Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

JCSPre: the Robot Edition to Control LEGO NXT Robots

Jon KERRIDGE, Alex PANAYOTOPOULOS and Patrick LISMORE

School of Computing, Napier University, Edinburgh UK, EH10 5DT
`j.kerridge@napier.ac.uk`, `alex@flocci.org`, `plismore@gmail.com`

Abstract. JCSPre is a highly reduced version of the JCSP (*Communicating Sequential Processes* for Java) parallel programming environment. JCSPre has been implemented on a LEGO Mindstorms NXT brick using the LeJOS Java runtime environment. The LeJOS environment provides an abstraction for the NXT Robot in terms of Sensors, Sensor Ports and Motors, amongst others. In the implementation described these abstractions have been converted into the equivalent active component that is much easier to incorporate into a parallel robot controller. Their use in a simple line following robot is described, thereby demonstrating the ease with which robot controllers can be built using parallel programming principles. As a further demonstration we show how the line following robot controls a slave robot by means of Bluetooth communications.

Keywords. parallel, robot controllers, JCSP, LEGO NXT, LeJOS, Bluetooth.

Introduction

The LEGO[1] NXT construction kit is often used for the construction of simple robots, in research and teaching and learning applications. Students and researchers of the CSP model would benefit from having a programming platform specifically for the programming of LEGO robots using CSP techniques. Researchers [1, 2] at The University of Kent have created their own parallel robot programming system which has many aspects in common with the environment described in this paper. However, their system is built on the Transterpreter platform, using the occam programming language, rather than the more widely used Java language. We therefore set out to create and develop an environment, for the LEGO NXT, based on a Java environment using a highly reduced version of the Communicating Sequential Processes for Java (JCSP) parallel programming environment, using an equally small Java virtual machine, called LeJOS. This means that developers of robot control systems can utilise parallel programming techniques based on a Java environment using a common development tool such as Eclipse.

The aim was to find out if a port of a reduced JCSP environment and LeJOS to the LEGO NXT was possible and feasible. Feasible is used in the sense that, even though a port was achieved, the size of control system that could be implemented would be so small as to render the system unusable. If the implementation proved feasible, then it would form the basis of a teaching module that introduces the concepts of process-based parallel programming using robotics as the underlying motivation. Furthermore, in using an environment based on Java and Eclipse we would be building on the students' prior experience and also showing them a concurrent programming environment that does not

[1] LEGO is a registered trade mark of the LEGO Group of Companies.

explicitly expose the underlying Java thread model. This may mean that we can wean students away from the widely-held belief that concurrent programming is hard.

The next section places the work in context. Section 2 describes the architectural framework and shows how LeJOS NXT is integrated with the cut-down JCSP and how this is then used to create active processes that can be used to build robot controllers. Section 3 explains how such robot controllers can be designed and section 4 shows an example of the design process for a simple line following robot. Section 4 describes the Bluetooth communication aspects that allow a slave robot to follow the movement of a master line following robot. The paper concludes with a summary of the achievements and the identification of future work.

1. Background

The LEGO NXT comes with a disc containing the proprietary LEGO programming language, called NXT-G. This is a visual language based on LabVIEW, another proprietary language created by National Instruments [3]. LabVIEW, and by extension NXT-G, has a control structure based upon that of dataflow diagrams [4]. While this means that there is an inherent parallelism in any LabVIEW program, it does not conform to the CSP model that this paper is most concerned with.

In addition, the LabVIEW language has two other shortcomings. Firstly, as a graphical programming language, it can be difficult for a more traditional programmer to learn [4]. Secondly, it is a closed source proprietary language, whose continued support is fully dependent on National Instruments. Any proprietary product raises concerns about the longevity of the format, since support for that format is dependent on the continued existence of the vendor [5].

The programming language occam (and its extension, occam-pi) is specifically designed around the CSP model of parallel programming [6], and it has been widely used in research [7]; but its use in the wider community is very limited in comparison to the world-wide use of Java.

One approach to the implementation of a CSP model on various devices is that of the Transterpreter project. This is a native occam runtime written in C, and developed at The University of Kent – originally for the LEGO Mindstorms RCX brick (the predecessor to the NXT brick) [1]. The Transterpreter is designed to be portable to a wide range of devices, and is currently being ported to the NXT brick by the Transterpreter team [2].

As well as running the Transterpreter as an interpreter, another approach is to use it to compile occam programs to native machine code, either directly or via an intermediary language such as C. This can be shown to give a measurable performance increase over interpreted occam [8]. The Transterpreter is not the only project aimed at creating a native occam runtime environment for embedded devices, [9] describes a similar project aimed at the Cell BE microarchitecture.

Although occam itself may be the most appropriate language for CSP programming, the language is still unfamiliar or unavailable to the majority of programmers. This has prompted the development of various occam-like libraries for more prevalent languages, such as C++ [10], Python [11], and notably Java.

The Java Communicating Sequential Processes (JCSP) framework is a library that allows a complete occam-like CSP programming package for Java, including all basic occam features [12], and some of the ideas of the pi-calculus/occam-pi [13], and whose channel classes have been formally proved to be equivalent to the CSP concepts upon which they are based [14]. JCSP was developed at the University of Kent, in consultation with the CSP and occam/transputer communities [15].

The JCSP framework is subdivided into four top-level packages, of which only parts of two of these are required for the JCSPre environment.

- `org.jcsp.lang`: This package provides the core of the CSP / occam core concepts. These are provided by classes such as CSProcess, Guard, Alternative and Parallel, each of which are named after the CSP concept which they implement. Most of the complexity of the `lang` package comes in the implementation of the channel classes, which are subdivided by source (One or Any), destination (One or Any) and type (integer or object). There are also "Alting" versions of each one, which can be included in an Alternative. To hide the complexity of the implementation from the user, a convenience class (Channel) is provided. The `lang` package also provides a few other classes not in the original CSP paper, but which have shown to be frequently used concepts.
- `org.jcsp.util`: This package provides some extensions to the simple channel behaviour of the CSP model. Various styles of buffering are provided to extend the simple semantics of the CSP channel model. A sub-class, o`rg.jcsp.util.filter`, allows objects to be transformed as they are transmitted over a channel – either at the input end, the output end, or both [16]. This package also contains some `exception` definitions that are required by the previous package.

LeJOS is a project to port a Java virtual machine to the various LEGO robotic bricks. The port for the NXT brick is known as LeJOS NXJ [17]. The most recent version, at the time of development, was LeJOS NXJ 0.3.0 alpha. Although the LeJOS virtual machine is based upon that of Java, there are several limitations which prevent it from being a fully compatible JVM. These include:

- No garbage collection is performed on the generated code, meaning that objects should be reused whenever possible,
- The Java `switch … case` statements are not implemented, meaning that a series of `if … then` statements need to be used instead,
- No arithmetic operations can be performed on variables of type `long`.
- Arrays cannot be longer than 511 items in length.
- The `instanceof` and `checkcast` operations are not properly implemented for interfaces and arrays.
- The core Java class `Class` is not implemented, and the `.class` construct is non-functional. This means that programming techniques involving reflection will not work.
- As well as the above design limitations, there are numerous bugs in the 0.3.0alpha release, as should be expected from any alpha release. One known bug is that multidimensional arrays of objects do not work correctly [18], (removed in LeJOS NXJ 0.6.0 beta).
- Inspection of the LeJOS API specification shows that the `java.lang` package, whose classes are described as "fundamental to the design of the Java programming language" by the core Java API, is missing a large portion of its classes [19], although the collection of classes implemented are similar to those implemented by other "micro edition" virtual machines, such as the CLDC [20].

The first two restrictions have been removed in the latest beta release of LeJOS but have yet to be incorporated into our implementation as the JCSPre as implemented had been tested and shown to be functional. In particular, reliance on garbage collection in small, resource constrained devices should be avoided whenever possible. The control processes described later all use `int` channels for data communication to avoid the creation of objects.

2. Architectural Framework

The system requires a multi-layer approach, with each layer adding more functionality and being more tailored to the specific outcome of a CSP-based LEGO robot.

2.1 Architecture: Description and Diagram

The basis of the architecture is the LEGO NXT brick itself, whose firmware supports the installation of a LeJOS kernel. The nature of the layered architecture is shown in Figure 1. The LeJOS NXJ abstraction layer provides a set of Java classes and interfaces that allow the creation of control systems using normal Java programming methods. The JCSPre is implemented using the LeJOS kernel abstraction layer. The Active Sensors then provide a process based realisation for the LEGO NXT sensors, see 2.5. The only part the final developer of a robot control system is concerned with is the high level Control aspect.

	Control	
		Active Sensors, Motors and other processes
LeJOS NXJ abstraction		JCSPre
	LeJOS kernel	
	LEGO NXT Firmware	

Figure 1: the layered architecture of the JCSPre environment.

The complete architecture has been integrated into the Eclipse development environment. In particular, a control system can be developed and downloaded into a LEGO robot from within the Eclipse environment.

2.2 LeJOS Kernel

The LeJOS virtual machine and core packages were described previously. During the development of the system, several bugs and shortcomings were uncovered. This prompted the development of workarounds, in the form of Java classes created in the package `org.jcsp.nxt.util`. These in the main dealt with `exception`s, which in the LeJOS implementation did not have a `String` parameter.

2.3 LeJOS NXJ Abstractions

The LeJOS project includes a set of classes specific to the functioning of the LEGO NXT robot, concerning input, output and communication. The packages provided for these functions are:

- `LeJOS.navigation`: A set of high-level classes which use the NXT internal compass and tachometer to move an NXT robot around at specified angles and distances.
- `LeJOS.nxt`: The majority of the robot input/output classes. Classes are provided to read the state of various input sources (battery level, push buttons, light and colour sensors, compass and tilt sensors, sound sensors, touch sensors and ultrasonic sensors), various output sources (motors, LCD screen, and speakers), as well as access to the flash storage facilities of the robot, and a "polling" method to wait for input from various sources.
- `LeJOS.nxt.comm`: Provides classes for communication with other robots and with a computer. These include interfaces for USB communication and Bluetooth wireless communication, and a class implementing the LEGO Communication Protocol used by all NXT devices.
- `LeJOS.subsumption`: This package contains classes which implement the LeJOS project's interpretation of the subsumption architecture.

2.4 JCSPre : Design and Evaluation

The core of the JCSPre takes the form of a ported set of classes from the JCSP framework; the essential subset. To identify this subset, a core set of features was identified as either necessary to the CSP model, or vital to the implementation of the JCSPre. The required classes were determined using the class dependency diagramming tools available in Eclipse. These features include, together with the JCSP required classes:

- The ability to declare an object to be a CSP process, and implement a run method for that object – the JCSP interface `CSProcess`.
- The ability to run several processes in parallel: the JCSP classes `Parallel` and `ParThread`.
- The ability to create channels, with a single input end and a single output end, thus allowing one process to pass an object to a second object. The object-passing channel variant is necessary in order to be able to pass `String` objects to the display. The integer-passing channel variant is necessary in order to provide a memory-efficient method of passing around large volumes of data, such as sensor levels. These are implementations of the JCSP channel interfaces `One2OneChannel` and `One2OneChannelInt` and their associated implementation classes. The Factory design pattern was widely used within JCSP, especially for the creation of Channels. These Factory classes were removed and the channel classes re-implemented appropriately to reduce the size of the final code footprint on the LEGO NXT. In addition none of the `Any` versions of the channels was implemented.
- The ability to choose between multiple input sources based upon the order in which they become ready, which requires the incorporation of the classes `Alternative`, `Skip` and `Barrier` and the abstract class `Guard`. In addition, the abstract classes `AltingChannelInput` and `AltingChannelInputInt` are required.
- The ability to set a timer; used for creating robot behaviours that last a specific length of time, or for creating "ticks" at which to schedule events. This requires the implementation of the class `CSTimer`.

The above comprises the complete set of classes and interfaces required from the core JCSP package `org.jcsp.lang`, which are included in the JCSPre package `org.jcsp.nxt.lang`.

2.5 Active Components

The main JCSP AWT libraries have a set of input/output components which it calls "Active Components". They include such common components such as text boxes, sliders and software buttons, but each is built to conform to the specification of a CSP process. Instead of accessing an active component through its listener, as would happen if using the standard Java libraries, an active component is accessed through its input and output channels.

Changes in an active component's state are represented by signals on the output channel. Changes in its configuration are effected by writing a signal to its input channel. So, for example, one might create an active "slider" process; configure its start point, end point, and interval via its input channel; and then connect another process to its output channel to read in a signal every time the user changes the value on the slider. This provides a much tidier interface to such active components because all the code associated with a particular component is contained within one class and the developer is unaware of the existence of the listener. In well written Java code the component instance and its listeners should be in different classes, which make understanding the code much more difficult because the reader has to refer to two class definitions.

The JCSPre mimics this active component style of implementation when implementing robot input and output components, for two main reasons. Firstly, as the JCSPre is meant as a port of the JCSP, a consistent approach should be used between the two projects. Secondly, the JCSPre, as a teaching tool, should seek to use a simple and consistent approach throughout its own library of classes. This approach is, necessarily, one which complies very closely with the CSP model. One possible disadvantage to implementing components in an active way could be the performance implications of having so many processes running in parallel on what is essentially a limited resource environment.

2.5.1 Transforming Listeners to Channels

The LeJOS implementation of the LEGO sensors (light, sound, ultrasound etc.) works in a typical Java fashion – a Listener object is created for various sensors, and when a sensor undergoes a state change, a method of the Listener object is called. The combination of the component object and its listener object can thus form the basis of an active component: the listener state change method, when called, writes a new token to the output channel of the active component. This style of implementation is required for any LEGO sensor that can be used to input data into the controller. This includes not only the input sensors, but motors which are able to input rotational information.

For the purposes of explanation we describe the implementation of the Active Touch Sensor process, simply because it is the least complicated but contains the essence of any of the active components.

The enclosing package `org.jcsp.nxt.io` is specified {1}[2] and then the required imports are defined from either the LeJOS NXT environment {2-4} or the JCSPre environment {5-7}. The specific classes are defined simply because the LeJOS environment only adds those required classes to the resulting downloadable classes, rather than the whole package. The class `ActiveTouchSensor` {8} implements the interfaces

[2] The notation {n} is used to indicate a line number in a code listing.

`CSProcess`, so it can be executed in a `Parallel` and `SensorPortListener` so that is can be notified of any changes in the sensor's state.

```
01  package org.jcsp.nxt.io;
02  import Lejos.nxt.TouchSensor;
03  import Lejos.nxt.SensorPort;
04  import Lejos.nxt.SensorPortListener;
05  import org.jcsp.nxt.lang.CSProcess;
06  import org.jcsp.nxt.lang.ChannelOutputInt;
07  import org.jcsp.nxt.lang.AltingChannelInputInt;

08  public class ActiveTouchSensor implements CSProcess, SensorPortListener {
09    private ChannelOutputInt outChannel;
10    private AltingChannelInputInt config;
11    private TouchSensor sensor;
12    private int lastValueTransmitted = -9999;
13    private int pressed;
14    private boolean booleanMode = false;
15    private boolean running = false;

16    public static final int ACTIVATE = -1;
17    public static final int BOOLEAN = -2;

18    public ActiveTouchSensor( SensorPort p,
19                              ChannelOutputInt outChannel,
20                              AltingChannelInputInt config ) {
21          this.sensor = new TouchSensor( p );
22          p.addSensorPortListener( this );
23          this.outChannel = outChannel;
24          this.config = config;
25    }
26    public void stateChanged( SensorPort p, int oldVal, int newVal ) {
27          if (booleanMode) {
28                if ( running && (newVal > 512) )
29                      pressed = 0;
30                else
31                      pressed = 1;
32                if (pressed != lastValueTransmitted) {
33                      outChannel.write( pressed );
34                      lastValueTransmitted = pressed;
35                }
36          }
37          else {
38                if (running && (newVal != lastValueTransmitted)) {
39                      outChannel.write( newVal );
40                      lastValueTransmitted = newVal;
41                }
42          }
43    }
44
45    public void run() {
46          int signal = config.read();
47          while( signal != Integer.MAX_VALUE ) {
48                if( signal == ACTIVATE )
49                      running = true;
50                else if (signal == BOOLEAN)
51                      booleanMode = true;
52                signal = config.read();
53          }
54    }
55  }
```

The private variables of the class are then defined {9-15} of which `outChannel` {9} and `config` {10} provide the channel interface to this process. Two publicly available constants are available, `ACTIVATE` {16}and `BOOLEAN` {17} that can be used to configure the component.

The constructor for the process {18-25} requires three parameters: the identity of the `SensorPort` (S1, S2, S3 or S4) {18} and the processes' output {19} and configuration {20} channels. The `SensorPort` identity is then used to create {21} an instance of a `TouchSensor` called `sensor`, which is private to this process. The process is then added as a sensor port listener {22}.

The interface `SensorPortListener` requires the implementation of a method called `stateChanged` {26-44}. The method is only active if the sensor process has received the `ACTIVATE` message on its configuration channel represented by `running` having the value `true`. The behaviour of the method depends whether the touch sensor is being operated in a 'pressed' mode or is required to output a value that is proportional to the amount of pressure applied to the sensor. In the 'pressed' mode the sensor simply outputs a 0 if the sensor is subject to no pressure and 1 if any pressure has been applied {28-31}. All sensors return a value in the range 0 to 1023 and we use `512` {28} simply as a means of ensuring that the sensor button has been pressed. A change of value is only written to the output channel provided the new value is different from that previously output {32-36}. In proportional mode the method outputs any change of value in the sensor provided it is different from the value that was last transmitted {38-40}.

The `run` method {45-54} is required by the interface `CSProcess`. Initially the sensor is inactive and so must receive at least one message on its configuration channel. The process thus waits until it receives at least one message on its configuration channel {46}, which is read into its `signal` variable. We have chosen not to use the poison mechanism [13] to cause a network of processes to stop processing but instead circulate `Integer.MAX_VALUE` as the universal system halt message. This choice was made on the grounds of reducing the size of the code required to support channels within the resulting system. The incoming configuration value is used to set the value of the local variables `running` {49} or `booleanMode` {51}. Values other than those expected on the configuration channel are ignored. The design of the run method is such that once the process has been set running and possibly been placed in the Boolean mode it waits for another input {52} on its configuration channel. The only sensible valid message is to read the halt value. Thus the process is, for the most part, waiting for a communication on its configuration channel, which consumes no processor resource. The underlying LeJOS kernel will only cause the invocation of the `stateChanged` method when it detects a change in the hardware state associated with this Touch Sensor's Sensor Port.

An additional complexity associated with the Active Light Sensor is that the sensor can be configured to recognise different input values: such as when detecting the input from either a black or white surface. It can also be configured to use, or not use, the sensor's floodlights. The `stateChanged` method can also be configured so that it only outputs changes in state that are larger than some configuration defined "delta" value.

Each sensor implemented in the LeJOS NXT environment has a corresponding implementation in the Active layer of the layered architecture shown in Figure 1.

3. Designing Robot Control Systems

The JCSPre environment comprises a number of elements contained within the `org.jcsp.nxt` structure which contains the packages `filters`, `io`, `lang`, `plugnplay`, `bluetooth` (see section 5) and `util`. The designer then utilizes processes contained within these packages to build their control system in conjunction with any control process(es) and user interface components that need to be coded specially. The designer has to construct a process network diagram of their controller, which will comprise mostly processes contained within the Active Sensor, Motors and other Processes layer (Figure 1). They then have to implement their control and user interface processes so the control system can be tested.

3.1 Filter Processes

In designing systems using this style of process construction it became apparent that a set of processes that could take an input on one channel and then output the value on a specific channel depending upon its value relative to some configuration parameters was very useful. Such processes are contained in the `filters` package Thus a Binary Filter process is provided that outputs an input value on one of two output channels depending on the input value relative to a single level value initially input on the process' configuration channel. Similarly, a Ternary Filter is provided that outputs on one of three output channels an input value depending on its value relative to two configuration values. Thus we can determine whether an input value is below a low value, between a low and high value or above the high value. These filter processes are generic in that all sensors return integer values typically between 0 and 1023 and thus the configuration values simply have to be set according to the requirements of the control application. Each filter process outputs the input value on whichever of its output channels is used. Figure 2 shows a 6-way filter constructed from one `BinaryFilter` and two `TernaryFilter`s. Each filter process has a configuration channel (c) which is used to initialise the filter with its boundary value(s). The output channels are labelled h – high, m – mid and l – low.

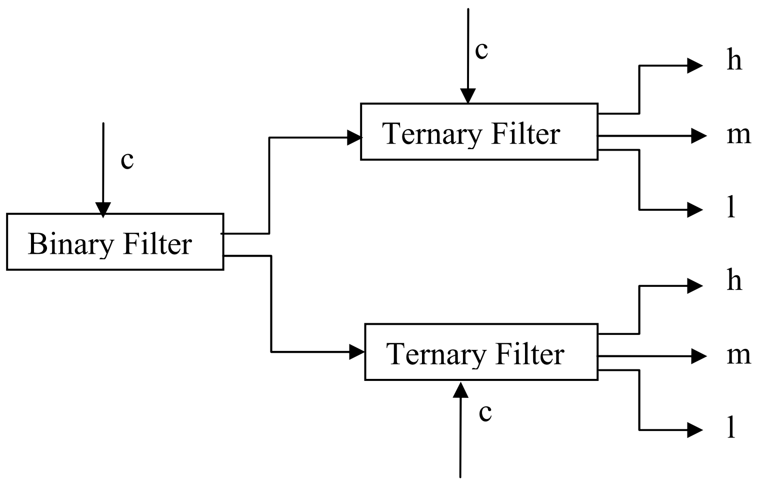

Figure 2: construction of a 6-way filter.

3.2 Input – Output Processes

This set of processes comprises the Active Sensor Processes and the Motor control processes. A variety of these motor control processes is provided depending on whether they are simply output processes or require some form of input capability because they are used as a sensor as well.

3.3 The lang and util Packages

These packages simply implement the JCSPre system and, as such, can be treated as black boxes.

3.4 The plugnplay Package

This package contains 'useful' processes that can be used to create common parallel network design idioms such as processes that copy an input from a single channel to a number of output channels as a sequence of channel outputs. In systems designed for large systems that use the full JCSP capability this is achieved using Delta and DynamicDelta processes. In the LEGO NXT environment this is not feasible because these processes use a large number of internal processes. This would tend to use scarce resources in an uneconomic manner.

3.5 Feasibility Testing

Initially, a program was created that used the underlying LeJOS thread model to discover how many child threads could be spawned. Each child process never terminates and it was discovered that 160 child processes could be spawned. A variant was created in which each child process held a simple piece of data and this resulted in the creation of 90 threads. This demonstrated that the number of threads was limited simply by the amount of memory used in each thread. A JCSPre program was created which comprised a pipeline of AddN processes, which add a constant value to an input value that is then output. An initial producer process sends an integer through the pipe of AddN processes to a consumer process that output a message on the robot LCD display. The system broke when the number of AddN processes reached 78. This was deemed sufficient to continue development because each AddN process has internal state and an input and an output channel. The point at which the system failed was indicated by an exception produced by the LeJOS virtual machine.

4. A Simple Line Following Robot

The photograph in Figure 3 shows a simple two motored robot with a rear trolley wheel (not visible) where each motor has an associated light sensor placed sufficiently far apart and to the front of the robot to be able to detect a line 15mm wide.

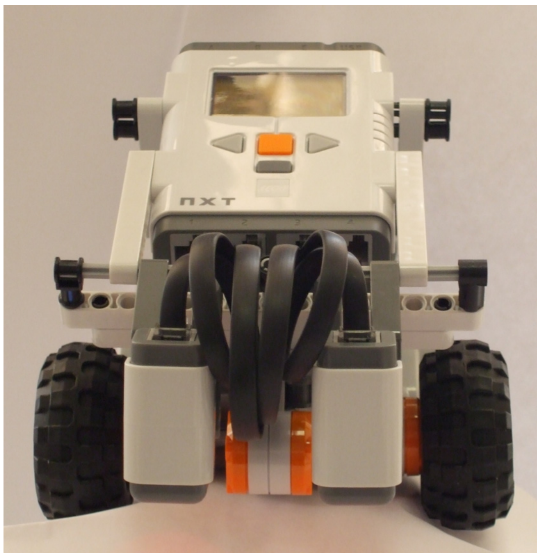

Figure 3: a simple two-motored robot with rear trolley wheel.

The control system is very simple in that if a sensor detects black, the colour of the line to be followed, the associated motor stops rotating, otherwise the motor rotates at a predetermined speed. Due to varying lighting conditions of the space in which the robot can operate it is necessary to configure the sensors so that it knows the sensor values for black and white. The speed at which the motors rotate can be set through a user interface, which also guides the user through the sensor configuration process.

4.1 The Control Process Network

Figure 4 shows the process network diagram for the line following robot. The left hand side shows the channels (solid lines) that are used during configuration of the system. The right hand side shows the channels (solid lines) used when the robot is running. In reality, all channels are required for the robot to function correctly.

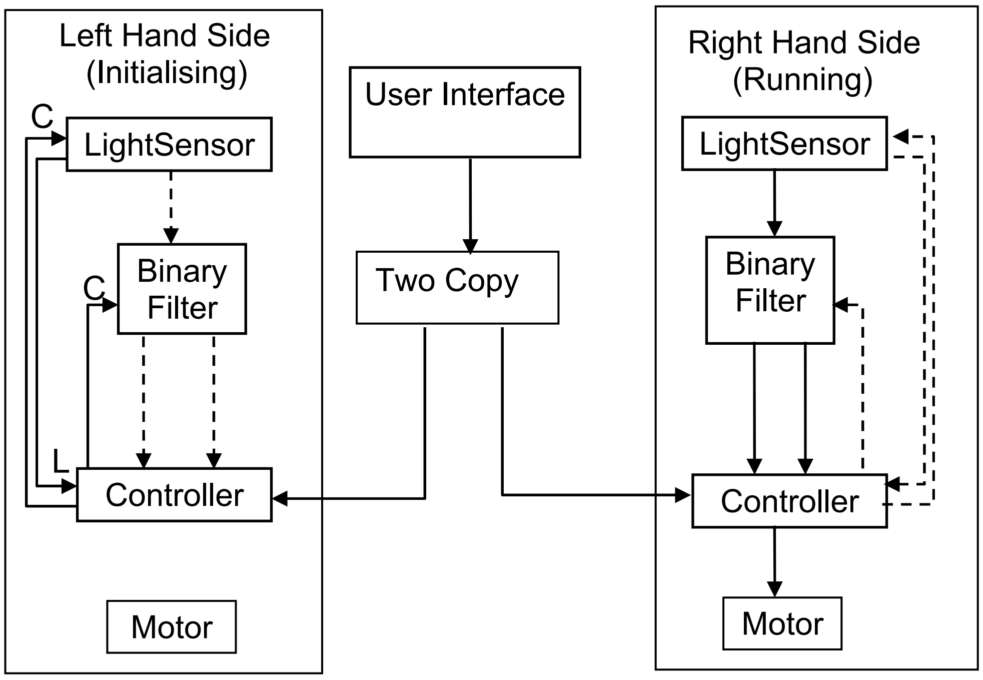

Figure 4: process network diagram for the simple line following robot.

The User Interface process guides the user through the configuration process. Messages are displayed on the LEGO NXT screen telling the user the stage reached. The user then responds by pressing buttons as required. Thus to configure the sensor black level the robot is placed over a black line such that both sensors are observing a black input. A button is then pressed, which is communicated to both sides of the control system to their Controller processes. This causes a message to be sent to the LightSensor process on its configuration channel (C) to determine the input level, which is returned to the Controller process using another channel, light level (L). The display then shows a message asking for the configuration of the white level, at the completion of which the Controller process has both the black and white levels. The point half way between these values is then sent to the BinaryFilter process on its configuration channel (C). The UserInterface process now requests the user to specify the speed at which the robot's wheels are to rotate. This is achieved using the left and right arrow buttons. The user then places the robot at the start of the line to be followed. Another button is then pressed to signify that the robot should start to follow the line which is sent to both Controller processes.

During movement the robot simply uses its LightSensor process (see the Right Hand Side Structure, solid lines) to detect a change in light value, which it outputs to the BinaryFilter process. Depending on the input value the BinaryFilter process outputs the input value on one of its output channels. The Controller process then waits for an input on either of its input channels and depending upon which channel the message is received it either outputs a stop or rotate message on the channel connecting it to the Motor process. This portion of the control loop is shown in the following code snippet.

An Alternative `a` is defined {55-56} which allows the Controller to choose amongst inputs from the Filter process on `filterHigh` and `filterLow` and the `buttonPress` channel from the user interface. The Controller comprises an unending loop {57-78}. The speed of wheel rotation is read from the `buttonPress` channel {58} followed by a simple button press that is interpreted as the go signal {59}. A further loop is now entered which terminates when the robot stops and a stop signal is received on the `buttonPress` channel {74}. A selection is made on the alternative `a` {62} which returns the index of the enabled input channel that has been selected. The following `if` statements {63-68, 69-72, 73-76} determine the action that should be undertaken depending upon the input that has been read. Normally, A `switch .. case` structure would be used but the LeJOS kernel does not contain such a capability as indicated earlier.

```
56  Alternative a = new Alternative(new Guard[]{ filterHigh, filterLow,
57                                               buttonPress});
58  while (true) {
59    rotate = buttonPress.read(); // read speed of wheel rotation
60    buttonPress.read(); // the go signal
61    boolean going = true;
62    while (going) {
63          altIndex = a.select();
64          if (altIndex == 0) {
65                filterHigh.read();
66                motorSpeed.write(ActiveMotor.SPEED);
67                motorSpeed.write(rotate);
68                motorSpeed.write(ActiveMotor.FORWARD);
69          }
70          else if (altIndex == 1) {
71                filterLow.read();
72                motorSpeed.write(ActiveMotor.STOP);
73          }
74          else (altIndex == 2) {
75                buttonPress.read(); // the stop signal
76                going = false;
77          }
78    }
79  }
```

In the case of an input read from `filterHigh` {64} then a sequence of messages are sent to the motor process{65-67} telling it to move forward at the indicated speed of rotation, which can be changed for each iteration of the main loop. In the case of an input from `filterLow` {70}, which represents the sensing of the black line, the motor process is sent the stop signal {71}. Finally, a button press can be read {74} which has the effect of terminating the internal loop by setting `going false` {75}. It should be noted that the robot comes to a stop if both its sensors sense black as the same time, which then enables the pressing of a button. Recall that each side of the robot has its own Controller process and thus these are both running concurrently.

In total, the system comprises 12 processes: a process for each side of the robot plus the processes shown in Figure 2.

During a series of Christmas Lectures [21], aimed at 14-17 school students, participants were asked to control the same robot, with no change to its `Controller` process to guide it through a slalom course using a black lollipop. The black lollipop was placed under the required sensor, by the participant, in order to control the movement of the robot through the slalom course. It was then shown that by placing a black line down the middle of the course the same effect could be achieved.

5. Slave Robots using Bluetooth

To evaluate the Bluetooth capability of the LEGO NXT robot an experiment was proposed in which one, master, robot followed a line and a second, slave, robot, without sensors, was sent messages telling it how to move. The slave robot was furnished with a pen so that it could draw the path it had taken on a sheet of paper. The path the master robot took could then be compared with that followed by the slave robot. The nature of the messages sent from the master to the slave is very simple. They were a single integer that indicated whether a motor was moving or rotating. It was presumed that a motor remained in the same state until the controller changed it.

In order to set up a Bluetooth connection an interaction between the two communicating robots is required. Each robot has to be made aware of the possible other devices with which it can communicate. Once this is established then subsequent connection at run-time is relatively easy. A Java `DataInputSteam, dis` {79} and a `DataOutputStream, dos` {80} are defined. A LeJOS Bluetooth connection `btc` is then defined {81}. The connection is made {82} after which the data streams can be opened {83, 86} and an initial interaction undertaken {84, 85}. The code snippet ignores exception handling and shows the output side of the initial interaction. The receiving process simply opens an input data stream, reads from it and then opens a data output stream..

```
80  DataInputStream dis = null;
81  DataOutputStream dos = null;
82  BTConnection btc;
83  btc = Bluetooth.waitForConnection();
84  dos = btc.openDataOutputStream();
85  dos.writeInt(1);
86  dos.flush();
87  dis = btc.openDataInputStream();
```

Taking the architecture shown in Figure 4, a further process, Bluetooth Transmit, was added to the master robot network, which had two input channels, one from each Motor process. A corresponding output channel was added to each Motor process. Whenever the state of a motor changed, it output a corresponding value to the Bluetooth Transmit process, which then caused this value to be transmitted to the slave robot over the Bluetooth connection.

The slave robot had a much simpler process architecture comprising a Bluetooth Receive process together with two Motor processes for each wheel of the robot as shown in Figure 5. The BTReceive process inputs data values from the Bluetooth data input stream connection that has been created. Depending on the value a message is sent to one of the Motor processes. A Motor process is expecting to receive a zero (0), indicating it should stop or a one (1) indicating it should rotate. The motor stays in that state until otherwise changed.

The code snippet {88-94} captures this behaviour. The channels outChannelLeft and outChannelRight provide the connection between the BTReceive process and the Motor processes respectively.

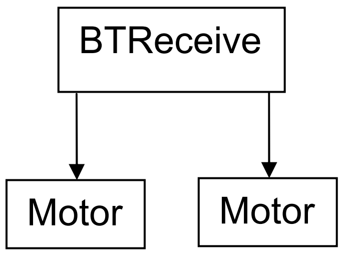

Figure 5: slave robot process network.

The Bluetooth Receive process receives an input {89}, which depending on the value {90, 92}causes a message to be sent to the appropriate Motor process {91, 93},which then effects the required change in motor behaviour. The values cause a motor to switch on or switch off.

```
88  while(true){
89    value = dis.readInt();
90    if(value == 0 || value == 1){
91          outChannelLeft.write(value);
92    }else if(value == 2 || value == 3){
93          outChannelRight.write(value - 2);
94    }
```

A video showing this basic operation is available [22]. It demonstrates that the basic operation of the slave robot is a reasonable mimic of the master robot. However because the robot simply moves its motors for the same time as indicated by the master robot it is possible for the slave to wander from the desired path. In particular, the orientation of the slave robot and in particular its trolley wheel alignment is crucial if the robot is not to wander from the path from the outset. This behaviour can be observed in the video, which also shows that once the trolley wheel effect has been removed the remainder of the movement is in fact a reasonable copy of the path taken by the master robot.

6. Conclusions and Further Work

The achievement of the work reported in this paper was that it demonstrated the feasibility of building robot controllers for small factor robot systems such as the LEGO NXT robot using parallel programming techniques in a Java environment. This was achieved by building a highly reduced version of an extant parallel environment (JCSP) and implementing this on top of a publicly available small Java kernel and virtual machine (LeJOS). A process based abstraction of each of the components that are available in the LeJOS NXJ packages were developed so that a truly parallel control network could be implemented. The basic functionality was demonstrated by means of a simple line following robot. Finally, it was shown that a master line following robot could control the operation of a slave robot such that the slave robot could mimic the path followed by the master using a very simple communication message sequence that utilized the Bluetooth communication capability of the LEGO NXT robot.

The LeJOS system contains an implementation of a subsumption architecture. This work is currently ongoing in terms of a process based implementation of such an architecture. The latest release of LeJOS is referred to as a Beta release and overcomes many of the shortcomings that caused much work, for example, the lack of `switch` statements. The release also contains a means of sending messages from the robot to the PC used to download the code in the form of a remote console capability. This capability needs to be incorporated into the Eclipse environment to ensure that users, and in particular students can develop and test their designs in a single development environment.

The major challenge though is to develop the system to a point where sufficient basic processes have been created to enable the easier design of parallel robot control systems. It is anticipated that this will be achieved as part of the development of an undergraduate module that will teach parallel programming techniques through the use of robot controllers.

Acknowledgments

Both Alex Panayotopoulos and Patrick Lismore acknowledge the support they received from the Student Awards Agency Scotland for payment of their fees for the programmes of study they were undertaking while they developed parts of JCSPre and the demonstrations. The work was undertaken as part of the project element of a Masters and Bachelors qualification respectively.

References

[1] J. Simpson, C. L. Jacobsen, and M. C. Jadud, "A Native Transterpreter for the LEGO Mindstorms RCX," in A. McEwan, S. Schneider, W. Ifill, and P. H. Welch (Eds.), Communicating Process Architectures 2007, pp. 339-348, IOS Press, Amsterdam, 2007.

[2] Transterpreter.org, "LEGO Mindstorms RCX and NXT," 2008. Retrieved 25 March 2008 from http://www.transterpreter.org/docs/LEGO/index.html.

[3] LEGO.com, "Mindstorms NXT Software," 2008. Retrieved 7 March 2008 from http://mindstorms.LEGO.com/Overview/NXT_Software.aspx.

[4] J. Kodosky, "Is LabVIEW a General-purpose Programming Language?," 2008. Retrieved 7 March 2008 from http://zone.ni.com/devzone/cda/tut/p/id/5313.

[5] A. Tan, "Call for Asia to Adopt ODF," 2006. Retrieved 12 March 2008 from http://www.zdnetasia.com/news/software/0,39044164,39380446,00.htm.

[6] G. Theodoropoulos, "Modelling and Distributed Simulation of Asynchronous Hardware," Simulation Practice and Theory, 7(6), pp. 741-767, 2000.

[7] C. G. Ritson and P. H. Welch, "A Process-Oriented Architecture for Complex System Modelling," in A. McEwan, S. Schneider, W. Ifill, and P. H. Welch (Eds.), Communicating Process Architectures 2007, pp. 249-266, IOS Press, Amsterdam, 2007.

[8] C. L. Jacobsen, D. J. Dimmich, and M. C. Jadud, "Native Code Generation Using the Transterpreter," in P. H. Welch, J. Kerridge, and F. R. M. Barnes (Eds.), Communicating Process Architectures 2006, pp. 269-280, IOS Press, Amsterdam, 2006.

[9] S. Jørgensen and E. Suenson, "trancell - an Experimental ETC to Cell CE Translator," in A. McEwan, S. Schneider, W. Ifill, and P. H. Welch (Eds.), Communicating Process Architectures 2007, pp. 287-297, IOS Press, Amsterdam, 2007.

[10] N. Brown, "C++CSP2: A Many-to-Many Threading Model for Multicore Architectures," in A. McEwan, S. Schneider, W. Ifill, and P. H. Welch (Eds.), Communicating Process Architectures 2007, pp. 183-205, IOS Press, Amsterdam, 2007.

[11] J. M. Bjørndalen, B. Vinter, and O. Anshus, "PyCSP - Communicating Sequential Processes for Python," in A. McEwan, S. Schneider, W. Ifill, and P. H. Welch (Eds.), Communicating Process Architectures 2007, pp. 229-248, IOS Press, Amsterdam, 2007.

[12] L. Yang and M. R. Poppleton, "JCSProB: Implementing Integrated Formal Specifications in Concurrent Java," in A. McEwan, S. Schneider, W. Ifill, and P. H. Welch (Eds.), Communicating Process Architectures 2007, pp. 67-88, IOS Press, Amsterdam, 2007.

[13] P. H. Welch, N. Brown, J. Moores, K. Chalmers, and B. H. C. Sputh, "Integrating and Extending JCSP," in A. McEwan, S. Schneider, W. Ifill, and P. H. Welch (Eds.), Communicating Process Architectures 2007, pp. 349-370, IOS Press, Amsterdam, 2007.

[14] P. H. Welch and J. Martin, "Formal Analysis of Concurrent Java Systems," in P. H. Welch and A. W. P. Bakkers (Eds.), Communicating Process Architectures 2000, pp. 275-301, IOS Press, Amsterdam, 2000.

[15] JCSP Project, "CSP for Java (JCSP)," 2008. Retrieved 3 April 2008 from http://www.cs.kent.ac.uk/projects/ofa/jcsp/explain.html.
[16] JCSP Project, "CSP for Java (JCSP) 1.1-rc3 API Specification," 2008. Retrieved 3 April 2008 from http://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp-1.1-rc3-doc/.
[17] LeJOS Project, "NXJ Technology," 2008. Retrieved 16 April 2008 from http://LeJOS.sourceforge.net/p_technologies/nxt/nxj/nxj.php.
[18] LeJOS Project, "README," 2008. Retrieved 16 April 2008 from software download of LeJOS-NXJ-win32, version 0.3.0alpha, available at http://sourceforge.net/project/showfiles.php?group_id=9339.
[19] LeJOS Project, "LeJOS NXT API Documentation," 2008. Retrieved 16 April 2008 from http://LeJOS.sourceforge.net/p_technologies/nxt/nxj/api/index.html.
[20] Sun Microsystems, "CLDC Library API Specification 1.0," 2008. Retrieved 16 April 2008 from http://java.sun.com/javame/reference/apis/jsr030/.
[21] Institution of Engineering and Technology, "Christmas Lecture 2007," 2007. Retrieved 18 June 2008 from http://tv.theiet.org/technology/infopro/898.cfm.
[22] P. Lismore, "2 NXT Robots With Parallel Java Brains Test 2," 2008. Retrieved 22 April 2008 from http://www.youtube.com/user/plismore.