Appendix A

Meshoils language description

This appendix describes the syntax and main logical constructs of the programming language called Meshoils. While intended to be as complete as possible, this description is not suggested to be the final blueprint for Mesh implementation.

Meshoils syntax is defined in a single XML schema called the Meshoils tester. The tester describes Meshoils's XML elements and their allowed physical relationships (see appendix C.2.1 on page 339).

This chapter covers:

• Syntax (page 278)—basic syntactical features of the language
• with bricks (page 290)—the data part of a Meshoils program
• do bricks (page 293)—the executable part of a Meshoils program
• Programming in Meshoils (page 302)—datatyping, concurrency, scope and local method calling
• Abstract API layer (page 307)—making use of the API provided by the platform where the Mesh engine runs
• Using namespacing (page 315)—the use of namespacing in program design
• Examples of more complex expressions (page 319)—examples of logical, numerical and string-based expressions.
A.1 Syntax

This section covers basic syntactical features of the language.

A.1.1 Meshoil’s five XML elements

Meshoil is a markup language written in XML. It consists of the following five XML elements:

- `<Meshoil>` as the root element
- `<a>` for annotation
- `<b>` for brick
- `<c>` for control
- `<d>` for data.

The root element is the single, enclosing element required by every XML file to indicate the nature of its contents. This leaves just four types of element with names like children’s ABC wooden blocks for writing code:

```xml
<1
  <b> for brick
  <a>
    <a>
      for annotation
    </a>
  </a>
  <c> for control
</b>
  <d> for data</d>
</1>
```

Figure A.1: Meshoil’s four XML elements for writing code

Code is expressed by the contents of these four elements and their relationship to one another. The XML source code for Figure A.1 is shown below:

```xml
<b>&lt;b&gt; for brick
 <a>
   for annotation</a>
 </a>
</b>
```

In XML ‘<’ and ‘>’ have the special meaning of tag delimiters. To enter them literally as character strings they have to be expressed by the special XML codes of ‘&lt;’ and ‘&gt;’. Thus ‘&lt;a&gt;’ displays as ‘<a>’.

```xml
&lt;a&gt;&lt;a&gt; for annotation&lt;/a&gt;
&lt;/a&gt;
&lt;/a&gt;
&lt;c&gt;&lt;c&gt; for control&lt;/c&gt;
&lt;d&gt;&lt;d&gt; for data&lt;/d&gt;
</b>
```
A.1.2  `<a>` for annotation

Annotation is any free text such as a comment. `<a>` elements can be nested within other `<a>` elements to structure annotations:

```
1st level annotation.
  2nd level annotation.
   3rd level annotation.
```

Figure A.2: `<a>` for annotation

The XML source code for Figure A.2 is shown below:

```
A.1.3  `<c>` for control

A control is a programming variable. `<c>` elements can be used as abstract representations of software or hardware features external to the program, as demonstrated here by the 2abc key on a mobile phone and Power for controlling a DC power circuit for a device:

Figure A.3: `<c>` for control

The XML source code for Figure A.3 is shown below:

```
<2abc key>Pow<er></2abc key>
```
APPENDIX A. MESHOIL LANGUAGE DESCRIPTION

A.1.4  <d> for data

Data is the setting of a control.

These might be the settings of the four controls in the previous example:

![Figure A.4: <d> for data](image)

The XML source code for Figure A.4 is shown below:

```xml
<d>Down</d>
<d>On</d>
<d>The fan is on</d>
<d>149</d>
```

A.1.5  <b> for brick

The brick is the atomic coding unit of a program. Each brick is built up from nested <a>, <b>, <c> and <d> elements.

This brick consists of two elements: one <c> and one <d>. It switches a device on by changing its Power setting to On:

![Figure A.5: <b> for brick](image)

The XML source code for Figure A.5 is shown below:

```xml
<b>
  <c>Power</c>
  <d>On</d>
</b>
```
A.1.6 Brick nesting

Meshoil is a modular programming language. Code is built up by using the XML elements rather like Lego bricks, with the brick element taking on the significance as the smallest identifiable piece of code. A program is just a collection of bricks arranged next to each other and nested within each other. Meaning is given both to what each brick represents and the relationship it has with other bricks.

Here are six bricks. The two, top-level bricks each contain two nested bricks of their own:

![Figure A.6: Brick nesting](image)

The XML source code for Figure A.6 is shown below:

```xml
<bx
    bx
        bx&gt;Brick 1&lt;/bx
    bx
</bx
<cx>&lt;cx
    bx
        bx&gt;Brick 1.1&lt;/bx
    bx
</bx
<dx>&lt;dx
    bx
        bx&gt;Brick 1.2&lt;/bx
    bx
</bx
<bx
    bx
        bx&gt;Brick 2&lt;/bx
    bx
</bx
<cx>&lt;cx
    bx
        bx&gt;Brick 2.1&lt;/bx
    bx
</bx
<dx>&lt;dx
    bx
        bx&gt;Brick 2.2&lt;/bx
    bx
</bx
```

281
A.1.7 Brick structure

A brick consists of two parts: the body of the brick followed by any child bricks it may have. The body consists of <a>, <c> and <d> elements. Child bricks follow these elements, logically nesting immediately beneath the <b> element of their parent.

The <a>, <c> and <d> elements of the body may contain nested <a>, <b>, <c> and <d> elements of their own. All such nested <b> elements are called body bricks, since at some point in their ancestry they are descendants of an <a>, <c> or <d> element belonging to the body of a brick.

Bricks that do not nest in the body of other bricks at any point in their ancestry are called main bricks. All their ancestors are <b> elements. Every brick in Meshoil is thus either a body brick or a main brick.
The example here consists of a total of four bricks: two main bricks and two body bricks. The parent main brick contains three bricks—two body bricks and a nested main brick in a parent-child relationship:

![Figure A.7: Brick structure](image)

The XML source code for Figure A.7 is shown below:

```xml
<><
  
  &lt;a&gt; Main brick: parent&lt;/a&gt;
  
&lt;/a&gt;
</><

<><
  
  Body brick: belongs to body of parent
  
&lt;/a&gt;
</><

<><
  
  Body brick: belongs to body of parent
  
&lt;/a&gt;
</><

<><
  
  Main brick: child of parent
  
&lt;/a&gt;
</><
```
A.1.8 Bricks with multiple \(<c>\) and \(<d>\) elements

When bricks have more than one \(<c>\) or one \(<d>\) element, every combination of a \(<c>\) with a \(<d>\) element is implied.

This example has two \(<c>\) elements and one \(<d>\) element, resulting in the Power and LED controls both being set to On:

![Figure A.8: Bricks with multiple \(<c>\) and \(<d>\) elements](image)

The XML source code for Figure A.8 is shown below:

```xml
<bp>
  <a>&lt;a&gt;Main brick: child of parent&lt;/a&gt;
  
  &lt;c&gt;</a>

  &lt;d&gt;On&lt;/d&gt;
</b>
</bp>

A.1.9 Brick names

Bricks can be assigned one or more names for identification.

Each name is specified in a \(<d>\) element that nests one level beneath a first level \(<a>\) element for the brick. Here, the brick is called **Start**:

![Figure A.9: Brick names](image)
A.1. SYNTAX

The XML source code for Figure A.9 is shown below:

```
<b>
  <a>
    <d>Start</d>
  </a>
  <c>Power</c>
  <d>On</d>
</b>
```

A.1.10 Brick reuse

Bricks can be reused elsewhere in a Meshoil program by referencing their name within another `<b>` element. At compilation or runtime, this results in inserting the contents of the referenced brick just as if it had been explicitly defined at that location.

This brick references and inserts the `Start` brick (inserted bricks display faint in the Meshoil viewer):

![Figure A.10: Brick reuse](image)

The XML source code for Figure A.10 is shown below:

```
<b>Start</b>
```

A.1.11 Brick namespacing

A namespace can be declared for a brick.

The namespace is specified in the namespace attribute of the `<b>` element. Namespacing applies to brick and control names, with the names prefixed by the namespace and a `.` delimiter. Here, the namespace is specified as `fan` so the name of the brick is now `fan.Start` and the control is now `fan.Power`:

![Figure A.11: Brick namespacing](image)
The XML source code for Figure A.11 is shown below:

```xml
<b namespace="fan">
  <a>
    <d>Start</d>
  </a>
  <c>Power</c>
  <d>On</d>
</b>
```

### A.1.12 Spin

A brick or control can be assigned a spin to indicate the type of brick or control it is, and how it is to be processed.

Spin is specified in the spin attribute of the `<b>` or `<c>` element. Here are three bricks. The first brick has `with` spin, the second has `do` spin, and the third has `inactive` spin:

![Figure A.12: Spin](image)

The first brick is an example of a `with` brick. `with` bricks are used to set up how controls are defined in the program. This brick sets up a control called **Counter** for use as a programming variable that takes defaults for all of its characteristics such as datatype and initial setting.

The second brick is a `do` brick. `do` bricks are used to change the settings of controls. This brick changes the **Power** control to its **On** setting.

`with` and `do` bricks are equivalent to the data and executable part of a conventional language. They provide the means to say: ‘*With* these hardware and software features, *do* these things’.

If the spin attribute setting in the `<b>` or `<c>` element is not recognised as a valid spin, that element and any nested elements it may contain are ignored. The third brick here has `inactive` spin. Since this is not a valid spin, this has the effect of commenting out the code that the brick represents.

The XML source code for Figure A.12 is shown below:

```xml
<b spin="with">
  <a>
    <a>...these controls...</a>
  </a>
</b>
```

286
A.1. SYNTAX

A.1.13 Spin controls and method controls

Spin controls and method controls are special controls that are equivalent to reserved words in a conventional language. Spin controls change how the brick is processed just like the spin attribute for the `<b>` element, but they allow finer control. Method controls are in-built methods or functions that perform basic operations such as addition and subtraction.

Different sets of spin and method controls apply to bricks of different spin. For example, with bricks can use the `integer`, `decimal` and `string` spin controls to specify whether the datatype of a control is to be defined as integer, decimal or string-type. do bricks can use the `if` spin control to change an assignment operation to a conditional one.

The first brick here sets up the `Counter` control as integer-type, since the brick includes the `integer` spin control. The addition of the `if` control in the second brick changes the brick from switching the `Power On` to checking whether the `Power` is `On`:

![Figure A.13: Spin controls](image)

The XML source code for Figure A.13 is shown below:

```xml
<b spin="do">
  
  <a>
    <a>...do these things...</a>
  </a>
  
</b>

<b spin="inactive">
  
  <a>
    <a>...never executes...</a>
  </a>
  
</b>

<b spin="with">
  
  <b>
    <c>integer</c>
    <c>Counter</c>
  </b>

  
</b>
```
APPENDIX A. MESHOIL LANGUAGE DESCRIPTION

A.1.14 Program structure

Program structure consists of with and do bricks nested within a single, parental brick that represents the whole program. Programs may have any number of with and do bricks.

with bricks are specified in the parent brick’s <c> elements. do bricks are specified in the parent brick’s <d> elements. This has the effect of associating every with brick with every do brick according to the rule for multiple <c> and <d> elements in a brick (appendix A.1.8 on page 284). In this way, Meshoil syntax scales naturally—as well as their role in associating single controls with single values, the <c> and <d> elements are used at the macro level of associating a group of controls (the with brick) with a group of dynamically changing values (the do brick):

![Figure A.14: Program structure](image)

While Meshoil syntax—as defined in the Meshoil tester—never changes, the meanings given to the relationships between the XML elements inside the with and do bricks differ.

Both with and do spin can be qualified by a Meshpointer suffix in parentheses, for example, with(mesh://fan) and do(mesh://fan). When a with brick is Meshpointer qualified, it is downloaded to the specified device to define control information there instead of in the local device. Similarly, when a do brick is Meshpointer qualified, the code it contains is downloaded to the specified device and executed there. with bricks are downloaded before do bricks.

The XML source code for Figure A.14 is shown below:

```xml
<spin="with">
  <with>
    ...consists of these controls...
  </with>
  <do>
    ...and does these things...
  </do>
</spin>
```
A.1. SYNTAX

...consists of these controls...

...and does these things...

...and does these things...
APPENDIX A. MESHOIL LANGUAGE DESCRIPTION

A.2 with bricks

This section covers the data part of a Meshoil program.

A.2.1 Spin controls

with bricks use three types of spin control for determining how a control is set up (Table A.1).

<table>
<thead>
<tr>
<th>Type of spin</th>
<th>Spin descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Direction</td>
<td>Specifies the direction of the control’s data flow between the Mesh engine and the external world:</td>
</tr>
<tr>
<td></td>
<td>(default) Directionless, no connection to external world.</td>
</tr>
<tr>
<td></td>
<td>in Incoming data flow.</td>
</tr>
<tr>
<td></td>
<td>out Outgoing data flow.</td>
</tr>
<tr>
<td></td>
<td>inout Bi-directional data flow.</td>
</tr>
<tr>
<td>2 Datatype</td>
<td>Specifies the control’s datatype:</td>
</tr>
<tr>
<td></td>
<td>integer (default) Integer-type.</td>
</tr>
<tr>
<td></td>
<td>decimal Decimal-type.</td>
</tr>
<tr>
<td></td>
<td>string String-type.</td>
</tr>
<tr>
<td>3 Values</td>
<td>Specifies various values for the control:</td>
</tr>
<tr>
<td></td>
<td>(default) Range unlimited, control can take any value.</td>
</tr>
<tr>
<td></td>
<td>min Minimum value for integer/decimal-type control.</td>
</tr>
<tr>
<td></td>
<td>max Maximum value for integer/decimal-type control.</td>
</tr>
<tr>
<td></td>
<td>step Step size of integer/decimal-type control.</td>
</tr>
<tr>
<td></td>
<td>list Enumerated list of values for string-type control.</td>
</tr>
<tr>
<td></td>
<td>initial Initial value at start of program execution.</td>
</tr>
</tbody>
</table>

Table A.1: Spin controls in with bricks

Values for range spin are used according to the control’s datatype. If integer-type or decimal-type, any specified min and max values are used. If string-type, any specified list values are used. Values specified for min and max are inclusive.

step specifies the positive integer increment for an integer-type control, or the positive decimal increment for a decimal-type control. The default is 1 for integer-type, and either the machine precision value ($\epsilon$) or the limit set by the Mesh engine application for decimal-type. Zero is made a valid step interval, or the min value if zero lies outside the allowed value range. If a value is assigned to a control and it isn’t a valid step interval, the value is automatically rounded to the nearest step interval.

If initial has not been used to set up the control’s initial value at runtime, a default is used. For an integer-type or decimal-type control the default is zero, or the min value if zero lies outside the allowed value range. For a string-type control the default is the first value specified by list, or a blank if no list has been specified.

1This feature is not yet implemented in the concept demonstrator.
This example sets up a control called `Counter`. Default settings are used since no spin controls are specified, i.e., `Counter` becomes a directionless integer of unspecified range:

![Counter]

Figure A.15: Spin controls in with bricks

### A.2.2 Connection with the outside world

Direction spin specifies the direction of data flow between the Meshoil program and the particular feature of the device or external system that the control represents. *in* indicates incoming information, *out* outgoing, and *inout* bi-directional flow. For hardware controls, which setting applies depends on how the device has been manufactured and what physical mechanisms are provided for communication with the device hardware.

Controls without any direction spin are equivalent to programming variables in a conventional language. They have no direct link with any physical feature in the external world.

Here are the definitions for three controls:

![Display, Key, Motor]

Figure A.16: Direction spin reflects hardware capability

The first control is called `Display`. It represents the display screen on a device. The *out* direction spin indicates that data flows one-way from the program to the device, reflecting the fact that the device hardware provides the physical means for uploading a message to the display, but provides no facility for interrogating the display to see what the current message is. The program is able to output messages to the display and can keep track of what is currently displaying by keeping a record of the last message output, but it has no way of verifying what is actually displaying should some external factor such as a power cut affect the device. The *string* spin specifies that the `Display` control handles string-type data.

The second control is called `Key`. It represents a physical push-button key on a device. Like `Display` it is string-type, but in *spin* indicates that data flows in the opposite direction from the device to the program. The hardware provides information to the program on changes to the state of the key, thus allowing the program to check for such changes and initiate processes accordingly. The program has no power to control the key by changing its up or down state since no such physical means are provided. The *list* spin restricts the list of acceptable string values to *Up* and *Down*.

The third control is called `Motor`. It represents the nominal speed of a motor in the continuous range of 0 (motor off) to 1.0 (motor on at maximum speed). The *inout* direction spin reflects the fact that there are hardware mechanisms for both setting the speed of the motor and for checking
its current value. This additional hardware capability allows the program not only to control the motor but also to monitor how it is actually running. The program can detect and respond to changes in speed caused by other external means of control.

### A.2.3 Flexible format in with bricks

Within the body of a main with brick, spin applies globally making the ordering and placing of spin controls unimportant.

These three main bricks all produce the same result of defining Motor as a decimal-type control that takes values between 0 and 1:

![Flexible format in with bricks](image)

**Figure A.17: Flexible format in with bricks**

### A.2.4 Defining identical controls

When more than one control has the same definition, the controls can be defined in a single brick.

This example defines the numeric keypad on a phone:

![Defining identical controls](image)

**Figure A.18: Defining identical controls**
A.3 do bricks

This section covers the executable part of a Meshoil program.

A.3.1 Assigning control values

These bricks perform three assignments:

The first brick sets the Power control to On.

The second brick shows how a control can be assigned the value of another control by nesting the other control under a $<d>$ element. This brick sets the value of Status to the current value of Power. Runtime casting between different datatypes is automatic. If the cast is impossible, such as assigning a string-type to a decimal-type control when the string-type control contains non-numeric characters, the assignment is not executed.

The third brick shows how specifying a blank $<d>$ element with a control causes the control to be reset to its default value. This brick resets the Power, Status and Counter controls to their respective defaults. This might be Off for Power, blank text for Status and 0 for Counter.

A.3.2 The order of processing bricks

A tree pathway is followed for processing bricks at runtime.

Here, Brick 1 and its two child Bricks 1.1 and 1.2 are processed before Brick 2 and its two children. The order is 1, 1.1, 1.2, 2, 2.1 and 2.2:
A.3.3 Spin controls

`do` bricks use five types of spin control for determining how a brick evaluates and changes the values of controls (Table A.2).

<table>
<thead>
<tr>
<th>Type of spin</th>
<th>Spin descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Branching</td>
<td>Sets type of expression:</td>
</tr>
<tr>
<td></td>
<td>(default) Assignment expression.</td>
</tr>
<tr>
<td></td>
<td>if Conditional expression.</td>
</tr>
<tr>
<td>2 Inversion</td>
<td>Controls inversion of boolean evaluation:</td>
</tr>
<tr>
<td></td>
<td>(default) No inversion.</td>
</tr>
<tr>
<td></td>
<td>not Inversion.</td>
</tr>
<tr>
<td>3 Group</td>
<td>Specifies how groups of conditions are evaluated:</td>
</tr>
<tr>
<td></td>
<td>all (default) True if all conditions are true.</td>
</tr>
<tr>
<td></td>
<td>any True if any one or more condition is true.</td>
</tr>
<tr>
<td></td>
<td>any(1) True if any one condition is true.</td>
</tr>
<tr>
<td>4 Timing</td>
<td>Sets the timing for evaluating a condition:</td>
</tr>
<tr>
<td></td>
<td>up (default) True if condition is true.</td>
</tr>
<tr>
<td></td>
<td>fall True at point when condition becomes false.</td>
</tr>
<tr>
<td></td>
<td>down True if condition is false.</td>
</tr>
<tr>
<td></td>
<td>rise True at point when condition becomes true.</td>
</tr>
<tr>
<td>5 Relational</td>
<td>Sets the inequality relationship for evaluating a condition:</td>
</tr>
<tr>
<td></td>
<td>eq (default) True if condition equals value.</td>
</tr>
<tr>
<td></td>
<td>lt True if condition less than value.</td>
</tr>
<tr>
<td></td>
<td>gt True if condition greater than value.</td>
</tr>
<tr>
<td></td>
<td>le True if condition less than or equal to value.</td>
</tr>
<tr>
<td></td>
<td>ge True if condition greater than or equal to value.</td>
</tr>
</tbody>
</table>

Table A.2: Spin controls in `do` bricks

A brick containing the `if` spin control marks the start of a conditional expression. When different types of spin control are specified within the same brick, they apply in the order shown in the list above regardless of where they appear in the brick. For example, if a brick contains both `any` (group spin) and `not` (inversion spin), the brick will evaluate as being true if no conditions are true. The implied meaning is `not any`, and not the alternative of `any not` that would cause the brick to evaluate as being true if at least one condition is false. (If `any not` is the desired condition, `not` would need to be specified in each brick that specifies one of the conditions making up the group.)

With the exception of `not`, spin is inherited within the body of a brick. Once a spin is defined by a spin control, it is inherited by all bricks nesting beneath the spin control until reset by another spin control of the same type. `not` isn’t inherited since this would unhelpfully cause boolean conditions to flip-flop at each level. `not` only ever has meaning when used to invert a single boolean state. Its effect is to invert the final boolean evaluation of the brick in which it has been specified.
A.3. DO BRICKS

This example uses the if spin control to define a Key press as the means for switching on the power for some device:

![Figure A.21: Spin controls in do bricks](image)

By default, a brick evaluates as true when executed. Adding if spin makes the evaluation of the brick conditional on its contents. At runtime, the state of the Key control is checked. If it is Down, execution passes to the child brick that switches the Power control to the On setting. If the Key is not Down, execution skips the child brick and continues with the next sibling of the parent brick.

The logic is equivalent to the following conventionally described expression:

\[
\text{IF (Key == Down)}
\begin{cases}
  \text{Power = On}
\end{cases}
\]

A.3.4 Method controls

Method controls are in-built methods for performing a standard range of numeric and string-based operations on the values of controls.

Table A.3 lists the current resident methods supported by the concept demonstrator. They consist of the four standard numeric operators +, -, * and /, plus the two string-based operators s+ and s- for performing generalised addition (concatenation) and subtraction (character removal) on strings.

<table>
<thead>
<tr>
<th>Type of method</th>
<th>Descriptions</th>
</tr>
</thead>
</table>
| 1 Numeric      | Performs a numeric operation:  
+ Add values.  
- Subtract values.  
* Multiply values.  
/ Divide values. |
| 2 String-based | Performs a string-based operation:  
s+ Add (concatenate) strings.  
s- Subtract (remove) characters from strings. |

Table A.3: Method controls in do bricks

Although many more method controls could be added to the concept demonstrator, this list has found to be sufficient to support all the working examples discussed in this thesis. The Mesh engine can be created as an application that supports any number of method controls—even none
APPENDIX A. MESHOIL LANGUAGE DESCRIPTION

at all—since when a method is needed that is not in-built, it is externally referenced (see appendix A.5 on page 307).

The four numeric operators are so basic that they are hard-coded into the syntax of every conventional language. The two string-based operators support functionality that is usually provided by a programming language’s API. For example, s- provides the functionality of the Java String-based methods replaceAll (replace all occurrences of one substring found in a given string with another substring), substring (return a substring of a given string by removing a specified number of leading and trailing characters), and split (search for a matching substring in a given string and split that string into two halves at that point).

Meshoil syntax allows each numeric or string-based operator to have any number of operands nesting beneath it. Operands are processed in the order they are specified, enabling the final result returned by the operator to be achieved in a number of predefined steps. For example, the result of an s- operator can be defined as the process of progressively and selectively removing characters from an initial string.

Figure A.22 shows the use of each of the six method controls in assigning a value to a control. The operands that a method control applies to are nested in <c> and <d> elements under the <c> element of the method control:

![Figure A.22: Using method controls](image)

The first four bricks in the example set the value of A respectively to B+10, B-10, B*10 and B/10 (the <d> elements enclosing each method control increase readability but are not compulsory since data assignment of A is already implied by use of the method control).

As well as performing their standard arithmetic functions, the four numeric method controls make special use of the case when they apply to single operands. The + method control adds all operands, but if there is only one returns its absolute value. The - method control subtracts the second and subsequent operands from the first, but if there is only one reverses its sign. The * method control multiplies all operands, but if there is only one returns its square root. The / method control divides the first operand by the second and subsequent operands, but if there is only one returns its inverse.

Any part of a calculation that cannot be performed is ignored, for example, division by zero or trying to cast a string-type control as a decimal when it currently contains non-numeric characters.

The last two bricks in this example demonstrate the basic way that the two string-based method controls work. The s+ method control concatenates all the string equivalents of the operands. The fifth brick uses s+ to set Status to be the result of concatenating the character string ‘The power is currently:’ with the current setting of the Power control. For example, if the setting of the Power control is On, Status will be the character string ‘The power is currently:On’.

The s- method control subtracts string equivalents of the second and subsequent operands from the first operand. The final string returned is the result of processing the operands in the order that they are specified. All instances of a character string are removed from the current string result.
A.3. DO BRICKS

The sixth brick in this example performs the reverse of the fifth brick. It sets the Power control by extracting information embedded in Status. The Power setting is the character string contained in Status minus any embedded occurrences of the character string ‘The power is currently:’. For example, if Status is the character string ‘The power is currently: Off’, the Power control will be set to its Off setting.

A.3.5 IF/ELSE IF/ELSE logic

In a conditional brick, child bricks execute by default when the parental condition is true. To make a child brick execute when the parental condition is false, the child brick is given not spin.

This example shows how IF/ELSE IF/ELSE IF.../ELSE chains of logic can be built up by nesting combinations of conditional expressions and child bricks with not spin (inclusion of ‘IF/ELSE IF/ELSE IF’ in the figure is purely for annotation purposes and is not part of the logic):

![Figure A.23: IF/ELSE IF/ELSE logic](image)

The logic is equivalent to the following conventionally described expression:

```c
IF (Motor == 0)
    {Indicate the fan is off}
ELSE IF (Motor == 0.5)
    {Indicate the fan is on at 1/2 speed}
ELSE IF (Motor == 0.75)
    {Indicate the fan is on at 3/4 speed}
ELSE
    {Indicate the fan is on but not at 1/2 or 3/4 speed}
```
A.3.6 Nested IF logic

Conventional nested IF logic using brackets is naturally accommodated by Meshoil’s modular structure. Here the logic is: IF (IF/ELSE) ELSE:

```
IF (Power == On)
  {Indicate the fan is on;
   IF (Pan == On)
     {Indicate the fan is panning}
   ELSE {Indicate the fan isn’t panning}
  }
ELSE {Indicate the fan is off}
```

The logic is equivalent to the following conventionally described expression:

A.3.7 Group spin controls

The concept demonstrator supports three group spin controls: all, any and any(1).

any and any(1) are examples of a generalised notation for an any-based group spin control in which the behaviour of the word ‘any’ is indicated by numbers in parentheses after the control name. These numbers, specified in colon indicated ranges or separated by commas, indicate how many conditions must be true for the group to evaluate as true. any and any(1) have been implemented in the concept demonstrator since they are the most useful variants.
Table A.4 gives examples of this notation.

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>The default specification without any parentheses. True if any one or more condition is true.</td>
</tr>
<tr>
<td>any(1)</td>
<td>True if any one condition is true.</td>
</tr>
<tr>
<td>any(2)</td>
<td>True if any two conditions are true.</td>
</tr>
<tr>
<td>any(2:3)</td>
<td>True if any two or three conditions are true.</td>
</tr>
<tr>
<td>any(1:4)</td>
<td>True if no condition is true, or any one condition is true.</td>
</tr>
<tr>
<td>any(1,3,5)</td>
<td>True if any one, three or five conditions are true.</td>
</tr>
<tr>
<td>any(2,3,6)</td>
<td>True if any two, three, six or more conditions are true.</td>
</tr>
<tr>
<td>any(n)</td>
<td>The literal ‘n’ can be specified to indicate whatever number of conditions are found to be in the group. This example is equivalent to all since it only returns true if every condition is true.</td>
</tr>
<tr>
<td>any(1:n-1)</td>
<td>Expressions are possible too. This example is true if any one, two, or up to the number of conditions in the group minus 1 are true. This is equivalent to what we understand by the word ‘some’; true is returned so long as at least one condition is false, ie, true as long as not all conditions are true.</td>
</tr>
</tbody>
</table>

Table A.4: Examples of the generalised notation for an any-based group spin control

These three bricks are identical except for having different group spin:

![Figure A.25: Group spin controls](image)

The first brick has the default all spin. When all bricks that are nested in the body of the parent brick evaluate to being true, its single child brick will execute. This means that all three keys need to be pressed in order to switch the **Power On**.

The second brick has any spin. One or more of the three keys need to be pressed to switch the **Power On**.

The third brick has any(1) spin. This means that any one of the keys must be pressed to switch the **Power On**. Since three key conditions are specified, any single key press will work, but not a combination of two keys or all three keys pressed together.
A.3.8 Timing spin controls

There are four timing spin controls: up, fall, down and rise. They allow the boolean evaluation of a brick to be made dependent on when control values change. Together, these four spin controls represent every way the value of a control can change with respect to a specified value.

up means that a control will evaluate as being true when it has the value specified in the brick. This is the default timing that is commonly understood for any type of conditional expression. fall means that a control will evaluate as being true only at the instantaneous moment that it no longer has the specified value. down is the opposite of up meaning that a control will evaluate as being true when it does not have the specified value. (down is the same as using the inversion spin control not in an up expression, and is provided for completeness of the method.) rise is the opposite of fall, meaning that a control will evaluate as being true only at the instantaneous moment that it reaches the specified value.

The first brick in Figure A.26 uses the default up. The Warning light will be On for as long as the Alarm sensor remains On. Note that this brick does not toggle the Warning light with the Alarm sensor, it only ensures that the Warning light will be On while the Alarm sensor is On. When the Alarm sensor resets to its Off position, the child brick that sets the Warning light On will simply no longer execute. From that point onwards, the Warning light will remain On until switched off by another brick or by some other means.

The second brick fires at the point when the Alarm sensor first detects a malfunction and comes On. Although the Alarm sensor may continue to remain On, this brick will not execute again. This allows the Siren to be switched Off by some other means before the Alarm sensor resets. The Alarm sensor must reset and come On for a second time for this brick to fire again.

Since the Alarm sensor only has the two states On and Off, the third and fourth bricks have an identical effect as the first two. They use the down and fall timing spin controls based on the condition of the Alarm sensor being Off.
A.3.9 Flexible format in do bricks

It is unimportant how spin controls nest within a brick. They can be freely arranged to maximise visual clarity of the logic.

These four bricks all describe the same logic of switching the Power On if A is greater than the sum of B, C and 149:

![Image of do bricks example](image)

Figure A.27: Flexible format in do bricks

The logic is equivalent to the following conventionally described expression:

\[
\text{IF } (A > (B + C + 149)) \{ \text{Power} = \text{On} \}\]
A.4 Programming in Meshoil

This section covers datatyping, concurrency, scope and local method calling.

A.4.1 Simplest program

The simplest program consists of a single do brick. The Mesh engine repeatedly parses the logic described in the do brick, executing bricks and changing values of the controls in the manner specified. In the Mesh engine, a do brick runs as a single executing thread.

Here is a Meshoil program that repeatedly assigns values to three controls: Counter is 149, Motor is 0.7, and Status is the character string ‘Everything is OK’:

![Figure A.28: Simplest program](image)

A.4.2 Automatic datatyping

Where possible in Meshoil, conventional programming tasks are automated to lighten the load on the programmer. The example here is the automatic declaration of variables and determination of their datatype. Casting between variables of different datatype is also automatic.

Controls do not need to be defined before they are used in a do brick since datatyping is automatically determined by the datatype of the values that the controls are assigned to.

The datatype of each control is determined at compilation time, or as the program is loading if code is interpreted. Each control is initially set up to be integer-type, the most restrictive of the three datatypes: integer, decimal and string. As necessary, the datatype of each control is progressively relaxed to decimal-type and string-type as all control assignments in the code are inspected.

In Figure A.29, Control 1 and Control 2 remain integer-type controls since the values they are assigned to are valid integers. Control 3 and Control 4 are relaxed to decimal-type since the values they are assigned to include decimal points. Control 5 and Control 6 are relaxed to string-type since neither of their values are valid integers or decimals.

![Figure A.29: Automatic datatyping](image)
A.4.3 Defining controls

When necessary, the characteristics of controls can be explicitly specified in with bricks rather than relying on such default behaviour as automatic datatyping.

In Figure A.30, the Counter control is explicitly set up as an integer that takes values between 0 and 100. The brick that sets Counter to 149 in the do brick will now generate an out-of-range compilation error. Automatic datatyping would have set up the Number control as a decimal but instead this control is explicitly set up as string-type. At runtime, Number will be assigned the character string ‘0.7’ instead of the decimal value.

![Figure A.30: Defining controls](image)

Setting the value of Counter to 149 in the do brick is an example of a single, runtime association: Counter is assigned the value of 149 when the code executes. Dynamic associations can be set up by declaring them in with bricks\(^2\), shown here in the example of the Ratio control. The association is permanently set up at compilation time so that the value of Ratio will constantly be updated to reflect the current ratio of Number divided by Counter at all times during runtime.

When the program interfaces with hardware, for example when it is the user interface for an electronic device, a with brick is used to define the complete list of controls that represent the raw, physical capabilities of the device. This list includes all hardware and firmware features of the device, forming an abstract hardware layer that describes every physical feature manufactured into the device.

The manufacturer is responsible for defining this hardware-related with brick and for supporting its physical connection to the device hardware. A runtime version of the with brick forms a link between the Mesh engine and device hardware, maintaining a snapshot of the current values of the various hardware controls. Values are constantly updated asynchronously, both by the Mesh engine and by the mechanisms manufactured into the device for recognising external hardware changes such as interrupt-driven, key press events. Detecting external events within the program and responding to them is polling-driven as the Mesh engine repeatedly parses the program logic.

\(^2\)This feature is not yet implemented in the concept demonstrator.
A.4.4 Concurrency and the scope of programming constructs

This program has six with bricks and four do bricks:

Each top-level do brick in a program (a do brick not embedded in another do brick) runs as a separate thread. Here, the code in Brick 1.1, Brick 1.2 and Brick 1.3 will run concurrently as separate threads. The more a program is broken up into separate do bricks in this way, the smaller and more specialised each do brick becomes. With less code to repeatedly parse, event management within each do brick becomes less polling-driven and more event-driven.

The scope of controls and program execution is defined by the way that with and do bricks are nested within the program. When a control is defined in a with brick, it remains in scope for all the do brick code defined at the same level. When a control is used in a do brick without being defined in any with brick, its scope is that single do brick.

Here, the scope of Control1 is Brick 1, ie, all executable code of the program. Defining a control
A.4. PROGRAMMING IN MESHOIL

at this location thus becomes the mechanism for declaring global programming resources that are shared between all threads—in this case the three threads represented by Brick 1.1, Brick 1.2 and Brick 1.3.

The scope of Control3 is the executable code of Brick 1.1.1 (ie, Brick 1.1.1.1). The scope of Control4 is the executable code of Brick 1.1.2 (ie, Brick 1.1.2.1 and Brick 1.1.2.2.1). The scope of Control5 is the executable code of Brick 1.1.2.2 (ie, Brick 1.1.2.2.1).

Control 2 is defined at two locations in the program. This creates two, completely independent instances of the control. A global instance is defined in Brick 1 and a local instance is defined in Brick 1.1.2. The scope of the global instance is the whole program, except for Brick 1.1.2 where the local instance is in scope.

The scope of Control6 is the two threads represented by Brick 1.2 and Brick 1.3. This control is defined too low in the program for it to be in scope for the first thread represented by Brick 1.1.

Nested do bricks (do bricks nested in other do bricks) define local execution loops similar to FOR/WHILE loops in a conventional language. Nested do bricks continue to execute exclusively while the condition defined at the top of the brick is true. When the condition is no longer true, looping ends and execution continues with the next sibling brick. Here, Brick 1.2.2.1 and Brick 1.2.2.2 will execute 10 times before control passes to Brick 1.2.3, since each time Brick 1.2.2.2 executes, the Counter control is incremented by one.

A.4.5 Local methods

Local methods in a Meshoil program—methods written directly in Meshoil and equivalent to user-created methods or functions in conventional languages—are declared in bricks given withmethods spin⁴. These bricks are positioned in the program like any other with brick. In the same way that with bricks are used in conjunction with do bricks to say ‘With these controls, do these things’, withmethods bricks are used to say ‘With these methods, do these things’.

Just like a control in a with brick, the scope of a local method depends on where it is declared within the program. If the withmethods brick containing the method is declared at the head of the program, the method can be used anywhere within the program.

Figure A.32 on page 306 is a program that performs two, identical string concatenation operations. For example, if the value of Counter is 29, Status will be set to the character string ‘Number of times =29’, the result of joining ‘Number of times =’ and ‘29’. The first operation uses the in-built method control s+ to perform the concatenation. The second operation refers to the local method control called join that is declared at the top of the program in the withmethods brick (which, in this simple example, doesn’t do anything more than use the same s+ method control of the first operation!).

Paralleling the way that controls are defined in with bricks, the brick labelled The join method that defines the local method control consists of a <c> element declaring join as the name, and a <d> element holding its functional description. This <d> element has nested with and do bricks for defining the data and executable parts of the method.

The nested with brick defines the two input parameters for the method control as String1 and String2. The in spin control is used in exactly the same as in a with brick to describe the direction of data flow in relation to the outside world. In a with brick, the outside world might refer to the hardware that lies outside the realm of the whole program. Here, it refers to the external code that calls to the local method.

⁴This feature is not yet implemented in the concept demonstrator.
The nested do brick defines the executable code for performing the join operation based on the two input parameters of String1 and String2. At runtime when the method is called, these parameters will be passed the values of the character string ‘Number of times =’ and the Counter control. The result of the operation will be returned as the value of join.

Figure A.32: Local methods
A.5 Abstract API layer

This section covers the abstract API layer that allows a Meshoil program to make use of the API provided by the platform where the Mesh engine runs.\(^5\)

A.5.1 Calling API methods

In Figure A.33, the program uses the same method control called `join` to concatenate two strings as in appendix A.4.5 on page 305, but here it is supported by the platform’s API instead of being written as a local method control.

All the code responsible for accessing the API of a particular platform is placed in a brick given that platform’s spin—for example, `java` spin for the Java API. These platform-specific bricks are positioned in the program just like with bricks. A typical multi-platform program would have a series of such bricks, one for linking to each platform that the Mesh engine for running the program might run on. This would ensure that, regardless of whether the Mesh engine was written as a Java application on a Java platform, a C++ application on a C++ platform, or as some other application on another platform, the Meshoil program would always run with all of its API method calls successfully serviced.

A simple way to manage these bricks and keep the code clean would be to use Meshoil’s brick insertion mechanism and refer to them by name rather than explicitly writing them into the program. If common naming of Meshoil methods could be agreed across the platforms of all these languages, each platform could provide such platform-specific API mapping code as standard.

\(^5\)This feature is not yet implemented in the concept demonstrator.
All that a Meshoil program would then require would be a single brick, one for each language, that made reference to the insertion of the standard block of API mapping code provided by the respective language—somewhat similar to the use of the `import` command in Java for importing a package of Java classes.

The Mesh engine running on each platform only recognises its own platform spin. In the example here, a Mesh engine running on a Java platform would recognise the brick with `java` spin and process its contents accordingly, but it would ignore the brick for the C++ platform since `c++` spin would be unrecognised.

The brick with `java` spin defines the `join` method control as the `concat` method of the `java.lang.String` class in Java’s API. It does this using the `class` and `method` spin controls that are platform-specific—each platform having its own set of such spin controls. Java’s `concat` method takes the form: `Arg1.concat(Arg2)`, appending the `Arg2` string to the `Arg1` string. At runtime when the `join` method is called, the character string ‘Number of times =’ and the value of the `Counter` control will be passed across to form the `Arg1` and `Arg2` parameters.

The brick with `c++` spin performs the parallel function for the C++ platform. Here, the suitable method in the C++ API for supporting the `join` method control is called `append`. No class definition is needed since this method belongs to the C++ standard template library.

### A.5.2 Abstract methods

The previous example of using the API of the Java platform to support the use of a method control called `join` worked fine, because Java’s API had just the right method to support the functionality required. Java’s `concat` method concatenates two strings—exactly what the `join` method control wanted. With this one-to-one correspondence between client-side (`join`) and server-side (`concat`) methods, all that the platform-specific Meshoil code had to achieve was to link the two together.

But what if this was not the case? What if the program used some method that had no equivalent in Java’s API?

Figure A.34 on page 309 shows such an example. The program uses the `join3` method control that concatenates three strings, adds two exclamation marks and converts the resultant string to upper case. For example, if the value of `Counter` is 29, `Status` will be set to the character string ‘NUMBER OF TIMES [29] SHOULD NOT EXCEED 10!!’. Naturally, no such method exists in the Java API.

The only way that the Java platform can support `join3` is by breaking up the task into two calls to `concat` to concatenate the three strings, another `concat` call to add the exclamation marks, and finally a call to the `toUpperCase` method to convert everything to upper case (Java’s `toUpperCase` method takes the form: `Arg1.toUpperCase()`, converting all characters of `Arg1` string to upper case).

The client-side Meshoil code could be written with an understanding of the limitations of the Java API by making each call match a known Java API method. But why should the Meshoil programmer’s task have to be complicated in this way? And anyway, doing this would be a bad move since it would reduce the universality of the program. Incorporating such Java-specific knowledge into the body of the program would bias it for running on the Java platform and potentially create problems if ever the program was to be run on a different platform in the future.

By providing a layer of abstraction between client-side and server-side method handling, Meshoil removes the problem from the Meshoil programmer’s purview. The programmer can focus on the job of writing code to solve the immediate programming task without having to think about which platform the program might run on.
Figure A.34: Abstract methods (1 of 2)
APPENDIX A. MESHOIL LANGUAGE DESCRIPTION

The example shows how this is achieved. Firstly, the join and uppercase methods are created in Meshoil by associating them with Java’s concat and toUpperCase methods. join and uppercase can now be used in the program as method controls if desired, but in this example, the purpose is more to put handles on Java’s concat and toUpperCase methods so that they can be used to describe the abstract join3 method. The data and executable parts of the join3 method are defined using nested with and do bricks in exactly the same way as for local methods discussed in appendix A.4.5 on page 305. The nested do brick defines how join3 is progressively built up from the component parts of the join and uppercase methods, the parameters passed across for these methods, and the character string ‘!!’.

A.5.3 Platform-specific coding

Method calling is one way that a Meshoil program can exploit the platform on which the Mesh engine is running, but it is also possible to write platform-specific code that the Mesh engine for that platform directly executes as an integrated part of the program.

Figure A.35 on page 311 shows a client-side program that refers to a control called Timer. When the Power switches on, this program starts the Timer running, counting up in seconds. When the value of Timer reaches 60, the program stops the Timer. The program relies on the particular platform where the Mesh engine is running to provide server-side support for such a timer function.

API support for the timer function is demonstrated here by an imaginary Java class called Timerkeeper. Java being an OO programming language, an instance of the Timerkeeper class has to be instantiated before Timerkeeper’s methods can be used. Instantiation is achieved by the Java call: timekeeper = new Timekeeper(Arg1, Arg2), where Arg1 indicates the units for the timekeeper (the character string ‘Seconds’, ‘Minutes’ or ‘Hours’) and Arg2 is the starting numerical value. Timekeeper has two methods: timekeeper.start(), to start the created timekeeper running, and timekeeper.stop(), to stop it running. timekeeper.stop(“Dispose”) can be used to stop and dispose of the timekeeper.

A withmethods brick defines the Timerkeeper class. This is followed by a do brick that only a Java version of the Mesh engine will execute since this brick belongs to the block of code given java spin.

The do brick contains two bricks. The first of these executes when the client-side, platform-independent code sets Timer to the character string ‘Start’, resulting in the three actions of: creating
a timekeeper that counts in seconds starting from zero (new is another Java-specific spin control), calling the timekeeper’s start method to start the timer, and setting up the association of the value returned by timekeeper with the Timer control used in the client-side code.

The second brick executes when the client-side code sets the Timer control to the character string ‘Stop’ after 60 seconds, resulting in a call to timekeeper’s stop method that disposes of the timekeeper.

Figure A.35: Platform-specific coding (1 of 2)
A.5.4 Building a GUI

Figure A.36 on page 313 shows an example of when platform-specific coding might be used to customise an application. The example is a user interface for a device that uses a GUI representation of keys to control device operation, since the device has not been manufactured with any keys and only features a 200x30 pixel-addressable screen.

The code shows GUI creation supported by the Java API. Black key and White key are defined as software controls. A brick with java spin creates the GUI using Java’s standard library of GUI widgets (Java swing objects). A withmethods brick defines the List and ComboBox classes based on Java’s JList and JComboBox swing objects. This is followed by a do brick that generates the GUI every time the Power for the device is switched on.

The do brick creates two widgets called Black key widget and White key widget. Widget definition consists of a label for the widget, the position of the widget’s top left corner, its size (width and height) and a list of the allowed states for the widget. Once the widgets have been created, the Black key and White key controls are mapped to the widgets, and the widgets are in turn mapped to the Screen. The widgets are positioned to be next to each other on the Screen (top left of the Screen is position 0,0, bottom right is position 200,30).

The GUI automatically adapts to the size of the available screen, represented by the setting of the Screen width control. This allows the user interface to be used without modification when the device is manufactured in a variety of physical sizes.

When the width of the screen is less than 600 pixels, the GUI economises on the display space by:

- switching from list widgets to combo boxes (combo boxes are drop down menus that use less space since they only show the currently selected state instead of the full list of alternatives)
- positioning the widgets closer together
- reducing the overall size of the widgets
- shortening the ‘BLACK KEY’ and ‘WHITE KEY’ character strings of the widget labels to ‘B-KEY’ and ‘W-KEY’.
Figure A.36: Building a GUI (1 of 2)
APPENDIX A. MESHOIL LANGUAGE DESCRIPTION

Figure A.36: Building a GUI (2 of 2)
A.6 Using namespacing

Namespacing is an organisational mechanism that can be used at any level in a program, whether as a convenient means of grouping a few, functionally related controls together, or at the other end of the scale in uniquely classifying and identifying large blocks of code. In programs that use brick insertion, it is particularly useful for avoiding naming conflicts.

A.6.1 Nested namespacing

Namespacing is inherited from parent bricks. Each time a namespace is specified for a brick, it is appended to any namespace the brick may already have. This allows nested namespacing schemes to be created in which each namespace belongs to the more general and encompassing namespace above.

This example shows a three-layered namespacing scheme:

![Nested namespacing diagram](image)

Figure A.37: Nested namespacing

A.6.2 Absolute naming within namespaces

By default, control and brick names are specified relative to the namespace to which they belong. When a name needs to be specified in absolute terms with the full namespace pathname because the brick or control doesn’t belong to the current namespace, the `<b>` or `<c>` element is given sic spin to indicate that the name is to be used exactly as written without automatic namespace prefixing.
This example shows the difference between relative and absolute naming:

![Diagram showing server-side and client-side locations with relative and absolute naming examples](image)

Figure A.38: Absolute naming within namespaces

The server-side location contains a brick called `brickname`.

The client-side location belongs to the `Client` namespace and contains two bricks. The first of these bricks uses relative naming by default. It tries to insert the server-side brick by specifying `<b>brickname</b>`, but fails because the brick searched for, `Client.brickname`, does not exist (the extra spacing around the period joining `Client` and `brickname` shows that the ‘Client’ part of the pathname is inherited rather than explicitly specified). The second brick remedies this by using absolute naming. To indicate that `brickname` is the literal name of the brick to insert, it specifies `<b spin="sic">brickname</b>`.

Use of absolute naming for control names is shown in the example of the control called `Control2`. In the first brick this control displays as `Client.Control2`, since its name is specified as `<c>Control2</c>` and it automatically gets prefixed with the `Client` namespace. In the second brick `<c spin="sic">Control2</c>` is specified, resulting in the control displaying exactly as written.

### A.6.3 Working with multiple namespacing schemes

As shown above, `sic spin` achieves absolute naming by effectively resetting (cancelling) the use of namespacing for the element in which it is specified. It is possible to reset namespacing, not just for the current element but for all nested elements too, by specifying a blank namespace in a brick’s namespace attribute (ie, `<b namespace=""/>`). Resetting namespacing signals a change from relative to absolute coding, since from then on down until a new namespace is set all controls will have absolute names.
A.6. USING NAMESPACE\CSHARPING

This example shows how resetting namespacing is particularly useful when inserting bricks, since it allows working with more than one namespacing scheme at a time and increases flexibility in code reuse:

![Diagram of working with multiple namespacing schemes]

Figure A.39: Working with multiple namespacing schemes

When the program is a user interface, namespacing is commonly used to identify which group of controls operate which particular device. It is a targeting mechanism that links software to the particular hardware it is meant to control. When the program includes brick insertion, this targeting can be done either client-side or server-side to suit the particular code reuse scheme.

This example represents code at a server-side and client-side location. The `Server` namespace defines the server-side location and applies to everything that it contains. Similarly, the `Client` namespace defines the client-side location. The server-side location provides two bricks, `Server.Brick1` and `Server.Brick2` which are each inserted twice at the client-side location.

The first brick to be inserted is `Server.Brick1`. Targeting begun server-side continues client-side,
since the control that the brick contains displays as Client.Server.Control1. This control was defined server-side as Server.Control1, but now it nests under the encompassing namespace of Client.

The second brick to be inserted is Server.Brick2. Namespacing has been reset within this brick at the server-side location so that its Control2 remains free of the Server namespace. Targeting is thus only performed client-side. When this brick is inserted, Control2 comes under the client-side namespace and becomes Client.Control2. It illustrates how generic server-side code can be written to integrate automatically with client-side code by taking on the namespace of the client.

Server.Brick1 and Server.Brick2 are then inserted the same way again, but this time with the Client namespace reset. As a result, no Client namespace is added to either of the controls and they appear as Server.Control1 and Control2, indicating that targeting is either performed server-side or simply not at all. Control2 is an example of absolute coding, since it remains free of all namespacing schemes.

Resetting client-side namespacing is useful when the client wants to insert ready-made code that performs some function external to its main one. For example, a program might want to influence the running of a system not central to the particular one it is designed to control.
A.7 Examples of more complex expressions

This section gives examples of logical, numerical and string-based expressions.

A.7.1 Logical expressions

In this example, switching the Power On depends on a complex pattern of key presses:

![Logical expressions diagram]

Since no group spin control is specified at the top level, the default all is implied. The logic is equivalent to the following conventionally described expression:

```c
IF (((A key == Down)
    AND (((B key == Down) OR (C key == Down)) OR
    ((D key == Down)
        AND (((E key == Down) OR (F key == Down) OR (G key == Down) OR (H key == Down)))))
    AND (((I key == Down) OR (J key == Down)) OR ((K key == Down) XOR (L key == Down)))
)
{Power = On}
```
A.7.2 Numerical expressions

This example uses a mixture of spin and method controls:

The implied group spin control for the three conditions is all. The first condition uses the inversion spin control not. The next two conditions use relational spin controls and method controls to define two mathematical inequalities.

The logic is equivalent to the following conventionally described expression:

```plaintext
IF ((A != 7) 
    AND (A > (B * C * 32 * ((D + E + F) / (G - H)))) 
    AND (A < ((J * K) - ((L / 14.9) / M)))
) 
{Power = On}
```
A.7.3 String-based expressions

In order to control string-based operations and overcome the fact that leading and trailing spaces are ignored in XML elements, special codes can be used with the s+ and s- string-based method controls. Special codes are specified in <d> elements following the special code identifier of ‘::’, for example <d>::3</d>.

For the s+ method control, the code is the number of spaces to insert.

For the s- method control, a numeric code indicates the number of characters to delete. A positive value deletes characters from the beginning of the current string result, a negative value deletes characters off the end of the current string result. If the code is specified with a leading zero, all leading or trailing characters are deleted (‘zeroised’) until just the specified number of characters is left. Nothing happens if the code is zero. For example, if the current string result is interface, ::3 would return erface, ::-3 would return interf, ::03 would return ace, ::-3 would return int and ::0 would return interface.

Special codes used with the s- method control can be enclosed in double quotes to make them string-based. Instead of deleting the specified number of characters, they delete all characters up to and including the specified matching string. Characters can be deleted from the start or end of the current string result by specifying a positive or negative match, but in both cases searching begins from the start of the string. If no match is found, nothing is deleted. For example, if the current string result is international, ::"na" would return tional, ::-"na" would return inter and ::" $" would return international unchanged.

Figure A.42 on page 322 shows three bricks that perform a number of string-based operations. The first brick creates a sentence using the value held in the Cost control. For example, if Cost is 3.55, Shopping report would hold the result ‘I spent $3.55 on cake.’.

The second brick is the reverse of the first. By removing information it just leaves whatever value of Cost was pasted into Shopping report. It then sets this value in the Price of cake control. For example, if Shopping report is ‘I spent $3.55 on cake.’, Price of cake would be 3.55.

The third brick summarises the total expense of the shopping trip. It extracts information about the cost of items in Shopping report, calculates their sum and outputs the result in Expense report as a sentence. It can handle the cost of each item in Shopping report being expressed between $ and space delimiters, or between AUD and space delimiters. For example, if Shopping report is either ‘I spent $3.55 on cake, $2 on cheese, and $1.40 on bread.’ or ‘I spent AUD3.55 on cake, AUD2 on cheese, and AUD1.40 on bread.’, Expense report would be ‘You spent a total of $6.95!’ . The first s- in Expense report extracts 3.55 by finding the first occurrence of $ or AUD, the second s- extracts 2 by progressively finding the second occurrence of $ or AUD, and the third s- extracts 1.40 by progressively finding the third occurrence of $ or AUD. Fourth and subsequent values in Shopping report are ignored, but less than three values cause items to be summed more than once.
Figure A.42: String-based expressions
Appendix B

Discussion on Meshoil

B.1 Language design

B.1.1 How easy is it to program in Meshoil?

Is Meshoil an easy language to use? It is easier to program in Meshoil than in a conventional language?

It is beyond the scope of this research project to answer questions like these, but Meshoil’s simple syntax, few coding rules and focus on reducing the traditional burdens placed on the role of the programmer, would seem to suggest yes. Caution is needed, however, since simplicity alone doesn’t prove the point. Sometimes, a simple language can mean more programming effort to achieve a given result, because the language lacks the sophisticated constructs that allow code to be written concisely.

In any event, it is likely that Meshoil would readily lend itself to the use of assistive tools to facilitate the programming process, as mentioned in section 3.12.1 on page 85, since the simpler the structure of a programming language, the easier it is to create such tools.

B.1.2 Increasing the descriptive power of a hierarchy

XML, on which Meshoil is based, is a hierarchical language. Hierarchies are so useful for representing the arrangement of information because they handle groupings so well.

Individuals in a hierarchy are defined by their three-way relationship to child, sibling and parent individuals. Each individual can have any number of children, any number of siblings, but only a single parent. This restriction of only one link to individuals above limits the usefulness of the hierarchy as a framework for describing any type of relationship.

Meshoil avoids this weakness by allowing an individual to have any number of parents (see appendix A.1.7 on page 282). This increases Meshoil’s descriptive power and allows, for example, the setting of a single variable in a conditional expression (represented in a child brick) to be based on more than one condition (represented in multiple bricks making up the parent).

B.1.3 Trading execution efficiency for usability?

A program written in Meshoil runs in the Mesh engine. The Mesh engine is itself a program—a Java application in the case of the concept demonstrator. Purely from the point of view of executional efficiency, it would be better to write the Meshoil program directly in Java and miss out the additional layer of the Mesh engine.
APPENDIX B. DISCUSSION ON MESHOIL

As part of Mesh implementation, Mesh engines might be coded directly on chips in machine language to avoid this inefficiency, but even if this did not happen and the additional layer of a conventional language was maintained, how much would the loss in executional efficiency really matter?

In many current areas of software engineering, of course, executional efficiency does matter and processing speed is critical. Take streaming on the Internet where compression techniques and anything else that can boost data throughput are seen as beneficial, even though they add to the complexity of the engineered solution.

But in other areas the Darwinian pressures are different, and becoming increasingly so as the processing power of computers grows. Useability, not processing speed, is the critical factor. This is particularly true in user interface design, since for much of the time a user interface is waiting for something to happen\(^1\).

\(^1\)Some parts of the Mesh infrastructure are likely to be more vulnerable to slow response times than others. For example, information passed across in ghost-in-the-machine API function calling involves the additional delay factor of an Internet leg, which might be an issue in some situations.
B.2 Beyond the functionality of a conventional language

B.2.1 Replacing programming constructs with more generic equivalents

Group spin controls are an example of the way that Meshoil seeks to replace conventional programming constructs with more generic equivalents. The all/any/any(1) group spin controls replace the need for AND/OR/XOR (exclusive OR) logical operators.

Group spin controls offer three advantages over the use of conventional logical operators. Firstly, they are a more general solution to the problem of how to achieve a boolean result from logical relationships. Conventional logical operators are limited to two operands. The operator is expressed syntactically as a conjunction that joins each operand pair. This restriction does not apply to group spin controls since the set of operands they apply to can be any number.

In the case where there are just two operands, group spin logic reduces to that of the conventional logical operators: both operands must be true for all to be true, either or both operands must be true for any to be true, and either but not both must be true for any(1) to be true.

For the AND operation, we can express this equivalence for the two-operand case as:

\[
\{ A, B \} \in \text{all} \equiv A \times B
\]  

(B.1)

While for the OR operation:

\[
\{ A, B \} \in \text{any} \equiv A + B
\]  

(B.2)

And for the XOR operation:

\[
\{ A, B \} \in \text{any}(1) \equiv A \otimes B
\]  

(B.3)

Secondly, group spin controls remove any semantic ambiguity. Consider the expression ‘A AND B OR C’. Where do the implied brackets lie? Is it ‘A AND (B OR C)’—meaning true if either A and B are true, or A and C are true? Or is it ‘(A AND B) OR C’—meaning true if either A and B are true, or C is true? In such cases, an arbitrary syntactic rule will apply that has to be remembered by the user.

Thirdly, the very economy of expression in being able to declare a set of operands of any size simplifies the syntax. There is no need to intersperse elements with repeated operators. A declaration like ‘A AND B AND C AND D’ is replaced by ‘All\{A,B,C,D\}’. The result is a syntax that is more natural in resembling how we think and speak. Consider the case of offering someone a box of chocolates and saying ‘Take all’. The meaning is clearly understood and avoids the tedium of having to say ‘Take this and this and this and...’
B.2.2 Confluent conditional expressions

Consider the code shown in Figure B.1 for shutting a system down in the event of a malfunction. Two types of malfunction are handled, as indicated by the Alarm LED showing red or orange. When the Alarm LED lights, the user must acknowledge the particular malfunction and shut the system down by pressing whichever of the two keys, Red key or Orange key, matches the type of malfunction indicated by the LED. Only a single key must be pressed.

![Diagram of emergency shutdown procedure](image)

Figure B.1: Emergency shutdown procedure—executionally inefficient

In a conventional language, this logic would be written as follows:

```plaintext
IF (((Red key == Down) AND (Orange key != Down) AND (Alarm LED == Red))
    OR ((Red key != Down) AND (Orange key == Down) AND (Alarm LED == Orange))
)
    {System = Stop}
```

Determining whether the conditional expression is true or false involves checking between three and six conditions, depending on the current settings of the key and LED controls. The code works fine, but is inefficient since it may lead to the same condition being checked more than once.

Take, for example, the case of the user pressing the Orange key in response to an orange-type malfunction. Four conditions are checked before the conditional expression evaluates as true and the system stops. The first condition for a red-type malfunction is false, since the Red key is not Down. This causes execution to jump immediately to the three conditions of an orange-type malfunction. The first of these checks whether the Red key is not Down. Although the Red key has already been checked and found not to be Down, the check is unnecessarily repeated here.
Figure B.2 shows how to use nested conditional expressions for programming the emergency shutdown procedure in a way that avoids these executional inefficiencies.

In a conventional language, the only solution would be to create a method consisting of the code that stops the system, and calling this method from the two locations where the code is currently duplicated. But this isn’t really an ideal solution, since invoking such programming constructs...
APPENDIX B. DISCUSSION ON MESHOIL

as methods complicates the task at hand by bringing up other programming issues, such as the change in scope for the variables used in the code caused by physically moving this block of code to another program location. Neither does it completely eradicate the code duplication problem, since there is still information that needs to kept identical in the form of duplicated method calls.

Figure B.3 shows how Meshoil provides a unique solution to the problem, achieving the executional efficiency of Figure B.2 but without any form of code duplication.

Figure B.3: Emergency shutdown procedure—executionally efficient without code duplication

In Meshoil syntax, a conditional statement is created by declaring conditions in the body of the brick and declaring the actions to be performed in child bricks. By enclosing all the branching conditional logic shown in Figure B.2 in a parental \(<\) element, all this logic becomes defined as part of the body of a single brick that has a single child brick for performing the system shutdown. The result is a *confluent conditional expression* in which different executional pathways through the conditional expression (achieved by nesting if spin controls) can still lead to the same end point and resultant executional outcome—something that is impossible to do in a conventional language. A language that allowed use of a GOTO statement would, of course, allow such programming, but Meshoil achieves the same effect in a structured way without any of the well-known problems associated with GOTO usage.

The example here was the simple one of a child brick consisting of a single assignment expression, but imagine if this was not the case and the child brick consisted of code for another similar procedure that involved branching code. This, in turn, might lead to deeper levels of code of similar complexity. The conventional solution of breaking up code into methods and having to use nested method calls would rapidly complicate program structure. Meshoil, however, makes it possible to represent such logic in simple, sequentially-written code.

---

\(^2\)This feature is not yet implemented in the concept demonstrator.
B.3. OO programming principles

This section discusses how Meshoil incorporates the following principles of an OO programming language:

- abstraction
- objectification
- code reuse
- inheritance
- encapsulation
- polymorphism.

B.3.1 Abstraction

Abstraction means being able to problem solve by focusing on the main issues at hand, free of the distracting details of how to implement the solution in the real world. It allows us to choose the most appropriate level of detail to be working with at any given time.

Meshoil is built on abstraction. To start with, the language itself is completely abstract. This is unexpected in an XML-based language and is usually only achieved in the syntax of a conventional language. For example, no UIDL (XML-based, User Interface Design Language) can claim to be an abstract language.

But it is not just the syntax. Meshoil’s functional connections to the outside world are abstract too, represented by the abstract hardware layer and abstract API layer. These two layers insulate the program sandwiched in the middle, allowing the solution that it provides to remain abstract and independent of the particular way it interfaces with the bigger engineering picture.

Finally, abstraction is also supported in the very process of writing a Meshoil program. The Meshoil programmer is able to start from a very simple idea of how something is to work, express it in abstract, declarative terms, and progressively build up a design until finally a fully-detailed, imperative, working version is reached (Figure B.4).

![Figure B.4: Abstraction](image)
APPENDIX B. DISCUSSION ON MESHOIL

B.3.2 Objectification

Code written in Meshoil is just a collection of bricks. Programs are built up by arranging bricks next to each other and within each other. In OO terms, Meshoil is little more than multiple instances of a single class of object (Figure B.5).

B.3.3 Code reuse

Mechanisms for code reuse are commonly supported in all OO programming languages. In Meshoil, the ability to reuse code is built into the very syntax of the language in the way that bricks can be inserted into other bricks (Figure B.6).
B.3.4 Inheritance

In Meshoil, bricks inherit information from the bricks above. Appendix A.6.1 on page 315 showed how namespace inheritance is used to create nested namespacing schemes. Here, the examples demonstrate control inheritance and spin inheritance.

Control inheritance

Definitions of controls in with bricks are inherited by all sibling do bricks. do bricks may contain embedded with bricks of their own, defining further instances of controls that are added to any already inherited from code above (Figure B.7).

Figure B.7: Control inheritance
Spin inheritance

Once a spin control has been used to set a certain spin for a brick, the spin is inherited by all bricks below where the spin was specified. The spin continues to be inherited at progressively deeper levels until explicitly changed.

Figure B.8 shows how the all, any and any(1) settings for group spin are set, reset and passed on down by inheritance.

Figure B.8: Spin inheritance
B.3.5 Encapsulation

Encapsulation is a privacy mechanism for restricting access to code. In Meshoil, code is made private and protected from change by isolating it, either logically or physically, from the rest of the program. Code can be logically isolated by holding it under a unique namespace, or it can be physically isolated by holding it in a remote location from where it can only be read (Figure B.9).

![Encapsulation Diagram](image)

Figure B.9: Encapsulation

B.3.6 Polymorphism

Polymorphism refers to the ability to perform functionally equivalent processing on dissimilar objects. The overall intent of the processing is the same in each case but the details differ.

In Meshoil, polymorphism is achieved by using namespacing and brick insertion in combination. On the server-side, each polymorphic version of a functionally equivalent brick is set up under a different namespace. The client-side code is able to insert whichever version of the brick is required without any direct reference to the version by simply belonging to the matching namespace. In this way, the alternative bricks held server-side act as polymorphic ‘methods’ that use namespaces for their ‘class names’.
APPENDIX B. DISCUSSION ON MESHOIL

In Figure B.10, code has been set up server-side to indicate device malfunction for two different devices: a TV and a fridge. The client-side user interfaces for the two devices make use of the server-side code by polymorphic reference. Not only are the user interfaces able to indicate device malfunction without any knowledge of the hardware features available to do this, but polymorphism means that identical error-handling code can be written for both.

The hardware provided to indicate device malfunction consists of an LED for the TV and a Buzzer for the fridge. When a user interface detects device malfunction, it is able to indicate this physically on the respective device by reference to the polymorphic brick called alarmcondition. Although alarmcondition is specified in the code, the actual brick inserted is different in each case, being either the server-side tv.alarmcondition or fridge.alarmcondition brick according to the namespace that the user interface belongs to. The action to take following the fault is also different for the two devices—polymorphism again being used to make sure the correct message displays on each device by reference to the polymorphic brick called display.

![Figure B.10: Polymorphism](image)

<table>
<thead>
<tr>
<th>Server-side location</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm conditions</td>
<td>Display the action to take after the alarm.</td>
</tr>
<tr>
<td>tv.alarmcondition</td>
<td>fridge.alarmcondition</td>
</tr>
<tr>
<td>LED</td>
<td>Buzzer</td>
</tr>
<tr>
<td>Beep</td>
<td>On</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Client-side location</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Somewhere in the TV user interface where a fault has been detected and needs to be indicated to the user, the code makes reference to a brick called alarmcondition and another called ‘display’ (the tv prefix is automatically added).</td>
<td>Somewhere in the fridge user interface where a fault has been detected and needs to be indicated to the user, the code makes reference to a brick called alarmcondition and another called ‘display’ (the fridge prefix is automatically added).</td>
</tr>
<tr>
<td>tv.alarmcondition</td>
<td>fridge.alarmcondition</td>
</tr>
<tr>
<td>tv.display</td>
<td>fridge.display</td>
</tr>
<tr>
<td>tv.LED</td>
<td>fridge.display</td>
</tr>
<tr>
<td>tv.Buzzer</td>
<td>On</td>
</tr>
<tr>
<td>On</td>
<td>Unplug fridge to allow to defrost</td>
</tr>
</tbody>
</table>

TV is faulty, return to manufacturer |

TV is faulty, return to manufacturer
Appendix C

Tools

C.1 The Mesh engine

The concept demonstrator version of the Mesh engine is a Java application written to run on the Java 2 Standard Edition platform (version j2sdk1.4.2_04).

C.1.1 The main window

The main window of the Java application controls the running of the Mesh engine. This window displays at the top left of the desktop when the application starts. Figure C.1 shows the appearance of the main window immediately after the application has started.

![Main window of the Mesh engine](image)

Figure C.1: Main window of the Mesh engine

The main window consists of five buttons and a status area underneath. The buttons control the running of the Mesh engine. Buttons that are disabled, such as the Run button before any Meshoil file has been loaded, display faint. Table C.1 on page 336 describes the buttons.

The main window uses traffic light icons to indicate the status of the application. Error and feedback messages display at the left of the status area. The cluster of three traffic lights to the right indicates whether any with bricks have been loaded, whether any do bricks have been loaded, and whether a compiled program is ready to run or is currently running.
APPENDIX C. TOOLS

### Table C.1: Buttons in the main window of the Mesh engine

<table>
<thead>
<tr>
<th>Button</th>
<th>Function</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quit</strong></td>
<td>Terminate the Mesh engine application.</td>
<td>This is the preferred way to terminate the application rather than using the standard close window button at the top right of the main window. It ensures that all running threads terminate correctly before the Mesh engine application itself terminates.</td>
</tr>
<tr>
<td><strong>Load</strong></td>
<td>Load and compile a Meshoil file.</td>
<td>The file name is specified in the field to the right of the Load button. The file is checked for being a valid Meshoil file and compiled into an executable program. Checking includes conformity with the basic Meshoil file structure specified in the Meshoil tester (XML Schema file, see appendix C.2.1 on page 339) and checking various syntax rules that are beyond the scope of XML schema files to represent. More than one Meshoil file can be loaded, allowing an executable program to be built up by loading files sequentially.</td>
</tr>
<tr>
<td><strong>Run</strong></td>
<td>Toggle the running of a program.</td>
<td>When the program is running, this button displays as Stop.</td>
</tr>
<tr>
<td><strong>Clear</strong></td>
<td>Clear the program.</td>
<td>Clears information about all loaded Meshoil files and reinitialises the Mesh engine so that it is ready to load a new Meshoil file.</td>
</tr>
<tr>
<td><strong>Save</strong></td>
<td>Save the currently loaded program as a Meshoil file.</td>
<td>The file name is specified in the field to the right of the Save button.</td>
</tr>
</tbody>
</table>

C.1.2 Control windows

When a Meshoil program is loaded, additional windows appear on the desktop. These windows display the list of all the controls read into the Mesh engine after one or more Meshoil files have been loaded. Together, these windows provide a complete snapshot of the current state of the program since each control represents a programming variable.

The type of GUI widget (Java swing object) used to display each control indicates its datatype. String-type controls display as text fields, if they can take any text, or dropdown menus if they have been restricted to a list of strings. Integer and decimal-type controls display as spinners showing numerical values. Decimal values are shown to three decimal places.

A single window is used to list all controls that have been explicitly defined by declarations in all the loaded with bricks. This window uses arrow icons that display to the left of each control name to indicate the direction of any data flow between the Mesh engine and the external world for that control (the direction spin).

Additional windows are used to list controls that have not been explicitly defined in any with brick. Each of these windows lists the group of undeclared controls from a single do brick that represents a thread.
Figure C.2 shows a typical example of the desktop during the running of a Meshoil program. This example is the user interface for the fan that was described in section 2.2 on page 26. The main window is on the left. Next to it are two control windows. The program has two do bricks and runs as two threads—one for the user interface and one for simulating the action of the keys.

The first control window displays the list of defined controls, representing all the hardware features of the device defined in the program’s single with brick. Direction spin is out for the Display and Pan controls, in for the Black key and White key controls, and inout for the Motor control.

The second control window called Key management displays the single, undefined Key timer control used in the second thread for timing how long to wait before returning the keys to their Up positions. No additional control window is needed for the first thread running the user interface, since all of the controls used in this thread’s do brick are defined in the with brick and are already displaying in the first control window.

In the concept demonstrator, a user can use the widgets in the control windows to change the value of any control manually at any time. In a real example of a Meshable device, the state of the program running in the Mesh engine would not be directly accessible like this. User interaction with the program would be limited to those controls mapped to physical means of user input, such as keys on a keypad, having in or inout direction spin. To reflect this fact, in and inout controls display bold in the control windows.
C.1.3 Implementation

Table C.2 lists the Java classes used in the code of the Mesh engine application.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brick</td>
<td>Holds the compiled version of the information held in a single <code>&lt;brick&gt;</code> element.</td>
</tr>
<tr>
<td>Control</td>
<td>Handles the management of each compiled <code>&lt;control&gt;</code> element.</td>
</tr>
<tr>
<td>ControlID</td>
<td>Handles the setting up and management of identifying information about a <code>&lt;control&gt;</code> element.</td>
</tr>
<tr>
<td>Do</td>
<td>Handles the thread for executing a <code>do</code> brick.</td>
</tr>
<tr>
<td>Engine</td>
<td>Sets up the main window of the application.</td>
</tr>
<tr>
<td>Loader</td>
<td>Handles everything to do with loading, compiling and saving Meshoil files.</td>
</tr>
<tr>
<td>Mesh</td>
<td>Main class for starting the application. Holds all global fields.</td>
</tr>
<tr>
<td>MeshException</td>
<td>Handles exception handling for the application.</td>
</tr>
<tr>
<td>NumericBrick</td>
<td>Holds the compiled version of Meshoil code that contains method controls for performing numeric and string-based operations on control values (eg, Meshoil code beginning with <code>&lt;control&gt;+&lt;control&gt;</code> that adds control values).</td>
</tr>
<tr>
<td>Window</td>
<td>Handles the GUI display of a <code>with</code> or <code>do</code> brick.</td>
</tr>
</tbody>
</table>

Table C.2: Java classes used in the Mesh engine application

The program starts by instantiating an `Engine` object. The constructor for the `Engine` class sets up the GUI window for the application. When the user enters the name of a Meshoil file and hits the `Load` button, methods in the `Loader` class are used to read in the Meshoil file and create a Java object tree that mirrors the contents of the Meshoil file.

Rather than use low-level XML parsers such as SAX and DOM for reading in an XML file into a Java program, the `Loader` class makes use of Java’s JAXB architecture for XML binding. JAXB binds components of a source schema to schema-derived Java content classes, effectively handling everything to do with creation of the Java object tree.

The simplicity of a Meshoil program consisting of no more than a collection of syntactically identical objects—instantiated forms of the `Brick` class—is reflected in the working heart of the Mesh engine application. When the user hits the `Run` button, the Java object tree is repeatedly parsed by a method called `executeBrick` in the `Do` class. This method is recursively called to process each nesting brick in the Java object tree.

The source code for the Mesh engine application is provided in electronic form with this thesis.
C.2 XML-based tools

C.2.1 The Meshoil tester

The Meshoil tester is an XSD (XML schema) file for validating the contents of a Meshoil file. It performs a first-stage validation on the file to ensure that it contains recognised XML elements that are in the correct order.

The Meshoil tester is listed below:

```xml
<?xml version="1.0"?>
<!--
================================================================================
MESHOIL TESTER
--------------

The Meshoil tester is an XML Schema file (XSD) that checks whether Meshoil files (XML) have the correct syntax. XML Schema is not powerful enough to fully validate these files.

The Meshoil tester checks that the XML elements are in the right order with the right names, but is not able to validate the contents of the XML elements. This has to be done in the Mesh engine itself.

Each Meshoil file references the Meshoil tester (in the schemaLocation attribute) so that at least a partial syntax check of its contents is possible by directly running XML Schema on the file itself.

The five XML elements in Meshoil:

<table>
<thead>
<tr>
<th>#</th>
<th>NAME</th>
<th>MEANING</th>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>meshoil</td>
<td>root</td>
<td>Standard XML attributes for an XML root element</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>annotation</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>brick namespace spin</td>
<td>any string (Mesh engine only recognises 'with/do/not/sic/online'. Other spins such as 'java/inactive' cause the brick to be ignored.)</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>control spin</td>
<td>any string (Mesh engine only recognises 'sic/online'. Other spins cause the control to be ignored.)</td>
</tr>
<tr>
<td>5</td>
<td>d</td>
<td>data</td>
<td></td>
</tr>
</tbody>
</table>

[Last changed: 3 May 07]

================================================================================

---
APPENDIX C. TOOLS

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.mesh.org"
    xmlns="http://www.mesh.org"
    elementFormDefault="qualified">

<!--
================================================================================
Root element: meshoil
The root element consists of:
  1/ One or more optional combinations of a’s and b's. These are top-level
     elements with a’s giving intro information about the b’s which form
     specification tables in the Meshoil viewer.
================================================================================
-->  
<xsd:element name="meshoil">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:group ref="abGroup" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:group name="abGroup">
    <xsd:sequence>
        <xsd:element name="a" type="aType" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="b" type="bType" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:group>

<!--
================================================================================
Type: bType
Each b consists of:
  1/ One or more optional a’s describing the brick.
  2/ One or more optional combinations of c’s and d’s, (their order has meaning
     in bricks performing numeric operations such as subtraction).
  3/ One or more optional b’s.
================================================================================
-->  
<xsd:complexType name="bType" mixed="true">
    <xsd:sequence>
        <xsd:element name="a" type="aType" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:group ref="cdGroup" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="b" type="bType" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="namespace" type="xsd:string" use="optional"/>
    <xsd:attribute name="spin" type="xsd:string" use="optional"/>
</xsd:complexType>

<xsd:group name="cdGroup">
    <xsd:sequence>
        <xsd:element name="c" type="cType" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="d" type="dType" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:group>

<!--
-->  
<!--

340
```
Type: cType

Same as bType but with no a's and one attribute.

```xml
<xs:complexType name="cType" mixed="true">
  <xs:sequence>
    <xs:group ref="cdGroup" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="b" type="bType" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="spin" type="xsd:string" use="optional"/>
</xs:complexType>
```

Type: dType

Same as bType but with no a's and no attributes.

```xml
<xs:complexType name="dType" mixed="true">
  <xs:sequence>
    <xs:group ref="cdGroup" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="b" type="bType" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Type: aType (Top level: a's in b's)

Each a consists of:
1/ One or more optional combinations of a's and d's.

```xml
<xs:complexType name="aType" mixed="true">
  <xs:sequence>
    <xs:group ref="adGroup" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

```xml
<xs:group name="adGroup">
  <xs:sequence>
    <xs:element name="a" type="aaType" minOccurs="0"/>
    <xs:element name="d" type="xsd:string" minOccurs="0"/>
  </xs:sequence>
</xs:group>
```

Type: aaType (Level 2 or deeper: a's in a's)

Each a consists of:
1/ One or more optional a's.

```xml
<xs:complexType name="aaType" mixed="true">
  <xs:sequence>
    <xs:element name="a" type="aaType" minOccurs="0"/>
    <xs:element name="d" type="xsd:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```
C.2.2 The Meshoil viewer

The Meshoil viewer is an XSL stylesheet for converting the contents of a Meshoil file to a graphical format. It reads in the file, adds descriptive and summary information, and outputs the result in an HTML-based, colour coded format suitable for display in a Web browser.

The logic of the viewer is broadly similar to that of the methods used in the Loader class of the Mesh engine application since its role is to process the contents of the Meshoil file by traversing a tree structure.

Being an XSL stylesheet, the source code for the viewer is written in XML. It consists of a series of XSL templates that are used to read the nested XML elements in a Meshoil file and display them as nested HTML tables for viewing in a browser. The viewer processes the Meshoil file in a single pass, traversing down the tree pathway represented by the nested XML elements in the file. When a brick contains references to other bricks to be inserted, these bricks are searched for in the Meshoil file and processed accordingly before processing of the tree pathway continues.

The source code for the viewer is provided in electronic form with this thesis.
The ‘source code for the Mesh engine’ is available on CD, which is included with the print copy held in the University of Adelaide Library.
Glossary

Abstract API layer  Maintaining platform-independent programming even when making use of a platform’s API.

Abstract hardware layer  The ability to control hardware without any knowledge of how it physically works.

API  Application Programming Interface.

Aspect-oriented programming  Type of programming where different aspects of a system’s behaviour are individually programmed and then woven together to produce the final code version.

AT command  Command language for the remote control of modems.

Atomic-level networking  Making physically discrete, concurrently executing programs interact in any way that is desired.

Biomimicry  Imitating designs found in nature to solve human problems.

Bloatware  A derogatory term for feature overflow in a user interface.

Building-brick design flexibility  Building up any level of required program functionality by mixing and matching discrete blocks of code.

Code filtering  Filtering out errors from a program without having to edit the code directly.

Code integrity through accreditation  Using a strategic approach to manage the problem of unreliable and malicious software.

Cost-free user flexibility  The benefits of a flexible design approach with none of the traditional drawbacks.

Cut and paste programming  Robust support of code manipulation and the freedom to paste code almost anywhere in a program.

Engineering integration by layering  The ability of a Meshoil program to fit into the bigger engineering picture as self-contained, easy-to-connect functional layer.

Esterel  An imperative, synchronous programming language with a strong bias towards programming control-dominated software and hardware reactive systems.

Futures programming  Writing programs that anticipate the future by controlling hardware that has yet to exist.
GLOSSARY

**Ghost-in-the-machine API** Instantaneously upgrading an electronic device, no matter how basic, to the level of a programming platform.

**Ghost processing** Boosting the computing power of a device through the use of virtual processors.

**Globally-unified hardware control** Standardising how hardware is controlled in the real world.

**Google** Popular search engine used in the Web.

**Grid computing** Distributed, parallel programming spread over a network of processors.

**GSM** Global System for Mobile communication.

**GUI** Graphical User Interface.

**Hardware reuse** Reusing the existing hardware features of a device to provide the user with what appears to be a different device.

**HTML** HyperText Markup Language.

**Interface agents** Making autonomous decisions designed to anticipate users’ needs and make life easier.

**Long tail** Recent social phenomenon describing the shift from mass markets to niches.

**Mesh** A parallel, dynamic world to the Web.

**Meshable device** Electronic device with installed Mesh engine.

**Mesh engine** Software application that drives Mesh operation.

**Meshoil** Universal markup language of the Mesh.

**Meshoil tester** XSD file for validating the syntax of a Meshoil program.

**Meshoil viewer** XSL file for converting a Meshoil program to a graphical format displayable in a Web browser.

**Meshpoint** Location of a Meshable device.

**Meshpointer** Specific location in a Meshoil program made accessible to any Meshable device.

**MindScript** Programming language for Lego robots.

**OO** Object-Oriented.

**OPT** Open-Plan Thinking.

**OSI** Open System Interconnection standard for worldwide, network-based communication.

**Point-accurate interoperation** Achieving maximum flexibility of device interoperation with minimum effort by modifying code at the exact location where interoperation is designed to have its effect.

**Program morphing** The process by which a program changes into a different program while running.
Responsibility optimisation  Redistributing the responsibilities of users, programmers and manufacturers in such a way to minimise the overall workload and achieve a better solution for all.

SIC  Set In Concrete.

Software-team scalability  Scalability that isn’t just a feature of the language but is represented in those using the language too.

Superprogramming  Unlimited flexibility in device interoperation across the Mesh.

Transparent programming environment  Making housekeeping tasks traditionally associated with programming transparent for the programmer.

Ubiquitous computing  Interoperation of disparate devices.

UIDL  User Interface Description Language, generic term for XML-based languages that describe user interface design.

UIML  User Interface Markup Language, a type of UIDL.

Universal agent  Replacing the need to program interface agents individually by using a universal agent capable of autonomously customising the operation of any type of device.

Universal usability  Design of products and services that are usable by all people.

URC  Universal Remote Console, a type of UIDL.

User-generated software  Web 2.0-style software created directly by the user.

UsiXML  USer Interface eXtensible Markup Language, a type of UIDL.

Virtual hardware  Adding hardware functionality to a device without physically installing anything.

Virtual logic  Changing the logic of how a program runs without editing code.


Web 2.0  Second generation of Web services that focus on user-generated content.

Wiki  A website that supports collaborative Web page authoring by allowing its users to add, remove, and modify its contents.

Wikipedia  Wiki-style online encyclopedia website.

XForms  XML Format Web Forms, a type of UIDL.

XIML  EXtensible Interface Markup Language, a type of UIDL.

XML  eXtensible Markup Language.

XSD  XML Schema Definition (part of the XML suite of technologies).

XSL  eXtensible Stylesheet Language (part of the XML suite of technologies).

XUL  eXtensible User Interface Language, a type of UIDL.

YouTube  Wiki-style video sharing website.
References


REFERENCES


REFERENCES


Index

3-in-1 calculator, 202
absolute namespacing, 315
abstract
  API layer, 74, 270, 274, 307–314, 329
  hardware layer, 28, 37, 45, 74, 303, 329
  method, 308
  programming, 18, 74, 80, 120, 122, 125, 202
abstraction, 273, 308, 329
adaptive systems, see interface agents
annotation element, 279
API, 70, 73–74, 206, 307
aspect-oriented programming, 25
AT command, 217
atomic-level networking, 36
biomimicry, 7, 17, 269
bloatware, 43
brick
  element, 280–286
  insertion, 41, 77–81, 83, 184, 258, 285, 307,
  316, 330, 333
building a GUI, 312
building-brick design flexibility, 36
C-Bus system, 14
calculator, 184
car with fan and radio, 182
case studies
  3-in-1 calculator, 202
  calculator, 184
  car with fan and radio, 182
  fan, 26
  mobile phone/emergency pager, 217
  radio, 152
  robot, 254
  safety fan, 128
  sports car, 171
  thermometer, 238
  two fans, 35
Clipsal, 14
cloning devices, 101
code
  efficiency, 107, 140, 323, 326
  filtering, 115
  integrity through accreditation, 116
  reuse, 25, 80, 184, 273, 285, 317, 330
concept demonstrator, 7, 109, 295, 298
GUI, 336
  sample run, 30
  tools, 20, 335
concurrency, 5, 36, 49, 107, 141, 182, 207, 271,
  304
  synchronisation, 205
confluent conditional expression, 326
consumer culture, 4, 272
control
  defining, 303
  element, 279
  feedback, 105
control-IT, 1, 3
conventional design, 15, 17, 41, 68, 77, 90, 97,
  119
  trade-off, 43
cost-free user flexibility, 38
cut and paste programming, 78
data element, 280
datatype, 49, 286–287, 290, 293, 302, 336
declarative programming, 122–126, 128, 253, 329
defining controls, 303
design flexibility, 36
device cloning, 101
DNA, 7, 17, 269
do brick, 293–301
dynamic code insertion, 25
dynamically extensible software, 25
embedded device control, 266
emergency pager, 217
encapsulation, 273, 333
engineering integration by layering, 74
INDEX

error handling, 207
Esterel, 266
examples, see case studies
executional efficiency, 107, 140, 323, 326
expression
  confluent conditional, 326
  logical, 319
  numerical, 320
  string, 321
fan, 26
feasibility of the Mesh, 272
future research work, 276
futures programming, 46, 235
general-purpose programming, 125, 237, 274–275
ghost processing, 109
ghost-in-the-machine API, 73
globally-unified hardware control, 80
Google, 85
grid computing, 5–6, 109, 271
GSM, 217
GUI, 126
  building, 312
  concept demonstrator, 336
  tools, 255
hardware, 80
  controls, 26–29, 46, 291, 303
  reuse, 86, 97, 217
  virtual, 64
hierarchical representation, 323
imperative programming, 122–126, 128, 329
implementation
  issues, 96, 273, 323
  of the Mesh, 272–273
inheritance, 37, 273, 331
interface agents, 5–6, 89, 271
Internet, 1–14
  Mesh protocol, 15, 25
  timing of information transfer, 83
keywords
  general, xv
  this thesis, xv

knowledge-IT, 1, 3
Lego, 18, 24, 40, 254, 281
literature review, 10
local method, 305
logic, 295–300, 319–322
logical
  expression, 319
  operator, 325
long tail, 4, 272
malicious software, 115
manufacturing, 38, 45, 63, 68, 75–81, 115, 303
Mesh, 3–16, 33, 270–273
  engine, 15, 20, 26, 74, 295, 307, 323, 335–338
  feasibility of, 272
  implementation, 272–273
  Internet protocol, 15, 25
  principles, 33
Meshable device, 15
Meshoil, 15–16, 125–126, 277–322
  annotation element, 279
  brick element, 280–286
  case studies, 127–235
  comparison to other languages, 237–267
  conclusions, 273–275
  control element, 279
  data element, 280
  datatype, 49, 286–287, 290, 293, 302, 336
  discussion on, 323–334
  do brick, 293–301
  logic, 295–300, 319–322
  method controls, 295–300
  overview, 23–32
  programming, 302
  spin, 286
  spin controls, 290, 294
tester, 20, 277, 339
tools, 20, 339–342
viewer, 20, 28, 41, 285, 342
with brick, 290–292
XML elements, 278
Meshpoint, 15
Meshpointer, 15
method
  abstract, 308
  calling, 70–74, 206, 305, 307

352
controls, 295–300
local, 305
MindScript, 254
mobile phone, 217
morphing, 84
multiple namespacing schemes, 316
multithreading, 5, 36, 49, 107, 141, 182, 207, 271, 304
synchronisation, 205

namespacing, 25, 37, 183, 285, 315–318, 333
absolute, 315
multiple schemes, 316
nested, 315
nested namespacing, 315
Nokia, 217
numerical expression, 320

object-oriented, 18, 24, 273, 329–334
objectification, 273, 330
OO, see object-oriented
open-plan thinking, see OPT
operator, 325
OPT, 17–19, 23, 122, 269–273, 275
OSI model, 15

path hijack and release, 59, 233
physical controls, 152
platform independence, 73, 120, 307–310
platform-specific coding, 310
point-accurate interoperation, 97
polymorphism, 273, 333

principles
abstract API layer, 74
abstract hardware layer, 37
atomic-level networking, 36
building-brick design flexibility, 36
code filtering, 115
code integrity through accreditation, 116
cost-free user flexibility, 38
cut and paste programming, 78
ingenious integration by layering, 74
futures programming, 46, 235
ghost processing, 109
ghost-in-the-machine API, 73
globally-unified hardware control, 80
grid computing, 109
hardware reuse, 86

interface agents, 89
OPT, 18
point-accurate interoperation, 97
program morphing, 84
responsibility optimisation, 68
software-team scalability, 77
superprogramming, 93
transparent programming environment, 48
ubiquitous computing, 67
universal agent, 90
universal usability, 41
user-generated software, 60
virtual hardware, 64
virtual logic, 58
profiles, 218, 233
program
morphing, 84
self-updating, 83

programming
abstract, 18, 74, 80, 120, 122, 125, 202
aspect-oriented, 25
declarative, 122–126, 128, 253, 329
dynamic systems, 171
general-purpose, 125, 237, 274–275
imperative, 122–126, 128, 329
in Meshoil, 23, 302
protocol for the Mesh, 15, 25
radio, 152
reliability, 114
remote control, 93, 97, 100, 217, 238
responsibility optimisation, 68
reuse
hardware, 86, 97, 217
software, 25, 80, 184, 285, 317, 330
reverse polish notation, 202
robot, 254

safety fan, 128
self-generating software, 84
self-updating program, 83
SIC design, 17
software
design, 44, 58, 77
dynamically extensible, 25
efficiency, 107, 140, 323, 326
reliability, 114
self-generating, 84
INDEX

trading, 62
software-team scalability, 77
solution space, 44–45, 86
spin, 286
  controls, 290, 294
sports car, 171
stack handling, 202
string expression, 321
superprogramming, 93
synchronisation, 217, 240
  between threads, 205
syntax, 278–289
tester, 20, 277, 339
thermometer, 238
timing of information transfer, 83
transparent programming environment, 48
Turing-complete, 123
two fans, 35
ubiquitous computing, 5–6, 67, 271
UIDL, 120–126, 238–253, 275, 329
UIML, 120, 122, 125
unique programming features, 275
universal
  agent, 90
  usability, 5–6, 41, 119–126, 238–253, 271, 275
unreliable software, 115
URC, 126, 238–253
user, 2, 6, 43, 45, 60
  interface description language, see UIDL
  solution space, 44–45, 86
user-generated software, 60, 274
UsiXML, 120, 122

vehicle dynamics, 171
viewer, 20, 28, 41, 285, 342
virtual
  hardware, 64
  logic, 58, 233

Web, 1–2, 15, 91, 117
wiki, 2, 91, 117
Wikipedia, 2, 117
with brick, 290–292

XForms, 120, 122, 126
XIML, 120, 122, 125

XML, 20, 24, 28, 120, 323, 329, 339, 342
XUL, 120, 122, 126

YouTube, 62