Chapter 6

Comparison to other languages

This chapter uses working examples to compare Meshoil with other programming languages. These head-to-head examples show how programs written in these languages can be rewritten in Meshoil. They demonstrate Meshoil’s versatility as a general-purpose programming language capable of emulating programming constructs of other languages:

- Universal Remote Console (page 238)—URC’s own example of controlling a digital thermometer remotely from a hand-held console
- MindScript (page 254)—a robotic application based on Lego’s robotic control language called MindScript.
6.1 Universal Remote Console—URC

6.1.1 Overview

The software design of the digital thermometer used in this example is described in a founding URC specification that has been published as part of an ANSI standard [21]. This standard advocates use of URC as a solution to the universal usability problem.

The Meshoil program in this example:

- rewrites URC’s high-level description of thermometer operation in the equivalent Meshoil
- emulates SIC features of URC syntax for handling communication between the console and the thermometer
- simulates the actual running of the thermometer.

The first two of these functions demonstrate how Meshoil is able to match the functionality of URC in a more abstract and generalised way. The third function of controlling the running of the thermometer is beyond the scope of URC.

6.1.2 The URC approach to universal usability

The URC approach allows users to control URC-compliant products remotely by using a PDA or similar device acting as a universal remote console. Detailed specifications were published in August 2005 as a set of five ANSI standards.

A white paper [46] produced by the URC consortium states that:

What is needed is a standardized, versatile user interface description for products, a “User Interface Socket” to which any URC can connect to discover, access and control a remote product. The User Interface Socket would provide an abstract description of the product’s functions, and the URC would build custom-tailored interaction mechanisms based on the description.

URC-compliant, remote products are called targets, a typical example being a domestic appliance such as a TV or alarm clock.

The user interface socket lies at the heart of the URC approach. The ultimate effectiveness of URC rests on how expressive this socket is in describing the capabilities of the target. If the syntax for a user interface socket only supports limited control of the target, then URC, for all its many advantages, can only hope to offer a limited solution for the universal usability problem.

The rest of the URC approach is concerned with issues resulting from the relationship between the console and its targets, such as uploading user interface sockets from the targets where they are stored to the console. These peripheral issues are not simulated in this working example.
6.1.3 URC’s example of a digital thermometer

URC’s digital thermometer example is useful because it highlights the main features of URC’s user interface socket syntax.

The thermometer is capable of displaying the current, minimum and maximum ambient temperatures. The URC user interface socket description for the thermometer is listed below:

```xml
<uiSocket about="http://www.mycorp.com/thermometer/socket"
    id="socket" xmlns="http://www.incits.org/incits390-2005"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <static id="modelNumber" type="xsd:double">
    570
  </static>
  <variable id="temperature" type="xsd:double">
    <dependency write="false()"/>
  </variable>
  <variable id="maximum" type="xsd:double">
    <dependency write="false()"/>
  </variable>
  <variable id="minimum" type="xsd:double">
    <dependency write="false()"/>
  </variable>
  <variable id="scale" type="scaleType" />
  <command id="reset" />
  <notify id="checkReset" explicitAck="false" category="alert">
    <dependency acknowledge="value(confirmReset)=done or value(cancelReset)=done"/>
  </notify>
  <command id="confirmReset" type="uiSocket:basicCommand">
    <dependency read="value(checkReset)=active" execute="value(checkReset)=active"/>
  </command>
  <command id="cancelReset" type="uiSocket:basicCommand">
    <dependency read="value(checkReset)=active" execute="value(checkReset)=active"/>
  </command>
  <xsd:schema>
    <xsd:simpleType name="scaleType" id="idScaleType">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="F"/>
        <xsd:enumeration value="C"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:schema>
</uiSocket>
```

The user interface socket syntax has two types of XML element for controlling the target: the `<variable>` and `<command>` elements. Variables are used to control simple states of operation of the target. For example, the thermometer can display temperatures in either centigrade or Fahrenheit. This is represented by the string type variable called scale which can be set to C or F.

For more complex operations, the method of using the states of variables to represent what the target is to do breaks down. Controlling the target is handled by the `<command>` element instead. The example is given of a reset operation that resets the thermometer’s current minimum and maximum readings to the current temperature. This operation is defined as a two-step process. The user is required to confirm or cancel the intended reset operation once the command for the reset has been issued.

Much of the user interface socket syntax is concerned with timing issues. The first such timing
issue is when commands are selectable. Command usage is controlled by use of `<notify>` elements which suspend normal user interface operation. A `<notify>` element can be `active`, `inactive` or `stacked`. This element is effectively a flag for controlling when a command can be executed. One command might set a notification that controls whether another command can be selected or not. For example, the `confirmReset` and `cancelReset` commands in the thermometer example are only selectable immediately following the selection of the `reset` command while the corresponding `<notify>` element for the `reset` command called `checkReset` is `active`. The dependency of the `confirmReset` and `cancelReset` commands on the value of the `checkReset` `<notify>` element is specified in `<dependency>` elements that make up the definitions of the two commands.

The second timing issue is two-way notification of events between the console and its targets. It may take time for a command issued from the console to be successfully executed in the target. To synchronise the process, the device informs the console of the current state of command execution. The state of a command can be `ready`, `inProgress`, `done` or `error`. The console can only issue a command when the status for that command is `ready`.

### 6.1.4 URC a command-driven language

Any local processing on the console remains limited to simple expressions. The user interface socket is basically a list of predefined commands that the target recognises, with information on when the commands can be used. Even the `scale` variable with C and F settings is really no more than shorthand for the alternative of having the two extra commands `Set to C` and `Set to F`.

We can conclude that URC’s approach to universal usability is fundamentally to support a command-driven system of device control. From the console’s point of view, the thermometer is a black box that will somehow successfully interpret the meaning of the commands it issues. For example, the console has no awareness of what the `reset` command means, merely that it is something that the thermometer must act on. For the thermometer, `reset` means updating the minimum and maximum values with the current temperature value.

### 6.1.5 Limitations of a command-driven approach

Even in such a simple example as a thermometer, the restrictions that a command-driven approach impose on design flexibility are apparent.

What if the user wanted to customise the user interface by having the ability to reset the minimum and maximum separately? Perhaps having commands called `resetminimum` and `resetmaximum` that could be issued at any time on their own to perform an individual reset, or together if a full reset was required? This would avoid the current situation of always being forced to reset the two together. There is nothing stopping the creation and use of such commands in the console, but they would not work because they would not match how the thermometer has been programmed to operate.

It isn’t possible for URC to control the behaviour of the thermometer in ways like these since thermometer operation is hard-coded and inaccessible to change. URC is only able to solve part of the universal usability puzzle. It can define the part of the user interface handled in the console but not what happens in the thermometer itself.
6.1.6 The Meshoil program

Figure 6.1 on page 242 shows the complete Meshoil program. It consists of one thread for running the user interface at the console, and another for simulating the operation of the thermometer.

In URC, communication between console and target may be instigated by the target in the form of an alert. This allows the target to inform the user at the console of some condition arising in the target that the user should be aware of. It also allows the target to influence the running of the user interface on the console by changing the status of notifications.

When a target activates a notification while another notification is already active, the second notification takes priority over the first. The first notification is put in a queue and its status changes from active to stacked. When the second notification becomes inactive, the first notification reverts to active.

A target generated alert is included in the Meshoil program to demonstrate this functionality, although it is not part of the original URC example. The alert is a checkFire notification that triggers whenever the sensor climbs past 100 centigrade. If the user is in the middle of the reset operation when the thermometer generates a fire alert, the user is required to acknowledge this alert before being allowed to complete the reset operation.
Figure 6.1: Simulating thermometer operation (1 of 6)
Figure 6.1: Simulating thermometer operation (2 of 6)
CHAPTER 6. COMPARISON TO OTHER LANGUAGES

Figure 6.1: Simulating thermometer operation (3 of 6)
Figure 6.1: Simulating thermometer operation (4 of 6)
Figure 6.1: Simulating thermometer operation (5 of 6)
6.1.7 Running the thermometer simulation

Figure 6.2 shows a sequenced example of running the thermometer simulation.

Frame 1  The window has three groups of controls. The top group represents the console. console display is the physical display unit on the console for displaying messages. scale is the variable that specifies whether the thermometer should be working in centigrade or Fahrenheit. command allows the user to select any one of the four available commands: Reset, Confirm reset, Cancel reset and Acknowledge alert.

The middle group of controls handle the communication between the console and thermometer. comms buffer represents the port used by the console to output commands to the thermometer. In the simulation, the thread running the thermometer constantly monitors comms buffer for changes to its setting made by the thread running the console. checkReset is the notification used during the reset operation. checkFire is the notification used during a fire alert condition generated when the thermometer detects the temperature exceeding 100 centigrade. stateReset holds the state of the reset command.

The bottom group of controls represents the hardware controls in the thermometer device itself. The temperature, minimum and maximum controls are display fields on the thermometer showing the respective current, minimum and maximum temperatures in whatever scale is currently selected. sensor is the physical sensor unit that outputs temperature readings, always in centigrade.

Figure 6.2: Running the thermometer simulation (1 of 7)
Frame 1  Here, all three temperature readings on the thermometer display are 20 centigrade since the thermometer has just been reset. Both the `checkReset` and `checkFire` notifications are currently inactive showing that the thermometer is operating normally.

Frame 2  The temperature rises (simulated by manually increasing the setting of the `sensor`). The thread representing the thermometer continuously updates the corresponding temperature, minimum and maximum values as necessary. Here, the maximum value is shown to be tracking the temperature rise.

Frame 3  The temperature falls. After passing the old minimum of 20 centigrade, the minimum value starts tracking the fall.

Figure 6.2: Running the thermometer simulation (2 of 7)
6.1. UNIVERSAL REMOTE CONSOLE—URC

The temperature climbs back up to 14 centigrade, resulting in the temperature, minimum and maximum values all being different.

On the console, the user switches to Fahrenheit by changing the scale setting from C to F.

The console uses the comms buffer to indicate the change in the scale setting and instruct the thermometer to switch from centigrade to Fahrenheit. The thread running the console has no understanding of what this means, relying on the thermometer to correctly interpret the command.

For the thread running the thermometer, changing to Fahrenheit means individually converting the currently displaying temperature, minimum and maximum values to Fahrenheit, and from then on converting each new centigrade reading output from the sensor to its Fahrenheit equivalent before using it to update the temperature, minimum and maximum values as necessary.

Figure 6.2: Running the thermometer simulation (3 of 7)
CHAPTER 6. COMPARISON TO OTHER LANGUAGES

Frame 6  The temperature falls from 14 to 10 centigrade. Although the sensor continues to generate readings in centigrade, temperature, minimum and maximum now track changes in Fahrenheit.

Frame 7  The user selects the Reset command to start the two-step reset operation (the console immediately resets the command control to Which command? ready for the user to select the next command).

The user is asked to confirm the reset operation. The checkReset notification changes from inactive to active. While in this state, normal user operation at the console is suspended. For example, changing the scale setting will have no effect.

Suspension of normal user operation only affects the console. The thermometer itself continues to run normally showing the correct temperature, minimum and maximum values as the sensor outputs temperature changes. Here, it has dropped another two degrees centigrade.

Figure 6.2: Running the thermometer simulation (4 of 7)
Frame 8 Before the reset operation can be completed by selecting the Confirm reset or Cancel reset command, a fire is simulated by increasing the temperature to 105 centigrade. At the point that the temperature rises past 100 centigrade, the thermometer generates a fire alert. The fire alert is indicated by the checkFire notification changing from inactive to active. This alert takes priority over the reset operation and causes the checkReset notification to change from active to stacked. The user must now acknowledge the alert by selecting the Acknowledge alert command before the reset operation can continue.

Frame 9 The user attempts to ignore the fire alert and continue with the reset operation by selecting the Confirm reset command. The console ignores the Confirm reset command and reminds the user that the fire alert must be acknowledged first. 

Figure 6.2: Running the thermometer simulation (5 of 7)
CHAPTER 6. COMPARISON TO OTHER LANGUAGES

Frame 10  The user selects the Acknowledge alert command.

Having acknowledged the fire alert, checkFire notification returns to inactive and checkReset becomes active once again.

Frame 11  Now able to complete the reset operation, the user selects the Confirm reset command.

Up to this point, control of the reset operation has remained local to the console, but now the console finally allows the Reset command to be sent to the thermometer via the comms buffer.

On receiving the command, the thermometer starts the reset process. It indicates this to the console by changing stateReset from ready to inProgress. The console asks the user to wait until the command has successfully been processed.

Figure 6.2: Running the thermometer simulation (6 of 7)
6.1. UNIVERSAL REMOTE CONSOLE—URC

On completing the task of resetting the minimum and maximum values to the current temperature, the thermometer indicates that the command has successfully been processed by changing stateReset to done. The console detects this state change and informs the user accordingly.

Somewhat belatedly, the user now investigates the fire!

Figure 6.2: Running the thermometer simulation (7 of 7)

6.1.8 Conclusion for the URC example

A URC program was rewritten in Meshoil, achieving a result that was not only more comprehensive but also less complex. In addition to providing a high-level description of thermometer operation, the Meshoil program was able to control the actual running of the thermometer device, something that was beyond the scope of URC. Table 6.1 shows how this was achieved by a solution that managed to halve the structural complexity of the URC program.

<table>
<thead>
<tr>
<th>Metric</th>
<th>URC</th>
<th>Meshoil</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of types of XML element required for the solution.</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Approximate number of lines of meaningful code in XML schema, as indicator of syntactic complexity.</td>
<td>120 (URC socket)</td>
<td>60 (Meshoil tester)</td>
</tr>
</tbody>
</table>

Table 6.1: Half the complexity for the Meshoil solution

The example served to demonstrate the greater range of Meshoil as a language for universal usability over URC and other UIDLs that suffer the limitation of being strictly declarative programming languages.
6.2 MindScript—Lego’s robotic control language

6.2.1 Overview

This example demonstrates the use of Meshoil as a programming language for a robotic application. A program for robotic control is first written in Lego’s proprietary MindScript language (Lego PBrick Script)—an application-specific language for programming Lego robots. The program is then rewritten in Meshoil to demonstrate Meshoil’s greater flexibility as a programming language and ability to emulate specialised programming constructs using generic methods. The Meshoil program also simulates the environment in which the robot operates.

The robot is a simple, wheeled vehicle made out of Lego bricks. It travels forward while there is nothing in the way. If it bumps into something, it tries to get round the obstacle by going backwards for a few seconds, swivelling left a little and going forward again. If conditions get gloomy, the side light comes on. If it gets too dark, the robot stops moving and the side light switches off.

6.2.2 Lego robots

Lego robots are built using special, programmable Lego bricks called PBricks. The PBrick used in this example is the RCX—a large, rectangular Lego brick containing a microprocessor that can be incorporated into Lego designs to make robots that move and react to external stimuli.

![Figure 6.3: Lego's RCX programmable brick](image)

MindScript programs are written on a computer and downloaded to the RCX to control the electrical components attached to the RCX. The RCX has six connectors for plugging in components. The three output connectors are typically used to drive motors. The three input connectors are typically used for touch and light sensors.
While MindScript programs can be written using a text editor, the language is designed to underpin future GUI programming tools that will simplify the task of programming. The user guide (called the ‘Lego P-brick Script Code Language’ [28]) states that MindScript is to be used:

as the implementation language for the graphical programming environment to be provided as part of the RIS2.0 Set, to implement other graphical environments and as a starting point for future scripting languages.

This would be an interesting future direction for Meshoil as well, since any attempt at providing GUI programming tools relies on the syntax of the underlying language being easily expressed in a graphical environment. Meshoil, with its highly modular structure and simple syntax, would seem to be a good candidate for this.

### 6.2.3 The MindScript program

Programs written in MindScript consist of five main structural components:

- **macro**—defines a sequence of commands that can be referenced by the name of the macro wherever those commands need to be executed
- **logical connection**—defines the link between a variable used in the program representing one of the robot’s components, such as a motor or sensor, and the physical input or output connector on the PBrick
- **event**—defines a particular state of a sensor as a recognisable event
- **main program**—where program execution begins
- **watcher**—defines the list of macros to execute when specified events happen.

The MindScript program for controlling the robot is listed below. It consists of five main blocks of code, one for each of the five structural components listed above:

```
program robot {

// Macros
// Define macros that make the robot move forward, turn left and move backward.
macro MoveForward {
    forward LeftMotor
    forward RightMotor
}
macro TurnLeft {
    backward LeftMotor
    forward RightMotor
}
macro MoveBackward {
    backward LeftMotor
    backward RightMotor
}
macro StopMoving {
    off LeftMotor
    off RightMotor
}

// Logical connections
```
// Define logical connections between the RCX's connectors and software.
output LeftMotor on 1
output RightMotor on 2
output SideLight on 3
sensor TouchSensor on 1
sensor LightSensor on 2

// Events
// Define touch-related and light-related events.
event BumpEvent when TouchSensor.pressed
event DayEvent when LightSensor is 51..100
event DuskEvent when LightSensor is 21..50
event NightEvent when LightSensor is 0..20

main {
    // When first switching on the robot:
    // 1. Start the robot moving forward if it dusk or day but not night.
    if LightSensor is 21..100 {
        MoveForward
    }
    // 2. Switch on the side light if it is dusk.
    if LightSensor is 21..50 {
        on SideLight
    }
}

// Watchers

// Bump Watcher
// Define a watcher for monitoring a bump event.
// When a bump occurs, make the robot reverse for six seconds, turn left
// for three seconds and resume going forward.
watcher Hit monitor BumpEvent {
    priority 2
    try {
        MoveBackward
        wait 600
        TurnLeft
        wait 300
        MoveForward
    } stop on fail
}

// Day Watcher
// Define a watcher for switching off the side light if it becomes day.
watcher Night monitor DayEvent {
    priority 1
    try {
        MoveForward
        off SideLight
    } stop on fail
}

// Dusk Watcher
// Define a watcher for making the robot move and for switching
// on the side light if it becomes dusk.
watcher Night monitor DuskEvent {
    priority 1
try {
    MoveForward
    on SideLight
} stop on fail
}

// Night Watcher
// Define a watcher for stopping the robot and switching off the side light if
// it becomes night.
watcher Night monitor NightEvent {
    priority 1
    try {
        StopMoving
        off SideLight
    } stop on fail
}
}

The `MoveForward`, `TurnLeft`, `MoveBackward` and `StopMoving` macros each hold the commands for making the robot move in a desired direction. For example, `TurnLeft` makes the robot spin anticlockwise on the spot by making the motors drive the wheels in opposing directions.

The logical connections define the associations between the `LeftMotor`, `RightMotor`, `SideLight`, `TouchSensor` and `LightSensor` variables used in the program, and five of the RCX’s connectors.

Four events are defined. `BumpEvent` is a touch-related event. `DayEvent`, `DuskEvent` and `NightEvent` are light-related events. The program splits ambient light into three levels: day for light sensor readings of 51%–100%, dusk for 21%–50% and night for 0%–20%. A light-related event is detected whenever the ambient light crosses the threshold between two levels.

The main program holds the code to get the robot working when it is powered up. If it is day or dusk, the `MoveForward` macro is called to get the robot moving. If it is dusk, the `SideLight` is switched on. This code is necessary because the Day Watcher and Dusk Watcher will only execute at the transition point when a DayEvent or DuskEvent occurs. They do not execute during steady-state conditions.

The four events are individually monitored by the Bump Watcher, Day Watcher, Dusk Watcher and Night Watcher. The three light-related watchers are given priority over the Bump Watcher by including priority and try statements in all four watcher definitions. The Bump Watcher is set at priority 2. The light-related watchers are each set at priority 1 (the highest). As a result, if a light-related event happens while the robot is trying to avoid an obstacle, it will abort the avoidance procedure and respond to the light-related event instead.

The program is simply written to demonstrate the main features of a MindScript program. It is not designed to be foolproof in the way that it controls the robot’s behaviour! For example, the robot is meant to stop moving at night. Although a NightEvent will stop the robot moving as night falls, bumping the stationary robot at night will still trigger the Bump Watcher and start the robot moving backward as part of the obstacle avoidance procedure.

### 6.2.4 Using Meshoil to simulate the robot’s environment

In addition to code equivalent to the MindScript program above that controls the robot, the Meshoil program contains code for simulating the running of the robot and controlling the ambient conditions under which the simulation runs.
Hardware controls in the Meshoil program consist of the Battery powering the RCX and all connected components, the Run button on the RCX, and the three Input and three Output ports on the RCX. The controls representing the various motors, lights and sensors remain programming variables that do not need to be declared as hardware controls in their own right since they are defined as being logically associated with the Input and Output ports.

The LeftMotor and RightMotor controls represent the motors driving the robot’s left and right wheels, either in a forward or backward direction. Mounted on the robot is the SideLight, TouchSensor and LightSensor. The TouchSensor responds to being touched by changing its state from Released to Pressed. The LightSensor indicates the strength of the ambient light as a percentage.

The Run button on the RCX switches the RCX’s Battery on and off. When the Battery is on, the part of the Meshoil program representing the MindScript program starts executing and the robot starts working. Pressing the Bump button simulates the effect of physically bumping the robot. The Sunshine control simulates the ambient light that the robot’s LightSensor detects.

The Status control displays graphically what the robot is doing at all times.

6.2.5 The equivalent MindScript program in Meshoil

Figure 6.4 on page 259 shows the part of the Meshoil program equivalent to the MindScript program for controlling the robot. The code has been written to mimic the structure of the MindScript program.

In MindScript, macros and events are mechanisms used to define blocks of code that can be referenced by name elsewhere in the program. Macros consist of a list of commands—code that is to be executed. Events consist of defined conditions—code that is to be interrogated on the basis of which other code will execute. Both constructs are used in watchers: the watcher monitors events and executes macros based on the events occurring.

In Meshoil, this SIC distinction between events and macros is unnecessary, since blocks of code can be inserted anywhere in the program. The functional distinction is made, not by using different statements and syntax, but by simply inserting code at different locations. A MindScript event is emulated by inserting a brick into the body of another brick to form part of a conditional expression. A MindScript macro is emulated by inserting a brick as a self-contained brick at the location where it is to execute.

The Logical connections brick is an example of the way in Meshoil that dynamic associations can be set up for programming variables that are maintained automatically during runtime (for details, see appendix A.4.3 on page 303).

Priority execution of the three light-related watchers DayEvent/DuskEvent/NightEvent over the Bump Watcher is achieved without the need for SIC programming constructs such as the priority and try statements used in the MindScript program. Instead, the program uses an additional control called Procedure. Execution of the Bump Watcher is made dependent on this control being set to Avoidance. All the light-related watchers have to do to abort continuing Bump Watcher execution is to reset this control.
6.2. MINDSCRIPT—LEGO'S ROBOTIC CONTROL LANGUAGE

Figure 6.4: Meshoii equivalent of the robot program (1 of 5)
CHAPTER 6. COMPARISON TO OTHER LANGUAGES

Figure 6.4: Mesoil equivalent of the robot program (2 of 5)
Figure 6.4: Meshoil equivalent of the robot program (3 of 5)
CHAPTER 6. COMPARISON TO OTHER LANGUAGES

Figure 6.4: Meshoil equivalent of the robot program (4 of 5)
6.2. MINDSCRIPT—LEGO’S ROBOTIC CONTROL LANGUAGE

6.2.6 Running the robot simulation

Figure 6.5 shows a sequenced example of running the robot simulation.

Frame 1 The simulation starts with the battery in the RCX switched off and the robot not working. With no power, the robot is asleep with its motors and sensors not functioning. The ambient light is currently set to 80%.

Figure 6.5: Running the robot simulation (1 of 3)
CHAPTER 6. COMPARISON TO OTHER LANGUAGES

Frame 2  The user presses the Run button on the RCX to start the robot working. The program loaded in the RCX starts executing. The block of code written to execute when the robot is first switched on checks the value of the LightSensor. The program uses this value to decide whether to start the robot moving and whether to switch the SideLight on. Since the reading of the LightSensor is currently 80%, the program classifies this as day. Accordingly, it starts the robot moving forward but does not switch the SideLight on.

Frame 3  The user presses the Bump button to simulate bumping the robot. The value of TouchSensor changes from Released to Pressed, indicating that this sensor has detected the bump. This change of state is defined as a BumpEvent. The Bump Watcher, monitoring for BumpEvents, is triggered. The Bump Watcher starts the avoidance procedure, as indicated by the Procedure control, and executes its first command. This is the macro MoveBackward which in turn contains commands to make the left and right motors run backward. The robot starts backing away from the perceived obstacle.

Figure 6.5: Running the robot simulation (2 of 3)
6.2. MINDSCRIPT—LEGO’S ROBOTIC CONTROL LANGUAGE

Frame 4  After six seconds, the next command in the BumpWatcher executes. This is the TurnLeft macro that makes the motors run in opposing directions. The robot starts turning left on the spot.

Frame 5  Before BumpWatcher can complete the left turn and progress to the execution of its third and final command—the MoveForward macro to make the robot move forward again—the user simulates the coming night by reducing the Sunshine to 45%. This causes the program to detect a DuskEvent at the point that the LightSensor reading drops from 51% to 50%—the threshold between day and dusk conditions defined in the program. Since light-related events have been programmed to have higher priority than touch-related events, triggering of the Dusk Watcher that has been monitoring DuskEvents overrides the still executing Bump Watcher. The Bump Watcher aborts. The commands in the Dusk Watcher execute, consisting of the MoveForward macro and the locally defined command to switch the SideLight on. The robot immediately stops turning left and starts moving forward with the SideLight now on.

Figure 6.5: Running the robot simulation (3 of 3)
6.2.7 Conclusion for the MindScript example

Meshoil was able to emulate all the main structural components of a MindScript program using a simpler, more generic syntax.

By simulating the operational environment for the robot as well as providing the code equivalent to the MindScript program for robotic control, this example demonstrated Meshoil’s greater strength as a general-purpose programming language.

Meshoil’s ability to support embedded device control, particularly in the field of robotics, has parallels with other programming languages such as Esterel [14]. Esterel is an imperative, synchronous programming language with a strong bias towards programming control-dominated software and hardware reactive systems. It first appeared in the early 1980’s and is still very much under development.
6.3 Summary of comparisons

This chapter demonstrated the ability of Meshoil to emulate the programming constructs of other languages. Programs written in two application-specific languages, one designed for user interfaces the other for robotic control, were successfully transposed into Meshoil and run in the concept demonstrator.

While some indication of the advantages of programming in Meshoil was suggested by Meshoil’s ability to achieve functional equivalence through the use of more general programming constructs and the creation of structurally simpler programs, other important measurements of how easy or hard it is to program in Meshoil, such as the time involved to design and write a program, were not taken into account. These Meshoil programs had to be hand coded using an XML editor since no specific tools currently exist for programming in Meshoil, other than the Meshoil viewer for displaying programs in graphical format.

While suggestive of the advantages of Meshoil, the conclusions in this chapter would clearly benefit from a more comprehensive series of comparisons undertaken in the future involving a wider range of languages.
Chapter 7

Summary

7.1 Conclusions

This chapter summarises the author’s claims based on the principles of the Mesh discussed in earlier chapters and the results obtained from running Meshoil programs in the concept demonstrator.

7.1.1 Open-plan thinking is useful

If there is any single, overriding conclusion to be drawn from this research project, it is that a fundamental shift in the way of thinking about design can bring unexpected benefits—the tree that is OPT (section 1.8.4 on page 18) is worth cultivating for its prolific growth of exotic and luscious fruit. The evidence for this is apparent in the breadth and diversity of the individual solutions expressed in the all-embracing concept of the Mesh that serves to tie all this research work together.

![Figure 7.1: Systematic application of open-plan thinking](image)

OPT, taking its inspiration from the DNA model, shows that the key to everything is design simplicity. Flexibility and unlimited descriptive power can only come about from a design approach
that is inherently very simple. This point is missed in conventional engineering design where many problems result from simplicity and flexibility being mistakenly viewed as opposing ideals.

### 7.1.2 Mesh principles are largely unique

The principles on which the Mesh are based range from the well-established, tried and trusted principles on which many engineering solutions are based, to those that are unique to the Mesh. An example of the former is engineering integration by layering (section 3.9.5 on page 74), a principle well understood and commonly practised. The only thing unique about its inclusion in the Mesh is the particular way that it has been achieved—the result of other aspects of Mesh design, such as Meshoil’s abstract hardware and API layers, and brick insertion mechanism.

Principles such as universal usability are also generally well understood, but they are problems that have yet to be fully resolved. Their successful implementation is still some way off, and they remain the focus of considerable research effort. Less mainstream still, are principles such the abstract API layer (section 3.9.4 on page 74), recognised as desirable but seemingly given little research attention because they appear even harder nuts to crack.

But the bulk of Mesh principles are, to the best of author’s knowledge, original. Table 7.1 shows how some of these principles are a direct expression of the Meshoil language while others are more to do with how the Mesh might be used.

<table>
<thead>
<tr>
<th>Language-derived Mesh principles</th>
<th>Usage-derived Mesh principles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract hardware layer</td>
<td>Futures programming</td>
</tr>
<tr>
<td>Atomic-level networking</td>
<td>Globally-unified hardware control</td>
</tr>
<tr>
<td>Ghost-in-the-machine API</td>
<td>Hardware reuse</td>
</tr>
<tr>
<td>Ghost processing</td>
<td>Responsibility optimisation</td>
</tr>
<tr>
<td>Point-accurate interoperation</td>
<td>Software-team scalability</td>
</tr>
<tr>
<td>Program morphing</td>
<td>User-generated software</td>
</tr>
<tr>
<td>Superprogramming</td>
<td>Virtual hardware</td>
</tr>
<tr>
<td>Universal agent</td>
<td></td>
</tr>
<tr>
<td>Virtual logic</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Derivation of unique Mesh principles
7.1. CONCLUSIONS

7.1.3 The Mesh addresses issues faced by current research areas

The Mesh encompasses a number of research areas working on the control issue at the big picture level. These include universal usability, ubiquitous computing, interface agents and grid computing.

The examples discussed in this thesis have shown how the Mesh would address various design issues that these research areas currently face. Table 7.2 lists these research areas, the issues addressed, and the various Mesh principles responsible, sorted according to the research area where they have greatest impact.

<table>
<thead>
<tr>
<th>Research area</th>
<th>Issue addressed</th>
<th>Responsible Mesh principles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal usability</td>
<td>Make user interfaces uniquely customisable to suit individual needs and run on any platform (section 3.3.2 on page 41).</td>
<td>Building-brick design flexibility</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code filtering</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code integrity through accreditation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cost-free user flexibility</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cut and paste programming</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Futures programming</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware reuse</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Responsibility optimisation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transparent programming environment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>User-generated software</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Virtual hardware</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Virtual logic</td>
</tr>
<tr>
<td>Ubiquitous computing</td>
<td>Make devices of all description easy to interoperate over a network (section 3.8.4 on page 67).</td>
<td>Abstract API layer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Abstract hardware layer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Engineering integration by layering</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ghost-in-the-machine API</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Globally-unified hardware control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Point-accurate interoperation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Software-team scalability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Superprogramming</td>
</tr>
<tr>
<td>Interface agents</td>
<td>Use a universal agent that not only simplifies the task of designing and and building agents for many different types of device, but also minimises the scrutability problem (section 3.13.2 on page 89).</td>
<td>Program morphing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Universal agent</td>
</tr>
<tr>
<td>Grid computing</td>
<td>Break down a program into abstract, modular threads able to exchange information at any time while running concurrently on a network of processors of any device type (section 3.20.3 on page 109).</td>
<td>Atomic-level networking</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ghost processing</td>
</tr>
</tbody>
</table>

Table 7.2: Addressing design issues faced by current research areas
CHAPTER 7. SUMMARY

7.1.4 The Mesh has wide application

In being underpinned by a general purpose, modular and intrinsically networkable programming language, the Mesh is potentially able to address a wide range of engineering problems. This is demonstrated by the way that not every Mesh principle would need to be implemented before the Mesh could be put to practical use. For example, Table 7.3 shows how even if the Internet was cut out of the equation and none of the Internet-related Mesh principles were implemented, the Mesh would still be able to address the problem of universal usability (section 3.3.2 on page 41).

Similarly, even if creation of the abstract hardware layer (section 3.1.4 on page 37)—and support by the manufacturing industry that this implies—was discounted, universal interoperation between programs embodied in the principle of superprogramming (section 3.14.3 on page 93) could work quite happily in such areas as commercial programming where there is no need to control anything physically.

<table>
<thead>
<tr>
<th>Internet-free Mesh principles</th>
<th>Internet-related Mesh principles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract API layer</td>
<td>Atomic-level networking</td>
</tr>
<tr>
<td>Abstract hardware layer</td>
<td>Ghost-in-the-machine API</td>
</tr>
<tr>
<td>Building-brick design flexibility</td>
<td>Ghost processing</td>
</tr>
<tr>
<td>Code filtering</td>
<td>Grid computing</td>
</tr>
<tr>
<td>Code integrity through accreditation</td>
<td>Interface agents</td>
</tr>
<tr>
<td>Cost-free user flexibility</td>
<td>Point-accurate interoperation</td>
</tr>
<tr>
<td>Cut and paste programming</td>
<td>Program morphing</td>
</tr>
<tr>
<td>Engineering integration by layering</td>
<td>Responsibility optimisation</td>
</tr>
<tr>
<td>Futures programming</td>
<td>Software-team scalability</td>
</tr>
<tr>
<td>Globally-unified hardware control</td>
<td>Superprogramming</td>
</tr>
<tr>
<td>Hardware reuse</td>
<td>Ubiquitous computing</td>
</tr>
<tr>
<td>Transparent programming environment</td>
<td>Universal agent</td>
</tr>
<tr>
<td>Universal usability</td>
<td>Virtual hardware</td>
</tr>
<tr>
<td>User-generated software</td>
<td></td>
</tr>
<tr>
<td>Virtual logic</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3: Internet dependency of Mesh principles

7.1.5 The Mesh is feasible and practical

Two factors make the Mesh a viable proposition: design approach and timing.

Trying to build the Mesh around a conventional language would fail, because the complexities of the attempted solutions would be self-defeating. The Mesh only works because its design is so simple. Many principles of the Mesh wouldn’t exist but for their extreme simplicity in design.

Secondly, the timing for Mesh implementation is right. It is only now that we have reached the point where the IT industry has the physical capacity to support a working Mesh. Should any issues arise in the ability of the Internet to handle the traffic generated by the Mesh, they can only lessen in time as Internet infrastructure continues to improve. But equally importantly, the timing is right socially in the way that our consumer culture is changing to one of niche markets and extreme personalisation, as described in the recent phenomenon of the Long Tail (section 1.2.3 on page 4).
7.1.6 The Mesh could be implemented incrementally

As mentioned above, it is not necessary to implement every aspect of the Mesh before it can be put to practical use. Since different parts of the Mesh can be used to address different types of problem, there is nothing stopping such problems being targeted one by one and implementing the Mesh in stages to provide a growing list of solutions to problems.

For example, the decision could be taken to postpone support for the abstract API layer and ghost-in-the-machine API (section 3.9.3 on page 73), and initially to provide Meshoil with no method calling functionality at all. All required method functionality would be provided in the form of in-built method controls, as is the case for the concept demonstrator in its current form.

7.1.7 The Mesh is greater than the sum of its parts

The Mesh is based on the unifying design philosophy of OPT, not on a collection of unrelated engineering techniques. There is a natural synergy between the various elements of the Mesh that make the whole greater than the sum of the parts. While progressive Mesh implementation might prove to be the most practical way to build the Mesh, new and novel solutions are likely to result when all the pieces of the Mesh are finally in place.

Applying superprogramming to user interface designs in order to link how users operate different devices is an example of this—the Mesh provides a way of doing something that was previously impractical. But the synergy of the Mesh might also mean easier ways to do existing things. For example, a Meshoil program can use the abstract API layer to remain independent of the particular platform that the Mesh engine runs on. But if an Internet connection is provided, universal platform independence can be won at a single stroke by switching to use of the ghost-in-the-machine API instead.

7.1.8 Meshoil is a unique language

Many Mesh principles are a direct expression of Meshoil, the language underpinning the Mesh. As a programming language, Meshoil (appendix A on page 277) is believed to be unique because it is:

- expressed in markup
- completely abstract
- general-purpose in application
- intrinsically networkable
- declarative/imperative.

7.1.9 Meshoil is object-oriented

Meshoil incorporates many of the hallmarks of an OO programming language including abstraction, objectification, code reuse, inheritance, encapsulation and polymorphism (appendix B.3 on page 329).
CHAPTER 7. SUMMARY

7.1.10 Meshoil is a general-purpose programming language

The point was made in Chapter 4 that, all things being equal, it is always better to use a simple tool in place of a complex one, so long as the tool can still do the job required of it. With an XML-based syntax consisting of just five elements, Meshoil would seem to qualify as a simple tool—at least syntactically—but would it prove to be sufficiently expressive for use as a general-purpose programming language?

The wide variety of working examples discussed in this thesis support the idea that Meshoil is capable of solving the problems usually ascribed to the use of general-purpose programming languages. Meshoil programs have been used to:

- Create high-level user interfaces for a range of electronic devices (section 5 on page 127)
- Simulate the working environments and running of such user interfaces (section 6.2 on page 254)
- Model dynamic, mechanical processes such as automobile engine performance (section 5.3 on page 171)
- Provide low-level method libraries (section 5.6 on page 202)
- Emulate programming constructs of other, more specialised languages such as those used in user interface design and robotic control applications (section 6 on page 237)
- Handle synchronisation issues involved in the communication between concurrently running threads (section 5.6 on page 202).

The expressiveness of Meshoil was demonstrated most clearly by the mobile phone/emergency pager user interface (section 5.7 on page 217). Even though the Mesh engine in its current concept demonstrator form is yet to support a number of fundamental Meshoil features described in this thesis, such as FOR/WHILE loop iteration and method calling, and even though current in-built functionality consists of nothing more than basic arithmetic and string-based operations, it was possible to emulate the working of a Nokia 6210 mobile phone right down to the level of AT command handling and calling protocols.

7.1.11 Meshoil is an empowering language for the user

The trend in programming language design is to simplify the housekeeping tasks that the programmer has to undertake, so that more effort can be directed towards useful problem solving. For example, proponents of Java claim that one of its big advantages over other OO languages is its ability to handle issues like garbage collection transparently.

Meshoil extends this trend. Take the example of how the abstract API layer (section 3.9.4 on page 74) supports abstract method calling, a feature that allows the Meshoil programmer to make use of an API without having to know much about it. The Mesh principle of a transparent programming environment (section 3.4.3 on page 48) automates such necessary programming tasks as object instantiation, variable creation, datatyping and concurrency definition.

While no work has been done to test the idea, these features of Meshoil might serve to lower the barriers for successful programming, making Meshoil an empowering language that encourages its users to get involved in the programming process and to create user-generated software (section 3.7.1 on page 60) that best serves their individual needs.
7.1.12 **Meshoil contains features not found in other languages**

While some constructs in Meshoil provide equivalent programming functionality of a conventional language in a simpler form, Meshoil also has a number of constructs that are, to the best of the author’s knowledge, either more powerful than those found in a conventional language or unique. These constructs allow programs to be written in Meshoil that could not easily be written in another language.

These extended language features derive from having followed OPT’s principle of encoding information in relationships and having developed a flexible syntax capable of expressing many different relationships between elements of a program. Examples include:

- structuring programs in unique ways for achieving programming solutions, typified by the principle of software-team scalability (section 3.10.4 on page 77)
- setting up dynamic associations for programming variables (appendix A.4.3 on page 303)
- calling abstract methods (appendix A.5.2 on page 308)
- using more powerful and generic logical operators (appendix B.2.1 on page 325)
- using confluent conditional expressions (appendix B.2.2 on page 326).

7.1.13 **Meshoil can be used as a universal usability language**

Although Meshoil has the advantage of simplicity in being XML-based like all UIDLs, being a general-purpose programming language gives it a clear advantage over these UIDLs as a language for universal usability.

The best indicator of this was seen in the working example of Meshoil going head-to-head with URC in the area where the two languages currently overlap (section 6.1 on page 238). URC is a recognised leader in the field, but Meshoil was able to match URC on its own terms in a solution that was only half as complex.
CHAPTER 7. SUMMARY

7.2 Future work

Clearly, with the immensity of the Mesh concept there are unlimited areas for future work.

7.2.1 Immediate focus

There is work to be done in developing the Meshoil features described in this thesis that are currently unsupported in the concept demonstrator.

Of immediate practical benefit would be the creation of a graphical tool for writing Meshoil that would save the Meshoil programmer from having to hand code at the XML editor level. It would then be possible to perform comparative studies to determine how easy or difficult it is to program in Meshoil.

The impact that the Mesh would have on the physical resources of the Internet needs to be investigated.

7.2.2 The longer term

The primary focus of this research has been to demonstrate the technical feasibility of the Mesh. Clearly, there are many implications of the Mesh that need to be explored to achieve a better understanding of its potential benefits and likely impact.

At the macro level, for example, how would the Mesh change the process of software design and affect current user, programming and manufacturing environments? What are the main business and financial issues? What would it cost to make a device Meshable? How would Mesh implementation actually proceed? And what about all the implied social issues such as how would users react to a world that they suddenly had more control over?

But there are many issues of a technical nature to explore further too. Ubiquitous computing, interface agents and grid computing have only received brief mention in this thesis. Use of the Mesh to support these mainstream research areas needs further analysis. Only passing mention has been made of work done in related software engineering fields such as those involved in embedded device control, concurrency management, aspect-oriented programming and dynamically extensible software. To what extent would the Mesh overlap or compliment ongoing work in these areas? Section 3.11.6 on page 84 briefly mentioned the idea of self-generating software and how this would be relatively easy to achieve, given the nature of the Meshoil language. Could work in this area lead to promising new applications?

Given the wide ranging scope of Mesh capabilities, none of the work covered in this thesis can be considered any where near complete. Even in universal usability, the research area that received the greatest attention, many interesting questions remain unanswered. For example, superprogramming is only one example of how the Mesh might be able to push the existing boundaries in universal usability. Other Mesh principles might expand them even further. Could program morphing, for example, prove to be a useful mechanism for customising a user interface for a particular user in realtime?