The Direct Manipulation Shell: Creating Extensible Display Page Editors

Michael E. Allen, Michael Christiansen
SSC Laboratory *
2550 Beckleymeade Ave.
Dallas, Texas 75237

Abstract

Accelerator controls systems provide parameter display pages which allow the operator to monitor and manipulate selected control points in the system. Display pages are generally implemented as either hand-crafted, purpose-built programs; or by using a specialized display page layout tool. These two methods of display page development exhibit the classic trade-off between functionality vs. ease of implementation. In the Direct Manipulation Shell we approach the process of developing a display page in a manifestly object-oriented manner. This is done by providing a general framework for interactively instantiating and manipulating display objects.

I. Introduction

We are developing a tool, known as the Direct Manipulation Shell (DMS), which will allow the construction of software applications in much the same way as modern hardware devices are constructed. That is, the user selects and combines together software components in an interactive, plug-and-play fashion when developing their applications software. DMS provides an environment in which software components are provided and directly manipulated (hence Direct Manipulation Shell) by the user.

A programming environment developed through DMS contains software components which address the needs of a single problem domain, e.g., accelerator parameter page development. This can be contrasted to a traditional programming environment containing compilers, linkers, editors, etc., which support no specific problem domain and provide no domain specific support for applications development. Using DMS, the user performing the applications programming spends most of his time browsing catalogs of domain specific components rather than developing algorithms and data structures. It is assumed that this applications programmer is knowledgeable in the domain supported by the specific environment, not necessarily in the domain of the computer sciences.

The goal of DMS is to provide users with a software development environment in which they construct solutions in problem domains about which they are concerned and knowledgeable. These domain experts are provided components that are presented and manipulated through terms and concepts found in this problem domain. Using the facilities provided by DMS, programming experts develop a set of interrelated software components which can be used to construct solutions to problems in this domain. This process of constructing a programming environment through DMS is similar to the process of constructing an expert system [2] using an expert systems development shell. When developing an expert system, a team of programmers and domain experts combine efforts to develop a set of rules which address problems in a specific problem domain. With DMS, a team of programming and problem domain experts construct a set of software components which can be accessed and manipulated through DMS. In either case the user is provided with an environment which can be applied to problem solving with little understanding of the underlying computing environment.

Of course, theses goals are not unique to DMS. Basically, DMS provides an interactive, interpreted, object-oriented, programming environment. Usually, such environments have the following major shortcomings:

1. performance.
2. availability of third party, off-the-shelf "components".
3. performance.

For an accelerator control application (1) and (3) (and to a lesser extent (2)) can be killers. The DMS environment is designed to specifically address these limitations. This is achieved in the current version of the DMS tool via the use of a modified Common Lisp [5] interpreter, XLisp [1].

XLisp has incorporate within it an object-oriented language constructs which allow classes to be defined and instances of XLisp object to be created. Our modifications to XLisp enable the user to interactively create and manipulate instances of XLisp objects which in turn create and manipulate instances of C++ objects. This is much more than just a foreign function interface, because the objects thus created are now managed by the DMS environment. This means, for example, that much of the memory management is taken care of automatically (garbage collection). Additionally, DMS knows about C++ data structures, so that unmodified C++ code can be linked directly into the DMS environment. Additionally, because of the object oriented extensions on the Lisp side, one can write straight forward Lisp code without continually bothering about how data is represented.

Within DMS one can move freely between the Lisp and C++ environments, taking advantage of the best features of both. In particular, one may take advantage of the speed and availability of C++ class libraries within an interactive Lisp programming environment. We have, for example,

*Operated by the Universities Research Association, Inc., for the U.S. Department of Energy under Contract No. DE-AC02-89ER40486

incorporated both the GNU [7] and InterViews [8] class libraries into a development environment for user interface development. The process of incorporating, or linking in, new C++ classes is discussed in some detail in a later section. It is important to note, however, that no modifications to the C++ code is required. In fact, one does not even need access to the source code in order to integrate a C++ class library into the DMS environment.

Another reason for integrating the Lisp programming language into the DMS tools is the opportunity to apply expert system, logic programming, and other knowledge based technologies in the development of domain specific development environments. Not only does the DMS tool support the interactive construction of software solutions, but embedded rules and constraints systems can guide the software developer in the correct manipulation and combination of software components into a needed solution.

II. The C++-XLisp interface

The connection between the XLisp and C++ programming environments is accomplished by providing an interface between the Lisp and C++ run-time environments. In general, a method call on a XLisp object is translated into a call on a C++ object's method. This translation is accomplished by a C++ function, called the Interface Function, which is generated specifically for the purpose of providing an interface between the Lisp and C++ environments.

Each C++ class which is imported into the XLisp environment is interfaced to DMS through an XLisp class. An XLisp class which imports and makes available a C++ class is called an "import class". Each import class duplicates the set of methods that the C++ class provides. When an instance of an import class is created, the constructor method for the import class constructs and makes available an instance of its corresponding C++ class. An instance of an import class is called an "interface object" and each interface object maintains and provides an interface to a single instance of a C++ object.

An Interface Function is created specifically for each class/method combination and is responsible for translating the Lisp arguments provided to the XLisp method call into C++ arguments which are passed to the C++ method call. The Interface Function also translates the value returned by the C++ method into a Lisp variable which is returned as the result of the XLisp method call on the interface object.

When an instance of a XLisp Interface Object is created through the interaction of the user with the XLisp programming environment, an XLisp constructor method calls an "interface function" which creates an instance of the imported C++ class. The pointer to this new C++ object is returned by the interface function and assigned to a pointer instance variable maintained by the interface object. This C++ pointer is then used as a target for all future interaction which occurs between the interface object and the C++ object it maintains.

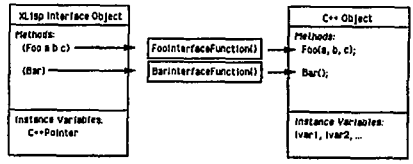

Figure 1: Interface between XLisp import object and C++ object.

The C++ pointer maintained by the interface Object to its C++ object instance can be manipulated and passed as an argument in other XLisp method calls. Thus, the address, or location, of a C++ object can be passed to other C++ objects. In this way direct interaction between C++ objects can established once references between these C++ objects have been passed through their interface objects. The importance of this direct interaction lies it the fact that once established, directly interacting C++ objects can execute method calls with the speed of compile code in a traditional, statically linked, programming environment. Solutions to problems (application programs) are then composed utilizing C++ objects which have been created and manipulated through their Interface Object instances.

III. Adding Interfaces For New C++-Classes to DMS

Integrating a new C++ class into the DMS image refers to the process of integrating the class data structure and its methods into the DMS process (or image). This integration can be accomplished either statically or dynamically. Static integration is implemented by simply linking the compiled object code which implements the class methods, data structures, and interface functions into the DMS image at link time. Dynamic integration is accomplished using a public domain library called Dld [4,3] which provides the ability to dynamically load and relocate object code into an executing image at run-time. * The Interface Functions needed for each method provided by an interface object are generated automatically using the development environment provided by the DMS tool.

The DMS development environment described in the above figure provides the ability to integrate new C++ classes into a DMS supported programming environment. The user wishing to integrate a new C++ class provides a description of the class and its methods in a form very similar to the typical C++ header declaration. This description is parsed for errors and is translated into a XLisp class declaration and a set of interface functions for each method in the C++ description.

*Dynamic integration is currently only offered in the Sun OS environment.

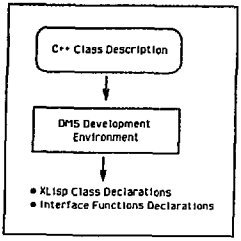

Figure 2: Incorporating C++ classes into the DMS environment.

In the case of the statically integrated DMS tool, the interface function declarations are compiled using the native C++ code development tools into an object code format. This object code is linked with the DMS object code, along with the C++ methods to generate the desired DMS programming environment. Also generated from the C++ class description are a set of XLisp class declaration which implement the import classes described above. These XLisp class declarations are loaded into a DMS environment when the tool is started.

IV. Concerns and Future Directions

As noted, DMS is currently implemented as an extension to the XLisp programming environment. DMS runs on on a variety of Unix workstations, and requires the X-Window system to support its user interface. Also integrated with the tool is a text editor which allows the development and execution of XLisp programs in a mouse driven editing environment. Within the editor the user is able to write XLisp source code with such capabilities as parenthesis matching, multiple widows, and text select, cut, and paste operations supported through the mouse interface. XLisp source code can be interactively executed directly from within the editor by selecting and evaluating the code of interest through the mouse interface. This feature is similar to Smalltalk's Workspace object, and allows work sessions to be saved and restored.

The primary programming environment to have been developed is an X-Window user interface development environment based on the InterViews [8] C++ class library and our extensions to that library, called *glistk* [6]. Using this environment the user is able to interactively create and exercise user interfaces implemented as InterViews and glistk objects. In addition, glistk's provide an underlying inter-object communication mechanism that has also been extended into the XLisp environment. Furthermore, there are also XLisp PushButtons and other Lisp based interactive objects. These Lisp based objects allow the development of higher level functionality within the XLisp programming environment. In particular, general Lisp expressions can be executed when these objects are selected.

There are two main concerns about the current DMS environment. The first is that XLisp, while providing an object-oriented interface to Lisp, is not standard. We would like to move DMS to a CLOS and Common Lisp environment. This would make accessible the rich and robust environments and tools available with commercial Lisp implementations. We are currently evaluating a few different options.

The second concern is actually more serious. Developing and maintaining a DMS programming environment is a reasonably complex process. When several dozen C++ classes are to be integrated into an environment, house keeping and version control become complex and error prone. Further, developing a new environment sometimes requires a in-depth understanding of XLisp internals. In the current version, integrating a new class involves several processing stages which could be combined into a few simpler steps.

This situation could be improved in a couple of ways. First, it should be possible to generate import classes and interface functions automatically from C++ header files. Another future enhancement would be to eliminate the need to generate and link Interface Functions for each method provided by an interface object. This might be accomplished using a byte code interpreter which interprets a set of byte codes describing the types of arguments expected for a method call and which uses these codes to translate Lisp to C++ arguments and then performs the C++ method call.

References

[1] D. M. Bentz. XLisp: An object-oriented lisp. version 2.0. Available as public domain software., 1988.

[2] F. Hayes-Roth, D. Waterman, and D. Lenat. *Building Expert Systems.* Addison-Wesley, Reading, Massachusetts, 1983.

[3] W. Wilson Ho. An Approach to Genuine Dynamic Linking. Available as Technical Report.

[4] W. Wilson Ho. Dld: A Dynamic Link/Unlink Editor, Version 3.2.3. Available as technical report.

[5] G. L. Steele Jr. *Common Lisp: The Language.* Digital Press, Bedford, Massachusetts, 1984.

[6] M. Kane, C. Saltmarsh, V. Paxson, M. Allen, and P. Veals. The GLISTK Manual. Available as internal technical memorandum.

[7] D. Lea. The GNU C++ Library. Available from the Free Software Foundation.

[8] M. A. Linton, J. M. Vlissedes, and P.R.Calder. Composing User Interfaces with InterViews. *IEEE Computer*, 22(2), Feb. 1988.