Sequential Operator

Last week we saw *action prefixing*. This operator provided a mechanism by which we could say that one action *must* occur first, before we behave as some process.

Imagine we already have two GUIs, G_1 and G_2 say, then if we want to build a new GUI (that allowed the user to interact with GUI G_1, and then GUI G_2), we need to use the sequential operator **;**.

Definition 1 *Let* `P` *and* `Q` *be two processes. The process* `P ; Q` *behaves exactly like the process* `P`*,* ***until*** *it* ***END****s. Once process* `P` *has hit the* `END` *process, then, and only then, do we behave as process* `Q`*.*

Notice, that if process `P` *never* becomes the `END` process, then `P ; Q` behaves exactly the same as `P`!

Modeling GUIs with *hidden* code

Often, GUIs have some code that controls how the user may interact with the GUI. Typically, one might have a GUI that opens up windows based on how the user interacts or *has* interacted with the GUI.

Server side scripting[a] in web applications is an excellent example of such GUI based applications.

Such server side scripts often demand the need for a greater range of control operators than we have seen thus far. For example:

- 2 pages may be displayed in a web application. The server side script uses the state of a radio button to determine which page is shown. This clearly implies the use of some sort of conditional operator.

[a] We can model client side scripting by executing the client side script on the server!

Guarded Actions

Whenever we wish to make an action *conditional* upon the *current machine state*, then we need to use *guarded actions*.

Definition 2 *If* `act` *is an action,* `test` *is a boolean expression and* `P` *is a process, then we may form the new process* `(when test act -> P)`*.*

This process will engage in action `act` *and then behave as process* `P` *if* `test` *is* ***true****. Otherwise (ie.* `test` *is* ***false****) this process can engage in no actions at all with the current machine state.*

Example 1 *Consider a website whose entry page has a radio button and a button on it (presumably all contained within a HTML form). When the button is pressed, a new HTML page is displayed by a server side script. The page displayed depends upon the state of the radio button.*

The following process models this behavior:

```
WebSite = EntryPage[0],
EntryPage[radio: 0..1] = (click -> EntryPage[(radio+1)%2]
          | when (radio == 0) button -> Page1 | when (radio == 1)
Page1 = END,
Page2 = END.
```

Note: *0 means the radio button is not clicked and 1 means it has been clicked.*

Conditional Operator

Sometimes it is more convenient to choose between different processes based upon some boolean condition. For example, we choose between two *existing* web sites based upon the state of some radio button.

In this case, we need a conditional operator that can act on processes directly (guarded actions are typically too low level).

Definition 3 *Let* `P` *and* `Q` *be two processes and* `cond` *be a boolean condition. The process:*

```
if (cond) then P else Q
```

behaves like process `P` *if the condition* `cond` *is true. Otherwise it behaves like process* `Q`*.*

Example 2 *Let* `Site1` *and* `Site2` *be two web sites that someone else has already setup.*

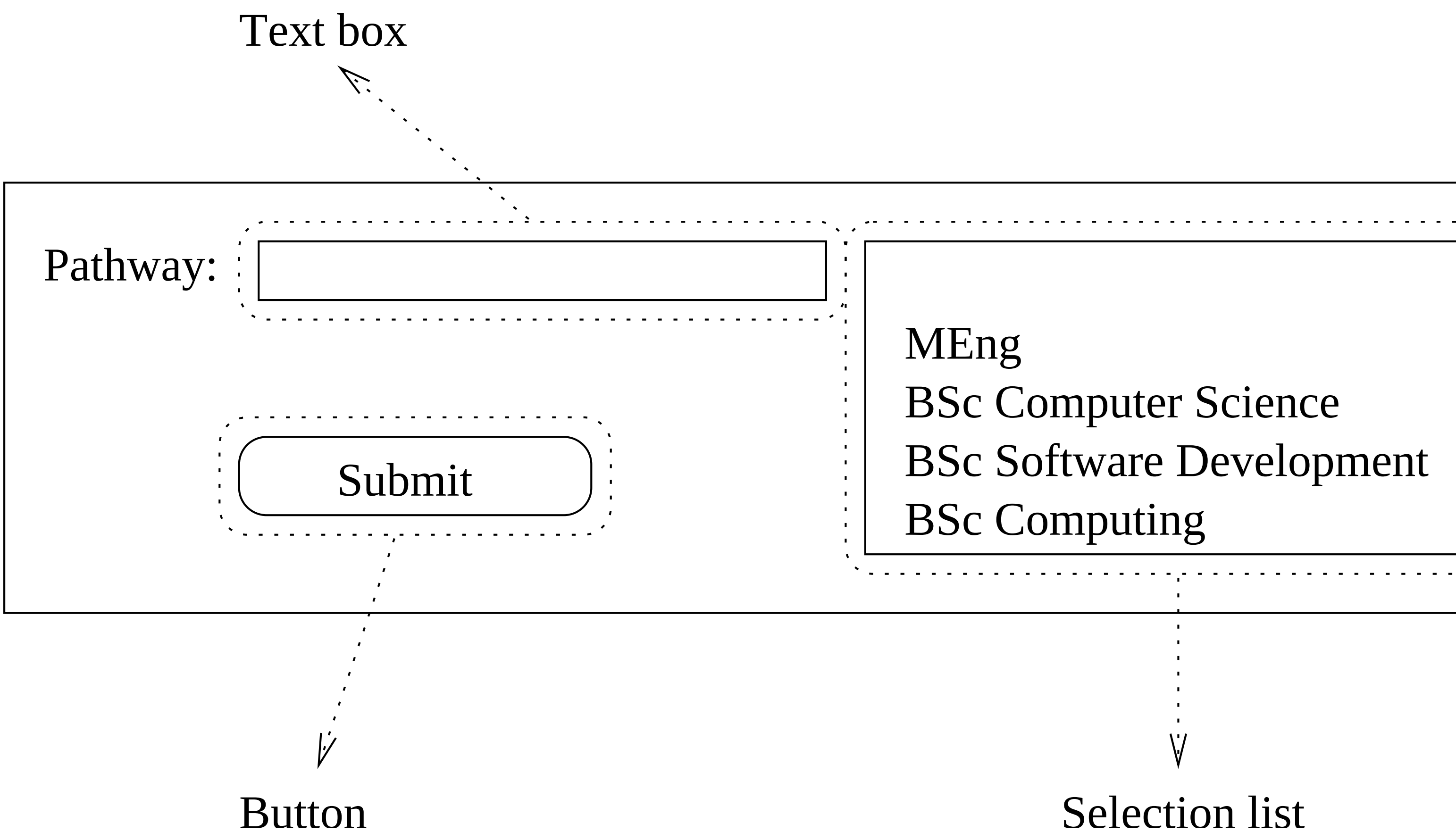
We may choose between these sites based on the radio button example above. This leads us to the following process definition:

```
WebSite = EntryPage[0],
EntryPage[radio: 0..1] = (click -> EntryPage[(radio+1)%2]
         | button -> Choice[radio]),
Choice[radio: 0..1] = (if (radio == 0) then Site1 else Site2).
```

Recall, that 0 means the radio button is not clicked and 1 means it has been clicked.

An Example

Example 3 *Consider the following GUI:*



A user may enter their pathway either directly into the text box, or by selecting one of the predefined pathways from the selection list.
Whenever a member of the selection list is chosen, the appropriate pathway name is written to the text box.

When the user clicks on the submit button, the the contents of the text box is submitted to the server.

How might we model the behavior of such a GUI?

Clearly, we need to maintain information about the current contents of the text box. This implies we need to index our GUI process.

Whenever we select something from the selection list, we need to know what was selected. This implies that the selection action/event should have an index as well.

When the submit button is pressed, we need to pass the current value of the text box to the server process. This implies that the server process should have an index.

This leads us to the following process definition:

```
set Selection = { empty, meng, compsci, softdev, comp }
set Text = ?? // see latter for its definition

GUI = GUI['empty],
GUI[text: Text] = (enter[newtext: Text] -> GUI[newtext]
        | select[selection: Selection] -> GUI[selection]
        | button -> Server[text]),

Server[text: Text] = END.
```

Note: *we have modeled the process* `Server` *as the process* `END` *because, in this instance, the user has no more interaction with the GUI application.*

All we have left to define is the set `Text`.

Recall, from last week, that all processes ***must*** *be finite. As a result, all index sets must therefore be finite.*

Hence, we can not model the set `Text` *using the Java type String!*
Instead, we must partition this infinite *set/type in some way.*

We choose to do this by identifying the members of the selection list explicitly and then grouping everything else within a userText *partition.*

Thus, we may model the set `Text` *as follows:*

```
set Text = { userText, Selection }
```