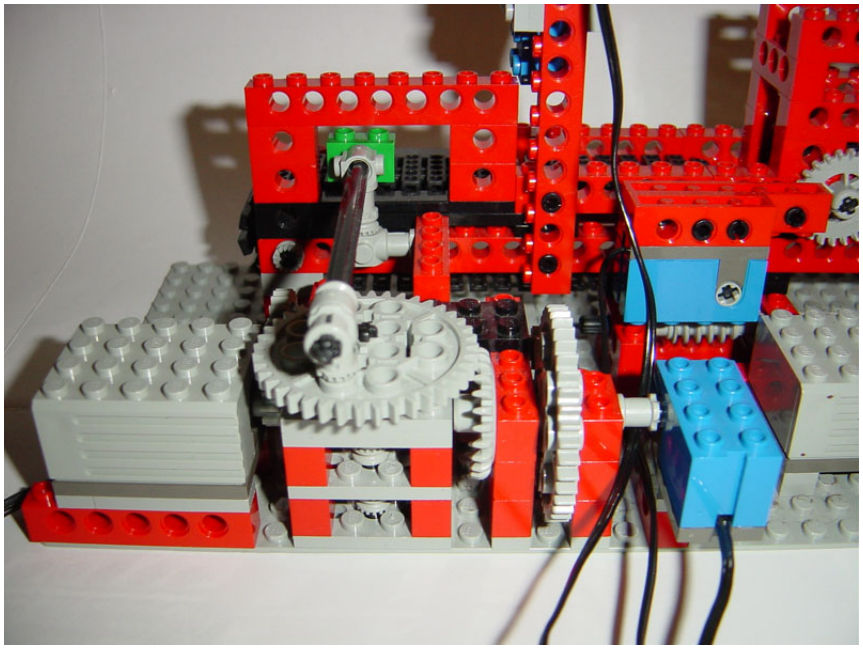

Figure 4.3: Sorter Subsystem

axle once a certain torque is applied. Thus the rotation sensor will detect that the axle has ceased turning and can switch off the conveyer belt motor. In the interim stage when the motor is turning and a jam is present, no harm will come to the motor since it will not be turning the axle that drives the belt, only the disengaged cog.

The system requires the following resources: three actuators and four sensors. However, due to the extra I/O capabilities provided on the RCX itself, it is not necessary to make use of a second RCX and the communication protocol developed in section 3.3. Two of the actuators are motors; one powers the conveyor belt and the second is used to control the sorter. The third actuator is the internal speaker in the RCX which is used as an alarm to signal when a belt jam occurs. Three external sensors are required, one light and two rotation. The light sensor is used to detect the colour of a brick. As mentioned earlier, one rotation sensor is used to read the speed of the conveyor belt and a second is used so the sorter arm can return exactly to its home position. The fourth sensor is one of the control buttons on the RCX used for resuming the system.

The proposed operation of the system is as follows: first a brick is moved out of the loader subsystem and placed on the conveyor belt. The conveyor belt then moves a brick underneath the light sensor which takes its reading. Next, depending on the colour of the brick, when it reaches the sorter arm it will either be pushed off the belt or allowed to move to the end. If at any time the belt's speed drops below a certain threshold then the system is put into a safety state where all actuators are switched off and an alarm sounds. This continues until the system operator removes the element that caused the jam and presses a touch sensor to resume. By the time the first brick is sorted, a second brick has already been loaded onto the conveyor belt and the process repeats.

4.2.2 SCADE Design

The design of the system is based on a top-down hierarchical order. As such the system will be described in terms of this hierarchical ordering. The discussion will begin at the highest level node.

Bricksort node

The highest level node, named `bricksort` is shown in figure 4.4. This node is responsible for taking all inputs and outputs of the system and passing them to the relevant sub-nodes for computation. A brief overview of the nodes will be given before a detailed discussion. The function of the `belt_safety` node is to monitor the conveyer belt for jams. `Input_filter` and `output_filter` set the system into a safe state when jams are detected. Finally, the node `sort` is responsible for detecting a brick's colour and activating the sorter arm to sort it accordingly. A full description of the SCADE design in given in appendix 4.

Sort node

The sort node is shown in figure 4.5. It begins with a simple comparison between the light sensor reading and a threshold. This threshold represents the reading at which the system will consider there to be a brick under the light sensor. Hence, this comparison operator will provide a true output for the duration of

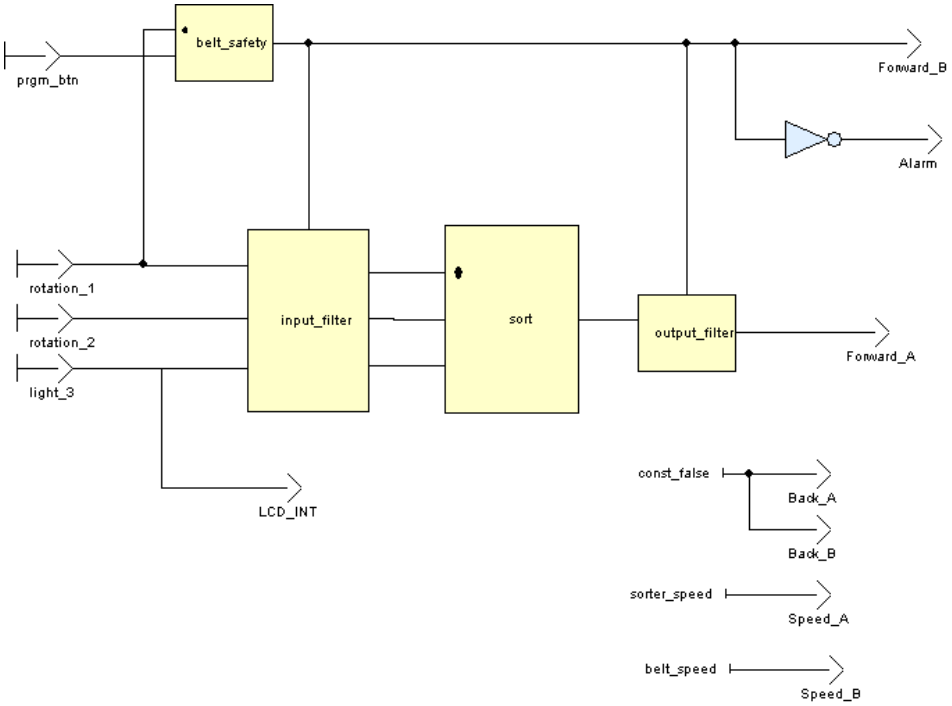

Figure 4.4: Bricksort Node

time a brick is under the light sensor. The output from the comparison is passed to a `rising_edge` node, which produces an output of true when its input has just performed a rising edge. Therefore, this will provide a true output for the first cycle where there is a brick under the light sensor.

The `rising_edge` output is used as the reset input for the node `compute_max`. After being reset, this node will output the largest value its input has taken so far. The input in this case is the light sensor and when the brick moves away from the light sensor, `compute_max` will have the highest reading the brick gave on its output. It is necessary to check a brick's reading using this method since, if only the first reading was taken, then it could recognise a higher valued brick as a lower valued brick. This can occur where the first reading is taken when the brick is not completely under the light sensor. It is only necessary for `compute_max` to be in operation when there is a brick under the light sensor. This is an ideal use for a separate clock and as such, `compute_max` is conditionally activated only when there is a brick under the light sensor. This is the first of two internal clocks that will be seen in this node; the second is responsible for activating `push_brick`. Therefore, the two clocks in the system are the output of the first comparison operator and the output of `on_until`.

The output of `compute_max` is now compared to a second threshold, which defines the value a brick must be over to be in the light coloured category. If it is false then this part of the system has completed its task and the brick will be allowed to run off the end of the conveyor belt. However, if it is true then a light coloured brick is present which must be pushed off the conveyor belt earlier.

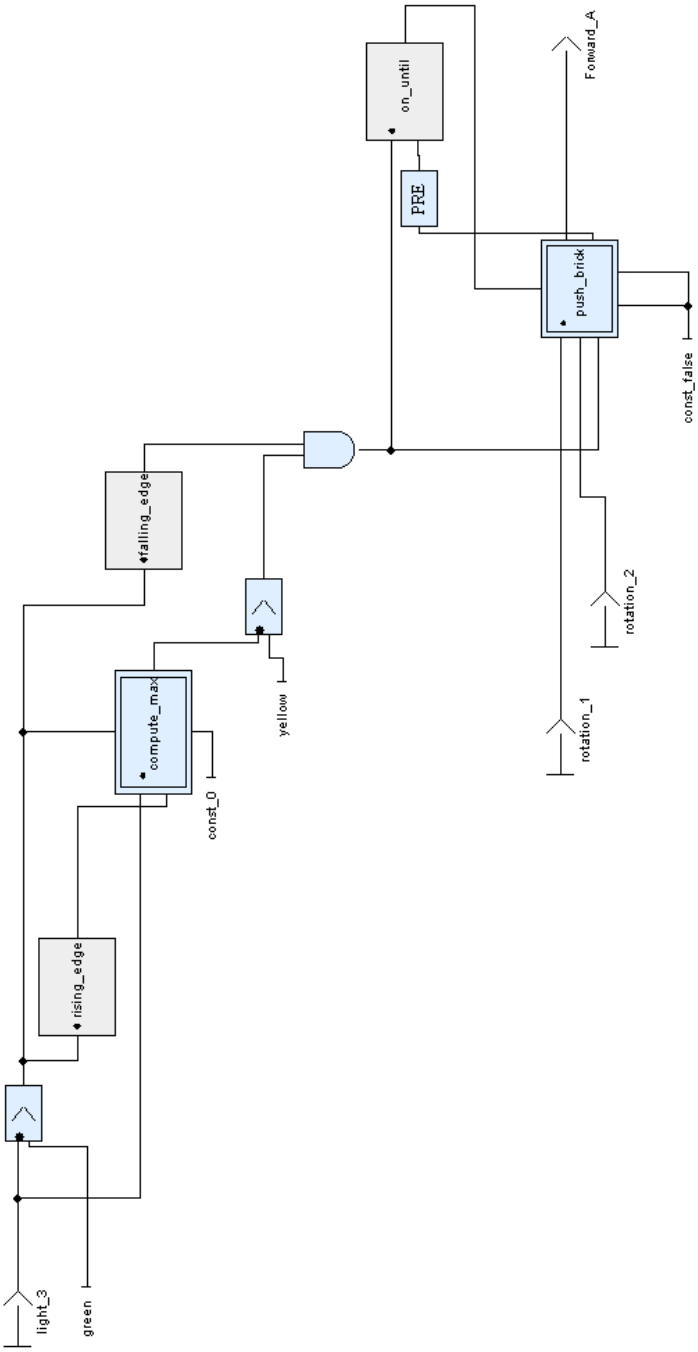

Figure 4.5: Sort Node

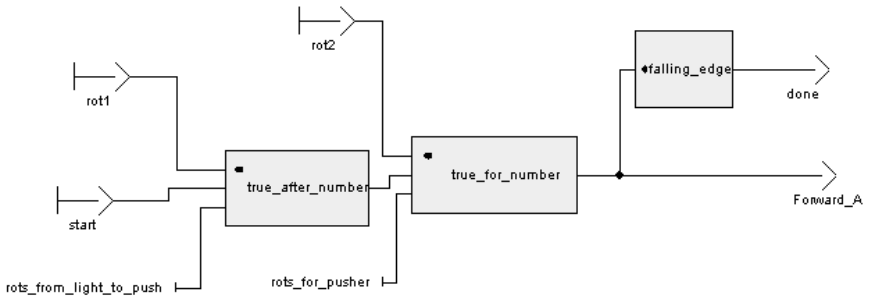

Figure 4.6: push_brick node

When the brick is no longer under the light sensor, the initial comparison will become false and cause the `falling_edge` node to output true. The output of the `falling_edge` node and the brick colour comparison are passed as inputs to an AND gate. Therefore, when the output of the gate is true, the brick has just moved away from the light sensor and it is a light colour, so it must be pushed off the conveyor belt.

The last part of the sort node is concerned with activating the sorter arm when the brick is in the correct position. The output from the AND gate mentioned earlier is passed into the first input of the node `on_until` which, once its first input turns true, will produce a true output until its second input turns true. The output of `on_until` (the second clock in this node) is used as the condition for activating the node `push_brick`. The `push_brick` node will be described later. For now, it is only necessary to note that it defines when it should switch itself off since its output is passed into the second input of `on_until`. The `push_brick` node also uses the output of the AND gate as a reset input to indicate when a new sorting process must commence. In addition it requires the two rotation sensors as inputs which it will need to activate the sorter arm at the correct time.

Push_brick node

The `push_brick node` (figure 4.6) is activated just after a brick has left the light sensor and is required to be pushed off the conveyor belt. The first node, `true_after_number` provides a true output after its input has increased by a certain number. In this case it is the number the rotation sensor reading must increase before the brick will be placed in front of the sorter arm. When the number has been reached, an output of true is produced which is used as a starting signal for the node `true_for_number`. This node effectively switches on the sorter arm for the amount of time it takes for it to complete a full rotation (returning to its home position). The amount of time it is switched on for is defined by the second rotation sensor's reading. When the number of rotations has been reached, `true_for_number` will output false resulting in the sorter motor being switched off. This will also trigger a `falling_edge` node, which is used to detect when the sorting process is finished. It communicates this out of the node by setting the `done` output of `push_brick` to true. As mentioned earlier, the `push_brick` node deactivates itself and it does this by using the

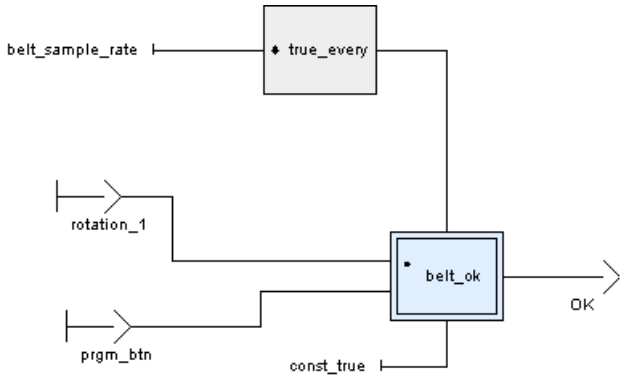

Figure 4.7: belt_safety node

`done` output. When `done` becomes true, the second input of `on_until` will also be true and therefore `on_until` will produce a false output. Since the output of `on_until` controls `push_brick`'s activation, `push_brick` will be deactivated.

Belt_safety node

We will now return to the top level of the design and analyse the `belt_safety` node (figure 4.7) which is responsible for ensuring that the system enters a safe state if a jam should occur on the conveyor belt. The sample rate of the system is much faster than the rate at which the rotation sensor monitoring the conveyer belt's reading increases. In other words, on each cycle of the base clock, the rotation sensor reading will not have increased even though the belt is still moving. Therefore, the rotation sensor measuring the belt must be sampled at a slower rate. This is another ideal use of clocks in a dataflow design and as such will be modelled in this way (making it the third clock in the design). A constant, `belt_sample_rate` is provided as input to a node called `true_every`. This node will provide a true output with period defined by its input, i.e. its output will be true every 20 cycles, if its input is 20. `true_every`'s output is used as the activation condition for the node `belt_ok`.

The `belt_ok` node checks that the rotation sensor responsible for monitoring the conveyor belt increases on each cycle that `true_every`'s output is true. If it does then `belt_ok`'s one output, `OK`, will remain true. However, if the rotation sensor value has not increased since the last time `belt_ok` was activated then its output will become false. The only way to reset this is for `belt_ok`'s other input, `prgrm_button` to become true. This signals that an operator has removed the problem and is now resuming the system. The system must be setup as it was when the jam occurred since the system is only paused when a jam occurs, not reset.

Now that the behaviour of `belt_safety` has been analysed, its effect on the system as a whole will be examined. As can be seen in figure 4.4, the output of `belt_safety` is used to control the conveyor belt motor, an alarm and is also used as input to the nodes `input_filter` and `output_filter`. When `belt_safety`'s output becomes false the system will move into a safe mode, turning off its actuators. One is obviously switched off since it depends directly

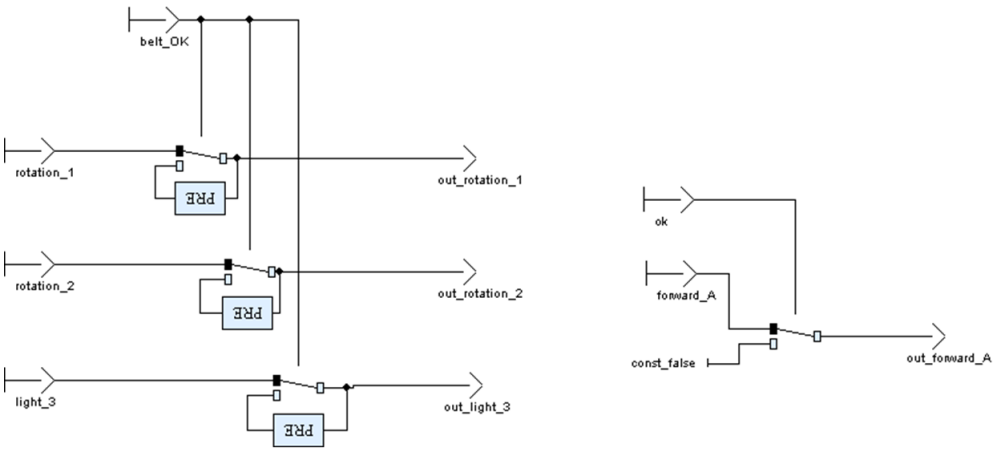

Figure 4.8: A) input_filter node, B) output_filter node

on the output of `belt_safety`. The other is controlled by `output_filter` (figure 4.8B). This will deactivate the motor if the output of `belt_safety` is false, or if it is true, it will let the signal pass through unaffected. While in this safe state the inputs must be protected from changing, since this could cause the system to malfunction as the motor outputs are being artificially stopped. For this reason the node `input_filter` (similar to `output_filter`) is used to hold the inputs at their previous values should the output of `belt_safety` become false. The `input_filter` node is detailed in figure 4.8A[1]. It is simply a switch on the output of `belt_safety`: if it is true then the inputs are passed through undisturbed but if it is false then the inputs are held at the value they had when the system was still functioning correctly (using the `PRE` operator described in section 2.3).

4.3 Implementation

The SCADE design was translated into code executable on the RCX using the methodology designed in section 3.2. First, SCADE2c was run on the project files and the output of this was used as the input to the SCADE2lego script. SCADE2lego was called as follows:

```
Scade2lego bricksort -i 3 rotation_1 rotation_2 light_3
-o 8 Forward_A Back_A Speed_A Forward_B Back_B Speed_B
LCD_INT Alarm
```

The control buttons on the RCX are unavailable in the script since they would not normally be used; however, in this case the `program` button is needed as an additional input. The input procedure is shown below. Observe that the variable in the context that it updates is liable to change if SCADE2c was rerun.

[1]Upsidedown SCADE operators, like the PRE operators in figure 4.8A, simply reflect the fact that their inputs are on the righthand side of the symbol (and their outputs on the left).

```
void I_btn_prgm(_C_bricksort* ctx, bool V)
{
   ctx->_I3_prgm_btn = V;
}
```

It is called in the main driving loop as follows:

```
if (PRESSED(dbutton(), BUTTON_PROGRAM))
    {I_btn_prgm(&the_ctx, true);}
else
    {I_btn_prgm(&the_ctx, false);}
```

4.4 Testing

4.4.1 Testing Methodology

The testing methodology used in this section will be applied to both this robot and the one built in chapter five. In order to develop the methodology, first the chain of events leading from design to implementation must be studied. At the highest level is the SCADE design, which may or may not be correct. The design can then be simulated and verified inside SCADE, but since the verification plug-in was unavailable, only simulations could be performed. The first link in the chain is SCADE's internal code generation. This is certified to be DO178b compliant and, therefore, its correctness has already been proven. Next, the SCADE2Lego script produced in section 3.2 is run on the output of the code generation. The script has already been tested in section 3.2.6 and for this reason we will assume that it is correct. The last part of the compilation chain is to compile the output of the script into machine code that is executable on the RCX. This uses the long established GCC tools whose correctness have been proven over time. What remains at the end of this compilation process is a complete robot system with both hardware and software components in place. Therefore, the robot can be tested as a black box. In many industries including aviation, automotive and telecommunications, this is normally how solutions are tested and for this reason we will apply the same style of testing to this project.

Since we are assuming that the compilation process is correct, when the design produced in SCADE is simulated it should directly reflect the behaviour of the final robot. Using the simulation for testing is very useful for finding functional bugs in the program. However, the robotic system must be tested as a whole to discover if the design's inputs and outputs are correctly mapped onto the sensors and actuators in the real world. An example of this would be during simulation the rotation sensor reading was always considered to be increasing. It was only when applied to the real world that it became apparent that the reading can decrease as well as increase since the motor it is monitoring can turn in both directions.

In conclusion, testing in this project will take the form of building test cases that the robot must pass. It should be noted that the simulation tool is a very powerful testing method. However, due to the argument for black box testing, only test cases on the completed robot will be considered.

4.4.2 Test Cases

The test cases the robot must pass are as follows:

- A brick is successfully loaded onto the conveyor belt and sorted to the correct bin based on its colour.
  - PASSED - this was trivial to test; the system is simply observed noting that no bricks were incorrectly sorted.
- It is not possible for two bricks to pass through the system in such quick succession that the control algorithm fails.
  - PASSED - the speed at which the loader introduces new bricks to the system is slow enough to ensure this problem will not occur.
- If the conveyor belt becomes jammed at any point during the system's execution, then the system should move into a safe state.
  - PASSED - the system was jammed at a variety of points each reflecting the different positions a brick or bricks could be in.
- From any point at which the system is set into a safe state, the resume button can be used to successfully resume operation.
  - PASSED - for each of the positions the system was jammed in as described above, the resume function was tested to ensure that the system continues functioning as normal.

4.5 Evaluation

The secondary aim of this project was to design, build and investigate robots built according to the synchronous approach. Furthermore, these designs should try to highlight key areas of dataflow synchronous reactive systems and evaluate them. In this section SCADE constructs such as the conditional activation operator, the "followed by" operator and the `PRE` operator have all been used, so the reader can see their role in a real-world context. Furthermore, the design has made use of hierarchical decomposition and because of this it has made explanation of the design considerably easier.

However, some aspects of the dataflow approach have been highlighted that are not so successful. The node `push_brick` is responsible for the timing and pushing of a brick off the conveyor belt. To model this, an idea of state is needed to keep track of the brick's location on the belt at various times. In the design, this transitioning through states is modelled as transitions through nodes, whereby each node has an input that informs it of when it should begin computing. This input is passed to it as an output of the last node and a chain is built between nodes acting like transitions in a state machine. This is an ugly use of the dataflow formalism and as such it would be much better expressed in a state based formalism such as safe state machines. Although it would be possible to do this within SCADE, the chapter will focus only on the dataflow formalism. An example involving state based control will be shown in the next robot.

However, there is an upside to this transitioning through nodes as if they were states. While the progress of a brick is being tracked so it can be pushed off the conveyor belt, another brick may enter the system and have its maximum value computed as it passes under the light sensor. So long as the old brick has been pushed off the conveyor belt before the new brick has passed the light sensor, the behaviour of two bricks in the system will be modelled correctly. This is a reasonable requirement since the speed at which the loader introduces new bricks to the system can be controlled by changing its gearing, therefore ensuring that two bricks cannot be in the same part of the system at the same time. This parallelism is undeniably useful as it significantly increases the throughput of the system.

Chapter 5

A Line Following Robot with Obstacle Avoidance

To provide a good overview of the different types of robots that can be built with Lego Mindstorms and programmed using synchronous languages, this section will focus on a robot that interacts with an external environment. This is opposed to the environment of a reactive system changing internally; an example of this would be the brick sorter (section 4). This project has already explored a reactive system of the former type, the line following robot of section 2.7. Since that robot was very simplistic, in this section a more complex robot of this variety will be built.

It is proposed that this robot will build upon the ideas developed in section 2.7. Therefore the requirements for this robot are as follows. Firstly, it must be capable of following a line. Secondly, it should be able to detect an obstruction on the line and manoeuver around it.

This robot will also demonstrate some features of SCADE that have not been described yet. In addition, the communicating RCX protocol developed in section 3.3 for expanding the RCX's input/output capabilities will be used.

Due to the size of the problem and the isolation between the software and hardware analysis, design and implementation, first the hardware side and then the software side for this robot will be presented.

5.1 Hardware

5.1.1 Problem Analysis

The hardware design for this robot is made particularly challenging by the need for the robot to perform precise maneuvers. This is necessary since the robot cannot calculate its position, it can only deduce where it is by the moves it has made so far. If these moves are inaccurate then an obstacle will not be avoided correctly. This complexity will be dealt with in the robot's wheelbase and, to design this, the different kinds of moves it is required to make must be considered. Firstly, the robot must be able to move forward and back in a perfectly straight line. In addition, it must be capable of turning on the spot by exactly 90 degrees. Furthermore, if the same turning method is to be used

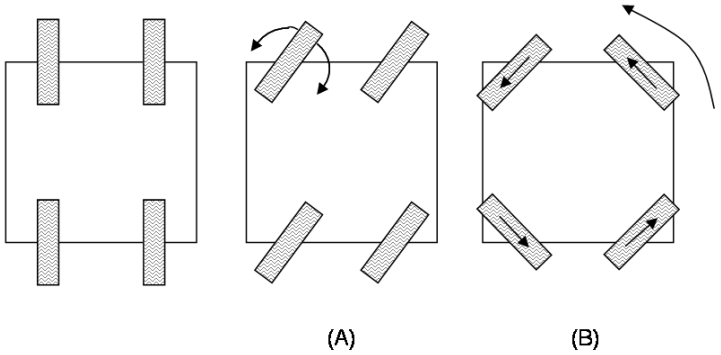

Figure 5.1: Different types of synchro drives

for following the line, then it must be able to turn by an arbitrary amount (e.g. keep turning until a light sensor detects it has rejoined a line).

The other major hardware feature the robot must provide is a sensor array that is always in front of the direction of travel. The sensor array must provide facilities for following a line and detecting an obstacle. Line following has already been discussed in section 2.7 and as such it will suffice to say that the two light sensor solution will be implemented.

5.1.2 Lego Design

A variety of wheelbases will now be assessed in an attempt to find the one most suitable for this application.

Firstly, the differential drive method (implemented in section 2.7) was considered. In this design the powered wheels' directions are fixed so the robot performs turns by supplying different amounts of power to different wheel sets. A caster wheel is usually used on the back of the robot for stability and to provide a tight turning circle. The advantage of this method is its extreme simplicity; however, it is very difficult to perform precise turns and the caster wheel can cause problems if the robot is required to move backwards.

Next, a skid drive was assessed. This design is very similar to the differential drive except there is not a non-powered wheel on the device. This is the drive method of tracked vehicles etc. To perform a 90 degree turn using this method, one side of the vehicle is powered forward and the other in reverse. The advantages and disadvantages are very similar to the differential drive method being a general lack of precision. The exception is that backwards movement is not a problem.

A car drive was also considered. This method is identical to a standard automobile, the rear two wheels are fixed and the front two handle the steering. However, once again, the ability for this type of drive mechanism to make precise turns is quite limited. The angle of the steering wheels must be closely monitored since a small error in their orientation can result in big errors in its movement. Furthermore, if the two rear wheels are powered together, then it is almost impossible for this robot to perform precise 90 degree turns.

Finally, a synchro drive was considered. This design is radically different to the others mentioned so far, because the orientation of the all the wheels on the robot changes to perform different maneuvers. Normally, a synchro drive moves in different directions by aligning all its wheels in the relevant direction and powering the driving motors (as shown in figure 5.1A). However, a sensor array is needed in front of the robot no matter what its direction of travel. For this reason a traditional synchro drive would not work unless the sensor array could be rotated round the robot. Since this is unnecessarily complex for the application, a different type of synchro drive will be investigated. In this method, when the robot performs a turn, it rotates all its wheels by 45 degrees corresponding to figure 5.1B. It then supplies power to two of the wheels and reverse power to the other two causing the robot to rotate on the spot. When it finishes rotating, the wheels turn out so that they are all identically aligned and the robot can move forward again. This method allows the robot to perform a turn on the spot with good precision (and no turning radius). The disadvantage is the speed at which it performs a simple turn, since it must turn its wheels in, rotate and then turn them back out again.

An evaluation of the possible wheelbases leaves the synchro drive as the clear choice. This is purely due to its ability to perform turns of precise angles. If a simpler, faster solution was desired, then the skid steer drive would fit the requirements well.

The synchro drive method requires the following resources to work. Three motors are needed, one to alter the orientation of the wheels and two to power the wheels. To ensure that the wheel orientation remains correct, a rotation sensor is also needed for monitoring the first motor. To turn through exact angles, a second rotation sensor is required to monitor either of the motors powering the wheels themselves.

The robot sensor array needs to provide the following functionality. Firstly, it must be capable of following a line which can turn in either direction. This will be implemented by employing the same solution as used in section 2.7, therefore requiring two light sensors. Obstacle avoidance will be provided using a bumper that triggers a touch sensor. In addition to the bumper, an experimental method for obstacle detection will be implemented that allows the robot to detect an obstacle before it bumps into it. This will be accomplished by using a light sensor as a range finder. When a light sensor comes closer to an object, no matter what its colour, its reading increases. Therefore, this provides an additional way to detect obstacles. Emitting extra infrared light increases the effectiveness of the light sensor as a range detector. This can be accomplished using the infrared transmitter on the RCX used for communication and downloading programs. However, the transmitter is not available in this application because, due to the number of sensors required, the infrared devices are needed for communicating between the RCXs.

5.1.3 Implementation

Implementing the synchro drive in Lego proved to be quite difficult. The first level of difficulty came from implementing the mechanics required to turn all four wheels simultaneously. This required careful consideration for routing the axles around the robot. Furthermore, a large amount of torque was needed to turn all the wheels. This was produced using a large number of gear stages.

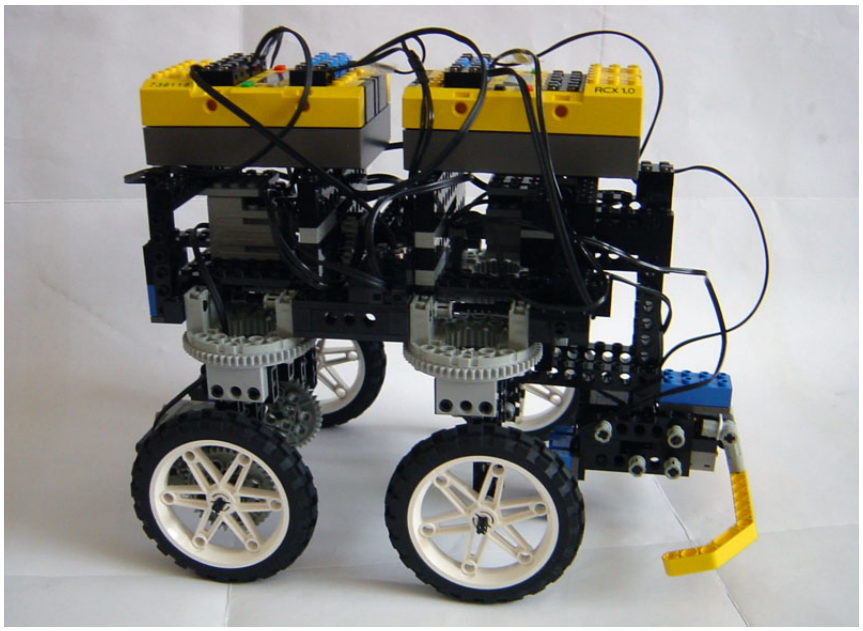

Figure 5.2: The robot capable of following lines and avoiding obstacles

However, it resulted in a robot that was exceptionally slow at turning. To provide a remedy, more torque was created at the motor stage by mechanically connecting two motors together and using them to drive the initial stage instead of one motor. Now that more torque was being provided, some of the gear stages could be removed resulting in faster turns.

The next problem to arise was how to power the driving wheels when a layer of mechanics must exist on top for the purpose of turning them. This was resolved by directly mounting the motors on the wheel supports below the turning mechanics. The motors drive 8t gears, which turn 40t gears that are directly connected to the wheel axles. This creates a gear ratio of 1:5 and is enough to make a standard Lego motor capable of driving the wheel.

5.2 Software

5.2.1 Problem Analysis

The two software requirements for this robot are quite distinct and as such will be modelled in this way. The only transitions between the two operating modes occur when a robot bumps into an obstacle and when it rediscovers the line after manoeuvring away from it. Although the requirements do not state it, it is clear that this problem will require, in part, a state based solution. This raises the question of which state based formalism within SCADE should be utilised. As mentioned in section 2.4, SCADE supports both simple state machines and safe state machines. Safe state machines are an order of magnitude more complex than the simple machines and for this reason are out of the scope of this project.

Simple state machines on the other hand provide a basic way for introducing some state control into a dataflow design. Simple state machines in SCADE have not been discussed before and since they will be used in the following design, a short explanation of their functionality will now be given.

State machines are constructed from three different components: an initial state, one or more internal states, and transitions. There must be exactly one initial state per machine. This is where execution begins, and leading from it (and other states) transitions are constructed so control can pass between the states. Transitions are given boolean expressions which represent the event that must occur for control to be passed from the state on one end of the arrow to the state on the other end of the arrow. Boolean expressions can either be true, indicating that a transition should be taken immediately, or some function of the state machines' inputs. State machines have as many outputs as states (excluding the initial state) and each maps to an internal state of the machine. Therefore, state machines communicate their internal state back to the dataflow design by setting the output corresponding to the current state to true and all others to false. Each state machine has an initialisation input and when this is set to true, the machine will reset on every clock cycle. When it becomes false, the machine begins executing.

The obstacle avoidance to be performed by the robot will be relatively simple. Therefore, for it to work correctly, the obstacle and its placement must meet certain requirements. Some examples of allowed obstacles are shown in figure 5.3. The part of the obstacle that obstructs the line (i.e. the part the robot will bump into) must be the furthest part of the obstacle to extend in that direction. In the direction at a right angle to the line, the obstacle may extend an arbitrary amount. Furthermore, the edges of the obstacle must be straight, containing no indents that the robot could get stuck in. The reason for placing such stringent restrictions on an obstacle's shape is because it is not intended for the path finding around an obstacle to obscure the rest of the system. Now the restrictions on an obstacle have been described, the navigation algorithm for avoidance (figure 5.3) will be presented.

The robot begins by backing up when it encounters an obstacle. This is so that when it rotates, it does not hit anything that might effect the exact rotation required. Next, it rotates to the left and moves forward for a set distance. It then turns to the right and moves forward. If the obstacle is large, then the robot might encounter it again at this stage (figure 5.3B) in which case it repeats this process again (backup, turn left, move forward, turn right, move forward). If it does not bump into the obstacle while driving forward, then after a set distance it will stop and turn right. Finally, it drives forward expecting to rediscover the line. If the obstacle is long then it is possible it will bump into it again (figure 5.3C). If this occurs then it backs up, turns left and repeats the last three steps mentioned above (forward, turn right, forward). This can repeat an arbitrary number of times. When it finally makes contact with the line, it turns left and continues to follow it.

5.2.2 SCADE Design

The design methodology is similar to that of the brick sorting robot, it is structured using hierarchical decomposition with higher-level nodes considered more abstract than their internal ones. However, there are some exceptions in this

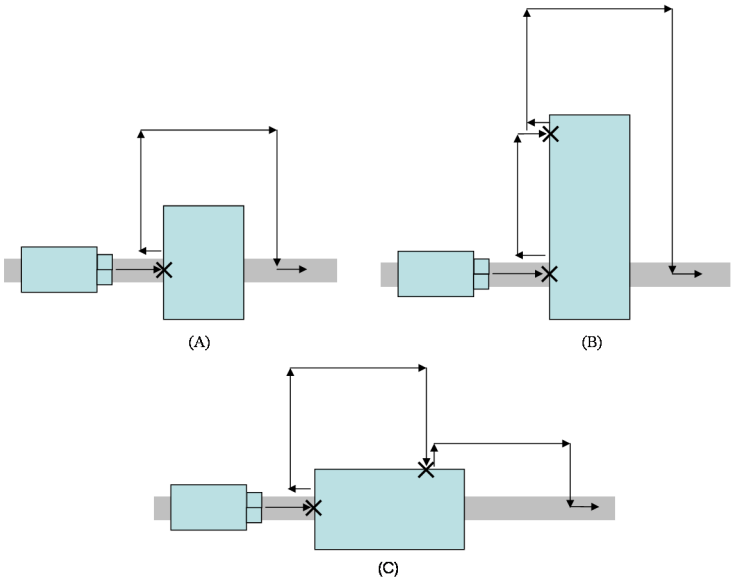

Figure 5.3: The obstacle avoidance algorithm

design since the main control for the robot must be placed at a low level in the design. This is because the control is modelled as a state machine, and these require a large amount of feedback from their outputs to determine input values. For example, if the state machine has just transitioned to a state where the robot must move forward for a certain distance, then it will be informed of when the distance has been travelled by using a rotation sensor. The number of rotations must be counted from when the state machine first enters the move forward state. This condition could be detected by a `rising_edge` node and then passed to a counter which would output true when the required revolutions were reached. Despite being a trivial example, it applies to some methods used in the design and shows how output feedback must be used for simple state machines. The full SCADE design for this robot is provided in appendix 5.

Synchro node

The `synchro` node (figure 5.4) is the highest level node in the system and it is named after the wheelbase chosen for this robot. It is responsible for processing the inputs and then passing them to the node `compute_state`. `Compute_state` contains the state machines and uses these to compute the system's global variables. These global variables are then used to derive the output values for the system. This method produces cleaner and easier to understand designs as opposed to passing the values as outputs of `compute_state`. Furthermore, it lets a designer produce a solution without worrying about the mapping of outputs to actuators in the real world. Different variables are set for the different actions that the robot should perform. In the synchro node, these are mapped to the system outputs in a way that makes the robot perform the relevant action.

Both inputs and outputs are controlled by filters, in a way similar to the

belt monitoring system designed in section 4.2.2. The monitoring in this case is of the communication between the two bricks. If a timeout is detected, then the `comms_ok` input will become false with the result that `input_filter` will hold all inputs at their previous values and `output_filter` will switch off all actuators.

The last part of the synchro node to discuss at this level is the input processing. The two rotation sensors are passed through unaffected since their readings will be required at a low level of the system. The touch sensor and `light_6` are used for obstacle avoidance. The light sensor's reading must be interpreted first and the node `obstacle_present` does this. Both these signals (the output of `obstacle_present` and the touch sensor) are then combined and passed to the `compute_state` node. The last two inputs, `light_4` and `light_5` are used for line following. As such they have their values pre-processed to pass a reading of true if they are over the line or false if not to `compute_state`.

Now, the `obstacle_present` node (figure 5.5) will be discussed before proceeding to the core of the system, `compute_state`.

Obstacle_present node

This node performs a simple function, first it calculates the ambient light value when the system is started and then it uses this ambient value and the current light sensor reading to detect obstacles. The ambient value is stored using the "followed by" operator, which on the first cycle of the system, lets the value of `light_6` through and stores this in the local variable `ambient`. On every cycle of the system after that, `ambient` is simply assigned to itself and therefore its value never changes.

A possible obstacle can be detected when the value of `light_6` becomes large in comparison to `ambient`. After experimentation, it was found the correct value of this is approximately 10. Therefore, when `light_6` is greater than the local variable `ambient + 10` the node has detected an obstacle and will set its output to true indicating this.

The state machines: state_control and turn_robot

To continue the analysis of the system by means of hierarchical decomposition, the node `compute_state` should be discussed next. However, before this the two state machines that are part of `compute_state` will be discussed since they are conceptually on a more abstract level.

The machine `state_control` (figure 5.6[1]) begins in the state forward where the robot moves forward until it encounters any of the conditions that will change the movement of the robot. This is where the two distinct parts of the system are separated: line following and obstacle avoidance. If state machines supported hierarchal decomposition this would be a good use for it but the formalism only supports "flat" designs. Line following will be explained first. Depending on which light sensor has moved away from the line, a transition will be taken to either `turn_right_and_find_line` or `turn_left_and_find_line`.

[1] All forward states in the state machine state_control represent the same action. They are only named differently since state names must be individual. The same applies to reverse, turn left etc.

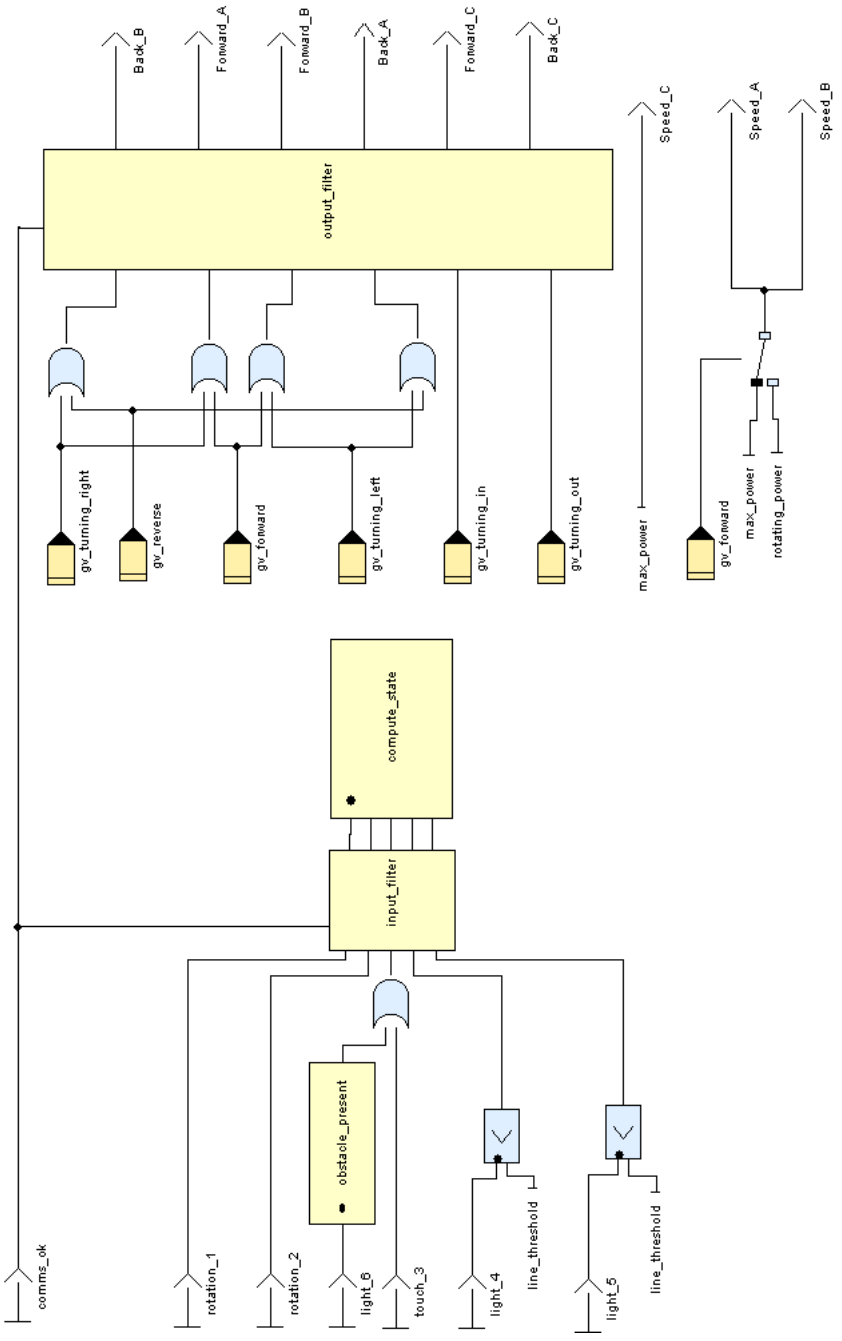

Figure 5.4: synchro node

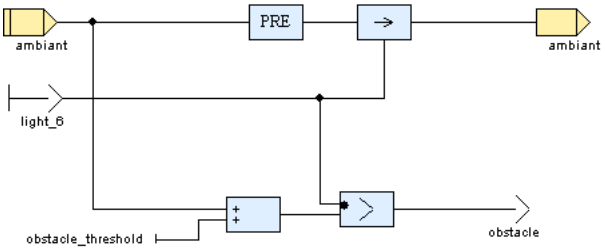

Figure 5.5: obstacle_present node

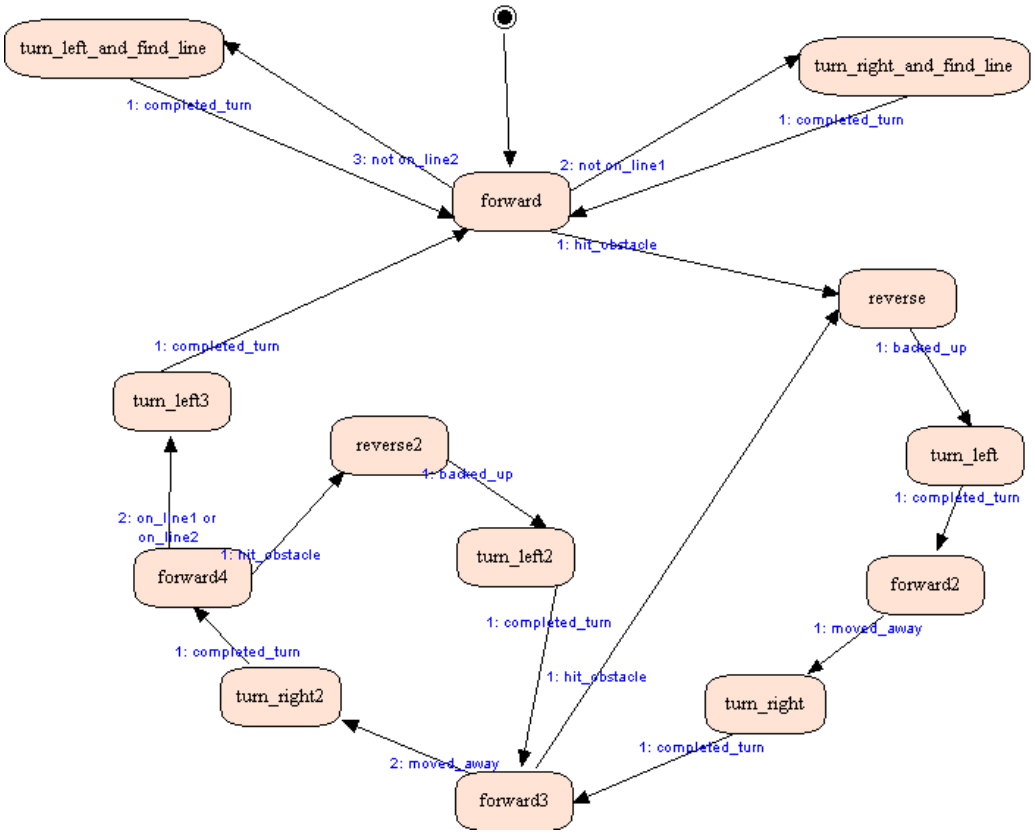

Figure 5.6: state_control state machine

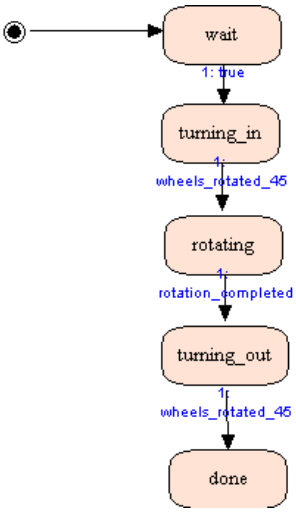

Figure 5.7: turn_robot state machine

Both these states will rotate the robot in the relevant direction until it rejoins the line. At this point a transition will be taken back to the forward state.

The robot switches from line following to obstacle avoidance when it is in the state `forward` and the `hit_obstacle` input becomes true. It then reverses until the `backed_up` input becomes true and turns left. When this has completed it moves forward until the `moved_away` input (based on a rotation sensor) becomes true. It then turns right and continues in a straight line. At this stage it can encounter the obstacle again and if this occurs (indicated by the `hit_obstacle` input) then it will transition to the `reverse` state and the process described above will repeat. This can continue as many times as necessary to move past the obstacle. On the other hand, if it moves forward without bumping into any obstacles then it turns right and moves forward again until it rejoins the line. Again, if it should hit an obstacle during this, it will perform the loop `reverse2`, `turn_left2`, `forward3`, `turn_right3`, `forward4` as many times as necessary to move around the obstacle. Eventually it will rejoin the line and when this occurs it will turn left and return to the main state, `forward`.

The second state machine used in `compute_state` (section 5.2.2provides the turning control for the robot (figure 5.7). When this state machine is reset it will first turn the synchro drive wheels in so it can rotate on the spot. Next, it enters the state `rotating` and this will continue until the `rotation_complete` input becomes true. Notice that the direction of rotation (i.e. turn left or right) is not of importance since this is derived outside the state machine. Lastly, it turns all the wheels out by 45 degrees thereby realigning them before moving to the state `done`. The done state is used to communicate out of the machine that the turning process is complete.

This machine is not active all the time and when it is inactive it is important that it does not enter any states which affect the global output variables. For this reason a `wait` state has been introduced as the first transition in the machine. Therefore, on each cycle the state machine's reset input is true; the only state

that will be reached is `wait`, which has no effect on the rest of the system.

Compute_state node

Now that the state machines embedded in `compute_state` have been described the rest of the node can be explained more clearly. First we will focus on the supporting code for the state machine `state_control` (figure 5.8). Despite the fact `state_control` has many outputs, they can be grouped into four different output situations. The states corresponding to forward, reverse, turn right and turn left are grouped using OR gates to produce one output for each. From now on, these will be referred to as the outputs of `state_control` rather than its real outputs. When the robot is turning to rejoin the line, it is just a special case of the turn right or turn left state, but instead of stopping after 90 degrees, it should stop after the relevant light sensor is on top of the line again.

First, input processing associated with the forward output will be discussed. When the robot must move forward for a certain distance, it must be informed when the distance has elapsed. This is computed using the node `true_after_number` which has already been seen in section 4.2.2. `True_after_number` is reset using a `rising_edge` node, which is triggered when the robot enters one of the `forward` states. It produces a true output when `rotation_2`'s reading has increased by the constant `rots_to_move_away`. Finally, its output is passed to the input `moved_away` of `state_control`.

An almost identical procedure is followed for the reverse output of `state_control`. The only exception is that as the robot will be moving backwards, the rotation sensor's reading will be decreasing therefore the node `true_after_negative_number`[2] is used instead. The distance the robot should backup is also different to that when it is moving forward around an obstacle. Therefore the constant `rots_for_backup` is used to determine when the robot has reversed far enough. The output of this is passed into the input `backed_up` of `state_control`. Note that `PRE` (previous value) operators are used to prevent recursion from appearing in the design.

The outputs of `state_control` that instruct the robot to perform turns are considerably more complicated than those for forward and reverse. Some of this complexity has been encapsulated in the `turn_robot` state machine, so its effect on the system must be considered before its impact on the inputs of `state_control` is discussed.

The subsystem for turning the robot does not distinguish between turns in either direction; the direction is only used for setting the global variables. For this reason, the outputs for turning left and right of `state_control` are combined using an OR gate. This is inverted and then used for initialising the state machine `turn_robot`. Before the inversion, it is passed to a `rising_edge` node which produces the reset condition for the node `turn_control`. `Turn_control`'s function is to pre-process the inputs for the `turn_robot` state machine and will be described individually later in this section. The inputs `on_line1`, `on_line2`, `rotation_1` and `rotation_2` are all needed to perform turns and as such they are passed to the `turn_control` node.

The `turn_robot` machine sets the output variable `gv_turning_in` when the synchro drive wheels are turning in and `gv_turning_out` when the wheels

[2] true_after_negitive_number is abbreviated to true_after_nve_number in the figures

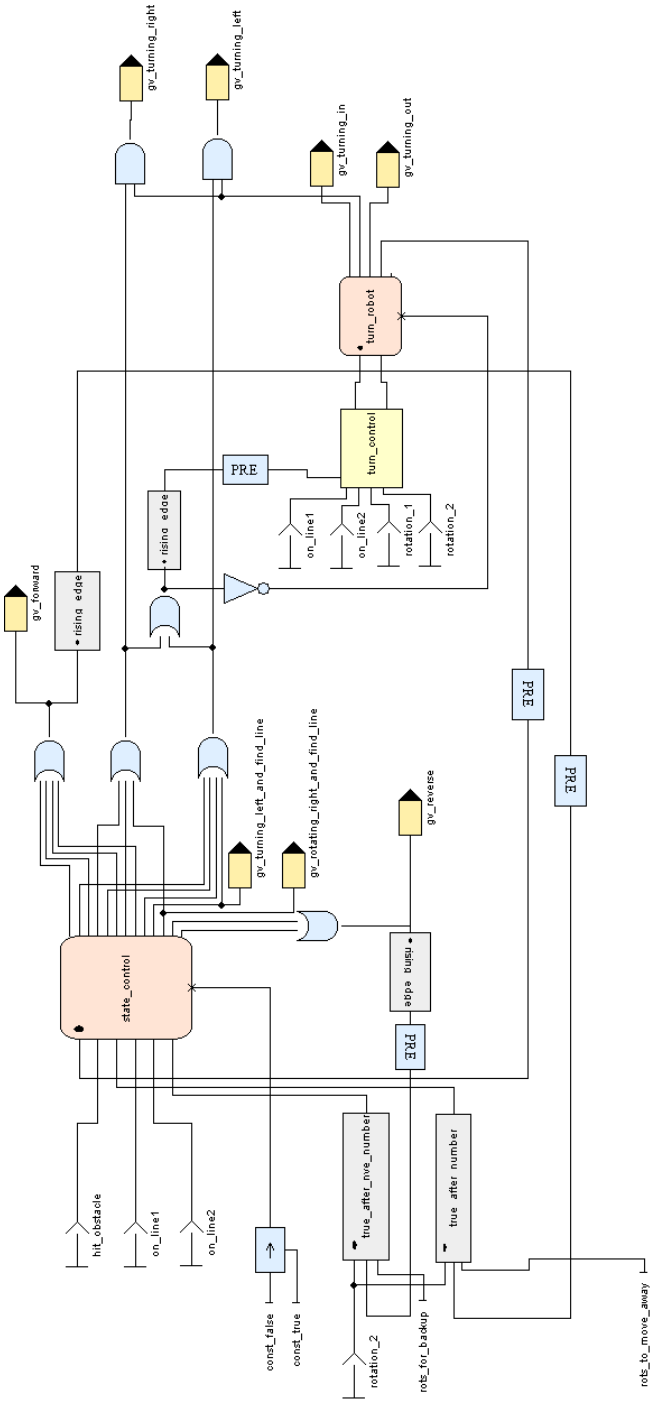

Figure 5.8: compute_state node

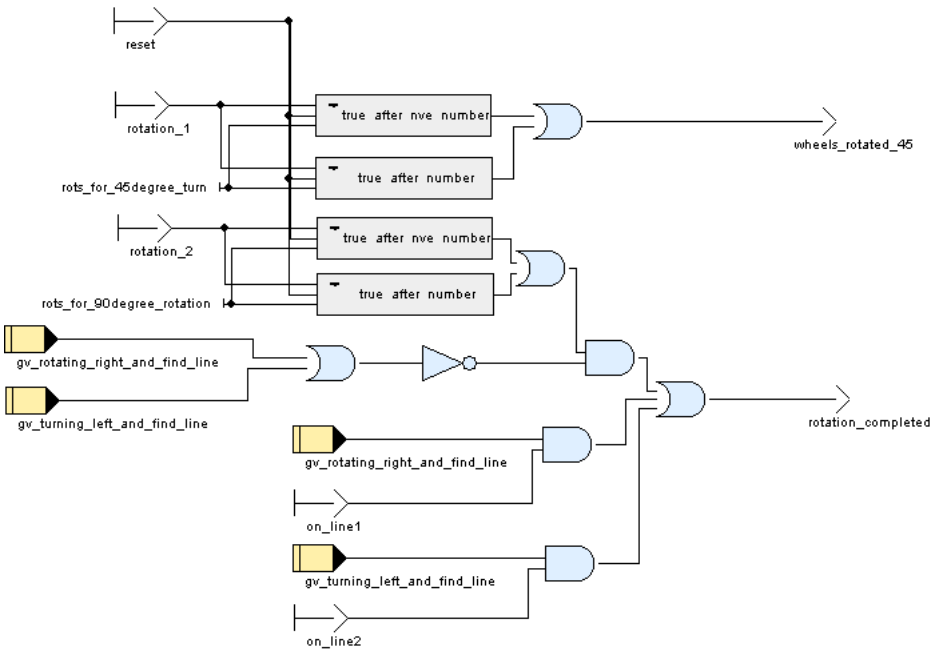

Figure 5.9: turn_control node

are returning to their normal position. The rotating output of `turn_robot` is then combined with the original turning outputs from `state_control` to define the direction in which the robot should be rotating. Therefore, the global variable `gv_turning_right` will become true if the robot is rotating right and `gv_turning_left` will be true if the robot is rotating left. When the turn is complete, the output `done` of `turn_robot` will become true and is passed back as an input to `state_control`. Thereby combining the different types of rotations (either 90 degrees or until a light sensor is on a line) into one input.

Turn_control node

`Turn_control` (figure 5.9) is responsible for creating the inputs `wheels_rotated_45` and `rotation_completed` for the state machine `turn_robot`. The input `wheels_rotated_45` is the simpler of the two since the condition for it to be true is identical for all kinds of rotations that the robot can perform. It uses the nodes `true_after_number` and `true_after_negative_number` to produce a true output when the wheels have been correctly turned in or turned out, respectively. It is because the wheels rotate both in and out that two nodes are needed for counting rotations since in one direction the rotation sensor's reading will be increasing and the other, decreasing. The output of both nodes is joined using an OR gate and produced as an output of the node. The reset condition for these nodes is handled externally and passed as an input to the system.

In the case where the robot must perform a 90 degree turn on the spot, the method described above is reused. However, the robot will also be rotating when it is trying to rediscover the line, thereby potentially triggering either of

the `true_after` nodes. Therefore, the OR gate output for a 90 degree rotation is only allowed to pass through if the global variables `gv_rotating_right_and_find_line` and `gv_rotating_left_and_find_line` are both false, indicating that the robot is not trying to find the line. The other cases, `on_line1` and `on_line2` are also filtered using the global variables that determine the current operation the robot is performing. All the outputs are then joined using an OR gate and passed by the output `rotation_completed` to `turn_robot`.

5.2.3 Implementation

Due to the large number of sensors required, the design was implemented on two RCXs using the communication algorithm from section 3.3. However, since only three actuators are required for the system, these can all be handled on the master RCX. Therefore, information only needs to pass in one direction, from the slave to the master. For this reason, the communication algorithm was altered to reflect the need for one-way only communication. This also had the advantage of reducing the latency from approximately 100 milliseconds to 50 milliseconds.

```
time_t receive_time;

int msg;
int timeout;

wakeup_t msg_or_timeout_wakeup(wakeup_t ignore)
{
  return ( msg || ((get_system_up_time() - receive_time) > 100) );
}

void my_integrity_handler(const unsigned char *data, unsigned char len)
{
  receive_time = get_system_up_time();
  copy_incomming_data(incomming_data, data, PACKET_SIZE);
  msg = 1;
}

int lnp_thread()
{

  while (1) {

    wait_event(msg_or_timeout_wakeup, 0);
    if (msg == 0) //no msg, but we've woken up > so must have timeout
    {
      //timeout occurred
      timeout = 1;
    }
    else
    {
      timeout = 0;
```

```
      //got reply
    }
    msg = 0;

  }
}
```

In the new algorithm (shown above), the master still has a separate thread for managing the communication but never sends any packets. The only function of the thread now is to check that messages are received from the slave regularly. If they are not, it sets a timeout variable that is passed to the reactive kernel in the `comms_ok` input. When a message is received, the time at which this occurs is stored in a variable `receive_time`. The function called by `wait_event` tests the difference between the current system time and the last time a message arrived. If it is greater than 100ms, then a timeout has occurred and this is set in `lnp_thread`.

```
int msg;
unsigned char packet[3];

int main(int argc, char *argv[])
{
  ds_active(&SENSOR_1);
  ds_active(&SENSOR_2);
  ds_active(&SENSOR_3);
  while(1) {
    packet[0] = LIGHT_1;
    packet[1] = LIGHT_2;
    packet[2] = LIGHT_3;
    lnp_integrity_write(packet,3);
    msleep(10);
  }
}
```

Since the slave program (above) no longer has to process incoming messages, it becomes very simple. It enters a non-terminating loop which repeatedly builds packets out of the light sensor readings, sends them and then pauses to reduce the chance of message collisions.

The remainder of the program was generated using SCADE2lego from section 3.2. It was called as follows:

```
SCADE2lego synchro -I 3 rotation_1 rotation_2 light_3
-o Forward_A Back_A Speed_A Forward_B Back_B Speed_B
Forward_C Back_C Speed_C LCD_INT
```

The script is only informed of the sensors that are present on the master brick. These include the two rotation sensors and the one touch sensor. They were chosen since they must always be reacted to instantly and cannot afford the 50 milliseconds delay from the slave. For example, a delay in reading the touch sensor might result in the robot crashing into an obstacle, and any delay on the rotation sensors will result in the robot not moving the expected amount.

The extra code for the master detailed above was inserted into the code produced by the SCADE2Lego script. Next, the main driving procedure was augmented with the code to support the communication as well as three extra input procedure calls for the sensors on the remote brick. One of the procedures is shown below in addition to how it is called in the main driving loop:

```
void I_light_4(_C_synchro* ctx, _int V)
{
   ctx->_I3_light_4 = V;
}

I_light_4(&the_ctx, incomming_data[0]);
```

5.3 Testing

The reasoning behind the testing of robots produced in this project is described in section 4.4.1 and this should be consulted before reading this section. The test cases developed for this robot are as follows:

- The robot correctly follows a line that turns both sharply and gently in either direction
  - PASSED - trivial to test, different lines were used with turns varying in sharpness and direction. However, the robot is very slow at line following and this will be commented on in the evaluation.
- The robot should successfully detect an obstacle using either its bumper or the light sensor detection mechanism.
  - PASSED - providing the obstacle is of the correct size to trigger the bumper it is always detected. The light sensor is less reliable at detecting obstacles, but with the bumper as a backup, obstacles are always eventually detected.
- When an obstacle is encountered, the obstacle avoidance algorithm should be correctly executed.
  - PASSED - testing the obstacle avoidance algorithm breaks down into three parts. The first is to test a small obstacle that the robot will not hit again after its initial collision. The next is an arbitrarily long obstacle in the direction at 90 degrees to the line. This requires successive bumps to manoeuvre around and therefore it is important that the robot keeps its orientation correct through precise turns. Finally, the case of an arbitrarily long obstacle in the direction of the line must be tested. This is identical to the last test except the direction of the turns the robot must complete are different
- Once an obstacle has been successfully avoided, the (line following) light sensors should correctly detect the line again and the robot should proceed to follow it.
  - PASSED - the last stage of a number of obstacle avoidance runs were observed for correct redetection of the line.

5.4 Evaluation

The synchro wheelbase used in the robot was a mixed success. It could repeatedly perform very precise turns with almost no error accumulation, but with an exceptionally slow turning speed. Currently, for the robot to perform a 90 degree turn, it takes about 10 seconds for the wheels to orientate themselves inwards and the same again outwards (the time to rotate is negligible), therefore, requiring a total of 20 seconds to rotate. For manoeuvring around obstacles, this is not a problem since precision is paramount. However, for following a line, many small corrections are required making this a very slow line follower. In retrospect it is possible that the two problem domains were too far removed meaning that a good solution for both was not achievable. In this case however, the choice had to be sacrificing line following efficiency over precise obstacle avoidance.

Some success was attained when using a light sensor to detect obstacles. If the lighting conditions were quite dark and the obstacles light-coloured, then the light sensor could usually detect the obstacle before the touch sensor. This is in spite of the delay brought about by the message passing to the master brick

The protocol developed for communication between multiple RCXs was finally put to use in this project. Regrettably, it was not implemented in its complete form, as no actuators were needed to be controlled by the slave brick. However, its value has already been proven since, if the extra sensors provided by the slave were not available, this robot would not have been possible.

Furthermore, in the design, a real-world example of a concept conceived in section 3.3.3 was implemented. The system is set into a safe state when communication fails between the two bricks. Using the `comms_ok` input and the input and output filters, the system effectively shields itself against any communication problems, thereby reducing the danger introduced by asynchronous communication in a synchronous reactive system. In this situation, communication never failed since the bricks are fixed to point at each other, but when tests were conducted involving intentionally stopping the communication, the system did correctly enter and exit the safe state.

The SCADE design for this robot has shown the use of state based formalisms inside dataflow equations to attain the best of both formalisms. Despite the fact that state machines are very simple the impact they have on dataflow designs is profound. Control concepts that were originally almost impossible to program in dataflow terms are now simply expressed as embedded state machines. This allows the designer to consider the system in an extra level of abstraction, since state machines can be created and simulated independently and then dataflow equations built up around them as supporting code. It should also be noted that this inverted the hierarchy of abstraction in the design, with the state machine responsible for the main control being placed in a low level node.

This idea recurs with the use of global variables. The designer can create a simple abstract mapping to global variables while working deep in the design. Then back at the top level of the design, these can be mapped to system outputs without concern for low level details. For example, the design in this chapter used a variable `gv_forward` to signal when the robot should move forward. At the top level (in the node `synchro`) this was mapped to the motor outputs in a way that made the robot move forward when it was true. Needless to say,

it also helps the understandability of the system by reducing the amount of connections needed.

Chapter 6

Conclusion

6.1 Overview of Work Completed

The first products of this project were scripts for enabling a Lustre or SCADE design to be translated into code that can be executed on the RCX. A script developed by Christophe Mauras provided a starting point for the work [22]. First, Mauras's work was extended by translating and updating the original script. The operation of the script was then further simplified while improving the functionality of light sensors and LCD control.

Next, several methods for performing the compilation of a SCADE design to BrickOS were investigated. The results of this concluded that the best way to translate a SCADE design to BrickOS code would be to run SCADE's own internal code generation and then execute a custom script on the output produced. The script developed is relatively easy to use and supports all SCADE features except Safe State Machines. Pre-built input and output procedures provide functionality for the most commonly used features of the RCX.

An RCX is severely limited in its input and output capabilities; therefore, the next piece of work aimed to address this. It involved developing a protocol for communication between two RCXs, thereby enabling one RCX to take control of another's sensors and actuators. The solution developed uses two RCXs, one executes the reactive kernel and is considered the master. The second simply passes its input/output capabilities to the master. The problems this introduces into the synchronous hypothesis were also investigated, and a possible solution was devised. This involved detecting that communication had failed and setting the system into a safe state until communication was restored.

With a method for compiling a SCADE design to the Lego architecture now in place, work could proceed on designing robots to illustrate the key aspects of programming in the synchronous language domain. The first robot produced was a simple line follower. The motivation behind this robot was simply to show the reader a basic design produced in Lustre and SCADE and compare the two. This link was established since all further designs in this project have only been considered in SCADE.

The next robot produced was a brick sorter. This had the advantage of only dealing with an internally changing environment; therefore, the system can only change in a limited number of ways. Because of this it was possible to introduce

some monitoring in the main element that manipulates the environment, the conveyor belt. Using a rotation sensor the robot could successfully detect, and with the aid of a human operator, recover from, a jam on the conveyer belt.

The last robot to be built was the most ambitious of the project; a line follower with obstacle avoidance capabilities. Most of the complexity arose from not knowing the nature of the environment beforehand. The Lego design in this case was entirely self built and utilised an advanced locomotion technique known as a synchro drive. This provided the robot with the ability to perform precise manoeuvres that were needed for obstacle avoidance. The SCADE design for the system was also an order of magnitude more complex than the previous robots since it involved combining state based control with data flow control. This was deemed necessary because of the state based nature of avoiding obstacles. In the design, it was shown how simple state machines within SCADE could be used effectively with data flow equations, thereby combining the best of both formalisms.

6.2 Conclusions

This project set out to show that Lego Mindstorms is a valid and useful architecture for implementing reactive systems produced using SCADE. This has been shown through development of the compilation script, SCADE2Lego and the robots designed using SCADE and then implemented in Lego. This also fulfilled a goal of the project that was to demonstrate the use of Lego Mindstorms in a teaching environment for reactive systems. It is clear that a course on synchronous reactive systems could be developed using SCADE for the design and Lego Mindstorms as the implementation platform.

However, it has been observed that data flow equations by themselves can only solve a limited set of problems. For problems such as line following and brick sorting, the data flow approach is an excellent one and produces concise and easily comprehensible designs. However, as soon as an element of state based control is required in the design, the data flow approach becomes insufficient. This is where a combination of the two synchronous formalisms should be considered. As illustrated in the final robot, combining state machines with data flow equations is indeed a valid way to design solutions.

Another point that became apparent is that some solutions may be better structured with a functional layer on top of the reactive kernel. Thus, the reactive system continues performing the function it is well suited for, but a higher level of control is present that manages functions that synchronous languages are not designed for, e.g. pathfinding.

A further area explored was constructing Lego robots. This was approached from two directions in this project: modifying designs of professionally constructed robots and, in the last chapter, building a robot from scratch. The versatility of Lego as a construction set is unrivalled and is an ideal choice for building robots to be programmed using any synchronous language.

Sadly, the verification plug-in for SCADE was not available on the system that was used for this project. Therefore, no verification could be performed on the designs produced. This is obviously a very important step in the methodology of designing synchronous reactive systems and as such it was regrettable that no verification could be conducted as part of this project.

Videos and more photos of the robots constructed in this project are available at http://www-student.cs.york.ac.uk/ dhw100/. The web site also explains how to setup BrickOS for use with the Perl scripts (legolus2 and SCADE2lego).

6.3 Further Work

One obvious area for further work is to extend the functionality offered by the SCADE2lego script. Support could be added for temperature sensors and the buttons on the RCX. A considerably more complex addition would be automated generation of code for both the master and slave RCX's when more than three sensors or actuators are required. The caller of the script would need to indicate which three sensors and actuators require the most precise timing and these would be assigned to the master RCX. Furthermore, although simple state machines are handled by the script, because they are embedded in other parts of the design, safe state machines are not. Therefore, another avenue to explore would be providing support for safe state machines [29].

Up to now the second RCX in the communication protocol has merely been considered a slave handing over all control to the master RCX. A better way of managing communication between two RCXs would be to develop two reactive kernels and define the communication protocol in terms of inputs and outputs of the designs, allowing communication to take place more abstractly. Some work has already been performed in the general case of distributed synchronous reactive systems [6] but applying this to the Lego Mindstorms architecture would prove an interesting challenge.

Reactive systems are good at what they do, i.e., based on stimuli they generate output signals. However, they are not well suited for expressing higher order control algorithms such as pathfinding or logic based reasoning. It would be interesting, therefore, to investigate whether or not a higher order control program could be implemented which would provide inputs to the reactive kernel in a more abstract form. An example of this would be a pathfinding program running externally to the reactive kernel that simply passes, as input, the next location to which the robot must move.

References

[1] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud: The synchronous data flow programming language LUSTRE, Proceedings of the IEEE , Vol. 79, No. 9, September, 1991, 1305-1320.

[2] A. Benveniste, G. Berry: The synchronous approach to reactive and real-time systems, Proceedings of the IEEE, Vol. 79, No. 9, September, 1991, 1270-1282.

[3] N. Halbwachs, P. Raymond: A tutorial of Lustre, January 24, 2002.

[4] P. Caspi, D. Pilaud, N. Halbwachs, J. A. Plaice: LUSTRE: a declarative language for programming synchronous systems, 14th ACM symposium on principles of programming languages, Munich, January, 1987.

[5] N. Halbwachs, F. Lagnier, C. Ratel: Programming and verifying realtime systems by means of the synchronous dataflow programming language Lustre, IEEE transactions on software engineering, Vol. 18, No. 9, September, 1992.

[6] N. Halbwachs (1993) Synchronous programming of reactive systems, Kluwer Academic Publishers

[7] Frdric Rocheteau and Nicolas HALBWACHS: POLLUX: A LUSTRE based hardware design environment 1994

[8] J. R. Mc Graw: The val language: Description and analysis, ACM TOPLAS, 4(1), January 1982.

[9] Kahn, G., The semantics of a simple language for parallel programming, Proc. IFIP Congress '7, North Holland, 1974.

[10] F. Boussinot and R. de Simone. The ESTEREL language. In Proceedings of the IEEE, pages 79(9):1293-1304, September 1991.

[11] D. Harel and A. Pnueli: On the devolpment of reactive systems, Logics and Models of Concurrent Systems, NATO ASI Series, Vol.13, K. R. Apt, Ed. New York: Springer-Verlag, pp. 477-498, 1984.

[12] A. Pnueli: Applications of temporal logic to the specification and verfication of reactive systems: A survey of current trends, in Current Trends in Concurrency, de Bakker et al., Eds., Lecture Notes in Computer Science, Vol. 224, Berlin, GermanyL Springer-Verlag, pp. 510-584, 1986.

[13] D. Baum, M. Gasperi, R. Hempel, L. Villa: Extreme Mindstorms - An advanced guide to Lego Mindstorms, 2000.

[14] M. Ferrari, G. Ferrari, R. Hempel: Building Robots With Lego Mindstorms, 2002

[15] Esterel Technologies - http://www.esterel-technologies.com

[16] Lego Mindstorms Website - http://mindstorms.lego.com

[17] pbForth Home Page - http://www.hempeldesigngroup.com/lego/pbForth/

[18] LeJOS: Java for the RCX - http://lejos.sourceforge.net/

[19] MindStorms RCX Sensor Input Page - http://www.plazaearth.com/usr/gasperi/lego.htm

[20] BrickOS Home Page - http://brickos.sourceforge.net/

[21] Scade v4.2 Reference Manual

[22] Christophe Mauras & Martin Richard: Reactive Languages and Lego Mindstroms - http://www.emn.fr/x-info/lego/

[23] VERIMAG - http://www-verimag.imag.fr/

[24] Lustre-V4 Manual - Pascal Raymond

[25] Nancy Leveson: Medical Devices: The Therac-45

[26] Lego Mindstorms Constructopedia

[27] Mindsensors - http://www.mindsensors.com

[28] An industral conveyer belt system - Lego Model Number: 9701-6, Lego Dacta Set

[29] Charles Andre: Semantics of S.S.M (Safe State Machines)

Appendix A

Code for Legolus2

```
#!/usr/bin/perl
# File              : legolus
# Author            : Christophe Mauras
# Created On        : december 1, 99
# needs             : lustre, poc
# Modified by David White

$name = shift(@ARGV);
$node = shift(@ARGV);
if ($name) {
  if (!-e "${name}.lus") {print "Sorry, ${name}.lus not found\n"; exit 1 ;}
         system "lustre ${name}.lus ${node} -o ${name}.oc";
         system "poc ${name}.oc -o ${name}.c";
         system "rm ${name}.oc ${name}.h";
  open (IN, "${name}.c") ||
    print "cannot open ${name}.c for reading \n" ;
  if (!open (OUT, ">${name}_${node}.c"))
  { close IN ; print "cannot open ${name}_${node}.c for writing \n" ; }
         print OUT <<Preamble;
/*****************************************/
/* Generated by legolus using lustre and poc */
/*****************************************/

#include <conio.h>
#include <unistd.h>
#include <dsensor.h>
#include <dmotor.h>
typedef int _boolean;
typedef int _integer;
typedef char* _string;
typedef float _float;
typedef double _double;
#define _false 0
#define _true 1
int modulo (a,b)
int a,b;
{return(a % b);}
_integer real2int (r)
```

```
_float r;
{return((int) r);}
_float int2real (n)
_integer n;
{return((float) n);}

/**********************/
/* LUSTRE Generated:  */
/*  - context struct  */
/*  - automata reset  */
/*  - inputs reset    */
/**********************/

Preamble
  $LineNumber = 0; $copier = 0;
  while (($in_line=<IN>)) {
    ++$LineNumber;
    if ($in_line =~ /^typedef/) {$copier = 1;};
                  if ($copier == 1) {  print OUT "$in_line";};
                  if ($in_line =~ /^}/) {$copier = 0;}
        }
        close IN;
  open (IN, "${name}.c");
  $LineNumber = 0; $copier = 0;
  while (($in_line=<IN>)) {
    ++$LineNumber;
    if ($in_line =~ /^\w*\s*void\s*\w+_reset/) {$copier = 1;}
                  if ($copier == 1) {print OUT "$in_line"; }
                  if ($in_line =~ /^}/) {$copier = 0; }
        }
  close IN;
        print OUT <<Output;

/********************/
/* Static Variables */
/********************/

static ${node}_ctx the_ctx;
static unsigned first_A, dir_A, first_B, dir_B, first_C, dir_C;

/**********************/
/* Output Procedures */
/**********************/

void ${node}_O_Forward_A(void* cdata, _boolean v)
       {if (v) dir_A -= 2;
        if (first_A) first_A = 0;
        else {motor_a_dir(dir_A);dir_A = 3;first_A = 1;}}

void ${node}_O_Back_A(void* cdata, _boolean v)
       {if (v) dir_A -= 1;
        if (first_A) first_A = 0;
        else {motor_a_dir(dir_A);dir_A = 3;first_A = 1;}}
```

```
void ${node}_O_Speed_A(void* cdata, _integer v)
       {motor_a_speed(v);}

void ${node}_O_Forward_B(void* cdata, _boolean v)
       {if (v) dir_B -= 2;
        if (first_B) first_B = 0;
        else {motor_b_dir(dir_B);dir_B = 3;first_B = 1;}}

void ${node}_O_Back_B(void* cdata, _boolean v)
       {if (v) dir_B -= 1;
        if (first_B) first_B = 0;
        else {motor_b_dir(dir_B);dir_B = 3;first_B = 1;}}

void ${node}_O_Speed_B(void* cdata, _integer v)
       {motor_b_speed(v);}

void ${node}_O_Forward_C(void* cdata, _boolean v)
       {if (v) dir_C -= 2;
        if (first_C) first_C = 0;
        else {motor_c_dir(dir_C);dir_C = 3;first_C = 1;}}

void ${node}_O_Back_C(void* cdata, _boolean v)
       {if (v) dir_C -= 1;
        if (first_C) first_C = 0;
        else {motor_c_dir(dir_C);dir_C = 3;first_C = 1;}}

void ${node}_O_Speed_C(void* cdata, _integer v)
       {motor_c_speed(v);}

void ${node}_O_LCD_0(void* cdata, _integer v)
       {cputc_0((unsigned) v);}

void ${node}_O_LCD_1(void* cdata, _integer v)
       {cputc_1((unsigned) v);}

void ${node}_O_LCD_2(void* cdata, _integer v)
       {cputc_2((unsigned) v);}

void ${node}_O_LCD_3(void* cdata, _integer v)
       {cputc_3((unsigned) v);}

void ${node}_O_LCD_4(void* cdata, _integer v)
       {cputc_4((unsigned) v);}

void ${node}_O_LCD_INT(void* cdata, _integer v)
       {lcd_int((unsigned) v);}

void ${node}_O_LCD_5(void* cdata, _boolean v)
       {if (v){dlcd_show (LCD_5_MID);}
        else {dlcd_hide (LCD_5_MID);} }

/*******************************************/
/* Input Procedures & Automata Step Procedure */
/*******************************************/
```

```
Output
         open (IN, "${name}.c");
  $LineNumber = 0; $copier = 0;
  while (($in_line=<IN>)) {
    ++$LineNumber;
    if ($in_line =~ /^.*_I_.*/) {$copier = 1; }
                  if ($copier == 1) {  print OUT "$in_line"; }
                  if ($in_line =~ /^}/) {$copier = 0;}
         }
  close IN;
         open (IN, "${name}.c");
  $LineNumber = 0; $copier = 0;
  while (($in_line=<IN>)) {
    ++$LineNumber;
    if ($in_line =~ /^\w*\s*void\s*\w+_step/) {$copier = 1;}
                  if ($copier == 1){  print OUT "$in_line";};
         }
  close IN;
         print OUT <<TheMain;

/*************/
/* Main Loop */
/*************/

int main(int argc, char *argv[])
{
  time_t temp_time = get_system_up_time();

  ${node}_reset(&the_ctx);
  dir_A = 3; dir_B = 3; dir_C = 3;
  first_A = 1; first_B = 1; first_C = 1;
  ds_active(&SENSOR_1);
  ds_active(&SENSOR_2);
  ds_active(&SENSOR_3);
  ds_rotation_set(&SENSOR_1, 0);
  ds_rotation_set(&SENSOR_2, 0);
  ds_rotation_set(&SENSOR_3, 0);
  ds_rotation_on(&SENSOR_1);
  ds_rotation_on(&SENSOR_2);
  ds_rotation_on(&SENSOR_3);
   while(1){
     if (SENSOR_1<0xf000)
       {${node}_I_touch_1(&the_ctx, _true);}
     else
       {${node}_I_touch_1(&the_ctx, _false);}
     if (SENSOR_2<0xf000)
       {${node}_I_touch_2(&the_ctx, _true);}
     else
       {${node}_I_touch_2(&the_ctx, _false);}
     if (SENSOR_3<0xf000)
       {${node}_I_touch_3(&the_ctx, _true);}
     else
       {${node}_I_touch_3(&the_ctx, _false);}
```

```
      ${node}_I_light_1(&the_ctx, LIGHT_1);
      ${node}_I_light_2(&the_ctx, LIGHT_2);
      ${node}_I_light_3(&the_ctx, LIGHT_3);
      ${node}_I_rotation_1(&the_ctx, ROTATION_1);
      ${node}_I_rotation_2(&the_ctx, ROTATION_2);
      ${node}_I_rotation_3(&the_ctx, ROTATION_3);
      ${node}_step(&the_ctx);
      msleep((int) 10 - (get_system_up_time() - temp_time));
      temp_time = get_system_up_time();
    }
}
TheMain
         close OUT;
         system "rm ${name}.c";
         system "rm -f ${node}.ec";
         print "C code in : ${name}_${node}.c \n";
         print "Now type   : make ${name}_${node}.lx \n";
         print "and then   : dll ${name}_${node}.lx \n";
}

if (!${name}) {
     print STDERR <<EndOfUsage;
Usage: $0 file nodename
file.lus must contain node:
node nodename (
                   touch_1: bool;
                   touch_2: bool;
                   touch_3: bool;
                   light_1: int;
                   light_2: int;
                   light_3: int;
                   rotation_1: int;
                   rotation_2: int;
                   rotation_3: int
 )
returns (
Forward_A, Back_A :bool ;
Speed_A : int;
Forward_B, Back_B :bool ;
Speed_B : int;
Forward_C, Back_C :bool ;
Speed_C : int;
LCD_0 : int;
LCD_1 : int;
LCD_2 : int;
LCD_3 : int;
LCD_4 : int;
LCD_5 : bool
);
EndOfUsage
exit 1;
}

exit 0;
```

Appendix B

Code for SCADE2Lego

```
#!/usr/bin/perl
# SCADE2lego - David White
# Parts taken from legolus by Christophe Mauras

$node = shift(@ARGV);
print "\n";
if ($node) {
  if (!open (OUT, ">out_${node}.c")) {
    print "Cannot open out_${node}.c for writing \n";
    exit 1;
  }
  print OUT <<Section1;
/*************************/
/* Definitions & Includes */
/*************************/

#include <conio.h>
#include <unistd.h>
#include <dsensor.h>
#include <dmotor.h>

#define true 1     // Required for booleans
#define false 0

typedef int _int; // Required for SCADE's own types
typedef int bool;

/*******************************/
/* Constants (SCADE Generated) */
/*******************************/

Section1

# Copy constants
  if (open (CONST, "${node}_const.c")) {
    while (($in_line=<CONST>)) {
      if (($in_line =~ /const (.*)/) && (!($in_line =~ /-const(.*)/))) {
        print OUT "$in_line";
```

```
      }
    }
    close CONST;
  }
  else {
    print "Cannot open ${node}_const.c for reading - no problem if your
           program dosn't use any constants\n" ;
    print OUT "// This design uses no constants.\n";
  }

  print OUT <<Section_global;

/*******************************/
/* Global Variables (SCADE Generated) */
/*******************************/

Section_global

# Copy global variables
  if (open (GLOBAL, "${node}_global.c")) {
    while (($in_line=<GLOBAL>)) {
      if (($in_line =~ /bool (.*)/) || ($in_line =~ /_int (.*)/)) {
        print OUT "$in_line";
      }
    }
    close GLOBAL;
  }
  else {
    print "Cannot open ${node}_global.c for reading - no problem if your
           program dosn't use any global variables\n" ;
    print OUT "// This design uses no global variables.\n";
  }

  print OUT <<Section2;

/***********************************/
/* Node Contexts (SCADE Generated) */
/***********************************/

Section2

  open (HEADER, "${node}.h") || die "Can't open ${node}.h - exiting...";
  $copier = 0;
  while ($in_line=<HEADER>) {
    if ($in_line =~ /typedef(.*)/) {
      $copier = 1;
    }
    if ($copier == 1) {
      print OUT "$in_line";
    }
    if ($in_line =~ /}(.*)/) {
      $copier = 0;
    }
  }
```

```
  close HEADER;

  print OUT <<Section3;

static _C_${node} the_ctx; // Main node (${node}) context
static unsigned first_A, dir_A, first_B, dir_B, first_C, dir_C;


/*********************/
/* Output Procedures */
/*********************/

void O_Forward_A(bool v)
{
  if (v)
    dir_A -= 2;
  if (first_A)
    first_A = 0;
  else
  {
    motor_a_dir(dir_A);
    dir_A = 3;
    first_A = 1;
  }
}

void O_Forward_B(bool v)
{
  if (v)
    dir_B -= 2;
  if (first_B)
    first_B = 0;
  else
  {
    motor_b_dir(dir_B);
    dir_B = 3;
    first_B = 1;
  }
}

void O_Forward_C(bool v)
{
  if (v)
    dir_C -= 2;
  if (first_C)
    first_C = 0;
  else
  {
    motor_c_dir(dir_C);
    dir_C = 3;
    first_C = 1;
  }
}
```

```
void O_Back_A(bool v)
{
  if (v)
    dir_A -= 1;
  if (first_A)
    first_A = 0;
  else
  {
    motor_a_dir(dir_A);
    dir_A = 3;
    first_A = 1;
  }
}

void O_Back_B(bool v)
{
  if (v)
    dir_B -= 1;
  if (first_B)
    first_B = 0;
  else
  {
    motor_b_dir(dir_B);
    dir_B = 3;
    first_B = 1;
  }
}

void O_Back_C(bool v)
{
  if (v)
    dir_C -= 1;
  if (first_C)
    first_C = 0;
  else
  {
    motor_c_dir(dir_C);
    dir_C = 3;
    first_C = 1;
  }
}

void O_Speed_A(_int v)
{
  motor_a_speed(v);
}

void O_Speed_B(_int v)
{
  motor_b_speed(v);
}

void O_Speed_C(_int v)
{
```

```
  motor_c_speed(v);
}

void O_LCD_INT(_int v)
{
  lcd_int((unsigned) v);
}

/********************/
/* Input Procedures */
/********************/
Section3

#Print Input Procedures depending on command line args specified
  $input_proc_calls[0] = "not_used";
  $input_proc_calls[1] = "not_used";
  $input_proc_calls[2] = "not_used";
  if (shift(@ARGV) ne "-i") {
    die("Incorrect Args (expected -i)");
  }
  $num_inputs = shift(@ARGV);
  for ($i=0; $i != $num_inputs; $i++) {
    $sensor = shift(@ARGV);
    if ($sensor eq "-o") {
      die("Number of inputs don't match");
    }
    #If sensor is light or rotation, print this kind of function
    if (($sensor eq "light_1") || ($sensor eq "light_2") || ($sensor eq "light_3") ||
        ($sensor eq "rotation_1") || ($sensor eq "rotation_2") || ($sensor eq "rotation_3")) {
      print OUT <<input_sensor_int;

void I_${sensor}(_C_${node}* ctx, _int V)
{
    ctx->_I${i}_${sensor} = V;
}
input_sensor_int

    }
    #If sensor is touch print this kind of function
    elsif (($sensor eq "touch_1") || ($sensor eq "touch_2") || ($sensor eq "touch_3")) {
      print OUT <<sensor_bool;

void I_${sensor}(_C_${node}* ctx, bool V)
{
    ctx->_I${i}_${sensor} = V;
}
sensor_bool

    }
    else {
      die("Unexpected Input: ".$sensor." - exiting...");
    }
    $input_proc_calls[$i] = ${sensor}
  }
```

```
#Copy initilisation and cyclic procedures from $Node.c
  open (NODE, "${node}.c");
  $copier = 0;
  while (($in_line=<NODE>)) {
    if ($in_line =~ /\/\* ============== \*\//) {$copier = 1; }
    if ($in_line =~ /SCADE_CG V4.2.1/) {$copier = 0;}
    if ($copier == 1) {  print OUT "$in_line"; }
  }
  close NODE;

#Print first part of main loop
  print OUT <<Section4;

/*************/
/* Main Loop */
/*************/

int main(int argc, char *argv[])
{
  time_t temp_time = get_system_up_time();

  ${node}_init(&the_ctx);
  dir_A = 3; dir_B = 3; dir_C = 3;
  first_A = 1; first_B = 1; first_C = 1;
Section4

#Print calls to set active sensors
  $num_rotation_sensors = 0;
  for($j=0; $j<3; $j++) {
    if (($input_proc_calls[$j] =~ /light/) || ($input_proc_calls[$j] =~ /rotation/)) {
      if ($input_proc_calls[$j] =~ /1/) {
        print OUT ("  ds_active(&SENSOR_1);\n");
      }
      if ($input_proc_calls[$j] =~ /2/) {
        print OUT ("  ds_active(&SENSOR_2);\n");
      }
      if ($input_proc_calls[$j] =~ /3/) {
        print OUT ("  ds_active(&SENSOR_3);\n");
      }
    }
  }

#Print initilisation calls for rotation sensors
  for($j=0; $j<3; $j++) {
    $sensor_num = $j+1;
    if ($input_proc_calls[$j] =~ /rotation/) {
      $num_rotation_sensors++;
      print OUT <<rot_setup;
  ds_rotation_set(&SENSOR_${sensor_num}, 0);
  ds_rotation_on(&SENSOR_${sensor_num});
rot_setup
    }
  }
```

```
#Print Delay stmt so rotation sensors have time to initilise
  if ($num_rotation_sensors > 0) {
    print OUT <<rot_delay;
  msleep(100);
rot_delay
  }

#Print While Part
  print OUT ("  while(1)\n  {\n");

#Print Input Calls
  for ($i=0; $i<3; $i++) {
    $uppercase = uc($input_proc_calls[$i]);
    $sensor_num = $i+1;
    if ($input_proc_calls[$i] ne "not_used") {
      print OUT ("    I_${input_proc_calls[$i]}(&the_ctx, ${uppercase});\n");
    }
  }

#Print call to main node's cyclic function
  print OUT ("    ${node}(&the_ctx);\n");

  (shift(@ARGV) eq "-o") || die("Expected -o as next argument");
  $num_outputs = shift(@ARGV);
  for ($i=0; $i<$num_outputs; $i++) {
    $output = shift(@ARGV);
    if ($output eq "Forward_A") {
      print OUT ("    O_Forward_A(the_ctx._O${i}_Forward_A);\n");
    }
    elsif ($output eq "Forward_B") {
      print OUT ("    O_Forward_B(the_ctx._O${i}_Forward_B);\n");
    }
    elsif ($output eq "Forward_C") {
      print OUT ("    O_Forward_C(the_ctx._O${i}_Forward_C);\n");
    }
    elsif ($output eq "Back_A") {
      print OUT ("    O_Back_A(the_ctx._O${i}_Back_A);\n");
    }
    elsif ($output eq "Back_B") {
      print OUT ("    O_Back_B(the_ctx._O${i}_Back_B);\n");
    }
    elsif ($output eq "Back_C") {
      print OUT ("    O_Back_C(the_ctx._O${i}_Back_C);\n");
    }
    elsif ($output eq "Speed_A") {
      print OUT ("    O_Speed_A(the_ctx._O${i}_Speed_A);\n");
    }
    elsif ($output eq "Speed_B") {
      print OUT ("    O_Speed_B(the_ctx._O${i}_Speed_B);\n");
    }
    elsif ($output eq "Speed_C") {
      print OUT ("    O_Speed_C(the_ctx._O${i}_Speed_C);\n");
    }
```

```
      elsif ($output eq "LCD_INT") {
        print OUT ("     O_LCD_INT(the_ctx._O${i}_LCD_INT);\n");
      }
      else {
        die("Unrecognised output: ".$output);
      }
    }
#Print loop end and timeing code

    print OUT <<end_loop;
      if ((get_system_up_time() - temp_time) > 0) {
        msleep((int) 10 - (get_system_up_time() - temp_time));
      }
      temp_time = get_system_up_time();
    }
}
end_loop

#Finish off
    close OUT;
    print "BrickOS compatiable C code in : out_${node}.c \n";
}
else {
    print "No args specified";
}
exit 0;
```

Appendix C

Code for Communicating RCXs

C.1 Code for Master Brick

```
#include <conio.h>
#include <unistd.h>
#include <stdlib.h>
#include <dsensor.h>
#include <dmotor.h>
#include <lnp.h>

#define PACKET_SIZE 3

tid_t lnp_thread_id;

time_t send_time;

int msg;
int timeout;

unsigned char incomming_data[PACKET_SIZE];
unsigned char outgoing_data[PACKET_SIZE];

void copy_incomming_data(char *dst, char *src, int size)
{
  int i;
  for (i=0; i<size; i++) {
    dst[i] = src[i];
  }
}


wakeup_t msg_or_timeout_wakeup(wakeup_t ignore)
{
  return ( msg || ((get_system_up_time() - send_time) > 100) );
}

void my_integrity_handler(const unsigned char *data, unsigned char len)
{
  copy_incomming_data(incomming_data, data, PACKET_SIZE);
  msg = 1;
}

int lnp_thread()
{

  while (1) {

    send_time = get_system_up_time();
    lnp_integrity_write(outgoing_data, PACKET_SIZE);

    wait_event(msg_or_timeout_wakeup, 0);
    if (msg == 0) //no msg, but we've woken up > so must have timeout
    {
      //timeout occoured
      timeout = 1;
```

```
    }
    else
    {
      timeout = 0;
      //got reply
    }
    msg = 0;

    msleep(100);
  }
}


int main(int argc, char *argv[])
{
  lnp_integrity_set_handler(my_integrity_handler);
  msg = 0;
  incomming_data[0] = 0;
  incomming_data[1] = 0;
  incomming_data[2] = 0;

  lnp_thread_id = execi(&lnp_thread,0,0,PRIO_NORMAL,DEFAULT_STACK_SIZE);

  while(1) {
    //read incomming data into context
    //call step procedure
    //write to outgoing data here
    if (timeout) {
      cputs("Tout");
    }
    else {
      lcd_int((incomming_data[0]*100)+(incomming_data[1]*10)+(incomming_data[2]*1));
    }
    //delay
  }
}
```

C.2 Code for Slave Brick

```
#include <conio.h>
#include <unistd.h>
#include <dsensor.h>
#include <dsound.h>
#include <dmotor.h>
#include <lnp.h>

int msg;
unsigned char packet[3];

void my_integrity_handler(const unsigned char *data, unsigned char len)
{
    //setup actuators here (as described in incomming packet)
    msg = 1;
}

int main(int argc, char *argv[])
{
  msg = 0;
  lnp_integrity_set_handler(my_integrity_handler);
  while(1) {
    if (msg == 1) {
      // create packet from sensors here
      lnp_integrity_write(packet,3);
      msg = 0;
    }
  }
}
```

Appendix D

Complete SCADE Design for the Brick Sorter

Appendix E

Complete SCADE Design for Line Follower with Obstacle Avoidance