EPCC-SS2000-10

Java MPI Simulator

Derrick Pisani
September 2000

Abstract

The goal of the Message Passing Interface (MPI) is to develop a widely used standard for writing message-passing programs. The interface has become an important and increasingly popular and portable message passing system that makes the development of practical and cost-effective large-scale parallel applications possible. EPCC has been developing an MPI Simulator to be used by the Training and Education Centre within a teaching environment. The aim of this Summer Scholarship Programme (SSP) project was to understand and document the existing code of EPCC's MPI Simulator and then go on to remedy some of the perceived problems, to improve its ease-of-use and to further extend its functionality.

Contents

List of Figures

1 Introduction

The aims of this project were to understand the code of the existing MPI simulator and go on to remedy some of the perceived problems with the existing MPI simulator, and to come up with a new more general formulation of the problem.

The problems perceived by EPCC staff prior to the start of the project mainly dealt with the fact that users of SOCCIM[1] found that it was not very intuitive to use and that the code lacked proper documentation.

The project eventually took the following course:

- Familiarisation with the application's usability;
- Documentation of inconsistencies and useful improvements to the user interface;
- Understanding individual functionality of the specific JavaBeans used in the application's code;
- General documentation of the JavaBean's functionality;
- Resolving conflicts which the Simulator had with the MPI-1 Standard;
- Addition of extra functionality within the GUI;
- Resolving structural inconsistencies within the code;
- Planning of future alterations that will harmonise point-to-point and collective communication simulation in the GUI.

It is hoped that with the understanding of the applet gained during the course of this project, and with the extensions made to the original model, EPCC will be in a better position to utilise this application as an educational tool.

2 Background

EPCC was established during 1990 as a focus for the University of Edinburgh's work in high performance computing during the previous decade. EPCC's task is to accelerate the effective exploitation of high performance parallel computing systems throughout academia, industry and commerce.

EPCC's goal is achieved through a range of activities including undergraduate and advanced training programmes.

EPCC first commissioned the development of a simulator of blocking point-to-point communication for MPI from SELLIC[2] in 1997 and thereafter pursued its interest by extending the previous application to also simulate collective communications through a Summer Scholarship Programme (SSP) project [1]. The application developed by SELLIC and the extension written by Tom Doel during the 1998 SSP were written in Java and designed to act as a classroom tool for EPCC's Training and Education Centre.

[1]Simulation of Collective Communication in MPI

[2]Science and Engineering Library Learning and Information Centre - web: http://www.sellic.ed.ac.uk/

EPCC used the simulator in a training environment, but felt that it was not very intuitive for users who were new to MPI. This SSP project dealt with the problem, documented the original code, provided additional functionality to the simulator and also made minor structural changes to the code.

3 Applet Assessment

The first stage of this project involved the assessment of the application's usability, familiarisation with the code's components and their functionality, and the composition of adequate documentation. This gave the project a direction that could be followed during the implementation phase.

Figure 1 shows the user-interface of the simulator. The blue circles represent processes and instructions may be added to the bigger text-box within the process representations. The array of six text-boxes in the lower half of the each process-circle represents the memory buffer. Buttons on each processs representation allow the user to add, delete and edit instructions.

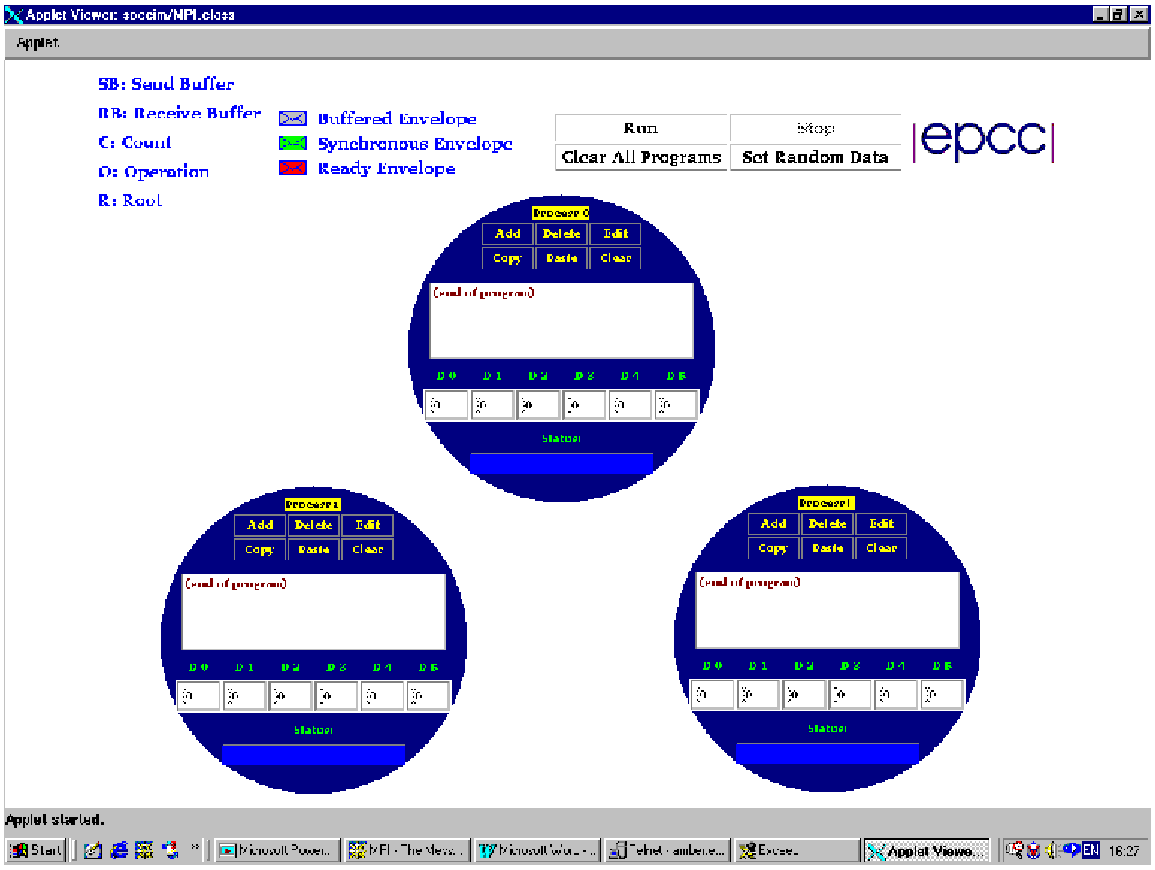


Figure 1: The Graphical User Interface of the MPI simulator.

3.1 Usability Assessment

The following list of comments were drawn up after careful examination of SOCCIM's usability:

1. The parameters passed to the Add Instruction dialog are inconsistent with the way they appear in the MPI Standard.
2. The interface is not very intuitive to the user and generally speaking, it is not very obvious what input is expected for the Simulator.
3. Clicking on EDIT brings up an "Add Instruction" dialog box rather than an "Edit Instruction" one.
4. Only the instruction which has last been input to the system may be modified by clicking once on the EDIT button when the instruction is highlighted.
5. Clicking on the EDIT button when "(end of program)" is selected (highlighted) brings up the Add Instruction dialog box, the user is then allowed to perform alterations to the last instruction and OK them. The modified instruction is added as a new instruction, but the edited one is not deleted.
6. It is only possible to insert instructions at the end the end of the processes's instruction list.
7. The CLEAR-ALL button does not clear the scrolling text of the Status field or the values in the Buffer text boxes.
8. The program does not terminate and the user does not know if the instructions have all been successfully completed or if the program has reached deadlock. The following may be implemented:
    - a rotating gear wheel for active processes that stops when the process has completed all instructions or shows a crossed out gear wheel if it reaches deadlock;
    - a 'traffic light' representation that shows Green for active processes, Amber for waiting processes (to demonstrate synchronisation) and Red when a problem (eg. deadlock) occurs. When all instructions are executed it can show show an All Blank (switched off lights).
9. The user does not have the possibility of running the instructions in Debug mode. Although this application is a simulator, it is meant to be used as a classroom tool and so it should give the user the ability to experiment with MPI instructions. The following may be implemented:
    - having 'On/Off release switches' for each instruction that will cause the process to pause at runtime if it finds an On switch on one of its instructions. The user will then be able to examine the output of the instructions, make modifications to unexecuted ones and then continue playing the simulation;
    - having the possibility of pausing a process, backtracking executed instructions, making any modifications and replaying.

It must however be remembered to implement a system where all processes recognise when a process that forms part of a synchronised system has paused so that affected processes may pause and wait for it to be reactivated (eg. by clicking on Play).

10. Data transfer between the Buffer text boxes is only simulated for collective communication instructions which is not the same as with the point-to-point communication where it is represented by ’flying envelopes’ that move from the sending process to the receiving process.
11. The Simulator is not at all flexible to modifications and this may be greatly improved if the user or teacher may specify:
    - whether C or Fortran-like MPI instructions are preferred;
    - the number of processes that are to be simulated;
    - the size of the data Buffer;
    - a pre-set initial state for the simulator.
12. It is necessary to add collective communication instructions to each process participating in the communication individually.
13. Only Blocking communication instructions may be simulated

3.2 JavaBean Component Functionality Assessment

The SELLIC simulator and eventually Tom Doel’s SOCCIM were both developed using JavaBeans. JavaBeans provide a framework for defining generic, reusable, embeddable, modular software components. The JavaBean specification does not set limits on the simplicity or the complexity of a bean. A bean communicates with the application in which it is embedded and with other beans by generating events.

The component beans in the SOCCIM simulator were only lightly documented and an understanding of their individual functionality within the application’s framework was not very well known at the start of the project.

API documentation was generated for the SOCCIM package using *Javadoc* and this provided a good general understanding of public classes, overridden and inherited methods and the class hierarchy. The "-private" option was used with *Javadoc* to gain a similar understanding of the private classes. In generating *doc* the "-use" option was also used to list which classes and packages are called by other classes and packages. JavaDoc documentation also provides an index of all classes, constructors, variables and methods which also proved to be extremely useful.

A typical *Javadoc* fragment is given in Appendix A. *Javadoc* can also be used to generate LaTeX documentation; this is covered in Appendix B.

The Unified Modelling Language (UML) package, MagicDraw was also used to generate a hierarchical structure diagram of the classes. Unfortunately, however, this package did not prove very useful because of SOCCIM’s flat structure.

JavaBean Structure

Figure 2 , below, shows how the main JavaBeans are related.

MPI extends Applet and is the class that is first called when the applet is loaded. It instantiates a Communicator object that handles all collective communication. Following this, it instantiates a number of objects (equal to the constant numProc, the number of simulated processes) of type Process. These objects take care of drawing individual processes on screen and handling the point-to-point communication between them. A ProcessDialog object is then instantiated by each Process object. The ProcessDialog object is shown on screen as the Add Dialog box. The visibility attribute of each ProcessDialog object is set to false so that it is not shown on screen until the user clicks on ADD or EDIT. Arguments are passed to this dialog from the Process object concerning the identity of the process, and the process instruction being added or modified. A Controller object is also instantiated by MPI for each simulated process within the same loop of the creation of the Process objects. Controller objects take care of sending messages to the Process objects when an event is triggered by the user clicking on one of the buttons in the main top control panel. Message, instantiated by both Process and Communicator provides a framework for collective communication messages.

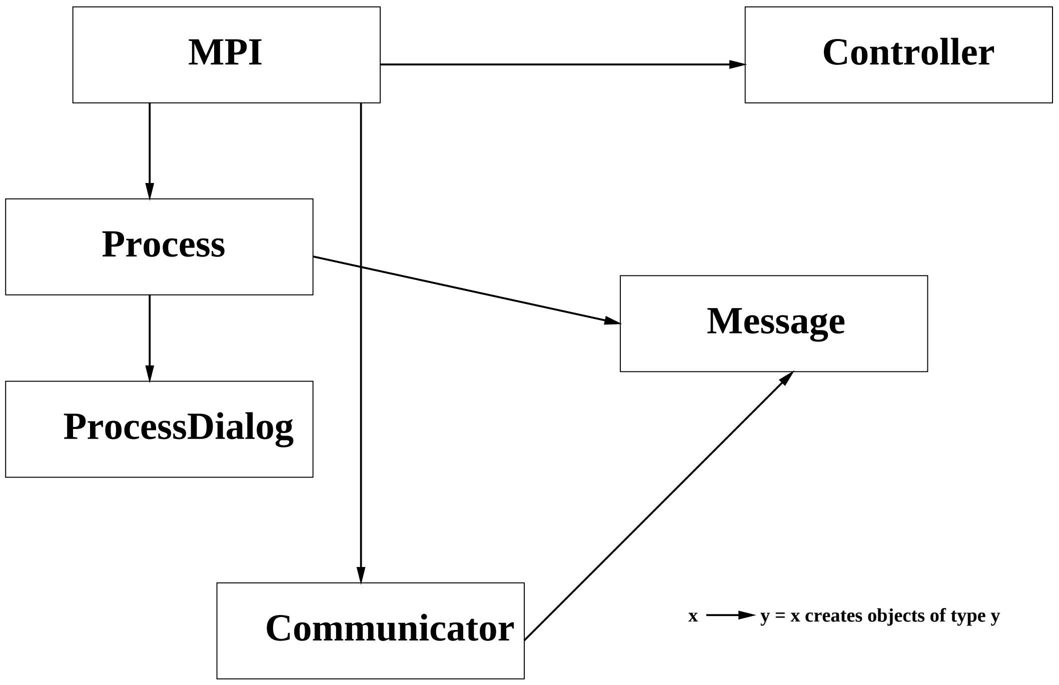


Figure 2: The Class structure in the applet.

Figure 3 shows the layout of the Add Dialog box. It is built from a set of cascading panels held in a Frame container. The Add Dialog box is an object called 'dialog' (of type ProcessDialog), and one instace is created by each Process object.

SOCCIM is a layer over SELLIC's original simulator code and so the code handling message-passing representation has not been properly inherited from one version of the simulator to the other. This has led to a major inconsistency : point-to-point communication being represented by flying envelopes and collective communication by the transfer of data values in the memory buffers. This is mainly due to the fact that the class Process builds the initial processes screen

(Figure 1) and then also handles almost all operations for point-to-point communications. This is part of the original SELLIC code.

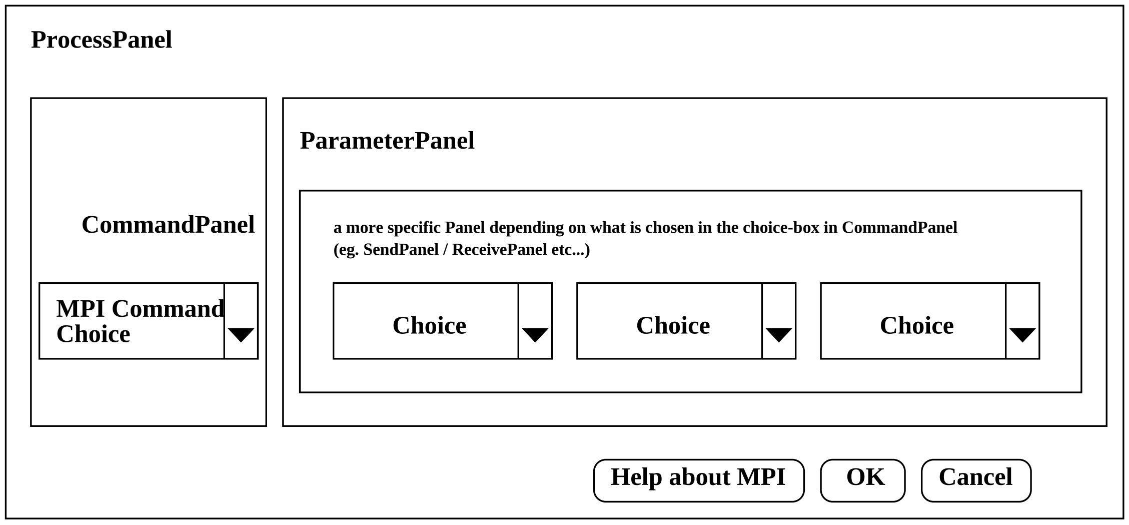

ProcessPanel
CommandPanel
MPI Command Choice
ParameterPanel
a more specific Panel depending on what is chosen in the choice-box in CommandPanel
(eg. SendPanel / ReceivePanel etc...)
Choice
Choice
Choice
Help about MPI
OK
Cancel

Figure 3: The Add Dialog structure.

4 Implementation

This part of the project occupied the last three and a half weeks of the duration of the SSP.

The JavaBeans specification includes the following definition of a bean: "a reusable software component that can be manipulated visually in a builder tool". Given the fact that all the components of the existing simulator program were JavaBeans, the idea of using a builder tool was quite appealing. However none was used for the reason that the work involved dealt more with applying modifications to existing beans and not in the construction of new ones or assembling existing ones into an application.

The API documentation previously generated by *javadoc* was very useful at this stage in tracking parts of the code that broke when modifications were applied to other parts of the code.

4.1 The Add / Edit Dialog Box

The Add Dialog box in SOCCIM has two uses, namely for adding and for editing program instructions. As pointed out in the Usability Assesment this dialog box always carried the name Add Dialog which was potentially confusing. It now carries the name Add / Edit Dialog. The most significant changes that were made to this part of the Simulator's GUI were:

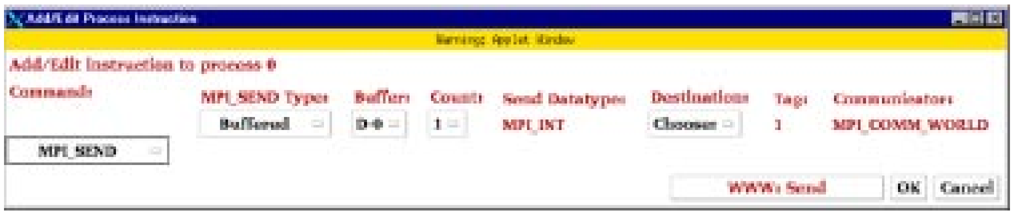
- making the data input consistent with what is expected by the MP1-1 Standard;
- allowing the modification of any process instruction;
- making it possible to edit any process instruction simply by double-clicking on it;
- type verification for Send and Receive operations.

It is believed that the most important characteristic of any teaching tool is for it to be as close to real life as possible. MPI is available for both C and Fortran 77; however this simulator presents all syntax in C-like form. As an improvement over the SOCCIM, all input fields were rearranged to appear in the order specified by the Standard and dummy fields introduced for Datatype and Communicator in the case of Send and Receive operations. In the case of Datatype, the simulator permanently displays MPI_INT. This is for two reasons, namely that:

- the data buffers only take integer input;
- there are different standard data types for C and Fortran 77 and so a C-like type was chosen.

The new Add/Edit Dialog box is shown in Figure 4, below.

The data buffers are represented as six text-boxes on each of the three blue process circles. The value specifying the size of the buffer is stored in an Integer variable *dataElements* and *dataText* is an array of Textfields of that size. The actual values of the buffer are stored in arrays of the same size called *data* and *pdata*, this latter being of type *dataElements* and the former of standard Java type Integer. The task of allowing different data-types on the Simulator would therefore involve changing the data type of *data* and *pdata* to a more general type such as Object, which would then allow the user to pass as data anything of a standard data type or even custom-defined data types defined differently for C and Fortran 77.



Add/Edit Instruction to process 0

Command:	MPI_SEND Type:	Buffer:	Count:	Send Datatype:	Destination:	Tag:	Communicator:
MPI_SEND	Buffered	D-0	1	MPI_INT	Choose	1	MPI_COMM_WORLD

WWW: Send

Figure 4: The new Add/Edit Dialog.

In the case that the Simulator is given the possibility of a choice between C and Fortran 77, then different contents have to be given to the following choice boxes in class ProcessDialog: *rootChoice*, *opChoice* and *commandChoice*, but no structural changes need to be made to the class. It is probably best to get ProcessDialog's constructor method to receive the language choice from Process when it is first instantiated. Other changes need to be made if a choice of preferred language is offered in the Simulator but these are out of the scope of this section and will be discussed later on.

Code has already been prepared (but commented out) to perform type-checking that takes place during a Send-Receive operation. This can only be tested when actual data types are introduced. The Add / Edit Dialog will not allow the user to add an MPI_RECV unless it has been type-checked with the respective MPI_SEND in the sending process. For this purpose the simulator needs to know which process the message is being received from and therefore disables the OK button until a valid Sending process has been selected. It should therefore be pointed out that for the scope of the simulator, when the change has been completely implemented, the addition of an MPI_RECV instruction should always be preceded by the addition of an MPI_SEND instruction.

4.2 The Instruction List

Instructions added through the Add / Edit Dialog appear in a scrollable list in the respective blue process circle and are selected when highlighted by a single mouse click. A highlighted instruction may be edited or deleted from the list by clicking on the respective buttons within the circle. The entire set of instructions may also be copied and pasted into other processes or cleared.

A MouseListener was added to *programList* (in class Process) that holds the program instructions. The *mouseClicked* method is called when an instruction is clicked on, and if it is a double-click the method ProcessDialog.setInstruction is called to set the Add / Edit Dialog to visible and show the instruction's arguments. The mode is also set to edit-mode to allow alterations to be made to the instruction.

When the processes are drawn on screen by the class Process, an item "(end of program)" is added to each of the program lists. This is eventually cleared when the first instruction is added to the process's instruction list and replaced by another that is always the last instruction in the list. When EDIT is clicked or an instruction double-clicked on, the program checks if the index of the selected item is less then the value equal to the size of the list, and is meant to bring the Add / Edit Dialog only if it is. Although this check works for the code that handles a double-click event, it does not work for when EDIT is clicked on. This bug could not be eliminated from the old code but should be pointed out to students.

4.3 The Data Buffers

Work was started on handling the problems associated with the data buffers, namely that the values:

- are not cleared when a Clear All Programs (main control panel) or a Clear (in process circle) is clicked;
- only participate in collective communication demonstrations.

This subject has already been partially dealt with in the subsection "The Add / Edit Dialog Box". The following is the way in which these problems were planned to be dealt with:

1. Coding of a routine to reset all data buffers to zero and to refresh screen when any of the Clear buttons are clicked;
2. Extending the routine which is used by point-to-point communication to simulate the flying envelope to exchange data between the appropriate buffers;
3. Checking that data is received intact;
4. Update on-screen buffer textfields.

The first part involving flushing the buffers was successfully implemented. Although the values on screen are not being reset, the arrays holding the data have been emptied by setting all values to zero. At this stage work on implementation was stopped due to lack of time.

4.4 Process ID Handling

Throughout the course of this project, a few structural changes have been made that will make the code less prone to break when modifications are made to it. The most significant one deals with the process id's. Although there was a pid variable in both Process and ProcessDialog classes, some instances of the code in ProcessDialog.java made use of the index of the item in the Process choice-box. In these cases, the reference is now made to the actual integer in the choice-box and not to its item-index. It was converted from String to Integer and the value is stored in the variable *procID*.

5 Future Recommendations

Although development work on the applet had to be stopped due to the limited duration of the SSP the following recommendations can form the basis for future work:

1. Changing the GUI code from AWT to JFC Swing;
2. Giving the Simulator the capability of demonstrating communications among any number of Processes;
3. Give the user the option of choosing between viewing/using Fortran or C-like MPI instructions;
4. Resolve the buffer issues;
5. Have NumProc and ProcID in central public variables;
6. Respective scaling of screen objects such as the blue process circles;
7. Redesigning the Add/Edit Dialog.

5.1 Changing the GUI from AWT to JFC Swing

JFC Swing is the way of building a system independent GUI in Java.

Tom Doel's applet was built using AWT. AWT uses the operating system's native graphics components and makes the applet look quite different when tested on Windows (TM) and UNIX environments. The applet was not tested in MacOS environment. Each AWT component has an OS-specific peer (native component that is implemented and/or handled by the OS itself) which contains the real implementation whereas Java functions as kind of an 'interface' to it. The synchronisation and communication overhead between the native component and the representation in Java is also an important issue.

Swing components, however, are light-weight meaning that they do not have an OS-specific peer. Native components cause troubles within the GUI because different operating systems treat their components in different ways (eg. a native button on Mac cannot get focus). Swing uses only one native component: the Window, all other components are light-weight and handled completely by Java.

Porting the code to Swing is also of considerable importance in this case because the simulator is meant to be used in a teaching environment and it needs to have a consistent look, whichever operating system the user’s computer is running. It will also make it much easier to develop the GUI of the application because it does not need to be constantly tested on both Windows and UNIX workstations to ensure that it looks good on both. It must also be pointed out that AWT is in the process of being deprecated.

More information on how to port AWT code to JFC Swing is available on Sun’s Java online pages on the Internet [5]. This is also outlined in Appendix C.

5.2 Simulating any Number of Processes

The SELLIC simulator and eventually Tom Doel’s SOCCIM were both designed to simulate the communication amongst up to three processes. It would be useful for the user or the instructor to be able to dynamically set the number of processes.

The following shows which parts of the code need to be changed:

MPI.java

The sections of code below make use of n_procs, a constant which is set to 3.

```
// Number of processes:
private int n_procs = 3;
```

The first instance creates three Process objects. The second instantiates six envelopes, which represent the six combinations of messages that may be exchanged between three processes and the third instance makes sure that these envelopes move at the same speed. The last instance within this class, however, does not make use of the constant n_procs, as each of the three statements adds a component (using the addComponent method defined in this class) to the Component Bag which will be displayed on screen.

```
 // Instantiate and set-up process JavaBeans:
       for (int i = 0; i < n_procs; i++) {
       soccim.Process p = (soccim.Process)Beans.instantiate
                                         (cl,"soccim.Process");
         proc[i] = p;
         p.setPID(i);
         controller.addPropertyChangeListener(p);


 .
 .
 .

// Instantiate and set-up envelope boxes with correct
// source and dest PIDs:
for (int i = 0; i < n_procs; i++) {
      for (int j = 0; j < n_procs; j++) {
      // Don’t need envelope boxes from process x to itself
            // so just break out of loop:
```

```
.
.
.

// Set all envelope boxes to move envelopes at
// the same speed (5):
for (int i = 0; i < n_procs; i++) {
     for (int j = 0; j < n_procs; j++) {
           if (i == j) continue;
           envbox[i][j].setFlightSpeed(5);
     }
}

.
.
.

// Add the processes to the applet:

addComponent(rightP, proc[0], gbl, gbc, 1, 8, 6, 6, gbc.BOTH);
addComponent(rightP, proc[1], gbl, gbc, 12, 17, 6, 6, gbc.BOTH);
addComponent(rightP, proc[2], gbl, gbc, 12, 1, 6, 6, gbc.BOTH);
```

Controller.java

The code below makes use of numProcs, a constant which is also set to 3.

```
public final static int numProcs = 3;

public void propertyChange(PropertyChangeEvent e) {
        String name = e.getPropertyName();
        Object value = e.getNewValue();

        if ((name.equals("stopProgram")) && (running) ) {
                      numStops++;
                      if (numStops>=numProcs) {
                        setInputs(false);
        pcs.firePropertyChange("allStopped", null, null);
                      }
        }
}
```

The number of simulated processes should be central stored variable within the application's code and should not be a hard-coded constant, but a number input by the user (or the instructor), or read from an applet parameter. An option box, implemented as a separate bean which is called by MPI.java, could appear when the applet is loaded. A text box is provided to the user to input the desired number of simulated processes and then this value passed and stored in a variable in MPI.java. The option box mentioned could then also be used for the user to input other preferences such as described in Section 5.3 below.

5.3 Choosing between C and Fortran

The issue of giving the user the option of choosing whether to view and use either C or Fortan-like MPI instructions has been already been mentioned in the Implementation section under the subsection called “The Add / Edit Dialog Box”.

The most sensible way of implementing this is to get MPI.java to bring a dialog box when the applet is loaded. The dialog box could be implemented as a separate class that instantiates a Choice-box with two options: C or Fortran 77. When the choice is OKed, this can then be passed to MPI which can thereafter pass it on to Process objects when they are initialised. Process needs to pass this information to other classes (eg. when initialising the Add/Edit Dialog by creating a ProcessDialog object or when adding instructions to *programList*, the instruction list in each process).

It is debatable whether or not the user should be allowed to change to a different language while using the applet. The user should not have the need for this, but if this option is implemented it must be remembered that the objects that have been drawn on screen will not be automatically refreshed to reflect the new change. The items in Vector *program* (in class Process) need to be regenerated. The ProcessDialog objects instantiated by the Process objects are only created once when these are initialised by MPI and their visibility attribute set to false when they are not required on screen. Ideally they should finalised or set null to ensure that they collected by the garbage collector when other ones are generated for another language[3].

The language option capability forms a basis for increased functionality of the simulator and is a very recommended future extension.

5.4 Buffer Issues

The Data Buffer issues have already been discussed in the context of harmonising the graphical user interface for point-to-point and collective communications in subsection 4.3, "The Data Buffers".

5.5 Centralising Important Values

The number of processes being simulated and Process objects’ ID are important values that do not change during runtime. Ideally these should be centralised so that they may be easily accessed by other classes.

The number of processes could then also easily be hard coded once, rather than having to go through all the instances where it is set and changed separately. One way of doing this is by saying:

[3]The use of Swing may eliminate some of the overheads caused by having multiple ProcessDialog objects, meaning two sets of objects can be initialised.

```
static final int numProc = X;
// where X stands for the number of processes to be simulated
```

This should be inserted in the class MPI.java since it is the Class that initialises the processes. References could then be made to this value using MPI.numProc.

An attempt should also be made to make use of a function that returns the ID of a process. Currently there are various ways in which the ID of a process is being retrieved, including the use of the item's index in a Choice-box and returning the String that gives the item and converting to an Integer. Although the latter method was implemented during the course of this project and makes sure that the process ID is always genuine, it would be much better to have a cleaner way of obtaining a Process object's assigned ID.

5.6 Respective Scaling of Screen Objects

This issue is raised when the Simulator window is not maximised or when the resolution of the screen in use is not high enough. The blue process circles scale up and down with the size of the window but their contents remain in their original size. Any modification to the code to correct this problem should, however, make sure that if the contents in the circle scale down the text remains legible; one way of doing this is to set a minimum size for the window. Alternatively a minimum size can be established for the circle and the scroll-bars attribute of the window enabled when the items cannot fit in the area of the window.

Figure 5 below, shows the effect of the scaling problem.

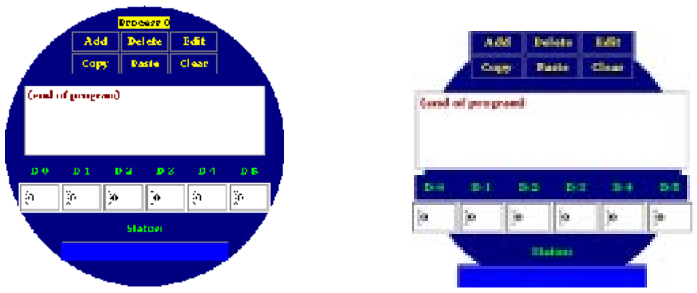


Figure 5: A Process before (left) and after scaling (right).

5.7 Redesigning the Add/Edit Dialog

The size of the ProcessDialog object (*dialog*) that is initialised in the class Process is set to a size of 970x200 pixels. The width of this is too big for screens running at normal resolutions such as 800x600. The dialog should be redesigned to allow for these eventualities.

6 Conclusions

This project has taken a different route then the one planned, but still produced a useful product that will hopefully be of great use to the expansion of the functionalities of EPCC's MPI simulator. Although, the implementation phase was short, significant changes were made and the old code is also now much better understood and documented by means of *JavaDoc*, other inline commenting and above all, in this document.

This project was supervised by Mario Antonioletti, Neil Chue Hong and Gordon Darling. Further information on the project is available online on the web:
http://www.epcc.ed.ac.uk/ssp/2000/Students/pisani.html

Throughout the course of this project I was given the opportunity to apply my OOP skills whilst using Java, I was introduced to better software engineering tools and techniques and feel to have benefited greatly from having worked in a team with my supervisors.

I have greatly enjoyed the social and cultural life in Scotland, especially during the Edinburgh festival in August. During these two and a half moths I shared an apartment with another seven people from six different countries, and have been exposed and come to better appreciate our different cultures.

Participating in an SSP has been a marvellous educational experience.

A JavaDoc Example

The following is a fragment of the *javadoc* generated for one of the classes called *Process*[4]:

Class **Process**

public class - A Java Bean which TRYING simulates an individual process.

This bean contains the program list, data and all the user interface components that are present on each process

Process.java creates a dialog box for instruction input and error output, and sends/receives messages to/from envelope boxes, the communicator and the controller.

Declaration

public class Process

implements java.io.Serializable, java.beans.PropertyChangeListener, java.awt.event.ActionListener,java.lang.Runnable,java.awt.event.MouseListener

Serializable Fields

- private int inputMode
  -
- private boolean running
  -
- private int pid
  -

Fields

- public static final int dataElements
  -
- public static final int addMode
  -
- public static final int editMode
  -

[4]This is an abridged version of the original Javadoc documentation. In particular, the lists of fields and methods have been pruned.

Constructors

- *Process*
  `public` **Process**`( )`

Methods

- *mouseClicked*
  `public void` **mouseClicked**`( java.awt.event.MouseEvent` **e** `)`
  - **Usage**
    * **This method captures a double mouse click on one of the Program Instructions and then fires the Edit Dialog box.**
      Notes on Implementation:
      Upon calling setInstruction (which is a method in the class ParameterPanelsetInstruction in ProcessDialog) for object ’dialog’ this will change the values in the dialog box but the object is still invisible. We then call showInput() to make it visible.

- *paint*
  `public void` **paint**`( java.awt.Graphics` **g** `)`
  - **Usage**
    * We first paint the background, then call super.paint which will paint all the contained components

B Converting *JavaDoc* Documentation to LaTeX

The following doclet was used to generate *JavaDoc* documentation in LaTeXformat:

com.c2_tech.doclets.TexDoclets.

This is **tar**red along with the project in a directory called UTILS.

C Example on How to Change AWT to JFC Swing

The following article show how to convert existing AWT code to JFC Swing. This is something which has been proposed as a future extension to the Java Simulator applet.

This is an abstract from the Javasoft webpage :
http://developer.java.sun.com/developer/technicalArticles/GUI/Transition/index.html

Swing 1.0.2 extends the Abstract Window Toolkit (AWT) with a full set of GUI components and services, pluggable look and feel capabilities, and assistive technology support. The Swing architecture is based on the AWT architecture, which makes it easy to port AWT-component programs to Swing. If you wonder why you should use Swing or convert an AWT-component program to Swing, consider these things:

- Swing components are more efficient and have a cleaner look and feel. Most Swing components are lightweight, which means they do not rely on user interface code that is native to the underlying operating system they run on. Being lightweight, they are more efficient than heavyweight components that rely on native user interface code.
- If you want a cross-platform look and feel, you can easily plug it into your program, and if you expect the application to run only on a certain platform, you can easily plug that look and feel in too. You can also set your application to plug the look-and-feel in at run time.
- Swing applications can support assistive technologies such as screen readers, screen magnifiers, and speech recognition devices, which make Swing applications accessible to a wide-range of persons with special needs.

You can find a lot more information on Swing in the Swing Connection[5], and in the article Introducing Swing[6].

Making the Changes

In many cases, it takes only a few adjustments to source code to convert AWT components to Swing. The list below summarises how the AWT source code for the demonstration application from the Exploring the AWT Layout Managers[7] article was changed to use Swing components.

[5]http://www.java.sun.com/products/jfc/swingdoc-current/index.html
[6]http://java.sun.com/products/jfc/swingdoc-static/what_is_swing.html
[7]http://developers.java.sun.com/developer/technicalArticles/GUI/AWTLayoutMgr/index.html

- Swing import statement added:
  import java.awt.swing.*;
- Swing container used:
  JFrame;
- Swing components used:
  JButton, JPanel, JTextField, JLabel;
- Main program specifies the look and feel:
  UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());

Here are the AWT and Swing versions of the source code:

- JDK 1.1 using AWT components[8]
- JDK 1.1 using Swing components[9]

Using a Content Pane

If you look at the LayoutExample constructor in the Swing version of the source code (excerpted below), you see calls to the getContentPane() method of JFrame for setting the layout manager and adding components.

Components are not added directly to a JFrame, but to its content pane. Because the layout manager controls the layout of components, it is set on the content pane where the components reside. A content pane provides functionality that allows lightweight and heavyweight components to work together in Swing.

```
public LayoutExample() {
   getContentPane().setLayout(new GridLayout(1,2));
   setFont(new Font("Helvetica", Font.PLAIN, 14));

   p1 = new JPanel();
   p1.setLayout(new GridLayout(8,1));
   p2 = new JPanel();


   p1.setBackground(Color.gray);
   p2.setBackground(Color.white);
   getContentPane().add(p1);
   getContentPane().add(p2);
```

You can find information on content panes, heavyweight, and lightweight containers (plus a lot more) in the article, Understanding Containers[10] by Eric Armstrong.

[8]http://developers.java.sun.com/developer/technicalArticles/GUI/Transition/1.1/LayoutExample.html
[9]http://developers.java.sun.com/developer/technicalArticles/GUI/Transition/Swing/LayoutExample.html
[10]http://www.java.sun.com/products/jfc/swingdoc-current/frames_panes.html

Different Methods

Some Swing components have different methods from their AWT counterparts. If you just change the component name without checking the methods, the compiler will point out all the AWT component methods that are not valid in Swing. In most cases, it is simply a matter of browsing the Java(TM) docs to find the correct name or parameter list for the equivalent method in Swing.

References

[1] T. Doel, *Java Simulation of MPI Collective Communications*, EPCC, University of Edinburgh, September 1998.

[2] D. Flanagan, *Java in a Nutshell A Desktop Quick Reference*, O'Reilly, November 1999.

[3] D. Flanagan, *Java Foundation Classes in a Nutshell A Desktop Quick Reference*, O'Reilly, September 1999.

[4] D. Kramer, *How to Write Doc Comments for Javadoc*, Javadoc Home Page: http://jsp2.java.sun/products/jdk/javadoc/writingdoccomments/index.html

[5] M. Pawlan, *JDK 1.1 and Beyond: Making the Transition*, Java Developer Connection webpage: http://developer.java.sun.com/developer/technicalArticles/GUI/Transition/index.html

[6] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra, *MPI The Complete Reference*, The MIT Press, 1996.

Derrick Pisani is an undergraduate at the University of Malta, reading for a Bachelor's degree in Information Technology with Honours in Computer Science and Artificial Intelligence. He has completed two years out of four and will graduate in June 2002.

E-mail: derrick@aost.com.