Document Centric Environments: combining the solving and documentation processes

Al Sadiq Ul Amin M. Madani Halepota\textsuperscript{1*}, Philip W. Grant\textsuperscript{1} and Christopher P. Jobling\textsuperscript{2}

\textsuperscript{1} Department of Computer Science, \textsuperscript{2} Department of Electrical Engineering, University of Wales Swansea, Singleton Park, Swansea SA2 8PP, UK
(email: csmadani, p.w.grant, c.p.jobling@swan.ac.uk)

SUMMARY

All disciplines of engineering rely on numerical processing and visualisation for solving design problems and therefore benefit from support environments. These environments should provide a mechanism for integrating different tools and streamline the process of solving other problems of similar type. Users can be provided with a consistent interface to the required tool allowing them to process problem specific data with ease and convenience. One important source of this convenience could be that the original problem solver completely documents the process of solving the problem. This is easier said than done, since the process of documentation is usually seen as a separate undertaking from the process of obtaining the solution. An answer to this problem is to combine the process of designing the problem solution with documentation of the process solution. To do this, a document centred environment is required which encourages designers to think of the process of problem solving as an activity, which stems from the process of documenting. This paper describes such an environment and various aspects of its realisation. A document paradigm is used from the user–interface of the environment to its repository. The whole environment is implemented in Java, with the eXtensible Markup Language (XML) used to describe the structure of the document. The documents produced by the environment act as the integrators of tools, which aid in the problem–solving process in various scientific and engineering domains. Two example problems, one dealing with control system design and other with computational fluid dynamics are used for illustration. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: Softw. Pract. Exper.; LATEX; class file

INTRODUCTION

The concept of a document centred (docucentric) approach towards environment building and designing can be used in all fields of engineering\textsuperscript{1,2,3,4,5}. Most engineering problems have to pass through several stages before a solution is obtained. At each stage, software tools are used to assist in the processing of the problem data. Some tools are more suitable for one type of problem than others. Many engineering and

\begin{flushleft}CCC 0038–0644/99/$10001–05$17.50 Received 18 Oct 1999\end{flushleft}

Copyright © 1999 John Wiley & Sons, Ltd.
research teams have already invested in coding and building a collection of tools inhouse over a long period of time using different programming languages and platforms. Others have opted for off-the-shelf solutions and have ended up with a collection of tools, which have diverse user-interfaces and proprietary data file formats.

To build a docucentric environment it is imperative that the environment should allow its users to easily integrate different tools. This was one of the design constraints when building the environment. The second constraint was the need to allow users to deploy this environment on any computer platform they favoured. Finally, the most important constraint was to present the whole environment in the form of a document so that user was encouraged to design by describing the process of designing. The constraints, mentioned above, indicate that the architecture of the environment should be component based. These components have to be capable of being plugged in and out, as needed, to adapt the environment to different tools and problems. The environment should be implemented on most platforms, or better still, be executable without recompilation. The Java programming language, with its virtual machine architecture, provides the solution to this design constraint. The third and most important constraint of implementing the whole design within a document interface using component technology was tackled by designing a framework. This framework incorporates an XML parser, a WYSIWYG document browser and editor, and an object storage engine. The latter performs object repository services during environment execution and provides a mechanism for creating documents from the states of the objects in the environment and recreating the environment object states from a document.

In this paper, after introducing the “docucentric” design concepts and various design constraints in the introductory section, the implementation details for building the framework for this environment are then described. Various techniques utilised to enable the users of the environment easy modification and component integration are indicated. The internal coding conventions for components, which can be integrated into the environment, are detailed in the subsection Component Anatomy. The process of creating a design document and its associated environment is described in the following subsection. In the Applications section, two example environments are demonstrated. In the first example, an integration of off-the-shelf tools for creating a computer-aided control system design environment is explained. For the second example, a command-line file-oriented FORTRAN code is transformed into a Java-based problem-solving environment for computational fluid dynamics problems. Neither of these examples relies on the Java Native Method Interfaces and so platform neutrality is preserved.

IMPLEMENTATION

This section details the implementation of the main components of the environment. The principal component of the environment is the browser/editor, which is described in detail. Other helper objects, which perform various tasks in the environment, are also outlined. Visualizing the occurrence of design patterns has helped in finding the programming solutions and therefore related patterns are also mentioned wherever applicable. Techniques for integrating components into the environment, for creating user customizable menus and creating new component are described.
Browser and Editor User–Interface

To encourage users to visualize the process of designing and environment building as an extension of the documentation process, the whole environment is built upon the document paradigm. This means that, not only is the user interface a document but also all objects and utilities available in the environment are viewed as sections of the document. This requires that users should see the environment as a document, interact with it as if editing a document and be able to record their interaction with the environment by ensuring the states of the document are persistent objects recorded in a document format.

The user interface of the environment is designed as a document editor. The differences between this editor and any other word processing editor are significant. A visible area on the screen is provided by the document editor which acts as a conduit for transferring the user actions to a component and may also return the resultant response from a component to the user. In short, this area acts in the document as a representation of an environment component by being a part or section of a document. Therefore, each component must have the ability to lay itself on the document by occupying a certain area. Some components, which provide a descriptive representation in a document, such as paragraphs or headings, must also have some text editing and formatting capabilities. If these components also act as providers of scripts for tools then a mechanism must exist for exporting their text contents as scripts and invoking the tool’s execution engine. The main component of the environment is implemented following the Singleton pattern as the Environment class. This class mediates between other objects in the environment, therefore the Mediator pattern suggested some coding solutions, which have featured in the implementation. The Environment class has been sub-classed from Frame Class from the Java Abstract Windowing Toolkit. As this class provides a container for other objects of the environment, it also implements those interfaces, which deal with receiving various events from the window, the mouse, and the keyboard. The Environment class also provides users with the menubar and various menu items for initiating events.

A flexible approach towards loading menu labels and events associated with them is taken. This facilitates easy adaptation of menu items, contained in the menubar of this object, to the different components integrated into environment by a user and is described as follows. At initial startup, a text file menus.ini is loaded which provides two resources. Firstly it supplies labels for a menu bar of the environment and secondly, it provides the name of the classes associated with these menu items. The environment takes advantage of the Java feature of instantiating objects from class names at run time. The MenuInitializer helper class reads the menu labels and associated classes so that the Environment instantiates the required components at run time. To cater for the extensibility criterion, the menus.ini file has been given a special format so that users can easily modify, add or remove labels and control the positioning of menu items, without the need for recompilation or creation of byte-code. In keeping with good user-interface design guidelines, permissible depths of menus have been kept to a maximum of three. Each line of the menu.ini file has the following format:

\[ x1 \ x2 \ x3 \ x4 \ LabelName \ ClassName \]

Where \( x1 \) are integers denoting:
\( x1 \) Main Menu Position.
x2 Menu Item Position.
x3 Sub Menu Level.
x4 Sub Sub Menu Level.

LabelName appears as a menu item (Figure 1) and is passed on to ElementLoader as a ClassName and used by the Environment class to instantiate the object. Parts of the menu.ini file for the menu in Figure ?? are seen in Table I.

<table>
<thead>
<tr>
<th># menu</th>
<th>Items in fourth menu</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 1 0 0</td>
<td>Image ElementLoader</td>
</tr>
<tr>
<td>3 2 0 0</td>
<td>PageBreak ElementLoader</td>
</tr>
<tr>
<td>3 3 0 0 p</td>
<td>ElementLoader</td>
</tr>
<tr>
<td>3 4 0 0</td>
<td>ParagraphBreak ElementLoader</td>
</tr>
<tr>
<td>3 5 0 0</td>
<td>Heading ElementLoader</td>
</tr>
<tr>
<td>3 6 0 0</td>
<td>Figure ElementLoader</td>
</tr>
<tr>
<td>3 7 0 0</td>
<td>Title ElementLoader</td>
</tr>
</tbody>
</table>

A MenuInitializer class contains a vector, which holds references to the classes already loaded, so that subsequent calls to methods of a given component class do not need its reloading or reinstantiating it. The Environment consists of two basic types of components. The first type of component acts on the document in the context of the

![Figure 1. Environment User Interface](image)
environment and performs actions such as loading and saving of the document. These components are concrete implementations of an abstract class `ActionProvider`. This ensures that all sub-classes have the `performAction` method, which is responsible for providing functionality of each object available through selecting a menu item. The second types of components are those elements of the document, which act in the context of the document. These include those components which format paragraphs, headings, figures or otherwise provide integration of legacy applications and tools. These components are represented in an XML document type definition (DTD) as document entities or document elements. They are sub-classed from class `ElementAction`. A wrapper or `Adapter` class `ElementLoader` allows the creation of these objects via menu item selection. This is necessary due to the flexible nature of the environment, in which a component could either become its part by virtue of being existent on a document and environment has loaded a class responsible for the instantiation of this component using `Class.forName` method. That is, the environment is acting as a browser of the document containing the object. Another scenario is one in which the advanced user has either recently created the component and wants to integrate the component as part of the environment, or just wants a new instance of it to be inserted in the document. In this case the user includes the class name in the `menu.ini` which then causes the `ElementLoader` class to load the component as `ElementAction` object.

Anatomy of a Component

In order to allow each component access to the document’s ‘real estate’ and interact with other classes of the Environment, certain coding conventions are adhered to. This section describes these conventions and the various methods and interfaces which have to be available in each component for it to become part of the environment.

Each component of the environment is a sub–class of class `ElementAction`, which itself is sub-classed from the class `ElementImpl`. `ElementImpl` class represents the default implementation of the Element interface created by the XML parser when using the default element factory. `ElementImpl` class is part of a public domain XML parser implementation available on the World Wide Web. While sub-classing from the `ElementImpl` class takes care of the necessary methods required for element to be able to conform to XML standard and be able to be part of an XML document, other methods are required for integrating this element into the environment. Table II lists the methods which each Environment component must implement to be able to become a part of the Environment.

Briefly, each component must be able to lay itself out in the environment and therefore should be able to display itself by using the `drawIt()` method. On scrolling or changing the attributes of the document, the component should be able to redisplay itself by using `reDrawIt()`. Each component has to have a reference to the containing Environment to be able to use its Frame as a container, get events from it and use other resources such as object store database. Therefore the component should make its physical position references for drawing itself, and the unique object reference to serve as a key in locating it in object store known to the Environment. The component should be able to register and un-register its interest in various events generated in the Environment to allow the user to send keyboard, mouse and other events to it while it is under focus. Each component should also instantiate an object with...
default attributes using `createThisElement` method. The methods like `init` and `performAction` provide convenient entry points for initializing the component and forcing it to do what it is supposed to do.

Apart from the methods described above, each element also has a corresponding attribute editing class. This class is sub-classed from `EditDialog` class and described in the section entitled Component Attribute Editing.

**DesignDocument**

This section describes the process of creating a design document in the environment. The default root element of the documents created in this environment is `DesignDocument` element. On loading a document, the `Environment` class uses an XML parser to create a parse tree of the document. Each element of the document forces the `Environment` to load its related component and call its `performAction` method. Each component uses the attributes from the document and creates the state of the object as preserved in the document (Figure 4 and 5). In case the element is being inserted in the document from a menu, i.e. a new instance is being created, it is given a state according to the associated DTD of the element. Default constructor attributes are also provided within the component code as shown in Figure 6.
Table III. DTD description of ‘p’ element with default attributes

```
<!ELEMENT p> (#PCDATA)>
<!ATTLIST p
  align (left|center|right) "left"
  font CDATA #IMPLIED
  style (plain|bold|italic) "plain"
  size CDATA #IMPLIED
  color CDATA #IMPLIED >
```

Table IV. ‘p’ element with attributes and character data from a document

```
<p align="left" style="plain" size="12"
font="Dialog" color="black">This is a test document. It contains p elements for handling of paragraphs.</p>
```

XML specifications 12 require all documents to have a single root or document element which is parent of all elements and is not a part of another element, therefore a sample DTD with one root element <DesignDocument> is used as a template for a blank document. The XML parser creates documents of the type DesignDocument and validates the document structure according to the DTD of the DesignDocument. Upon initiating the new Document menu event, a new blank document is opened in the Environment. User wishing to modify the Environment can insert different elements into this document from the available components in the Insert menu to assemble a new document and environment. An expert user wishing to extend the Environment can edit the blank.xml document to update the DTD with the attribute specification of their new components. Each document element is given a visible space on the document according to its content. An internal data structure in Environment class keeps track of this space and allows interaction with it via a mouse click. Clicking with right mouse button brings up a dialog window (Figure 7) for attribute modification. A left mouse button click activates direct interaction, such as data entry or text editing by calling registerListener() method of the component. If the component is designed for text based interaction then it must provide a mechanism for laying out its text content with various font attributes, such as typeface, size, style and layout attributes such as alignment and justification using font metrics. Some example components such as “p” have been implemented with these facilities using well-known text processing algorithms 13.

Component Attribute Editing

Each component also has an associated EditDialog Class. In order to cater for components which may have varying numbers of attributes and may require a different
number of widgets in the attribute editing window, labels, names of button and list contents are added at the beginning of the EditDialogClass as elements of an array. The numbers of elements in each array are also assigned to variables, which are used by the button, label and list making methods as illustrated in Figures 6 and 7. This arrangement permits very easy modification to sub-classes of EditDialogClass, by simple text editing at the beginning of the class and then allowing the system to create and layout the widgets on the fly. This mechanism is used here instead of using resource bundles for two reasons. Firstly the number of resources may vary from component to component and the modification of dialog is not just limited to the use of different label strings but different layouts, number and types of widgets as well. The second reason for implementing this feature may prove useful for machine generation of the EditDialogClass in later implementations.

Table V: Various parameters used for iterative creation of the widgets in EditDialog Class

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>numberOfButtons</td>
<td>2</td>
</tr>
<tr>
<td>numberOfChoices</td>
<td>9</td>
</tr>
<tr>
<td>numberOfLabels</td>
<td>9</td>
</tr>
<tr>
<td>numberOfTextBoxes</td>
<td>4</td>
</tr>
<tr>
<td>buttonPanelwidth</td>
<td>200</td>
</tr>
<tr>
<td>buttonPanelheight</td>
<td>20</td>
</tr>
<tr>
<td>numberOfTextFields</td>
<td>4</td>
</tr>
</tbody>
</table>

```java
int numberOfButtons = 2;
int numberOfChoices = 9;
int numberOfLabels = 9;
int numberOfTextBoxes = 4;
int buttonPanelwidth = 200;
int buttonPanelheight = 20;
int numberOfTextFields = 4;
String buttonLabels[] = {"Update","Cancel"};
String labelLabels[] = {" Align"," Font"," Style"," Size"," Colour"," Plane", " Linear"," No. Of Quad Pts>12"," Fixed Boundry"," Make File", " Output File"," Prerequisite"," Postrequisite"};
String choiceLabels[][] = {
{"left","center","right"},
{"TimesRoman","Helvitica","Dialog","Courier","Symbol"},
{"plain","bold","italic"},
{"8","9","10","11","12","14","16","18","20","22","24","28"},
{"black","white","red","green","yellow","blue"},
{"true","false"},
{"false","true"},
{"false","true"},
};
```

Using the above described component building techniques the environment is populated with components which handle basic document layout such as titles, headings, figures and paragraphs. The actual problem solving capability of the environment comes from those components, which use the parts of document as their visual representations but also have some mechanism for activating and communicating with tools and other applications. This will be illustrated in the two examples appearing in the next section.

Table VI. makeButtons method with a for loop for iterative creation of widgets

```java
private void makeButtons()
{
    int x;
    for (x = 0; x<numberOfButtons; x++)
    {
        buttons[x] = new Button(buttonLabels[x]);
        buttons[x].addActionListener(this);
        buttonPanel.add(buttons[x]);
    }
}
```

APPLICATIONS

To illustrate the creation and integration of problem solving components and techniques for communication with tools in a platform-neutral manner, two environments dealing with problem solving in two different domains of engineering are described.

Control System Design Example

Control system design like all computation intensive disciplines requires help from various tools. Matlab\textsuperscript{14} and Simulink have become de-facto standard this field\textsuperscript{15,16}. Matlab is pre-dominantly a text oriented tool, requiring the user to interact with it by typing commands or loading text based script files into the command processor. The Simulink extension to Matlab allows users to create simulations of systems by manipulating graphical representations of the blocks of various system sections. Each block in Simulink diagram has associated with it parameters which can be loaded into Matlab using command line text. To demonstrate the applicability of document centred design approach in the context of computer aided control system design, we have created components which facilitates the integration of Matlab and Simulink into the document as document sections. These sections communicate with Matlab from Environment providing mechanism for conveying user commands to Matlab and rendering results obtained from it into the document. Placing Matlab and Simulink components along with other document formatting components in a document creates a design document and a design Environment simultaneously. This Environment allows users to modify different parameters and experiment with the results and record all interactions in a single document.
The mechanism employed for integration, using and invoking Matlab requires use of a component, which must not only have facility for textual display but should also support script entry and editing.

When a Matlab element occurs in a document, a component dealing with its DTD description is instantiated, using the attributes defined in its tag, an example of which is shown in Figure 7.

Matlab uses special script files, called m-files, that store sequences of commands. On initial startup, Matlab reads a special script file named startup.m. This file is used by the Matlab component to invoke Matlab with a sequence of instructions taken from the document. Necessary m–files, with pre and post conditions as described by the above Matlab tag (Figure 8) and set up using the dialog shown in Figure 8, are created automatically. Matlab is then invoked using the startup.m script file, which is also created on the fly, with the necessary instructions. Two copies of the Matlab workspace, one in ASCII and other in binary MAT format, are saved, so that the results can be used within the document and the variables and workspace can be reloaded into Matlab for later interaction. The need for saving the same data in two formats arises due to the fact that Matlab can only reload variable names and values from MAT files properly, whereas ASCII files are needed for insertion into a document.

To integrate Simulink into the document environment requires a component with slightly different presence on a document. Simulink, being a graphical application, allows visual building of models and simulations and requires a graphical component.
rather than a textual one. The images representative of the Simulink models must allow the user to invoke Simulink for modifying model or its parameters. Simulink may also require that Matlab must contain the values of the different variables in its workspace. Therefore, the Simulink component is designed with the facility to start Matlab with a loading script file before loading a Simulink model.

As an example, a document for solving F-14 benchmark problem $^{18}$, a problem used to evaluate control system design tools, has been constructed. The user interface for this document is shown in Figure 9.

The Gruman F–14 Benchmark problem document consists of various paragraphs describing the problem and Matlab and Simulink components to setup the problem and perform the simulation. Matlab component creates a m-file with necessary data from the document and generates a startup script which causes this new script file to run by obtaining a process object on the host machine (Process \texttt{p = Runtime.getRuntime().exec(‘matlab.exe’);}). The Simulink component has image of its model on the document and clicking on it can start Simulink user-interface in Matlab and load the model and its parameters. The modified models are saved as images and put into the document to serve as hyperlinks.

COMPUTATIONAL FLUID DYNAMICS PROBLEM

We now consider a very different field of research, that of Computational Fluid Dynamics \textit{CFD}. This is a branch of science, which relies heavily on the use of numerical
tools for problem definition and solution. Problems in CFD are usually solved in three stages.

The first stage is pre-processing, in which problem parameters and the geometry are defined. For this purpose, mesh-generating software is utilised which can save a mesh in any standard format for use by the solver.

In the second stage of problem solving, the problem is simulated using time stepping code written in high level language. In this specific case a FORTRAN 90 code is used.

The post processing stage deals with the visualisation of the various variables obtained in the processing stage within the domain of the problem.

**Problem Solver Component**

In this specific example the heart of the environment is the code written in FORTRAN. This code was written on a UNIX platform and in its current form is totally reliant on text editing by a programmer to change its compilation and execution paths. To include or exclude various modules of the code from execution a set of parameters are defined using #define pre-processor directives. A makefile is hand edited by a programmer. In this file, different flags are set or un-set to compile the sections of the code pertaining to the solution of problems under consideration. For example, one problem may require a LINEAR model of formulation to obtain its solution whereas another may have FIXEDBOUNDARY conditions set to FALSE.

We want to make the process of experimenting with different parameters, geometries etc. easier for the researcher and at the same time provide a means of documenting the steps. In order to create an environment for CFD problem solving, using the above-mentioned FORTRAN engine, requires a component for integrating this code into the Document Centred Framework. This component should not only allow easy modifications of the flags controlling the compilation and execution path but should also require no changes to the original FORTRAN code.

The component, which deals with this FORTRAN code, is appropriately named CFD.class. The responsibility of this component is to provide the following facilities:

1. To provide visible space in the document for a description of the program fragment or related information and allow dialog based interaction for parameter modification.
2. To instantiate other helper classes, in particular a class named CFDEditDialog, for setting up flags and parameter modification, employed during compilation and execution.
3. To generate an appropriate 'makefile' automatically reflecting the changes in the parameters and flags.
4. To compile the FORTRAN code by invoking the make utility and executing the executable file thus generated to produce desired results.
5. Allow the visualisation of results using an appropriate tool.

**Problem Specifications**

For demonstration purposes, the numerical example employed in this study is a mixing and separating channel flow of visco-elastic fluid. In this problem fluid enters from two opposite inlet channels, which are separated by thin wall with a gap at
the centre of the channels to allow mixing of the fluid. The domain is divided in the regular and mapped triangular tessellation biased in centre of domain. A known parabolic velocity profile for axial velocity component, vanishing cross component and stress boundary conditions are prescribed at the inlet. To model this complex flow of viscoelastic fluid a shear-thinning Phan Thein Tanner model for constitutive equation and semi-implicit Streamline-Petrov-Galerkin/Pressure-Correction scheme has been chosen.

The problem data relating to the mesh specification, initial and boundary conditions are stored in a text file, which is read by the FORTRAN code module.

Creating the Environment

A document describing the environment needs a DTD to define its structure and to validate the attributes of various components as described above. The DTD also serves the purpose of providing initial conditions for the initialisation of the objects. The DTD description for the CFD object is shown in Figure 10.

Table VII. DTD description of CFD processor component

```xml
<!ATTLIST cfd
    align (left|center|right) "left"
    font CDATA #IMPLIED
    style (plain|bold|italic) "plain"
    size CDATA #IMPLIED
    color CDATA #IMPLIED
    pre CDATA #IMPLIED
    post CDATA #IMPLIED
    evaluate (false|true) "false"
    Output CDATA #IMPLIED
    MakeFile CDATA #IMPLIED
    Plane (false|true) "true"
    Linear (false|true) "false"
    QuadPoints (false|true) "true"
    FixedBoundry (false|true) "false">
```

The components required for composing a CFD problem solving environment not only need a textual description but also need support for modifying different flags, which control the flow of the compilation process of the FORTRAN problem solver.

The first part of the DTD description deals with the textual properties of the component such as font used and the alignment of the text in the container. The second part of the DTD description deals with the description of the flags controlling the compilation and execution path of the code. Depending on the setting of these flags a Makefile is generated. Figure 11 shows a typical Makefile generated by the CFD component.

To facilitate easy modification of the flags without resorting to text editing a simple dialogue window, as shown in the Figure 12, is made visible by clicking the right mouse
Table VIII. Makefile generated by the CFD component reflecting the flags set by a user

```
# This is machine generated File
MAKEFILE= Makefile
PROBLEM= main_stream
PROBLEM_DIR= .
CODE_DIR= $(PROBLEM_DIR)
INCLUDE_PATH= -I$(CODE_DIR)
DEFINES= \n-DNQPTW \n-DPLANE
CPP=/usr/ccs/lib/cpp
FC=/usr/bin/f90
targt:
  $(CPP) $(DEFINES) $(INCLUDE_PATH) inpout_strm.F inpout_strm.f90
  $(CPP) $(DEFINES) $(INCLUDE_PATH) main_stream.F main_stream.f90
  $(CPP) $(DEFINES) $(INCLUDE_PATH) streamuty.F streamuty.f90
  $(CPP) $(DEFINES) $(INCLUDE_PATH) stream_cb.F stream_cb.f90
  $(FC) $(FFLAGS) -o $(PROBLEM) $(PROBLEM).f90
FFLAGS= -O5 -C -w -automatic $(INCLUDE_PATH)
dummy: cc -o $(PROBLEM)
CACHE_LINKS: FORCE
  @ echo Making $(CACHE_DIR).....
  @ if test ! -d $(CACHE_DIR); then mkdir $(CACHE_DIR); fi
  @ cd $(CACHE_DIR); \n  for SOURCE_FILE in $(CODE_SOURCES); \n    do ( echo $$SOURCE_FILE ) done
  clean: FORCE
    rm -f *.o *.out *.f90 work.* *.trace $(PROBLEM);
FORCE:

```

button anywhere within the area occupied by the CFD component in the document.

For the post processing of the results, the graphical facilities of Matlab are used. In order to invoke the Matlab with the suitable script files, the same techniques as described in the context of Control System Design Environment are utilised. Opening up the menu on the main window of the environment allows us to select the CFD object, which is placed into the document. Clicking with the right mouse button brings up a popup menu from which the user can choose either to edit the attributes or start the execution. Modification of the attributes triggers an `itemStateChanged` event invoking methods changing the XML attributes of CFD object. Initiating the 'execute' event causes the CFD component to write out a Makefile reflecting the current state of the attributes, and then run this Makefile using the make utility resulting in compilation of the FORTRAN code into an executable file which is then executed.

The data generated by the FORTRAN executable is written in a text file. Various
Matlab scripts are run on this data to plot the figure. This figure is then inserted into the document as a figure object (Figure 13).

CONCLUSIONS

The emergence of new technologies and the need for integration of old and reliable tools into a format which makes them more useful to a wider spectrum of users requires building environments which may provide easily extendible and modifiable frameworks. The use of a document as user interface has been in use for quite some time now.\textsuperscript{20,21} The document/browser paradigm is becoming more popular day by day, not only as seen in the context of the world wide web and Internet applications but also as a user interface to one of the major operating systems.

By building a document centred environment with a component architecture and XML mark-up scheme we have put to the test the available technology. We have suggested methods of keeping Java applications platform neutral yet enabling them to integrate tools and applications without relying on native methods interfaces, which would render Java applications tied to a specific platform.

The emergence of XML is being talked up as the latest greatest thing.\textsuperscript{22} The ability of XML to represent object states in a document has been combined with behaviour in the environment, thus allowing creation persistent object stores in document format. The resultant environment has provided a framework for documenting the processes.
Mixing and Separating of Viscoelastic Fluids

In this problem fluid enters from two opposite inlet channels, which are separated by thin wall with the counter of the channels to allow mixing of the fluid. The domain is divided in regular and mapped by tessellation based in centre of domain. Known parabolic velocity profile for axial velocity component, component and stress boundary conditions are prescribed at the inlet. Whilst pressure datums at outlet walls. To model this complex flow of viscoelastic fluid a shear thinning Plan Tien Tanner model for equation and semi implicit Streamline-Petrov-Galerkin Pressure Correction scheme has been chosen.

We need to run various Matlab scripts on the output.dmp file to filter out the data and vectors.

Figure below above velocity vectors displayed using Quesa Plot function of Matlab

Figure 5. First page of the environment incorporating the CFD problem solver.

and creating integrated suits with ease as a product of creating the document.

To illustrate the flexibility of our approach we have taken examples from two very different fields of engineering and constructed appropriate design environments.

ACKNOWLEDGEMENT

We would like to thank Dr. A. Baloch, Senior Research Officer, Institute of Non Newtonian Fluid Mechanics, University of Wales Swansea, for providing us with the CFD code used as one of the examples.

REFERENCES


[8]. Deborah Hix and H. Rex Harton, Developing User Interfaces: Ensuring Usability Through Product and Progress, John Wiley & Sons, inc., 1993


[10]. Needs title, author (if any) Institution. WWW.microsoft.com/standard/xml/xmlparser.htm


[18]. Reference for benchmark problem itself rather than this technical report is preferable.


