TOWARDS A DOCUMENT CENTRED COMPUTER-AIDED CONTROL SYSTEM DESIGN ENVIRONMENT

Al Sadiq ul Amin M. Madani Halepota ∗∗ Christopher P. Jobling ∗ and Philip W. Grant ∗

∗ Centre for Communications and Software Technologies, University of Wales, Swansea, Singleton Park, Swansea SA2 8PP, UK. Email: C.P.Jobling@Swansea.ac.uk
∗∗ Topline Data Limited, Wiltshire, UK.

Abstract. Most conventional design processes produce documentation as a by-product: design is done within an environment or collection of tools and the design outputs, models, diagrams, results, etc are gathered together along with commentary (usually with hindsight) into some kind of document at the end of the process. Inspired by ideas taken from literate programming and notebook interfaces to design tools, a prototype document centred design document has been developed. In such an environment, the document is the main product and the design is a by-product of documenting the design. By implementing the framework components of the system in Java and by using the extended markup language (XML) to encode the contents of the documents, the system is made platform neutral and therefore portable. An example document centred environment for computer-aided control system design (CACSD), that integrates MATLAB and SIMULINK with the document-based framework, is briefly described and indications for future possibilities for this technology are given.

Keywords. Computer-aided control system design, computer-aided engineering, document centred design environments, literate programming, Java programming language, framework reference model, markup languages.

1. INTRODUCTION

One of the earliest applications of computers was to help with the creation of documents. Indeed, document creation is arguably the most important application of computers today. Computers have also always acted as a general purpose calculating tool in the support of the work of design. A by-product of many design studies is a design document and indeed, many quality assurance procedures require documentation of each step to enable traceability of design decisions and to provide a record of the components that make up the designed artefact. However, it has been argued that the “view that documentation is something added to [a design artefact] after it has been commissioned seems to be wrong in principle, and counter productive in practice. Instead documentation must be regarded as an integral part of the process of design...” (Hoare, 1973). Knuth, taking the idea further, introduces the idea of literate programming, a style of documentation in which the design is embedded in the documentation, with the words “instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to humans what we want the computer to do” (Knuth, 1984).

Most design documentation is separate from the system it documents: we call it design centred documentation. To achieve the paradigm shift suggested by Hoare and Knuth requires document centred design, for which the document would also be the design (Halepota et al., 1998).

In this paper, we survey the tools and environments that have been developed to support CACSD. In Section 3 we introduce document centred environments and in Section 4 we look at the so-called CACSD reference model and discuss its relationships to a document centred environment. Finally in Section 5 we describe the implementation of a prototype document centred environment for CACSD that was implemented.
as part of Madani Halepota’s PhD thesis work (Madani Halepota, 2000). In the conclusions, we give some indications of how this work could be taken forward.

2. TOOLS AND ENVIRONMENTS FOR CACSD

The computer-aided control system design (CACSD) community has long sought the “holy grail” of the perfect environment for control systems design. The earliest environments were little more than a set of library functions loosely coupled with a menu-driven text-based user interface and exchanging information via common blocks of memory (Munro, 1972; MacFarlane, 1973).

The emergence of command-shell driven interfaces popularized by the UNIX operating system and the personal computer, inspired Cleve Moler to adopt a similar interface for the original “matrix laboratory” [Matlab] (Moler, 1980). Matlab itself was rapidly adopted and adapted by the control systems design community and several forerunners of modern Matlab appeared in the mid 1980s. Indeed, Rimvall (1987) makes an extremely convincing case for the command language interface and the matrix data structures in control systems design software.

In the late 1980s, the focus of the CACSD research community changed to the integration of groups of tools, e.g. systems modelling, simulation, analysis, and code generation. A number of environments were proposed which attempted to achieve this integration with computer graphics (Barker et al., 1988), 4th-generation programming tools (Munro and Jobling, 1994), expert system shells (Boyle, 1994), and menu and forms based user interfaces (Rimvall et al., 1989). The key drivers for these developments were the emergence of the windows-based graphical user interfaces. The commercial vendors of CACSD tools produced versions of the Matlab derivatives (including a new version of Matlab itself) that fully exploited the capabilities of these interfaces, and (apparently) gave the end-user community of practicing control engineers access to the tools that the researchers could only demonstrate in research prototypes. (One major effect of this consolidation was that research funding for integration research largely dried up in the early 1990s.)

Today, no one would dispute that the market leader MATLAB, now its 6th edition, deserves its place as the essential weapon in the control engineer’s armory. MATLAB provides powerful task management and data integration services through its programmable scripting language and built-in support for multi-dimensional arrays, records and objects. The availability of sophisticated (user programmable) user interface services (implemented in Java), and a communications services infrastructure that allows it to participate, both as client and server, in integrations with other tools. Modelling services provided by the modern Control System Tool-

An important feature of the environments cited above, was their built-in support for data persistence—the automatic storage of data objects to ensure their existence beyond the running of an interactive session. Such repository services are still not provided by the MATLAB (except through the manual method of loading and saving the current workspace).

3. DOCUMENT CENTRED SYSTEMS

The inspirations for a document centred environment for CACSD were Knuth’s literate programming (cited in the introduction) and the “notebook systems” exemplified by CaminoReal (Arnon et al., 1988), claimed to be the first system to make the integration of documents and computation central to its operation. CaminoReal was an early notebook interface to a computer algebra system that was developed as part of early research on document architectures performed at Xerox PARC. It provided a user interface based on a multimedia editor through which mathematical functions could be evaluated and the results displayed in a natural manner. The notebook interface provided by Mathematica is a natural successor to this early system and is largely similar to the interfaces provided in Mathcad and to MATLAB (via Microsoft Word).

Using a notebook system, the user creates word processed documents with embedded code that is executed by the back-end processor. The results of computations are returned as formatted outputs or graphical objects. These notebooks are active in the sense that any change in the document can be re-evaluated in the back-end and updates the documented design. However, they are limited in that the work with only one back-end system, and are portable only to a limited range of computing platforms.

Another pioneering system was Designer's Notepad (Haddley and Sommerville, 1990) which used a notepad metaphor in an attempt to provide the means of annotating the outcomes of conceptual design, design evaluation and detailed design in the systems engineering context. Although not a tool integration mechanism, it demonstrated that a documentation paradigm can be used to support and capture the outcomes of a design process. A significant example of the use of a document editor as an extensible framework for creating systems is the tool Write (Szyperski, 1992) which is built upon, and implemented in the object-oriented programming system Oberon. Write allows its user to document extensions to the Oberon system as the extensions are created. Another system of historical interest is HyperCASE an example of an hypertext based software engineering environment that used features adopted from artificial intelligence and hypertext to
support the creation of self-consistent system design documents throughout a system lifecycle (Cybulski and Reed, 1992).

For the purposes of the work presented here, the development of most direct relevance has been in the compound document software technologies that provide the building blocks for so-called active documents. Active documents are the class of documents that have some sort of active content (English et al., 1990): for example, they may take the form of a document containing the block diagram of a system that when activated becomes a simulation which itself is linked to a time response shown as an embedded figure. Techniques for creating such systems in a platform dependent way, e.g. Microsoft’s ActiveX component architecture, have been available for some time. A recent attempt to create a platform neutral framework for such compound documents was called OpenDoc which, though largely unsuccessful, was a major inspiration for the prototype described later.

4. DOCUMENT CENTRED INTEGRATION ENVIRONMENTS

As a result of an integration effort in the field of computer-aided software engineering, a reference model for computer-aided software engineering (CASE) was developed (ECM, 1990). The so-called “toaster model”, adapted from the CASE reference model for computer-aided control systems design (CASCD) by Barker et al. (1991) is illustrated in Figure 1. The diagram indicates that a CACSD environment should provide shared front-end (user interface and task management) and back-end (modelling/data integration and repository) services and communication services for the tools that use the infrastructure.

![Diagram](image)

Fig. 1. A reference model for CACSD also known as the “Toaster Model”

Taking the CASE framework reference model as a starting point, Wasserman (1990) has described integration in detail using the following headings:

- **Platform Integration** which is associated with operating systems services and communication services.
- **Presentation Integration** which is associated with user interface services.
- **Data Integration** which is associated with model and data integration services and repository services.
- **Control Integration** associated with task management services.
- **Process Integration** which is concerned with how the components work together during the design process.

In the following section we summarize the implications of these features of an integrated environment and how they have been satisfied in the document centred design environment for CACSD. Much more detail is to be found in Madani Halepota (2000).

4.1 Platform Integration

Platform integration is essential for leveraging the available tool sets without being tied to a single platform. It should allow tools to be inter-operable. In the prototype, platform integration has been achieved by implementing most of the document centered environment in the Java programming language which can then be executed on any platform that supports a Java Virtual Machine.

4.2 Presentation Integration

Often the stated aim of presentation integration is to provide a consistent “look and feel” for all the tools in an environment. A consistent look and feel makes it easier for the users to learn a new application by building on their previous experience in the environment. Java provides support for switchable look and feel even on the same platform so this issue is easily addressed. At an environment level, the document metaphor is a natural user interface paradigm: tools simply have to present themselves as parts of a document. Access to the tools is achieved by selection of part of the document (with mouse or keyboard) and activation either directly within the document, or indirectly by opening a separate window. Familiar visual cues, such as underlined text for hyperlinks and consistent changes of font size, style and colour for special sections of the document, further reinforce the document centred view of the design process and complete the user experience.

4.3 Data Integration

Data integration allows tools to share information. Tools produce and process data. Unless data produced by one tool is made available to the other tools to follow in the design process, the environment cannot fulfil its requirement to support the design process. Tool integration in an environment requires that tools can share
the data and the environment can manage the relationship among data objects produced by the tools.

Electronic documents have long used mark-up schemes to distinguish between different parts and to describe the attributes of their content. The Standard Generalized Mark-up Language (SGML) (ISO, 1986) has been in use for a long time. Its well known subset, the Hyper Text Mark-up Language, has been one of driving factors behind the World Wide Web, and more recently the eXtensible Mark-up Language (XML) (Bradley, 2000) has been formalised to help with the issues of arbitrary data content and meta-data description of the electronic documents.

Using XML for document mark-up allows tool designers and environment extenders to describe not only the attributes of the tools but also to provide a description of the data which the tools must manipulate.

Data persistence in a document centred environment is simply a matter of saving the current document: the states of the components embedded in the document are converted to suitable XML tags and attributes and the result is simply a marked-up text file!

5. A PROTOTYPE DOCUMENT CENTRED DESIGN ENVIRONMENT FOR CACSD

This section describes various aspects of the realization of a prototype document centred CACSD environment which can be executed on all platforms, without recompilation, with a facility for easy extension and modification.

The document paradigm is the linchpin of the design, from the user interface of the environment to its repository. The whole environment is implemented using the Java Programming Language. A document type definition is used to describe the structure of the documents and documents are stored in XML. The documents produced by the environment act as the integrators of tools which aid in the problem solving process in the control system design domain.

5.1 Implementation of the environment

To encourage users to visualize the process of design 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 or as operating upon 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 the document format.

The user interface of the environment is a document editor. A visible area on the screen is provided by the document editor, 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. Each component must have the ability to integrate itself into 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 contents as scripts and invoking the tool’s execution engine.

The main component of the environment is a Java class which mediates between other objects in the environment and also provides users with the menu bar and various menu items for initiating events. At initial startup, a text file is loaded which provides two resources: firstly it supplies labels for the 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 being able to instantiate objects at run time. As the menu initialization class reads the menu labels, the environ-
ment class instantiates the required components (implemented as JavaBeans). It is easy to extend the environment by simply editing the file or by loading additional classes from the menus.

The environment consists of two basic types of components. The first type of component acts on the document in the context of the environment and performs actions such as loading and saving of the document. All other service providing classes designed to perform actions in the context of the document and include components that format paragraphs, headings, figures or otherwise provide an interface matching for the integration of legacy applications and tools. These components are represented in an XML DTD as document elements.

Components are able to register and un-register their interest in various events generated in the environment, can participate in parent-child relationships with other components, and have attributes that can be edited via dialogue boxes.

5.2 Document loading and saving

On loading a previously created design document along with its DTD, the environment uses an XML parser to create a tree of the document. Each element of the document forces its related component to be loaded. Each component uses the attributes extracted from the XML attributes to re-create the state of the object as preserved in the document. Each document element is given a visible space on the document according to its content. An internal data structure keeps track of this space and allows interaction with it via the mouse and keyboard.

When a document is saved, the current state of all the contained objects is translated back into elements and attributes and saved as an XML tagged text file.

5.3 Integrating MATLAB and SIMULINK

To demonstrate the applicability of a document centred design approach in the context of computer aided control system design, components have been created which facilitate the integration of MATLAB and SIMULINK into a design document. These elements communicate with MATLAB from the document environment and provide a mechanism for conveying user commands to MATLAB and rendering results obtained from MATLAB into the document. Placing MATLAB and SIMULINK components together with other document formatting components creates a design document and a design environment simultaneously.

When a MATLAB element occurs in a document, a component providing communication with MATLAB is instantiated. This component passes the commands embedded in the matlab element to MATLAB and stores the results returned in the document. To achieve platform neutrality, simple file access is used for the communication and return of results.

SIMULINK is integrated into the document environment in a slightly different way. A graphical representation of a SIMULINK model is shown in the document which, when activated, calls MATLAB with instructions to load the SIMULINK model and initialize its parameters. The model is manipulated in a separate window and when the interaction is complete the graphic is updated (if necessary) and any results returned to the document. The user interface for a typical CACSD design document is shown in Figure 5.3.

6. CONCLUSIONS

The prototype CACSD environment described above was implemented about three years ago and was necessarily limited by the tools and technologies available at that time. Since then, technologies, particularly in presentation, task management and communication services have developed significantly.

An implementer of a document centred design environment starting work today could make use of modern web browser technology to remove the need for the environment to handle document display details. For example, modern browsers support the extended style language transformation (XSLT) standard and cascading style sheets, and this means that documentation layout and display can be done directly from the XML code. The developer simply delegates these actions to the browser. Similarly, the document object model allows the manipulation of the internal tree structure to be manipulated by the browser’s built-in scripting language. Component and communication infrastructures provided by enterprise editions of Java (and in the near future Microsoft’s .NET) should simplify the integration of external tools.
The unanswered question is whether such document design is actually beneficial to the control systems designer. To answer this question, we will need to build a more robust version and try it out. This said, we believe that the small advances made in the pioneering work described here indicate that this would be a worthwhile activity.

7. REFERENCES


