

## CS637 Labwork Background (ProcessWeb)

Name .....

MSc Service Guest login .....

Version 2003

## Introduction for CS637 MSc Module (2003)

This booklet provides the information about *ProcessWeb* that you will need for the labwork. You will be given separate sheets to be completed and handed in. These will be marked, and along with your background reading report, they will form your individual labwork mark.

Note that there are no specific lectures on *ProcessWeb*. You will learn about it through the pre-course work and the labwork. There will be time for discussion about the labwork if there are any queries.

You do not have to do everything in this booklet to complete the labwork. Remember it is the separate sheets that you hand in that are marked. However, there may be an optional exam question that is based on the labwork as a whole (i.e. this booklet) and the appropriate parts of the course text (i.e. mainly chapter 10).

The Case Study is group exercise which involves applying the OPM method in a realistic context. In contrast the labwork consists of a number of smaller exercises which are to help learn OPM, and give some exposure to the issues involved in developing software to support processes. The particular process technology used is the IPG's *ProcessWeb* system.

For additional information the following are useful starting points:

- The CS637 Web page - <http://www.cs.man.ac.uk/ipg/CS637/>
- The *ProcessWeb* home page - <http://processweb.cs.man.ac.uk/>

## 1.0 What is ProcessWeb?

ProcessWeb is a system which supports collaboration through an executable process model. A ProcessWeb user connects to the system through a web browser.

Web Servers normally treat the users who connect through browsers as anonymous. Everyone gets the same information. In contrast, ProcessWeb identifies its users as individuals, and the web pages which they receive are personal. (This does not preclude using ProcessWeb to deliver the same information to several people if that is what is required.) The web pages received can include forms which enable users to provide input to the process. The web pages are thus both the mechanism of output and input.

A process will normally involve the communication of information between a number of people. For example, if user A supplies some input, not only will user A's web page change reflecting the new state of the process, but the web pages of others involved in the process can also change. In ProcessWeb this is achieved by dynamically creating the web pages on demand. When a user connects to ProcessWeb, or provides input, this is passed onto the process support component (ICL/TeamWARE's ProcessWise Integrator system) which calculates the web page which should be returned, and any updating of other users' pages which is required. Within ProcessWeb it is useful to distinguish between the essential communication between users and the system which updates the state of the process, and the detail of how such communication is presented which is determined by the particular web pages calculated by the process model.

Hypertext Markup Language (HTML) is the text markup language currently most used on the World Wide Web. It is not difficult to learn the basics: there are many books and information on the Web itself. There are proposals for more sophisticated languages such as Extensible Markup Language (XML) but HTML is still the most widely used. The basic principle of HTML is that markup commands tell the browser software the structure of the document and how the content is to be displayed. For example if you want to display a section of text in bold, you surround the text with the bold markup tags `<B>` and `</B>`. The ProcessWeb user interface is in terms of HTML pages. This is illustrated in section 2.0.

### 1.1 ProcessWeb Models

It is a common case that there can be multiple instances of a process. For example, a bank might have an overdraft application process; a new instance of this process is created in response to each new overdraft request. These process instances are distinct: each will have its own state and its own users. ProcessWeb can be used to support multiple instances of many different processes.

Each distinct process in ProcessWeb is described as an executable process model. This model is written in the Process Modelling Language (PML) of ICL/TeamWARE's PWI system, using some extensions specific to ProcessWeb. The model is described in terms of a set of classes. The most significant classes are the role classes which are the major unit of structuring.

A process model instance consists of a number of role instances connected by interactions. There will normally be one "Main" or "boot" role class within a model which is responsible

for setting up the set of connected role instances required. A process model instance is then created by starting an instance of this “Main” or “boot” role class.

The *ProcessWeb* system has a library of models. Any user can create an instance of any of these models from the library. The user who created a model instance does not have any special status except for the ability to delete the model instance.

## 1.2 Interacting with *ProcessWeb*

From a user perspective, interacting with *ProcessWeb* is similar to performing a series of searches, using a search engine such as [www.google.com](http://www.google.com). A user supplies some input by clicking on a button (on link) and eventually receives a web page (HTML) in response.

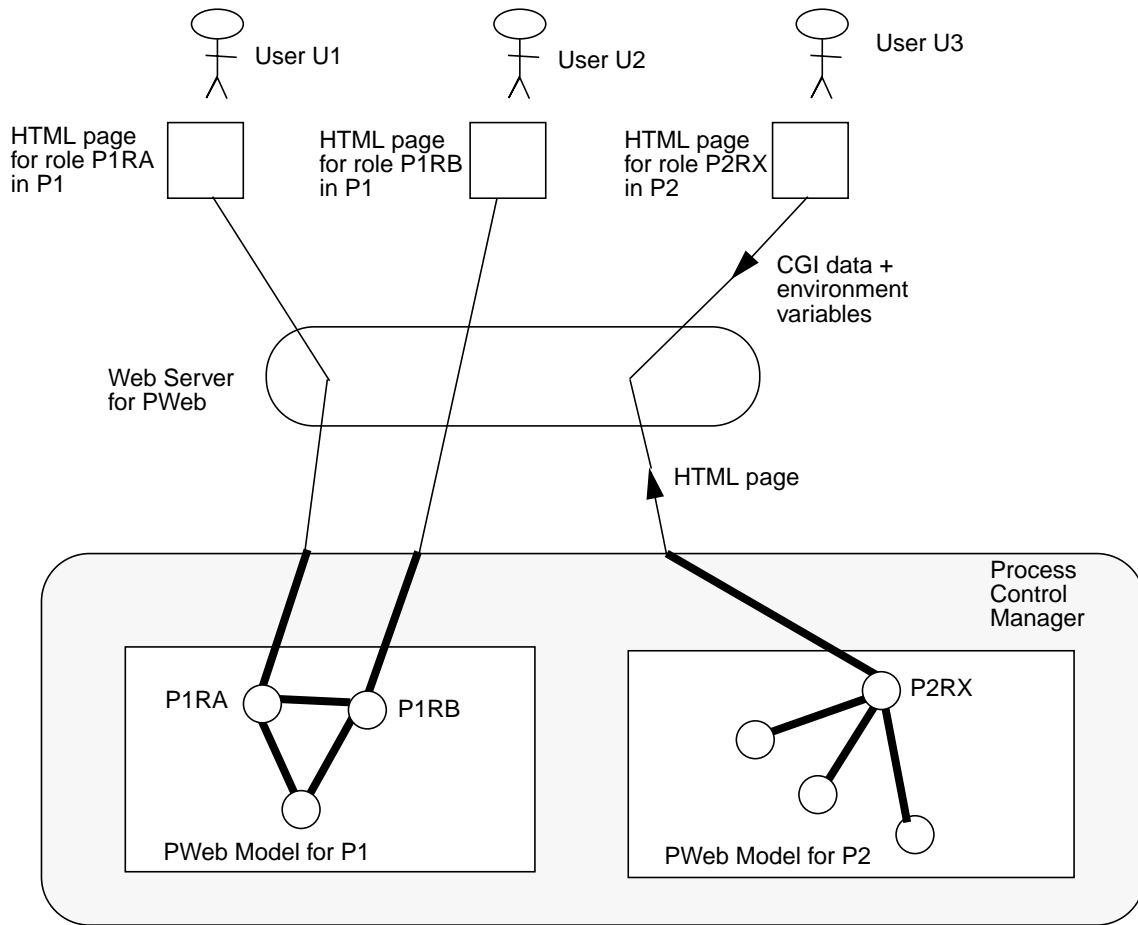
First think of the stages involved in making a search on the net. You:-

1. Locate the page that provides the search facility
2. Input information to be used during search (local to browser)
3. Submit search (data input transferred to server and search program)
4. Receive search result page.

Differences in *ProcessWeb*:

1. You login to *ProcessWeb* as an identified user.
2. The *ProcessWeb* pages you use to provide input to a model are personal to a user role in the model. No two users can have a specific user role at the same time.
3. *ProcessWeb*'s output depends on the state of the model. It is possible for a user's display to change because of another user's input updating the state of the model.

**FIGURE 1. An Overview of the ProcessWeb User Interface Architecture**

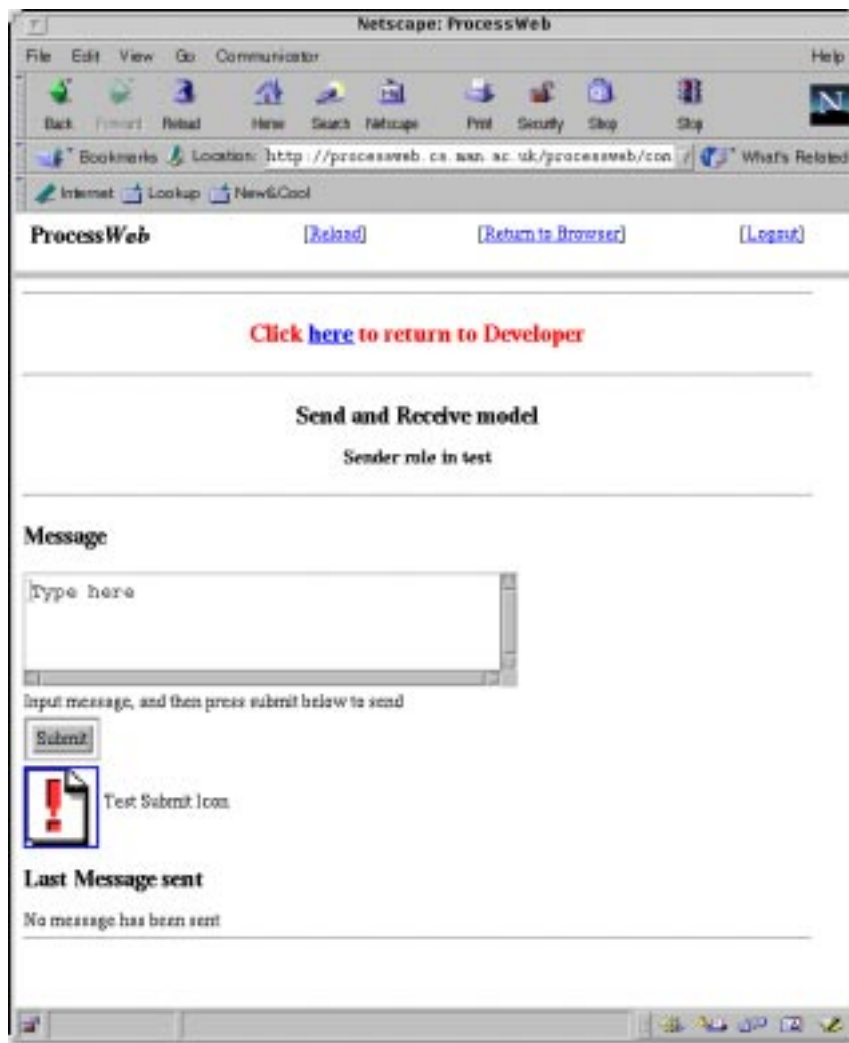


## 2.0 HTML (Briefly)

HTML (Hyper-Text Markup Language) is a standard way of writing web pages. An HTML page can be written as an ASCII file using a text editor (almost all of the IPG web pages are done this way). HTML files can also be created by generating them from a word processor (e.g. Word) or through using HTML editing facilities incorporated in many browsers. To understand a system, like *ProcessWeb*, which has an interface where HTML is generated it is useful to have some knowledge of the relationship between “HTML source” and what is displayed on your browser.

For most web pages you can see the source either by doing “View Page Source” in your browser, or by saving the page to a file (as Source) and viewing with a text editor.

**FIGURE 2. Sender role example - display**



You should be able to identify the corresponding parts of the HTML source (Figure 3) and the browser display (Figure 2).

You will note that Figure 3 does not include the HTML source for “Reload”, “Return to

Browser” and “Logout”. These are in a separate HTML frame. The complete display consists of this standard header frame along with the HTML which is specific to the role in the model instance.

**FIGURE 3. Sender role example - HTML source**

```

<HTML>
<HEAD>
<BASE HREF="http://processweb.cs.man.ac.uk/processweb/connect">
</HEAD>

<body bgcolor="#FFFFFF" vlink="#0000FF">
<hr>
<h3 align=center><font color=red>Click <a href="/processweb/connect?operation=jumpToRole&roleID=R253">here</a> to return to Developer</font></h3>
<hr>
<h3 align=center>Send and Receive model</h3>

<h4 align=center>Sender role in test</h4>
<hr>

<h3>Message</h3>

<form method=POST action="/processweb/connect">
<textarea name="text" rows=4 cols=40>
Type here
</textarea>
<br>
Input message, and then press submit below to send
<br>
<input type="submit" value="Submit">
<br>
<input type="IMAGE" src="http://www.cs.man.ac.uk/icons/alert.red.gif"
alt="Test Image Submit" height="60" align="middle" name="Submit"> Test Submit Icon
</form>

<h3>Last Message sent</h3>
No message has been sent
<hr>
</HTML>

```

(If you are having problems relating Figure 2 and Figure 3, look at the examples in Appendix C on page 35.

### 3.0 Users, Models and Roles

ProcessWeb supports many users. Each user may be involved in zero, one or many models. Each model will have a number of user roles. Each user role is a distinct view on the process. A user role may either be Unbound, i.e it is not currently assigned to any user, or it can be bound to a single user. ProcessWeb takes a very egalitarian approach: any user can bind to an unbound role. This binding between users and roles is described in more detail on the ProcessWeb home page (<http://processweb.cs.man.ac.uk>) in the section “Creating and Using models”.

The following table gives an example sequence of mappings between two ProcessWeb users, user1 and user2, and two user roles, Sender and Receiver.

The first unsuccessful action is because user2 cannot bind to Receiver while user1 is already bound to Receiver. The binding of user1 to Sender cannot be relinquished while user1 is connected to Sender, hence the second unsuccessful action. Check that you see the reason for the third unsuccessful action.

Action	Success	Sender Binding	Receiver Binding
user1 login	Y	-	-
user1 create model (Send Receive)	Y	Unbound	Unbound
user2 login	Y	Unbound	Unbound
user1 request binding Sender	Y	Bound user1	Unbound
user1 request binding Receiver	Y	Bound user1	Bound user1
user2 request binding Receiver	N	Bound user1	Bound user1
user1 connect Sender	Y	Bound user1	Bound user1
user1 relinquish Sender	N	Bound user1	Bound user1
user1 logout	Y	Bound user1	Bound user1
user2 request binding Receiver	N	Bound user1	Bound user1
user1 login	Y	Bound user1	Bound user1
user1 relinquish Receiver	Y	Bound user1	Unbound
user2 request binding Receiver	Y	Bound user1	Bound user2
user2 connect Receiver	Y	Bound user1	Bound user2
user1 connect Sender	Y	Bound user1	Bound user2

The following pairs can be considered inverse actions:

- login and logout
- bind and relinquish
- connect and return to browser

However they are not exact inverses. The login action always returns a user’s ProcessWeb browser page, while a user can logout from any page. A user can therefore achieve the effect of return to browser by a logout followed by a login.

## 4.0 Planning Example - ( Labwork Part 2 )

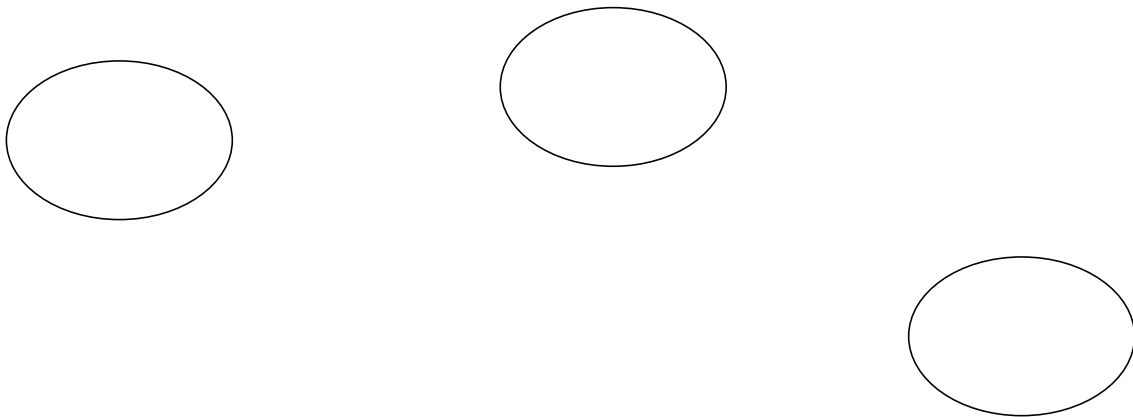
Based on the description below, complete the OPM models for this example.

Complete your answers on the separate labwork 2 sheet. The gaps below are included in case you wish to sketch out your system and goal models first.

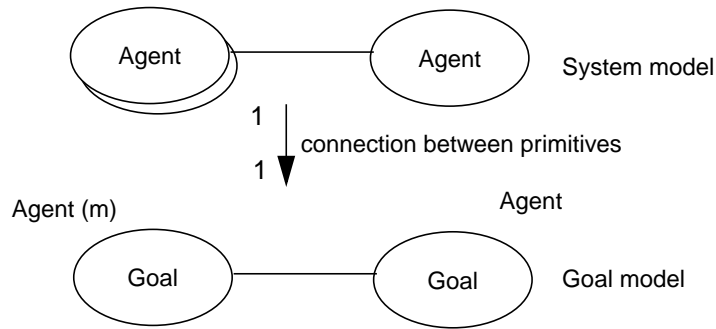
In many circumstances, building alterations require the permission of the local council. Local councils have an obligation to deal with such planning applications in a reasonable timeframe, but also have to ensure that appropriate laws are not broken. A simplified version of the process is as follows.

When a member of the public (the client) wants to make a planning application, they obtain an application form from the council clerk. After completing the application form, they return it to the clerk. The clerk examines the form to see if it can be approved without consulting the planning committee. If it can the clerk responds to the client directly. Otherwise it is referred to the committee. When the clerk receives the committee's response, this is passed onto the client.

### 4.1 Planning Example - System Model

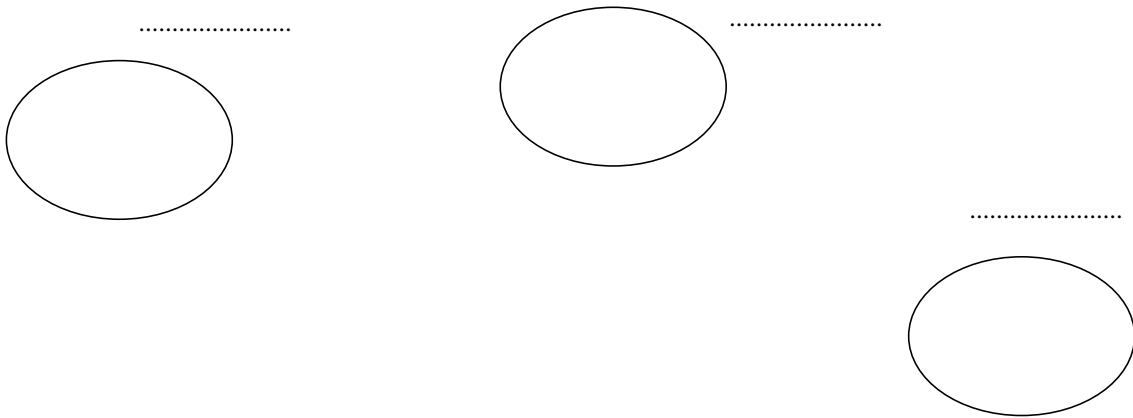


The conversion from a system model to a goal model



This is the revised version of Fig 7.8 on page 113 (in second printing)

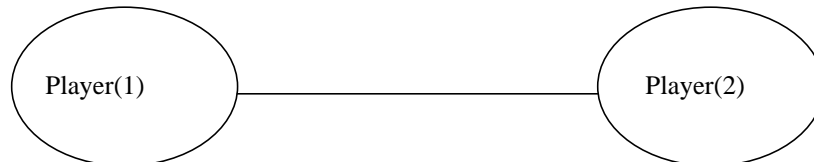
## 4.2 Planning Example - Goal Model



## 5.0 ProcessWeb Scissors model (Labwork Part 2)

Scissors is one of the first simple models that was developed for *ProcessWeb*. In the labwork you will be asked to look at the Scissors model and recreate part of its original design. This will help you to understand the relationship between an enacting process model and its static process description.

A rough sketch of the Scissors model is::



The lowest level description of the Scissors model is its PML code. The PML code for Scissors consists of three new PML classes

- **CalculateResult** - this is a PML action class. It takes the selection of each player, represented as an Integer, and returns a Boolean (true - win, false - lose, nil - draw)
- **Player** - this is a PML role class. It performs input and output with the user using the standard *ProcessWeb* facilities. It communicates with the other player through its resources (local data variables) **giveMine** (giveport Int) and **takeYours** (takeport Int).

For a description of how PML role instances use giveports and takeports to communicate through interactions see section 10.8.1 in “Business Information Systems: a process approach” or section A.1 and section A.5 in this document.

- **ScissorsBoot** - this is the PML role class used to create an instance of the Scissors model. When an instance of the **ScissorsBoot** class is created it creates the network of role instances and interactions for a scissors model. It creates the interactions, and the **UserRoles**. It then creates two instances of class **Player** (p1 and p2) and supplies each instance with appropriate initialisation parameters. For example, for one integer interaction the **giveport** will be given as the initialisation value of **giveMine** in p1, and the **takeport** will be given as the initialisation value of **takeYours** in p2. This interaction now links the two **Player** role instances. When p1 send a message using **giveMine**, it will be received by p2 using **takeYours**.

The Scissors model makes use of several standard *ProcessWeb* classes: **CreateUserRole**, **HKClient2** (the superclass of **Player**), **ModelUserRecord**, **SendToUser**, **WWW\_file**,

It also uses built-in PML action classes: **ConvertFromString**, **Duplicate**, **GiveCopy**, **NewInteraction**, **StartRole**, **Take**.

And built-in PML primitive entity classes: **Bool**, **Int**, and **String**. There is also mention of the PML class **Any**: this is the superclass of all PML classes.

## 6.0 Send Receive - OPM

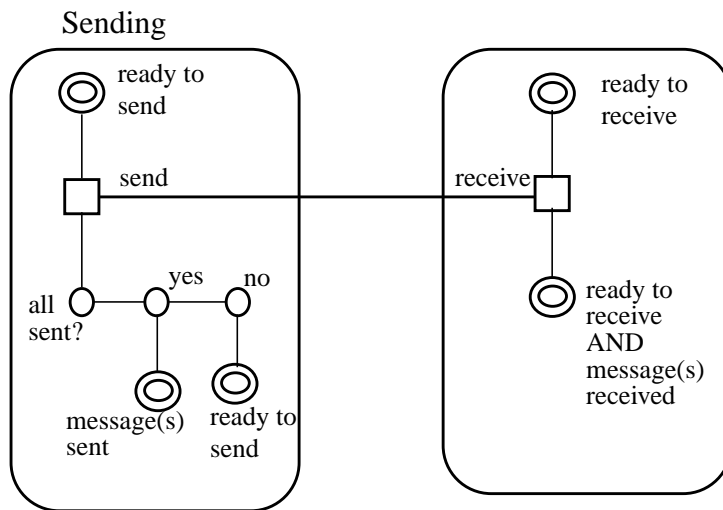
Send Receive is a very simple *ProcessWeb* model which you can experiment with to see how a *ProcessWeb* model is developed. With OPM we can see just how simple send receive is. System model:



Goal model:



Method model



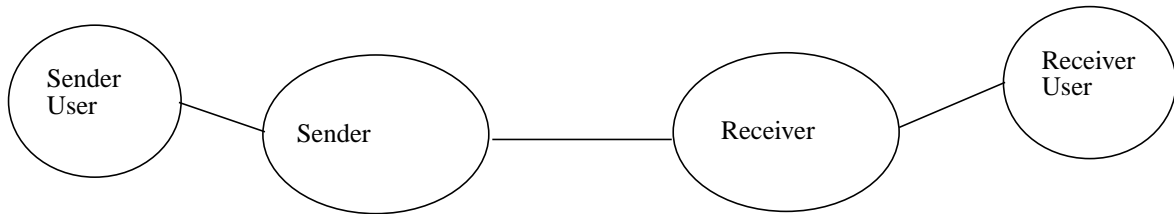
If you have not already done so, create an instance of the Send Receive *ProcessWeb* model and satisfy yourself that this is a reasonable OPM model for the process.

However, this is a simplification of what happens in *ProcessWeb*. In *ProcessWeb*, for each role that a user can interact with, e.g. Sender, there is a “user role”, e.g. Sender User. The user role stores a copy of the HTML which represents the current state of its role. In this way the user roles provide an indirection which means that the model roles do not have to worry about

users logging in, logging out, browsing around the roles, etc. (Consider the table in section 3.0, all the actions could be done without any message actually being sent from sender to receiver.)

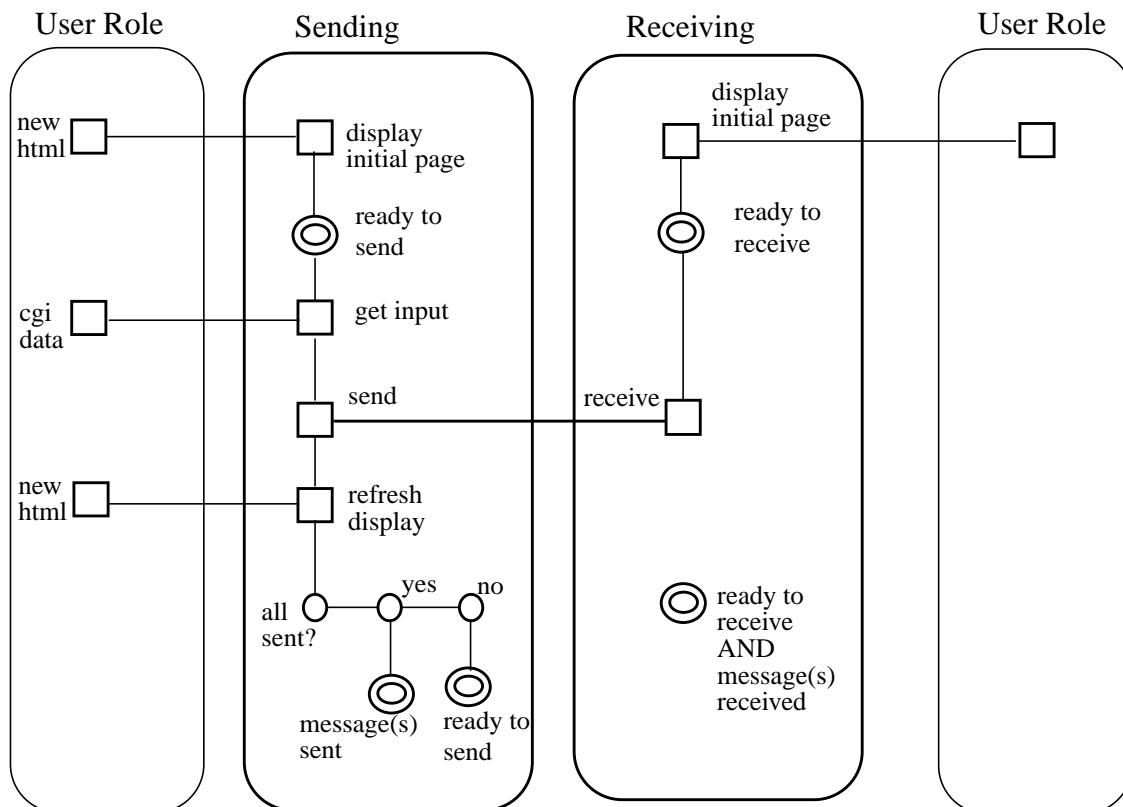
We can adapt OPM to illustrate what happens in more detail.

System model: (The User Roles are not strictly separate agents)



Method model (The User Role is a standard facility provided by *ProcessWeb* so modelling the internal states of Sender user and Receive user is not appropriate.)

Complete the Receiving role and its corresponding user role. Remember that the Receiving role does not offer its user any chance of inputting data; it just displays the data that it receives from Sending



## 7.0 Send Receive Development - ProcessWeb (Labwork Part 3)

The Send Receive model is very simple. This exercise is to illustrate how ProcessWeb models are developed through making simple changes to a send receive model. The model which we will use is Send Receive Develop.

### 7.1 Starting and killing a model

Start an instance of the Send Receive Develop model. Connect to the single role, and then use the Start option to create a sender and receiver within this Send Receive Develop model instance. The role instances can be deleted, by selecting them in the role instances list and clicking on the Kill option.

### 7.2 Example PML outline

The PML code for the **Sender** role is shown in Figure 4.

The **Sender** class is a subclass of **HKClient3**. This class is part of the ProcessWeb system and includes some standard resources which are used by several model roles.

The **Sender** class declares two local data items in the resources section: **messGP** is the variable whose value will be the giveport of the interaction which connects the Sender and Receiver, **m** is the string variable used to store the message being sent.

The **Sender** class has three action parts: **init**, **receive\_selection**, and **sendmes**. Each action part has a label, a body (a sequence of statements enclosed in {}), and a guard, a boolean expression which follows the keyword **when**. There is a scheduler which chooses which action part to run according to the value of the guards. Once an action part has been selected, all the statements in the body are performed, and then the scheduler re-evaluates the guards to choose the next action part.

The **init** action part creates the initial HTML page and uses **SendToUser** to send it to the user role. If a user is logged in and connected to this role, then the page will be sent to their browser. (See <http://processweb.cs.man.ac.uk/doc> for the full details on **SendToUser** and other ProcessWeb facilities) The **init** action part is only run once.

The **receive\_selection** action part is run whenever CGI data is received from the user role. The guard “**when userRolePorts ~= nil**” simply checks that this role has a valid connection with a user role. This action part will be run whenever there is a data item in the interaction indicated by the PML pre-defined action **Take**, which is the first item in the body. When this action part is run, the first data item from the interaction is placed in the role’s local variable **cgi\_data**, and the variable **doNext** is set to the value **SendMessage**.

The **sendmes** action part sends a message to the **Receiver** role. It will be run when **doNext** is set to the value **SendMessage**. To send a message it uses the pre-defined action **Give** and its giveport value which identifies the interaction which connects the sender and receiver role instances. The action part also updates the HTML page for the sender role. Finally it sets

Sender isa HKClient3 with

resources

! these are additional to resources inherited from HKClient3

messGP : giveport String

m : String

actions

init: {

if wwwFile = nil then

wwwFile := configuration.processWebHome ++ configuration.templateDirectory ++ 'examples/send1.htm';

end if;

parseTable('\$msgsent') := 'No message has been sent';

parseTable('\$modelName') := modelName; ! <modelName> should have been set by Developer

SendToUser(

gram=WWW\_file( file\_name=wwwFile,  
replacement\_table=parseTable),  
connection=userRolePorts);

} when

init = nil & ! i.e. only do the <init> action once

configuration ~= nil ! <configuration> is set by an action in HKClient3

!!!! receive and interpret data coming from cgi process on web server

receive\_selection: {

Take(gram=cgi\_data, interaction=userRolePorts.userTakeport);

if cgi\_data = nil then

cgi\_data := tableof String(); ! for ease of parsing

end if;

doNext := 'SendMessage'

}

when userRolePorts ~= nil

! sendmes sends current message to the receiver

sendmes: {

! copy input to local resource variable m and send using giveport meeGP

m := cgi\_data('text');

GiveCopy( interaction = messGP, gram = m );

! update display with input received

parseTable('\$msgsent'):= m;

SendToUser(

gram=WWW\_file(  
file\_name=wwwFile,  
replacement\_table=parseTable),  
connection=userRolePorts);

! update doNext to avoid looping and repeatedly sending same message

doNext := 'Idle'

}

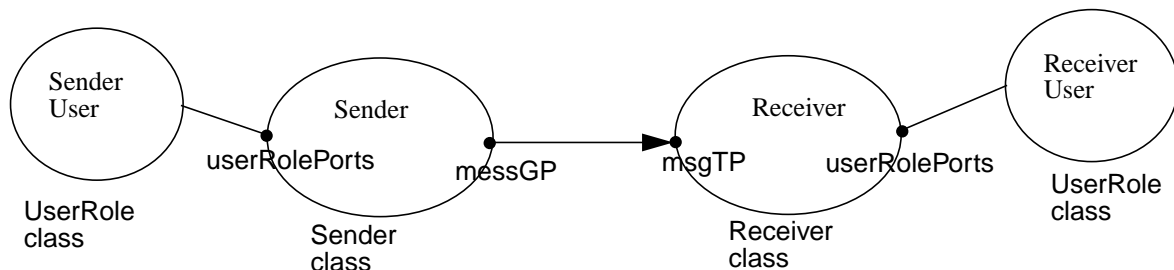
when doNext = 'SendMessage'

end with ! role Sender

## FIGURE 4. PML Code for Sender

`doNext to Idle`. This indicates that there is nothing for this role to do until another message is received from the user role.

You should be able to relate the **Sender PML** code in Figure 4 with the detailed RAD in section 6.0. The following diagram should help.



### 7.3 Revising and Re-compiling PML

Note that in PML you should always use the quote (`'`), not the backquote (```), symbol to delimit strings. (The different appearance of opening and closing quotes in the PML in this document is due to the default smart-quote setting for the wordprocessor used to write it.)

Using the Compile Sender Class Definition option, in the action part `sendmes`, change the line:

```
GiveCopy( interaction = messGP, gram = m );
```

to

```
GiveCopy( interaction = messGP, gram = m ++ 'Testing');
```

(Testing should be enclosed in single quotes, as elsewhere in the example. The operator `++` is the PML string concatenation operator)

Start a new instance of send receive and observe the difference.

### 7.4 Revising the HTML Template

In the code of **Sender**, the HTML is sent to the user using `SendToUser`. The data sent includes two parts: a file name of an HTML template file and a table which maps between markers in the HTML template file and strings to substitute for the markers.

The basic scheme works as follows:

1. The role instance send to its user role the name of a template file and a substitution table.
2. The user role reads in the template file. When it comes across a string in the template file that matches a key in the substitution table, then it replaces this with the corresponding value from the substitution table.

3. The user role adds the standard HTML headers, and sends the HTML to the user. (If there is no user currently bound and connected to this role, then the HTML is saved. It will be sent whenever the user does connect to this role.)

Some *ProcessWeb* models use another scheme where the role instance manipulates a number of HTML fragments, as PML strings. When it wants to send output to the user it concatenates (++) several of these HTML fragments together, and send these to the user role. The user role then only has to do part 3 above. There are strengths and weaknesses to both approaches. The templates make it clear that the template HTML is about the presentation of the model not its essential content (You could have the same PML code with different templates), but the developer has the extra complication of understanding the template substitution. The concatenation of HTML fragments is simpler to understand, but tends to give PML code that is longer and more difficult to read.

### FIGURE 5. HTML Template for Sender

```
<h3 align=center>Send and Receive model</h3>
<h4 align=center>Sender role in $modelName</h4>
<hr>
<h3>Message</h3>
<form method=POST action="$cgiScript">
<textarea name="text" rows=4 cols=40>
Type here
</textarea>
<br>
Input message, and then press submit below to send
<br>
<input type="submit" value="Submit">
<br>
<input type="IMAGE" src="http://www.cs.man.ac.uk/icons/alert.red.gif"
alt="Test Image Submit" height="60" align="middle" name="Submit"> Test Submit Icon
</form>

<h3>Last Message sent</h3>
$msgsent
<hr>
```

By comparing the HTML template (Figure 5) with the HTML source after the template markers have been replaced with the corresponding strings (Figure 3) and the PML code (Figure 4), you be able to identify the template markers in this example, and the corresponding values.

**template marker**

**string substituted for marker**

Note that in Figure 3, part of the HTML source is a standard header which is added to the template to create the HTML page. In addition, one template marker is a *ProcessWeb* standard one which is not included in the table which is sent from the sender role instance.

In most models the HTML templates are standard and shared by all the instances of the model. The Send Receive Develop model makes its own local copy of the HTML page for Sender so that each model user can experiment with changing their own copy. To do this you need to download a copy of the HTML template, edit it, and then upload the new version.

You should now be able to make some small improvements to the HTML template of Sender.

## 8.0 Send Receive Change on the fly - *ProcessWeb* optional exercise

The Send Receive Develop model (section 7.0) can be viewed as a simple meta-process which supports changes to the **Sender** class definition, and running of test Send Receive models. After a new definition of the **Sender** class was compiled successfully, new instances of the model would use the new definition.

With an additional facility it is possible to have the new **Sender** class definition applied to already existing instances of the Sender role.

- Create an instance of the Evolve SndRcv model from the library  
This will look very similar to the Send Receive Develop model. There is just one extra option, Modify.
- Start an instance of the Send Receive model  
Send a message from the Sender to Receiver.
- Compile the **Sender** class definition, making the same change as before. (section 7.3)  
Send another message to check that the current role instance has not been changed.
- Modify the existing Sender role instance. You need to select both Sender in the Role Classes list and the Sender instance in the role instances list.  
Send another message - this should show that the role instance has been changed
- Repeat this making another change to the PML for **Sender**.

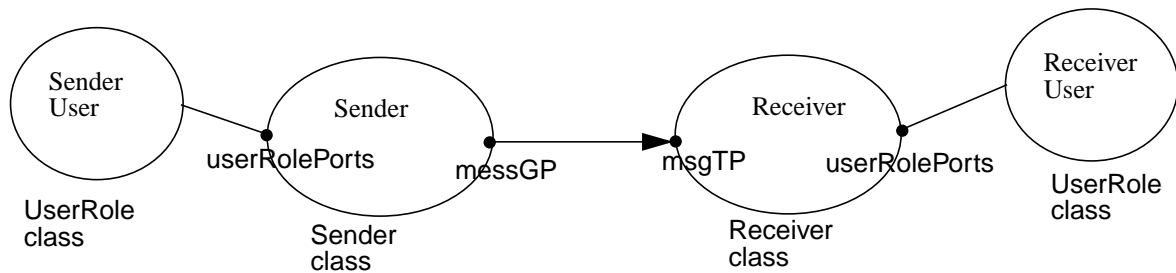
The Evolve SndRcv model is might be considered a meta-process for the Send Receive model. You should note however that there is no provision for feedback from the enacting Send Receive instance to the Evolve SndRcv model.

## 9.0 Role Interaction Networks (labwork Part 3)

One way of getting a top-level design view of a *ProcessWeb* model is through a role-interaction network diagram.

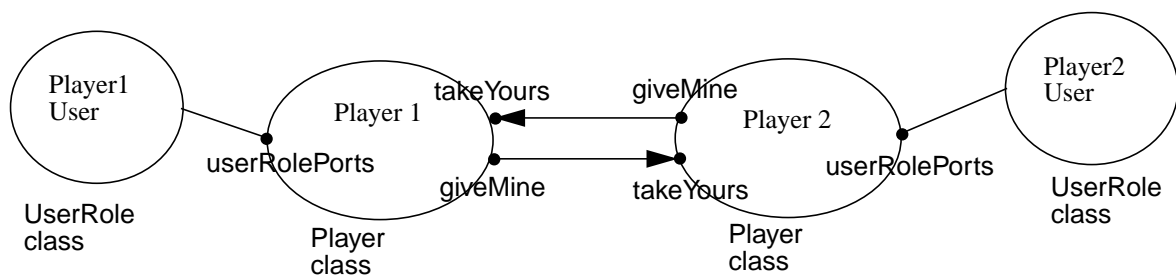
### 9.1 Role Interaction Network - Send Receive

You might want to compare this with the RAD on page 13.



You might think that the port names `messGP` and `msgTP` are not really necessary here - it is enough to show that there is an interaction linking the instance of class **Sender** and the instance of class **Receiver**. However, there are times when names are essential for clarification as the next example shows.

### 9.2 Role Interaction Network - Scissors



The reset interactions, which are used when a player selects 'play again' have been omitted

The User Roles, and the `userRolePorts` connection, could be omitted since these are standard in *ProcessWeb* models. The input and output connection with a User Role, and hence with a user, is achieved by `userRolePorts` being an entity including one `giveport` for output and one `takeport` for input.

We can extend the role interaction network by giving the interaction type. For example:  
`giveMine : giveport Int` and `takeYours : takeport Int`

## 10.0 Planning Example - Evolution

Consider the planning example:

Give examples of the information which might be gathered to make sure that the process was running smoothly.

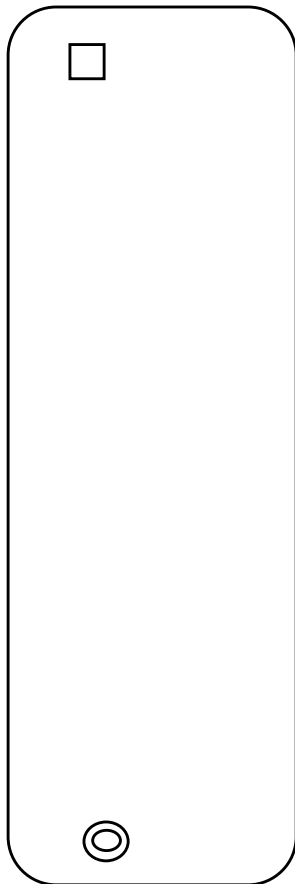
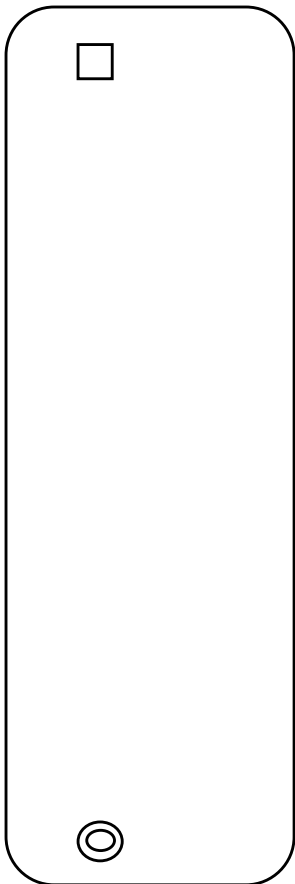
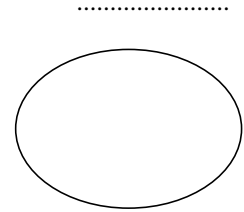
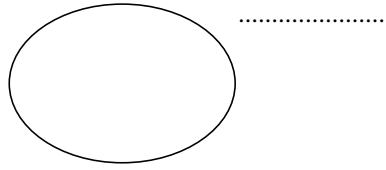
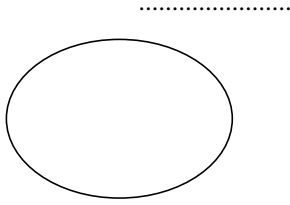
Someone has proposed that there should be a specific process for re-applications. If a planning application is essentially acceptable but there are a small number of issues to be addressed, then the client could be invited to re-apply. The outstanding issues from the initial application could be recorded, say in a database, for future reference. On receiving a re-application, the clerk could then check whether all the outstanding issues had been addressed properly without having to check the complete application. If the planning committee was involved in the original application, then the clerk would advise them of the outcome of the re-application.

This re-application facility could be modelled by extending the existing model. Another approach is to develop a second goal model and method model for the re-application process.

### 10.1 Planning Re-application Example (Labwork Part 3)

Based on the description above, complete the OPM models for this example. The system model is unchanged so that does not need to be repeated.

Complete your answers on the separate labwork 3 sheet. The gaps below are included in case you wish to sketch out your models first.



The related models can be related using a table:

<b>Model</b>	<b>No.</b>	<b>Description</b>
<b>System (<i>what</i>)</b>		Interaction of Client, Clerk, and Planning committee
<b>Goal (<i>why</i>)</b>	1	Evaluating Planning Proposals
<b>Method (<i>how</i>)</b>	1	Evaluating Planning Proposals
<b>Goal (<i>why</i>)</b>	2	Evaluating Planning Re-applications
<b>Method (<i>how</i>)</b>	1	Evaluation Planning Re-applications

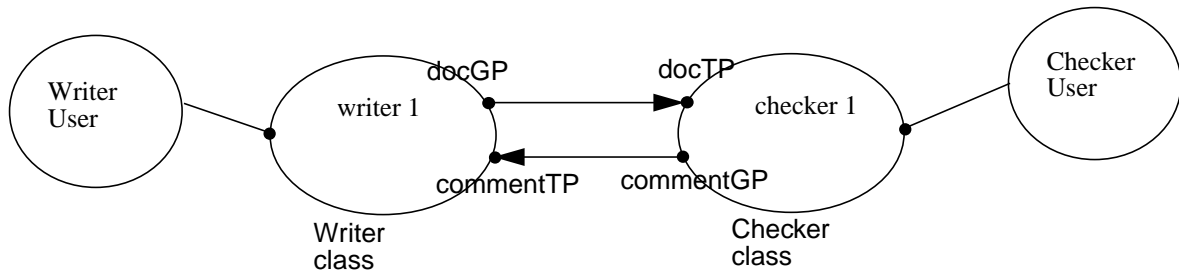
## 10.2 Planning Example - Technology Support

Encouraged by an e-government initiative, the council are considering whether the planning application process could be improved. There are a number of issues.

- Use technology within the council to make the process more efficient. This would aim for quicker processing of applications, quicker responses to client queries, and the timely introduction of applying new guidelines to applications.
- Provide web facilities so that applications could be submitted electronically by clients.
- The planning application process is related to other council processes. For example, the council collects a community charge from householders which is based on the value of their property. After planning is approved and a property improved, then the council may need to re-assess the value. This raises the issue of whether any new system to support this process should be stand-alone, or should integrate with other council systems, existing or planned.

What are your thoughts, comments, observations:

### 11.0 Role Interaction Network - Writer Checker (labwork Part 3)



docGP : giveport Document  
commentTP : takeport Comment

docTP : takeport Document  
commentGP : giveport Comment

## Appendix A Some PML (and ProcessWise Integrator) Background

At the heart of ProcessWise Integrator is the Process Control Manager (PCM), or process engine, which acts as a central server. It interprets the process descriptions which are written in PML. In addition it also handles communication with the clients (in the case of *ProcessWeb* these include the cgi programs started by the server on behalf of the Web browsers).

The PCM has been implemented using a persistent store technology. This is very important. Business processes may last a long time, even continue indefinitely, and do not suddenly disappear just because a powercut has caused a computer to go down. In addition, it is common for processes which last for a long time to evolve during their lifetime. The process descriptions and their states are held in a persistent store. This means that the PCM can be stopped for tasks such as routine maintenance and then subsequently restarted with the models in the same state.

PML is an object-oriented language, which has been designed to provide the expressive power to model potentially complex, and dynamically changeable processes involving multiple people and software applications.

Processes typically have many activities in progress at the same time. A related group of activities is represented in PML as a role. Each role is capable of responding to a range of messages. Interactions provide the mechanism for delivering these messages, which may originate from other roles or from external sources such as people or applications.

A process is modelled as a network of roles and interactions using role/interaction definitions. These are interpreted by the system, creating a dynamic environment in which the participants (both human and computer based) take part in the process.

### A.1 Concepts of PML

PML is an object based language. The basic building block of an application are program objects called *roles*, and a means of communication between roles which are called *interactions*.

Some *roles* are programmed by the application writer. Such a role is an object that includes code (**actions**) and data (**resources**). It receives messages passed from other roles and from outside the system, processes them, and optionally send back one or more replies. It encapsulates its data, and external access to its data is allowed only through special mechanisms intended for application modification or diagnostics. Some roles are standard and may not be modified by the application.

*Interactions* are uni-directional asynchronous communication channels. If one role wishes to communicate with another it requires two interactions, one to send and one to receive. Interactions are primarily a means of sending values not addresses, which helps to preserve the integrity of a role. It is possible to send roles and interactions through an interaction. In the case of a role a reference is sent, but a reference to a role is useful only for program control and diagnostic purposes. In the case of an interaction it is not the interaction as a whole which is sent, but one end of it, the **giveport** or **takeport**. In this case integrity is maintained by removing the port from the role which is sending it.

PML is a class based language with single inheritance. There are three class hierarchies: **Role**, **Entity** and **Action**. A class may be based on the definition of another class in which case it inherits all its code. It may add to the class definition, or modify it, subject to certain constraints. With all three classes, inheritance is a means of reusing code. With entities subtyping is also supported.

A class defines a template for the creation of an instance of the object. In the case of a role, an instance is a program object which interacts with other roles. In the case of an entity an instance is an item of data. An **Action** class is the equivalent of a subroutine in other languages, and an instance is a particular call of an action. This instance has no life after the call has terminated.

## A.2 Roles

A role class definition consists of a number of properties. The most important are **resources** properties and **actions** properties. There are also **initially**, **always**, and **termconds** properties. There is an inheritance hierarchy for roles.

Resource properties define the local data for a role. There is no global data in PML. All resource properties are introduced with `<name> : Type`, and optionally an initial value can be supplied. The Type can either be a PML pre-defined type, an Entity type name, (or a constructor and Type). A role can redefine a resource property defined in a superclass.

The resources properties which use the constructor **giveport** and **takeport** are of particular interest as these are the only way one role can affect another (by giving and receiving messages).

Action properties define what a role instance does when it executes. This includes: changing the values of its local data, giving a data value to an interaction **giveport**, receiving a data value from an interaction **takeport**.

An action property consists of `<name>: { <command list> } when <expression>`. The expression is the trigger which determines when the action property is selected for executions (see Scheduler section A.3). The command list defines what the action property does. (The { } brackets can be omitted if there is only one command.) The name identifies the action property: it determines if it overrides a superclass definition, and introduces a **action/part** variable which is initially nil and set to 'nonnil' when the action property has been executed.

It is not unusual to have an initialisation action property which is only to be executed when the role instance has just been created. These often have the form:

```
init: {<command list> } when init = nil ! the name init is not significant
```

The command list can include:

- assignments to resource properties
- calls of PML actions (including PML actions), in which case parameters, including result parameters are given, in any order by `<parameter name> = <value>`
- conditional (if), or iteration (**while**, **forevery**, **for**) statements

(There are also some pre-defined actions which allow the manipulation of roles: `StartRole`, `FreezeRole`, `UnFreezeRole`, `ExtractRoleData`, `InsertRoleData`, `BehaveAs`, `GetRoleState`, `GetRoleClasses`. Apart from `StartRole`, which is used to create role instances, these are not used unless the dynamic change of role instances is required.)

Note there is no facility in PML for introducing a variable (resource property) which is local to an action property. All a role's local data is collected together in the resources section at the start of the role.

### A.3 Scheduler

The execution of a role's action property, or an action's parts property, is governed primarily by triggers, the boolean expressions following the `when` keyword. An absent trigger is treated as one which always evaluates to true. Only actions or parts properties whose triggers evaluate to true are candidates for scheduling.

Once the candidate properties have been determined one is chosen using the following priorities:

1. A property which invokes a `Give` or `GiveCopy` as its first command, and the value bound to the interaction property of the `Give` or `GiveCopy` is not nil
2. A property which invokes a `Take` or `ListenMultiInteractions` as its first command, and there is at least one interaction which contains data which can be taken
3. A property which does not invoke `Give`, `Givecopy`, `Take` or `ListenMultiInteractions` as its first command

If there is no property which satisfies these conditions then the role, or action, will wait in a quiescent state until a value is placed in an interaction so that a property will now satisfy criteria 2 above (or until another role performs a dynamic change on this role).

A role is single threaded; once one actions or parts property has been chosen, it will be completed and then all the triggers will be re-evaluated.

There is no concept of fairness in the scheduler, the same property may be chosen repeatedly if its trigger evaluates to true. The only way to guarantee that a property will be chosen is to ensure that all other triggers evaluate to false.

### A.4 Actions

In PML an Action class can be defined to represent a computation which may be called many times with different parameters. An Action class should not be confused with an action property of a role. There are a large number of pre-defined PML Action classes.

Like roles, actions are defined in terms of their properties which fall into a number of categories.

- `in` and `out` define the parameters - `in` parameters are call by value, while `out` parameters are 'call by value result' (the final value is copied back to the calling role or action). An action only changes the state of the calling role or action through its `out` parameters.

- **resources** the actions local data
- **parts** the statements to be executed (like actions for a role)
- **preconds** a boolean expression which should be true when action is invoked
- **postconds** a boolean expression which should be true on completion of an action

Actions are similar to procedures. However because there is no global data in PML, the only values which an **Action** can access are its parameters and local data (**resources**).

### A.5 Interactions

An *interaction* is comprised of a data queue, and a set of ports which give access to the queue. An interaction is like a pipe which connects role instances together allowing them to cooperate. Interactions are typed.

Only one role can hold an interaction port. If one is given from one role to another, the value in the giving role will be set to nil.

The pre-defined PML actions for interactions are:

- **NewInteraction( giver = gp, taker = tp )** - creates a new interaction and binds the ports to resource properties **gp** and **tp**. After this action call the role could use **gp** and **tp** to send messages to itself. It is usual that once an interaction is created, one or both of its ports are passed to another role, for example in the bindings table on **StartRole**.
- **Give( interaction = gp, gram = <...> )** which appends the **gram** value to the queue and assigns nil to **gram**. **GiveCopy** appends a copy of the **gram** value to the queue leaving it unchanged
- **Take( interaction = tp, gram = var )** extracts data from the queue and updates the value of **var** to be the extracted data. There is **ListenMultiInteractions( collection = <collof takeport>, gram = var, takeCollIndex=<Int>)** which will take data from any of a number of interactions.
- **Duplicate( original = gp, duplicate = gp2 )** creates another giveport connected to the same queue. This gives interactions their many to one capability.
- **Disconnect( connection = gp )** closes a giveport indicating that it is no longer required. When a role terminates its giveports will be automatically disconnected
- **QueryInteraction( taker = tp, hasGram = <Bool>)** or **QueryInteraction( giver = gp, hasGram = <Bool>)** checks for data in a interaction's queue

A dynamic network of interactions can be created by including giveport and takeport values in the data communicated through interactions. Interaction ports can also be transferred from one role to another using the bindings to **StartRole**, or **BehaveAs**.

### A.6 Entities

*Entities* are the data on which roles and their actions operate. There are four primitive entity classes: **Bool**, **Int**, **Real** and **String**. There is also a pre-defined class **Entity** which is the superclass of all entity classes.

There is an inheritance hierarchy of entity classes. New classes may be subclasses of **Entity**, or subclasses of a previously defined entity class. These entity classes are similar to record structures. A number of parts properties can be defined. These are similar to the resources properties in a role or action class and can be initialised in the same way.

Entity properties are selected using the 'dot' notation. (<entityname>.<propertyname>)  
One of the commonest errors in PML is attempting to read or change a property of an entity with value nil. This gives a run-time error.

### A.7 Constructors

There are two constructors for aggregating values in PML:

- **collof** for collections - A collection is an ordered set of values of a type T (or subtype of T). The empty collection has a length of 0
- **tableof** for tables.. A table is a mapping from key values, of type **String**, to values of a type T (or subtype of T). Table lookup takes the form <tablename>(<keystring>). If there is no value in the table for <keystring> then nil is returned.

### A.8 Comments

Comments can be included in PML following a '!' character. The comment lasts till the end of line. For multi-line comments there must be '!' on each line.

### A.9 Naming Conventions

PML class names must start with an uppercase letter. PML instance variables must start with a lowercase letter.

### A.10 The 'nil' value

All variables may be assigned the special value nil. If they are not supplied with an initial value on declaration, then they will have the value nil. Variables can also become nil through the use of **Give**, and when a **giveport** or **takeport** are transferred between roles.

**A.11 Example Syntax**

```

<EntitySubClass> isa <EntitySuperClass> with
parts
    {data-decl}
end with

```

where:

```

data-decl    = instanceVariable {, instanceVariable} : type := expression
type        = {constructor} Class
constructor  = collof | tableof | giveport | takeport

```

```

PersonDetails isa Entity with

```

```

parts

```

```

    personName : String := 'name undefined'
    personAge   : Int     ! default initial value will be nil
    personMale  : Bool   := true

```

```

end with

```

```

<ActionSubClass> isa <ActionSuperClass> with

```

```

in

```

```

    {data-decl}

```

```

out

```

```

    {out-decl}

```

```

resources

```

```

    {data-decl}

```

```

parts

```

```

    {exec-decl}

```

```

preconds

```

```

    [expression]

```

```

postconds

```

```

    [expression]

```

```

termconds

```

```

    [expression]

```

```

end with

```

where:

```

data-decl    = instanceVariable {, instanceVariable} : type := expression
out-decl     = instanceVariable {, instanceVariable} : type
exec-decl    = actionTag : command [when expression]
type        = {constructor} Class
constructor  = collof | tableof | giveport | takeport

```

Note = not all properties (in, out etc.) are required but the above order must be preserved

```
<RoleSubClass> isa <RoleSuperClass> with
resources
  {data-decl}
actions
  {exec-decl}
initially
  [expression]
always
  [expression]
termconds
  [expression]
end with
```

where:

data-decl = instanceVariable {, instanceVariable} : type := expression

exec-decl = actionTag : command [when expression]

type = {constructor} Class

constructor = collof | tableof | giveport | takeport

Note = not all properties are required but the above order must be preserved

## Appendix B PML for ProcessWeb

ProcessWeb provides a number of facilities in addition to the PML language. There is a Manager role which provides basic facilities - user login and logout, starting instances of library models, ... For each user there is a Proxy role which handles the user's browser page. For models access to the Web connection facilities is given by a UserRole. On output the UserRole keeps a copy of the html sent, so that it can be redisplayed when required. These facilities mean that a ProcessWeb model does not need any code which deals with users logging in and out, binding and relinquishing roles, or switching between different roles which they are bound to.

ProcessWeb provides a number of PML Actions which supplement the standard pre-defined Actions in the language.

For ProcessWeb, many role classes are subclasses of HKClient2. The HKClient2 class provides some standard resources and functionality (used for example in deleting models.)

There will usually be one action property which receives the cgi data. This will have an initial statment - `Take( interaction=userRolePorts.userTakeport, gram=cgi_data )`. The entity `userRolePorts` is the role's interface with its UserRole, its `userTakeport` part has type `takeport` colof `String`. This action property will usually examine `cgi_data` and update the role's state as appropriate. There values in `cgi_data` examined will correspond with the html.

There may be one action property which sends the html, or there may be a number of places where this is done. Either `Give( interaction=userRolePorts.userGiveport, gram=...)` or the much more common `SendToUser( ... )` is the code to look out for.

The code for a ProcessWeb model will consist of a number of PML class definitions. (The programmer can decide how these are split into files, any class definition must be wholly contained within one file.) One class definition will be of the 'main' or 'boot' role which creates the initial role interaction network for the model, and links it into the standard ProcessWeb facilities.

This 'main' or 'boot' role class definition will have:

- resources for the giveports and takeports - These will hold the values when the interactions are created and later be placed in the bindings table of the appropriate role
- some resources which will be used in the calls to `StartRole`
- usually only one action part - This role is often programmed to terminate once everything has been set up
- a number of calls to `NewInteraction` to create the interactions requires
- a call of `CreateUserRole` for each role to be started (strictly each role which has a PWeb user interface). The name parameter is the string which will appear on the browser page, the out parameter `newUser` gives the `userRolePorts` value to be given to the role.
- calls of `StartRole` - Note the `className` parameter is given as a `String`. The `roleInst` parameter should be bound so that the Housekeeper can be informed of the new role. The `inputClasses` parameter has a value which is usually initialised by a prior call of

**GetRoleClasses.** The bindings parameter contains a table which will be used to initialise various resources properties in the newly created role. The warnings parameter is required (If the **StartRole** fails it can be accessed through the Developer diagnostic facilities.)

- calls of **GiveCopy**( interaction=sendReq, gram=HKRequest(...) ) which informs the Housekeeper of the new role instance
- initialization of the bindings table which is passed to each call of **StartRole**. These bindings should include **roleName**, **modelName**, **sendReq**, **userRolePorts**, and the names of all the giveports and takeports which need to be initialised

It is a useful check that all of the giveport and takeport resources are mentioned in one of the **NewInteraction** calls, and in one (and only one) of the binding tables which are given as a **StartRole** parameter.

In the simplest case of a one role model, there is no need for a 'main' role as the **Developer** will automatically do the **CreateUserRole** and start the role with the bindings for **roleName**, **modelName**, **sendReq**, **userRolePorts** and update its Housekeeper.

### B.1 Developer

The Send Receive Develop model, described in section 7.0, is a specialised version of the general **Developer** model used to write and test out *ProcessWeb* models.

*ProcessWeb* provides an incremental compilation facility. New source code will consist of one or more PML class definitions. These are compiled in the context of a set of existing class definitions (held in a PML variable of type **Classes**). If the compilation is successful then the new class definitions will be added to the set of existing class definitions which forms the context for the next compilation. If the compilation fails then an error message will be produced and there will be no change in the existing class definitions.

**Developer** is a *ProcessWeb* model which allows users to develop and test further models. It maintains a set of class definitions which provide the context for compilation. Initially this set consists of the standard *ProcessWeb* class definitions. **Developer** also has a Housekeeper role which enables it to delete, diagnose and modify role instances.

There is an 'Information' link on the **Developer** page which gives details of its facilities.

The following sequence describes how you might compile and run a new model.

- **Compilation** - used the **Developer** compile option to compile in the file(s). This is when compilation errors will be given. On successful compilation the **Developer's** list **Role Classes** will be updated.
- **HTML template files** - If the model uses HTML template files, you need to use this option to load these files onto the server. The directory and filenames should match those in the PML code.
- **Running a model** - Once compilation is complete, select the 'main' role class in the **Developer's** **Classes** list, supply a name and select **Start**. If all goes well the roles will appear in the **Developer's** **Role Instances** list, and you will be able to select the **UserRoles** either from **Developer** or from your browser page

- If the model runs ok, but you want to make further improvements, you can delete the model by selecting the roles from the Developer's instances list, and selecting Kill.
- If there are problems with the model, you might want to look at particular role instances using the diagnostic facilities. Select the role in the Developer's instances list and select Diagnostics. Useful diagnostic command include 'lsr', 'lsr <resource property>', 'cr', and 'his'. If the the role fails in the middle of an action you may want 'lsa' rather than 'lsr'.

The Developer's Modify option allows you to change the class definition for a role instance. The role instance is frozen, its data extracted, it is modified to the new definition, and data is replaced. ( This means that Modify will not update resources properties unless explicit code is written. )

The Export to Library option is for use when you have a tries and tested model which you want to make available to other *ProcessWeb* users.

## **B.2 External Applications**

The PWI system has facilities for interacting with external applications. An external application is presented to a PML role as an entity with a number of giveport and takeport values.

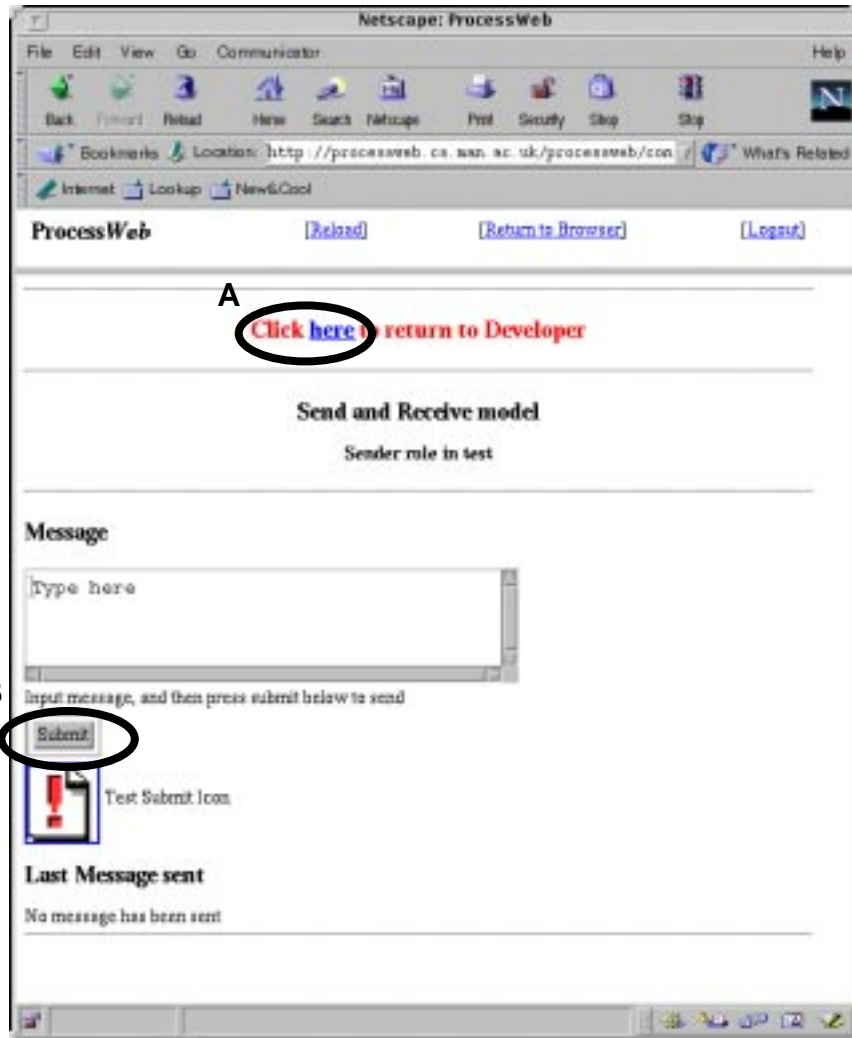
*ProcessWeb* uses these facilities to connect PWI with the Web server. The details of this are hidden from the *ProcessWeb* developer. The *ProcessWeb* developer only needs to know about UserRoles and the interface to them.

Some *ProcessWeb* models for interacting with external tools have been developed. (Mainly Oracle, but also a simple UNIX interface.) These have resulted in some standard PWeb actions to support this. Developers still have access to the underlying PWI external application capabilities if required.

## Appendix C HTML Examples

This appendix shows some additional examples of the correspondence between the HTML source text and the corresponding browser display.

**FIGURE 6. Sender role example - display**



Mark the corresponding parts of the HTML source (Figure 6) for the parts indicated by A and B in Figure 2.

**FIGURE 7. Sender role example - HTML source**

```

<HTML>
<HEAD>
<BASE HREF="http://processweb.cs.man.ac.uk/processweb/connect">
</HEAD>

<body bgcolor="#FFFFFF" vlink="#0000FF">
<hr>
<h3 align=center><font color=red>Click <a href="/processweb/connect?operation=jumpToRole&roleID=R253">here</a> to return to Developer</font></h3>
<hr>
<h3 align=center>Send and Receive model</h3>

<h4 align=center>Sender role in test</h4>
<hr>

<h3>Message</h3>

<form method=POST action="/processweb/connect">
<textarea name="text" rows=4 cols=40>
Type here
</textarea>
<br>
Input message, and then press submit below to send
<br>
<input type="submit" value="Submit">
<br>
<input type="IMAGE" src="http://www.cs.man.ac.uk/icons/alert.red.gif"
alt="Test Image Submit" height="60" align="middle" name="Submit"> Test Submit Icon
</form>

<h3>Last Message sent</h3>
No message has been sent
<hr>
</HTML>

```

**FIGURE 8. Another piece of HTML source**

```

<table>
<tr>
<td>
<br><br>
<B>Click on an arrow to select an operation:</B><br>
<form method=POST action="/processweb/connect">
<input type=image align=center border=0 src="/images/arrow.gif" name=StartRole>
<B>Start</B> an instance of a send receive model <BR>&nbsp;Enter name: <input
type=text index size=15 name="RoleName" value="SndRecvStart">, for model startup
role<br> X
<input type=image align=center border=0 src="/images/arrow.gif" name=EndIt> <B>Kill</
B> roles. Choose one or more instances.<br>
<input type=image align=center border=0 src="/images/arrow.gif" name=CompileInput>
<B>Compile</B> New Class Definitions
<SELECT name="Code Input Method" size=1>
<option selected>Text Input
<option>Browser Upload
</SELECT><BR>
<input type=image align=center border=0 src="/images/arrow.gif" name=EditSndHTML>
<B>Edit</B> HTML Template for Sender Role
  <BR>
<CENTER>
<table>
<tr><td>Role Instances:</td><td><select name="Role Instances" size=5 multiple>
<option value=R254>Developer (R254)
<option value=R395>SndRecvStart (R395)
<option value=R393>test3 (R393) Y
</select>
</td>
<tr><td>Role Classes:</td><td><select name="Role Classes" size=1>
<option>DevSndRcv
<option>Developer
<option>HKClient3
<option>Receiver
<option>Sender Z
<option>SndRecvStartupRole
<option>StartHK3
</select>

</table>
</CENTER><BR><input type=image align=center border=0 src="/images/arrow.gif"
name=GetHistory> <B>History</B> of executed actions<br><br>
</form>

```

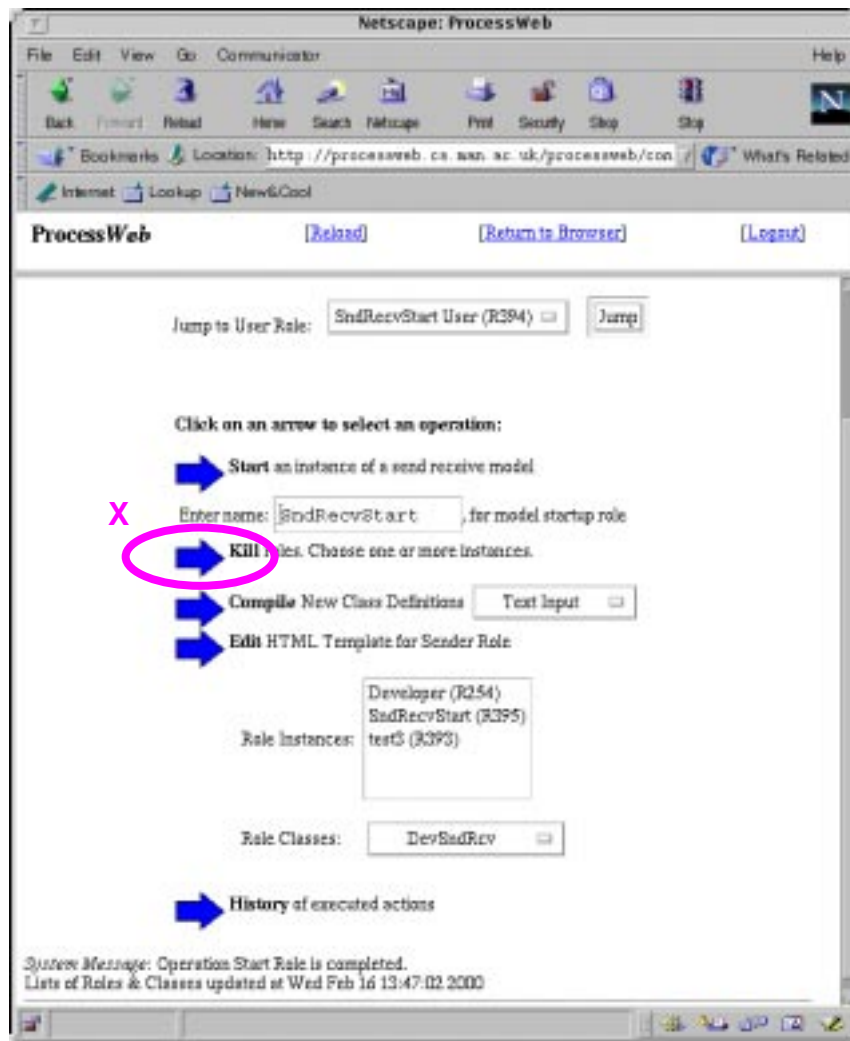
<I>System Message</I>: Operation Start Role is completed. <br>Lists of Roles & Classes

Figure 8 shows part of the HTML source for the page shown in Figure 9.

Mark the display which corresponds to the HTML form image element indicated by X.

Why does “test3” (Y) appear in the display but “Sender” (Z) does not?

**FIGURE 9. Another example - display**



Why does “test3” (Y) appear in the display but “Sender” (Z) does not?

Y is the 3rd element in a list box of size 5, while

Z is the 5th element in a list box of size 1.