University of Waterloo
Department of Computer Science
Waterloo, Ontario, Canada

Technical Report Series

CS-93-17

Towards CAAI: Computer Assisted Application Integration

by

D.D. Cowan C.J.P. Lucena R.G. Veitch

October, 1993

Towards CAAI: Computer Assisted Application Integration

D.D. Cowan R.G. Veitch
Computer Science Department & Computer Systems Group
University of Waterloo
Waterloo, Ontario
Canada
N2L 3G1
dcowan@watcsg.uwaterloo.ca

C.J.P. Lucena
Departamento de Informática
Pontifícia Universidade Católica
Rio de Janeiro, 22453-900, RJ,
Brazil
lucena@inf.puc-rio.br

October 7, 1993

Abstract

As the manipulation and use of information forms the base of our economic structure, knowledge workers will become software application integrators. That is, knowledge workers who are experts in their specific domain and its related software, will need to "glue" together databases, analysis, simulation, modelling and visualization tools in order to produce timely information for their organization. In order to perform application integration easily we first need to define a coherent supporting architecture and a programming model. This paper outline this architecture and the programming model and indicates that many of the components to implement this model are available. Unfortunately, the design of many of these components is not consistent, thus, making it difficult for a knowledge worker to integrate applications without a large amount of "inside" technical information. Through the definition of these models we have highlighted the problems that need to be resolved. The architecture and programming model described in this paper are based on a large number of experimental systems developed as part of our research program.

1 Introduction

As we move more closely to an information and symbolic economy, most of the population will be primarily knowledge workers manipulating and using information, and only a small percentage will be employed directly in the production of manufactured goods and agricultural products [Tof91]. These knowledge workers who will be experts in their own specific application domains, will re-

quire computer systems with access to databases[1] and various analysis, simulation, modelling and visualization tools in order to produce the type of timely information which will be essential to the successful corporation, government and public or private interest group. Part of the knowledge worker's or domain expert's toolkit will be the ability to use novel and timely approaches in the manipulation of information to provide a specific group with a competitive advantage. Such methods will often require new ways of viewing and processing information. Thus, some knowledge workers will become **application integrators** in that they will use their understanding of software and programming in their specific domain of expertise to create domain-specific applications. These approaches will require that software components be combined in ways that could not be anticipated when the components were designed. Successful knowledge workers will require access to "instant" software which will be developed by the application integrator as a specific need is perceived. There will not be time to wait for the lengthy design and implementation cycle prevalent in many organizations, rather a prototype cycle [B+92] will be followed as the software system will in all probability be constantly evolving.

The need for "instant" software imposes certain constraints on application integrators and on the underlying software system they use to support their software building activities. Application integrators must have access to a repository of predefined domain-specific large software components which can be combined together in new configurations. They will not create these predefined components themselves, since in general such components will require specialized knowledge of both their application programming interface (API) design and internal structure. The repository must contain graphical user interface (GUI) components which will allow the application integrator to interact easily with software components and to display results of computer processing. In the extreme, such displays could be in an interactive multimedia format. The application integrator must have access to software environments and technologies which will allow the predefined components to be combined easily into new software systems. In this paper we call these software technologies "glue" technologies, because they are primarily used to glue or bind together reusable application and user interface components. These various components must be supported by a software development environment suitable for application integration, a computer assisted application integration (CAAI) workbench.

Many of the modern concepts in programming which involve: abstract data types (ADTs), object-oriented programming, graphical user interfaces (GUIs), event driven programming, servers such as databases and their corresponding clients, and script and macro languages, can be used by application integrators in implementing applications. Before these can be combined, a generic framework or applications architecture is required to provide a disciplined approach to this type of software engineering.

Smalltalk [Kay69, GR83] was one of the first attempts to provide a software technology to allow the creation of powerful personal information systems. Smalltalk has not been broadly accepted by groups such as application integrators, at least partly because of the overhead in mastering its complexity. However, Smalltalk has made significant contributions to many of the programming concepts which could make widespread application integration feasible.

[1] The term databases will be used in this paper to describe any collection containing entities such as structured data, text, knowledge, pictures, graphics and sound.

Current glue technologies are usually based on a programming language[2] and range from the shell languages prevalent on Unix systems to many of the new micro-based "object-oriented" systems such as Hypercard [Goo90], Toolbook [Asy89], Visual Basic [Cor91], VX·REXX [VRe93], or ARexx [Haw87, ZS91]. The micro-based technologies often allow the use of applications and the construction of user interfaces. Future glue may be based on more visual systems such as those represented by VX·REXX [VRe93], NeXTstep on the NeXT computer [NeX91], or as shown in the work on demonstrational interfaces [Mye91]. We expect that the glue technologies will be based on the programming language paradigm for quite a length of time as we make progress toward a more friendly method of combining software components. Effective application integrators will have to become programmers for the foreseeable future because they will combine reusable software components using methods based rather loosely on a programming language approach. In previous papers [CIS92, LCP92, CILS93a, CILS93b] we have described design and implementation models for factoring user interfaces which we believe makes them more reusable. In this paper we examine an underlying architecture, language structures and environment required to support the construction of a wide range of applications.

2 Sample Applications

The requirements for an architecture are normally defined by the classes of applications to be supported. This section describes two applications which might be considered typical and which provide the background for subsequent discussion about the architecture. These specific examples are representative of classes of applications which have been identified through a substantial amount of observation and experimentation.

2.1 A Database Application

The government of a city has databases containing information about the location of toxic substances which are stored by various companies, schools and other organizations within the city boundaries. The databases contain information about location, and attribute information about types of substance, and parameters about the conditions of storage. A new environmental law has been passed which requires the storage of toxic substances to meet certain minimum requirements. The city is required to determine which organizations do not comply with this new law and notify them of the legal requirements within 30 days. Failure to comply by the city or the organization storing the substance within 60 days could lead to fines or other penalties. Obviously an application that determines violators of the new law and notifies them must be created in a timely fashion. The application is outlined in the next few paragraphs and a pictorial description of the components and their relationship is presented in Figure 1. Working applications similar to the one described here have been created in our laboratory by one or two "application integrators" within time periods of a few hours to a week.

The location of the toxic substance is stored in a geographic information system (GIS) while a corresponding relational database contains attributes such as the address of the corresponding

[2] We use the term *programming language* with the more traditional meaning of a text to be interpreted or compiled.

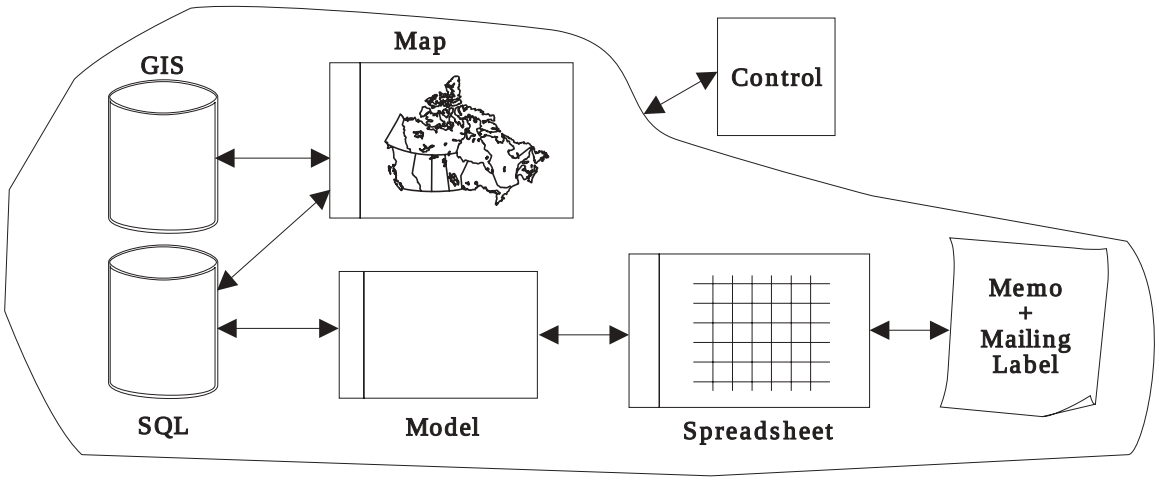

Figure 1: A Sample Database Application

property, the name of the occupant, and the toxic substance storage conditions. The location of each toxic substance is displayed on a map and accessed through a map interface. A modelling program uses the storage conditions to show how the toxic substance will disperse in the soil if the substance is spilled and is used to determine if the legal requirements are being met. The spreadsheet is used to display the data produced by the modelling program to the city engineer so that cases which are close to violating the law can be examined more carefully. The spreadsheet is configured to mark any data where there are any doubts about the answers on dispersion. Finally the word processor is used to produce letters from the database which are addressed to the legal occupant of the property. A control program with an appropriate user interface performs the following functions:

- links the map interface to the GIS database.
- uses the property number from the GIS database to find the owner of the property and the soil type in the relational database
- directs the soil type to the modelling program
- directs the output of the modelling program to the spreadsheet
- directs the name and address of the occupant to the word processor
- directs the data from the spreadsheet to the word processor

In this case the complete application consists of a number of components which are transferring information and/or commands among themselves. The entire process is coordinated by a control program. The components can be categorized into three different forms:

- a component which manages data and data structures but which is not normally considered to have any input/output functions. The GIS and relational database fall into this category. In subsequent presentation such a component is called a “server”.

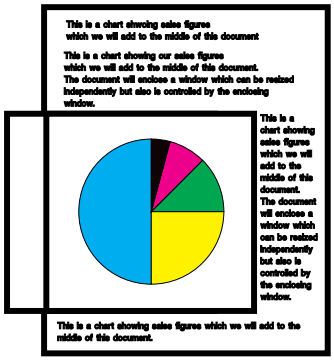

Figure 2: An Application with Embedded Documents

- a component which is a graphical user interface or GUI and does not normally store and manage data. The map interface would be in this category. This component is called an "abstract data view" or "ADV" and has been described in detail in [CILS93a, CILS93b].
- a component which combines both the previous categories, it stores and manages data and also has a graphical user interface which allows direct manipulation of the data. This combined component is called an "application".

2.2 A Document Application

Some applications require that documents be embedded inside each other. For example, a chart showing sales figures might be contained in a report. Most systems support cut-and-paste methods whereby a static copy of the chart is copied into the report. Opening a window in the document which would contain the chart displaying the data would be more appropriate. The window would close and open with the enclosing document, and resizing the enclosing document or the enclosed window would cause automatic reconfiguration of both windows. Also the chart would always be current since it would represent the real data. A printed version of the report would contain the latest static copy of the chart. Such an application is illustrated in Figure 2.

This particular application could be viewed as an interaction between a word processor and a spreadsheet with a chart drawing capability. The data structure supported by the word processor must contain information about the location of the chart in the document, but the primary interaction is between the user interfaces or ADVs of the two applications. The implication is that there should be some communication between the two ADVs, so that they can synchronize their interaction. Such a structure is described in [CILS93b].

3 Software Architecture

The sample applications presented in the Section 2 can be viewed as being decomposed into a number of interacting elementary components. The components transfer data among themselves and control may reside in one or more components at any one time. There may be a master

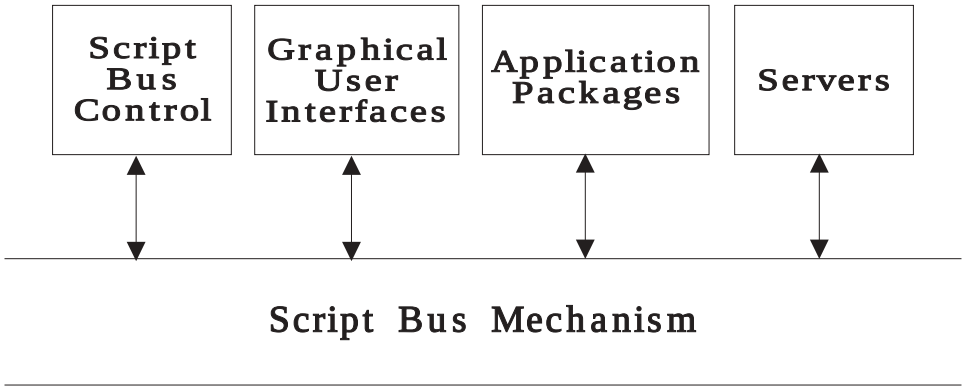

Figure 3: The Proposed Software Architecture

control such as the single component labelled "Control" in Figure 1 or control may be distributed among components as described for the example in Figure 2. If there is a conflict for control then some form of arbitration may be required to resolve the issue. The configuration of the elemental components and the data transfer and control are illustrated in Figure 3. Transfer of data and control are depicted by the "Script Bus Mechanism" and control of the entire application is shown by the box labelled "Script Bus Control". Although control is depicted explicitly in this diagram, it may be distributed over the components. Details of the other components are provided in the next few paragraphs.

There are two basic components: graphical user interface (GUI) elements such as the map interface, and data structure managers such as databases. The GUI elements are just input/output mechanisms; they display data stored in some data structures such as a database and they also allow the user to interact with this data. The data structure managers which are called servers in the rest of this paper, do not have any input/output mechanisms although they provide an interface which allows external agents to query and/or change their state. A server may also provide a notification mechanism so that an external agent may determine that the server's state has changed. The GUI component is linked to one or more servers to implement the input/output functions for a specific application. The method used to achieve the clean separation of a user interface from a server is explained in Section 5.

Application packages such as word processors and spreadsheets are usually integrated combinations of servers and user interfaces. These packages contain both data structures and user interfaces and are configured for a specific class of applications. Quite often the design of an application package allows the user to modify or specialize its behavior through an accompanying macro or script language; such techniques are used in both Word for Windows [Mic91b] and Excel [Mic92a].

Each of the elements described in the previous paragraphs is shown attached to the script bus mechanism in Figure 3.

4 Servers

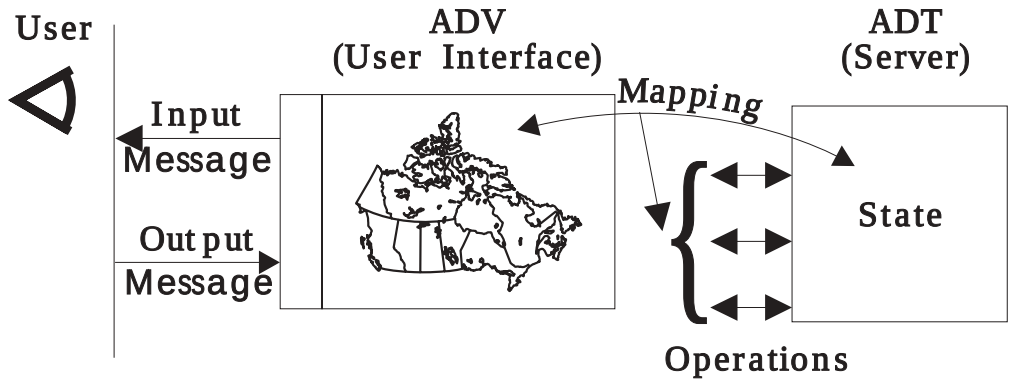

Figure 4: The Structure of a Server and its User Interface

A server is a black box which can have its state queried or changed through an interface. A server may also provide a notification mechanism so that an external agent may determine that its state has changed. The user of the server should know nothing about its internal structure, only the definition provided by the interface. A server can be viewed as a program which manages data structures and responds to commands presented at the interface. From this viewpoint a server can be viewed as an abstract data type (ADT) [Hoa89]. An abstract data type (ADT) is an object in the object-oriented sense of program design. We use the term ADT because we are mostly interested in its properties as a type. An example of an ADT as a server with an accompanying user interface is illustrated in Figure 4. The arrows labelled "mapping" depict the operations to query and change the state of the ADT and to provide an external notification of a change in state.

There are several excellent examples of servers with well-defined interfaces: SQL databases, name servers such as the X.500 directory, communication servers, user agents and mail agents as in the X.400 mail service and X-Windows servers. These types of servers as well as being an ADT have one other overriding characteristic, they do not have a user interface to allow the external user to interact with the ADT interface.

Many potential servers still have a closely integrated graphical user interface (GUI) which makes it difficult to intervene to access the data. Ideally the server should be able to connect to different GUIs, the type of GUI should usually depend on the user's view of the application. Is such clean separation between the server and the GUI possible? Although we have mentioned examples of servers with well-defined interfaces, it should be possible to build all applications in the form of server ADTs. Numerous experiments in our research laboratory in which we have implemented applications such as graph editors, cooperative drawing tools, distributed ray tracers and applications similar to the one in Figure 1 have convinced us that such clean separation is possible. The approach is outlined in Section 5.

5 Graphical User Interfaces – Abstract Data Views (ADV)

Constructing graphical user interfaces (GUIs) that can be used for a broad range of applications and thus are reusable should be possible. For example, the same interface might be used for a drawing

program, or a package to draw graphs, linear graphs or maps, since all the images are constructed from basic syntactic primitives such as rectangles, straight lines and circles. The semantics of the image is a function of the method of drawing the images, the database or server containing the images and their relationships, and the user's interpretation of those images.

A GUI is a sophisticated input/output mechanism in that it allows the user to interact with the data contained in a set of data structures and display the results of manipulating that data. Is it possible to separate the user interface from the server portion or ADT part of an application? In this way we could design classes of reusable user interface components in much the same way that we currently are designing reusable servers. Several GUI architectures which are reviewed in [CCCL93] have been proposed where clean separation is one of the goals. However, in most software designs the GUI becomes strongly connected to the ADT or server and then a clean separation is difficult to achieve. For this purpose we have created a general concept of a GUI that we call an Abstract Data View or ADV [CILS93a, CILS93b].

Abstract data views (ADVs) are ADTs which have been modified to support the design and implementation of general user interfaces and to achieve a separation of concerns by providing a clear separation at the design level between the application as represented by an ADT, and the user interface or ADV.

ADVs are objects in that they have a state, and a functional interface; they have also been extended to support nesting. ADVs are active objects and can be viewed as processes which are activated by external agents. Also an ADV is not an independent object, it must be associated with one or more ADTs. ADVs support a mapping which allows the ADV to query the state of an associated ADT and to change the state of that ADT through its functional interface. This mapping approach illustrated by the labelled arrows in Figure 4 allows controlled access to the state of the ADT, thus preserving the hiding principle [Par72].

In the ADV model, several ADVs can be associated with a single ADT and through a mapping each ADV can provide different control functionality or a different view of the ADT. Since an ADT has no knowledge of input or output, it does not need to refer to any ADV. As a consequence, there is *not* a symmetrical arrangement with the ADT. In the ADV model, the mapping between an ADV and an ADT is represented by the variable **owner**. An ADV instantiation is associated with one and only one ADT instantiation, namely the ADV's owner. The opposite is not true. Because the association relationship comes from the ADV, and not the other way around, the owner variable can be in the ADV for implementation purposes.

The functional interface is not invoked through the usual procedure or function calls but by input messages. Input messages can be triggered by external operations such as input events caused by a keyboard or a mouse. ADVs can produce output messages that use the mapping to query the values of the variables in an accompanying ADT. An output message is sent every time the ADT operations change the state of the ADT. Output messages in terms of a user interface are the display commands which paint various views on the screen. Of course output messages do not change the state of the associated ADT. Figure 4 illustrates these concepts.

In the context of user interfaces and other transformations from one medium to another the input and output messages can be viewed as medium transformers. For example, the input message or event takes a user action and transforms that action into a sequence of ADV operations. The output message or display takes a sequence of ADV operations and transforms them into a viewable

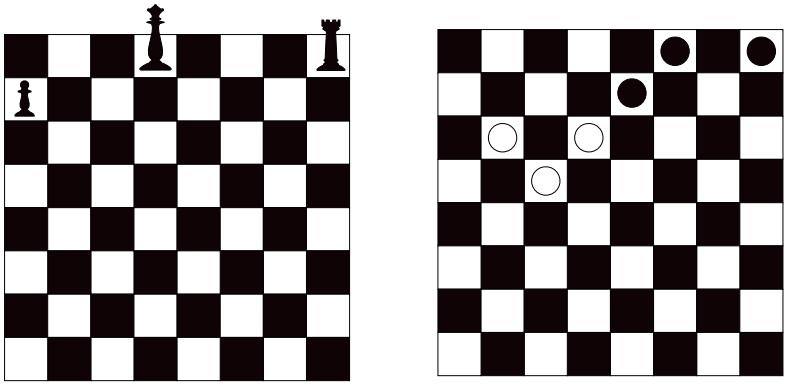

Figure 5: An Illustration of Composition of ADVs

or tactile phenomenon. Roughly speaking, an ADV observes and manipulates the ADT which is its owner; from the point of view of the ADT, the observations are invisible and the manipulations are anonymous.

We should note that some ADVs require such a simple ADT that it would just duplicate internal data structures supported by the ADV. For example, a paint program might be such an ADV.

5.1 Communication Between ADVs and ADTs

Communication between an ADV and an ADT occurs when the ADV executes synchronous invocations such as procedure or function calls, to the ADT [CCCL93]. An ADT never generates events that must be handled asynchronously by the associated ADV. This approach contrasts with other user interface models, where a component must handle both synchronous and asynchronous invocations from other components. Handling asynchronous invocations is considerably more complex than handling synchronous invocations since it requires error-prone mechanisms such as signals, interrupts, or callbacks. With an explicit mapping we enforce a one-way communication, and, as a consequence, have fewer interconnections, thus, ensuring that the role and scope of the interface are defined unambiguously. Asynchrony is needed in other models because the "official" locus of control is in the non-user-interface portion of the application. This approach is consistent with "slightly-interactive" programs, which mostly compute but occasionally prompt the user for input. In highly interactive applications, however, the actual locus of control is associated with the user. The tension between these two loci of control is the source of the complexity. The ADV model avoids this tension by placing the main locus of control in the ADV. Fundamentally, the ADV model is based on the program waiting for the user rather than the user waiting for the program.

5.2 Composition of Abstract Data Views

Applications such as the chess and checkers game illustrated in Figure 5 are often formed by composition. That is the game is composed of a board and playing pieces and we would expect the corresponding ADT to be composed in the same manner. Since ADVs represent the state of an ADT they should also be formed by composition. In this example the ADV is built from two types

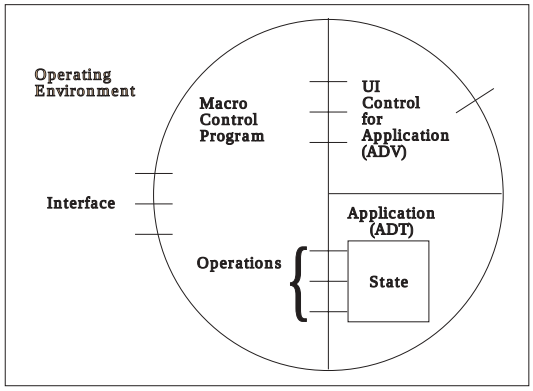

Figure 6: The Structure of Applications

of components; an 8X8 board and playing pieces. The board can be used for any game which uses an 8X8 grid and only needs to be modified by the specific pieces which are required. Hence we have developed the concept of embedding ADVs within other ADVs. For the chess game illustrated we would require the 32 ADVs for the playing pieces to be embedded inside the board ADV. Each of these embedded ADVs can be connected to its owner ADT as well.

5.3 Embedding Abstract Data Views

Embedded documents require that each ADV be connected to its owner ADT and also that communication exist between enclosed ADV(s) and enclosing ADV(s). This communication between ADVs must exist in order to relate behavior. For example, if the inner ADV in Figure 2 is resized the enclosing ADV must cause a reconfiguration of the text. This means that each ADV must have an accompanying script which supports communication among ADVs.

Of course we should also note that when the inner ADV is first opened there must be some way within the system to connect to its application. We could either offer a menu of legal ADV types or once the ADV is open a menus of legal applications. These methods of connecting user interfaces or ADVs together so that they can influence each other's behavior are described in detail in [CILS93b].

6 Application Packages

Most applications can be viewed as two components, namely a server or ADT which manages the underlying storage and data structures for the application data and a user interface or ADV which allows the user to see the data and modify it by direct interaction. Some applications such as an SQL database only have a server component. Of course the user interface and the server must interact with each other and the surrounding operating environment.

As was already indicated in the example in Figure 1, we often want to configure these general-purpose applications into a specialized application. In order to perform this specialization we need to be able to query and change the state of both the server portion and the user interface of the application. We will also need to make decisions based on the current values of the state. What

is needed is a programming language which can perform these tasks. Early applications such as Lotus 1-2-3 had a record facility which would capture and store user interactions. These could then be stored and played back. Current software such as the word processor Word for Windows, the spreadsheet Excel and the database Watfile [CGMW89] contain programming languages often called script or macro languages which can query and change the state of both the server and the user interface. These languages provide a high degree of flexibility. Each of these different programming languages is embedded in the software application and contains all the standard programming language constructs. Also each software system is able to receive and process commands provided through a program interface. This means it is possible for an external program to control the application by providing a programming script. This concept is illustrated in Figure 6 where the lines leading from the different sections of the application indicate possible communication paths with other components.

A more appropriate model would be to build the software with all the interfaces to both the user interface and the server accessible outside the application. In this way a language which supported some form of linking (preferably dynamic) would be able to control the software. This would have one other advantage: only one language would be needed and the application integrator configuring these applications would find the task much less arduous. Although an interface would have to be learned with each piece of software, knowledge of only one programming language would be required. This approach of providing access to all aspects of the application is used in applications such as PMEdit [IBM92] which are built for OS/2 and use REXX as the macro language.

The interface specifications and their use could be supplied as help hypertext with the application. In this way the application integrator would have appropriate on-line help as they developed their specialized application. Such an approach has been implemented in VX·REXX [VRe93].

7 The Script Bus Mechanism

The "script bus mechanism" shown in Figure 3 consists of "script" programs and standard interfaces to allow communication and control among the programs. Control of the application may reside in the script bus or in one of the components. In other words it may be possible to have several operational threads of control. The "script bus control" is a special user interface or user console which allows the operator of the application to start the application and to set parameters for the various scripts. The standard interfaces are the communication capabilities and the event control.

The communication interface supports full asynchronous communication which could be implemented as blocking send, blocking receive and non-blocking reply or some other suitable set of primitives. The event mechanism allows input events initiated through the various stand-alone or application-based user interfaces to trigger actions of the script bus mechanism. For example, initiation of the "save" command in the spreadsheet might cause the database to save some data. Some current attempts at standard interfaces are DDE, OLE and similar mechanisms in Windows and OS/2, and Apple Events and Tooltalk [Hel91].

The communication and event handling which might be considered part of the operating system are accessible through the script bus mechanism; the other aspects of the operating system such as utilities and the file system are just clients or servers.

```
select x,y from datatable where x < maximum
select count(x) from datatable
select max(y) from datatable
plot.range(maximum)
plot.domain(max(y))
i := 0
while i < count(x)
 select x,y from database
 plot.point(x,y)
 move to next x,y in database
endwhile
```

Figure 7: A Simple Data-type Transformer

Script programs often act as data-type transformers between two or more software systems. For example, consider an script program which extracts data from an SQL database and then produces a graph or chart. The script program would have a user interface which would allow the user to specify the domain of the independent variable and the name of the database. Access to the database and presentation of the graph would probably involve a type conversion between the data structures supported by the SQL database and those handled by the plotting package. The commands to the database will be in SQL and the results will be a table, while the commands to the plotting package could be in the form of drawing commands with the data represented as a sequence of strings. Currently, a script program would copy the data from the database, transform the data into its new format, and then pass the data to the plotting program. Such an approach could have a significant impact on performance in both space and time, since at some point in the computation three copies of the data exist, and all the data has been transformed twice. In order to minimize the copying of an entire data structure, the interface to the plotting package should contain functions which would allow specification of the range and domain of the plot and would pass the package the y-coordinate for a given x-coordinate. The script program would transform these functions into equivalent database retrieval commands. This mechanism is outlined in Figure 7.

Both the SQL database and the plotting package are predefined components and each component would be compiled and linked. Implementing the code for the example in Figure 7 requires that the function invocations in the script program be dynamically linked to the function bodies defined in the reusable components. One implementation of the dynamic linking concept is embodied in the Dynamic Link Library (DLL) of Windows 3.0 [Pet90].

Although the performance penalty in the previous example may not be too high, consider a further example where the database contains a map stored as a large graph and you wish to operate on the graph with a graph package containing a graphical user interface and various graph algorithms. The graph package is independent of the access mechanism for the graphs, all the package requires are nodes, arcs and cost functions associated with the arcs. The graph package will display a portion of a map and allow operations such as selecting two points in order to compute the shortest path. This example introduces further complications. First, the graph package uses

```
-- Initialization
   bind graph.origin to origin
   bind graph.destination to destination
   bind graph.arcs to arcs
   bind graph.cost to cost

-- Transformer Functions
   function origin (x : arc)
     select x from arc_table
     y := arc_origin(x)
     return (y : node)
  endfunction
  function destination (x : arc)
    select x from arc_table
    y := arc_destination(x)
    return (y : node)
  endfunction
  function arcs (x : node)
    select x from node_table
    for all arcs
      append arc to arc_list
    return (y : arc_list)
  endfunction
  function cost (x : arc)
    select x from arc_table
    return (arc_cost : integer)
  endfunction
```

Figure 8: A Transformer Using Dynamic Binding

a portion of the graph for display purposes and a different portion of the graph to compute the shortest path; in both cases the entire map may not be necessary. Second, the entire application is operated from the user interface of the graph package rather than from the script program.

In the map example illustrated in Figure 1, functions would be a much better way to pass portions of the graph, since the portions can be requested on demand. This means that the graph package should have a functional interface which allows requests for nodes and arcs and cost functions. The script program will contain transformation functions which will change the requests for nodes and arcs into suitable database retrieval requests. Since the script program and the graph package will be connected after the graph package is compiled and linked, there must be a method of dynamic binding[3] which will connect the transformation functions to the function interface of

[3] Dynamic binding is also often called late binding.

the graph package. Dynamic binding requires more than dynamic linking. An outline of the code for this example is included in Figure 8.

In later sections we provide a description of a programming language and its associated programming environment which support application integration (CAAI).

8 Characteristics of Programming for Application Integration

Section 2 provides an informal description of the type of applications likely encountered by an application integrator. Based on this description, the script bus mechanism in Section 7, and various application integration experiments some of which are presented in [CIS92, CILS93b], we can now describe the main characteristics of programming for application integrators. In this description we emphasize the differences between this paradigm and the more conventional programming model. We call this type of programming, application-integration programming or AIP.

Certainly, the main distinguishing feature of AIP, as opposed to more conventional programming, is the fact that the programmer and one of the major users of the application software are the same person. The application integrator will not be creating software with the intention of distributing it to a large group of clientele. This software will be created, maintained, modified and informally distributed by the application integrator. For this reason, this type of software will not go through the usual software engineering and development life-cycle, in fact these should be completely eliminated, otherwise we will defeat the entire raison d'etre for AIP.

Since the application integrator and the end-user of the software are closely related, there can be a different approach to program design and development. The programmer can create experimental prototypes with the results being throwaway or disposable code, since there should be minimal impact on other users. There is no need for intermediate documentation between requisite analysis and program code. On the other hand, the disposable nature of this kind of software means that the prototype is the final product. In fact the program probably is always a prototype, as we expect most users in consultation with the application integrator will modify the code extensively to add new functionality.

Programming languages, at least for the foreseeable future, will be an integral part of AIP. What is their current status and how are they likely to evolve? The class of non-programmers which obviously includes many application integrators are often wary of programming languages [ACL+91]. Methods of programming without languages such as exemplified by visual programming [Cha90, Gli90] will continue to be investigated and progress will be made. However, currently such systems still do not have enough flexibility to be a generic software development tool.

An intermediate approach which has met with considerable success is in the area of end-user interfaces. These interfaces are composed of such elements as buttons, dialog boxes, list boxes, and selection mechanisms which allow the user to interact with other software. Although programming user interfaces of this type is usually complex, a level of abstraction can be introduced which simplifies this task immensely and can transform constructing user interfaces into a form of visual programming. There appear to be two reasons for this ability to simplify; first, user interfaces are composed of a small number of different objects, and second, user interfaces are primarily visual in nature. Thus, AIP can be visual while supporting basic tasks such as building user interfaces, and yet retain the more traditional programming model for the tasks which do not currently lend

themselves to automatic program generation. It is the authors' expectation that the traditional programming model will always be part of AIP, but that it will be used less and less as we discover how to modify the programming paradigm for various components of AIP tasks.

Examples of systems which use this intermediate approach are Hypercard, ToolBook, Visual Basic and VX·REXX; all these systems uses some aspects of the ADV model, particularly the locus of control. A common characteristic of all these systems is that many facets of the end-user interface can be constructed without programming. However, only the most basic functionality can be achieved this way; anything a little more complex still needs some form of programming.

Application integrators do not develop applications, rather they use the existing general-purpose applications and features of the operating system to develop software tailored to a specific set of tasks. An attractive approach, already introduced in Sections 6 and 7, to reusing these various software components is through a glue language (also called *command* or *script* language). The application integrator would write a program in the glue language to pass data among components and send them commands. Many of these applications, which are a form of predefined component, have a well-defined programming interface or port, and/or their own scripting language. Thus, an application can often be fully operated both through the normal user interface and also with a set of language-based commands passed to the port. Unfortunately, because each application often has its own version of a script language the application integrator's programming task becomes even more difficult. Developing a standard script language would alleviate some of this problem[4].

9 Characteristics of a Programming Language for Application Integration

Section 8 of this paper provided a description of the type of programming likely encountered by an application integrator. In this section we use these observations to motivate the features which should be available in a programming language designed for this class of programmers.

Most programs produced by application-integrators will be short initially, although they may grow to several hundred lines as they evolve. Application-integrators will be "amateur" programmers in every sense of the word and are not likely to have the disciplined training in software engineering that we should expect from so-called professionals. We can not expect application integrators to use anything resembling a formal design cycle or other notions of programming such as encapsulation or inheritance. Amateurs may have learned their programming from a single class or by experimenting with a few examples in the manual. Ideally application-integrators will only program for their own particular use or the use of close colleagues. However, we know that within a group such as an office, programs are frequently circulated. Because of this amateur status and the fact that application-integrators are unlikely to receive much more substantive training in programming and software design, we should protect application-integrators from many of the common programming and design mistakes. This "protection" can be built into the language and the environment in which that language is implemented. We discuss the language in this section, the programming environment is described in Section 10.

[4] Visual Basic and VX·REXX are two such attempts at creating a standard script language for a range of software applications.

Although application-integrators are not professional programmers, they often operate in a procedural domain, since instructions for operating computer software, appliances and machinery are primarily procedural in nature. The procedural model is one which has evolved through the experience and seems to be natural to most users. This argument is used to justify choosing an application-integration programming language (AIPL) which is primarily procedural rather than functional.

9.1 Control and Communication Constructs

An AIPL should contain standard control constructs such as sequence, loop and choice. An AIPL should also support an event-driven model, since programs will be activated by external programs such as those in a user interface or a device driver which needs attention at random intervals.

Since programs written in an AIPL will be gluing together predefined components, the programs must be able to communicate with these components in a reasonable manner. The language should be able to support both synchronous and asynchronous communication. The ability to handle asynchronous events can be extended to any type of communication not just user interface input events.

The language should contain some form of procedure so that common code can be encapsulated into a more meaningful operation. Procedures of course allow some form of reuse within a program.

Some function invocations in predefined components may be related to function code in a program. Thus, the language and its environment must be able to support some form of dynamic binding so that reusable components may be connected to a program.

There are many languages which implement many of the features just described, particularly in the Unix environment. These languages, often called shell languages, include the C-Shell, Korn-Shell [Par89], Perl [Wal92], and Expect [Lib90] all Unix-based languages; Visual Basic for Windows [Cor91]; and REXX [Cow90] for VM [Cha89], MVS[Joh89], DOS [Mic91a], Windows [Mic92b], Windows NT [Far93] and OS/2 [IBM92]. However, most of these languages have deficiencies. Typically they lack user interfaces, some form of event support or allow only simplex communication. The last two items in this list are related.

The languages themselves do not have to be complex. For example, the fundamental constructs of Visual Basic or VX·REXX are quite similar to standard sequential programming languages such as Pascal. However, they do look somewhat different because they also support events and communication as well as the standard control constructs of sequence, loops, choice, and function and procedure calls. The language will likely be interpreted since interpretation offers more flexibility in implementation and error handling. Compilation may be available although the efficiency offered by compiled program will not be necessary since most of the computational time will be spent in the clients and servers rather than on the script bus mechanism.

9.2 Types

The types of variables are often redundantly specified in declaration statements to assist with compile-time verification or static type-checking, and performance improvement. Since application integrators should be in intimate contact with their programs and are present during program operation to correct errors, the need to find errors at compile-time is significantly reduced. Also

the program is usually not in control during most of a computation and hence performance should not be an issue. Since introducing this redundancy adds complexity to the programming process with little or no gain in either compile-time or run-time performance, explicit declaration of types should be avoided in an AIPL.

Many programming languages use the concept of primitive types and then use them in different ways to create more complex type structures. The primitive types must be chosen carefully to be highly readable to the application integrator. Moreover they must be completely abstracted from the underlying implementation. For example, an integer type which allows the manipulation of its constituent bits or overflows without warning, is not an appropriate representation. In the same vein the application integrator should not be concerned with mundane issues such as memory allocation.

More complex notions such as abstract data types, classes and inheritance do not need to be accessible to the application integrator. The authors do not see any way that notions common to object-oriented programming can be easily incorporated into the AIPL. The concepts of inheritance and data encapsulation are motivated by software engineering arguments which are currently beyond the experience of most application integrators. Motivating these concepts seems to be difficult, although a rationale may be discovered which can present these concepts in familiar terms. Obviously, types provided in a language can be abstract data types, but requiring the application integrator to produce code which requires such facilities is not appropriate.

Class inheritance also should not be a necessary part of the AIPL. Systems such as Hypercard and ToolBook provide a form of dynamic inheritance. Procedures are attached to specific structures such as a card, background or stack in Hypercard, or objects, groups, pages, background, book or system in ToolBook in a manner which might be considered a weak form of inheritance. These structures are arranged in a hierarchy so that if a procedure is not defined at one level of the hierarchy, the underlying implementation will look for an equivalent routine at a higher level. This concept provides a form of inheritance as the lower-level data objects can use procedures attached to objects higher in the hierarchy. If a procedure is not encountered in the hierarchy then an error message is produced. This form of inheritance is not explicit, but rather implicitly available as part of the implementation[5].

9.3 Safety

The AIPL should be safe. It should be impossible to combine types and produce a nonsense result. Some forms of both compile-time and run-time type checking should be part of the implementation. It also should not be possible for the program to leave the thread of control. For example, creating an undefined pointer[6] which could create havoc in a program should not be allowed.

[5] This type of inheritance although useful can be very confusing. Even in a modest size application it becomes quite difficult for a programmer to trace the thread of control by following the static program text.

[6] We are assuming here, that the reusable components are "correct" and do not create undefined pointer problems.

10 Application Development Environments

Application integrators will not in general create large programs; they are programming to make the environment in which they perform their current set of tasks more productive. In some sense the application integrators will want to automate their access to computer tools they already use. Also from a programming point of view, the user and the programmer are identical, and so the distinction between compile-time and run-time becomes blurred. For example, both a compile-time and a run-time error are really forms of output, unlike a production system, where a run-time error is unacceptable.

10.1 Compile-time and Run-time Environment

Amateurs are more likely to make errors and wish to experiment with various programming constructs, in effect they are almost always in a learning and debugging mode. Timely feedback is important and the programming environment should support a fast edit-compile-run cycle for ease of experimentation. Thus, compile-and-link performance is as important as run-time performance.

Clear error diagnostics at both compile and run-time are obviously essential characteristics; the error should be described in a normal language and its location should be determined as accurately as possible. A source-level debugger to allow tracing, and setting breakpoints would be another essential part of the environment.

However run-time errors should only be generated by the application integrator's code, not by predefined components. Errors in predefined components can not be traced by application integrators since they had no control over the construction of such a component. Error conditions of this type are discussed more in Section 10.3

10.2 Reusable Components and Object Libraries

Most of the code in a program will be concerned with connecting together various software tools, user interface objects[7] and device drivers, since the application integrator will want to automate access to computer tools already being used. Extensive libraries of tools and objects will have to be available with well-defined interfaces and command languages. Each component in a library will have to have extensive associated help files which can be accessed by the application integrator, so that it is clear how the object can be activated or used. In some cases the access to the components will be through some set of tools which will use a paradigm such as visual programming.

Predefined components will have to be encapsulated so that they have an interface which can be operated through function calls. Some of these functions will be bound to the program at compile or run-time. In general, predefined components should support dynamic binding.

Each predefined component or object should be able to describe its own interface and how it might be used in applications so that the domain expert can easily find help. Also the domain expert should be able to augment this information as more experience is gained. Some form of hypertext help mechanism might be available to manage this help text.

[7] The term object used in this paper is the usual English word and is not necessarily used in the sense of object-oriented programming.

10.3 Communication and Communication Safety

Access to software tools, user interface objects and device drivers through the AIPL, implies that the AIPL and the underlying environment must support two-way communication with other software systems. Such communication with other software systems should be safe, since lack of response is often a difficult error to trace.

There are at least three possible conditions which have to be considered to ensure safe delivery from the program viewpoint: Was the message delivered?, Was the message correct?, Was the message executed? The reusable software component or the underlying system should send a response indicating the outcome of the communication.

The underlying system must also provide support for synchronous and asynchronous communication. The system must provide support for synchronous communications, any timing critical problems should be handled by the predefined components rather than the program.

10.4 Code Management and Presentation – A CAAI Environment

The application integrator will need to be presented with a programming environment to support this type of programming activity which we have called CAAI: Computer Assisted Application Integration. Our goal in creating this CAAI tool is to make application integration and the management of the resulting programs as simple as possible. For example, many of the current script or glue programming languages do not support modularity and information hiding easily. Early version of CAAI tools will have to assist with these problems by hiding some of the "ugly" facets of these languages. The environment provided with VX·REXX [VRe93] provides some of these facilities.

A CAAI environment must be flexible and be easily programmable by the application integrator. As the knowledge of the application integrator grows the CAAI environment should be able to change appropriately. Being able to change the user interface, incorporate new tools into the environment or add help text are three examples of the type of flexibility required. In this way we will gradually create an application development environment which will become more "user-friendly" for the individual over time.

11 Conclusions

The paper has described an architecture for implementing applications which uses predefined large objects. The architecture uses reusable interface objects, and application objects connected together through a scripting language which forms a script bus mechanism. The control of this mechanism is implemented as a set of programs written in a scripting language.

Since the programming required to "build" these applications is primarily concerned with control rather than constructing sophisticated objects it should be possible to teach the programming skills required in a short period. Thus a domain expert should be able to build applications rather than becoming reliant on a programming "team".

Many of the tasks outlined in the paper such as: building user interfaces, tailoring applications to specific tasks no longer have an apparent programming component. They can either be performed graphically or by tracking user behavior through a record function.

Finally any application development environment that supports application integrators in building applications must be simple to use and must produce safe code. That is, the script must report to the application integrator when some computation or communication has failed.

Many experiments have been conducted to prove out the concepts described in this paper, since an architecture must have some hope of realization if it is ever to be used. The ADV concept which allows the construction of reusable user interface has been implemented in several systems and is described in [LCP92, CCLP93, C+92]. Specifically ADVS have been used in a graphics editor, a GUI development environment and in several distributed client server graphics applications. The concept of distributed scripts or holes has been implemented in Smalltalk and is described in [CILS93b]. Also the GUI components in VX·REXX [VRe93] and their interaction with the REXX language closely follow the ADV model. Finally a number of application systems similar to the one shown in Figure 1 have been implemented using VX·REXX, various databases, spreadsheets, word processors and editors.

References

[ACL+91] D. Allen, D. Crabb, L. Loeb, R. Malloy, W. Nance, B. Rash, K. Sheldon, and P. Wayner. What is a Programming Language. *BYTE*, pages 103–104, August 1991.

[Asy89] Asymetrix Corporation, Bellevue, WA. *Using Toolbook, a Guide to Building and Working with Books*, 1989.

[B+92] R. Budde et al. *Prototyping, An Approach to Evolutionary System Development.* Springer-Verlag, 1992.

[C+92] D. D. Cowan et al. Program Design Using Abstract Data Views–An Illustrative Example. Technical Report 92–54, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, December 1992.

[CCCL93] L. M. F. Carneiro, M. H. Coffin, D. D. Cowan, and C. J. P. Lucena. User Interface High-Order Architectural Models. Technical Report 93–14, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, 1993.

[CCLP93] M. Coffin, D. D. Cowan, C. J. P. Lucena, and A. B. Potengy. Distributed Abstract Data Views: Design and Implementation. Technical Report 93-61, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, 1993.

[CGMW89] D. D. Cowan, T. G. Galvin, S. G. McDowell, and T. A. Wilkinson. *WATFILE/Plus.* WATCOM Publications, 1989.

[Cha89] Paul Chase. *VM/CMS: a user' s guide.* John Wiley, 1989. QA76.76.O63C455.

[Cha90] Shi-Kuo Chang, editor. *Principles of Visual Programming Systems.* Prentice-Hall, 1990.

[CILS93a] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien. Abstract Data Views. *Structured Programming*, 14(1):1–13, January 1993.

[CILS93b] D. D. Cowan, R. Ierusalimschy, C. J. P. Lucena, and T. M. Stepien. Application Integration: Constructing Composite Applications from Interactive Components. *Software Practice and Experience*, 23(3):255–276, March 1993.

[CIS92] D. D. Cowan, R. Ierusalimschy, and T. M. Stepien. Programming Environments for End-Users. In *Proceedings of IFIP 92, Volume III*, pages 54–60, 1992.

[Cor91] Microsoft Corporation. *Microsoft Visual Basic Programmier's Guide.* Microsoft Corporation, 1991.

[Cow90] M. F. Cowlishaw. *The REXX Language: A Practical Approach to Programming.* Prentice-Hall, 2nd edition, 1990.

[Far93] Rik Farrow. Understanding Windows NT. *UNIX/World*, 10(2):47, 1993.

[Gli90] E. P. Glinert, editor. *Visual Programming Environments.* IEEE Computer Society Press, 1990.

[Goo90] Danny Goodman. *The Complete HyperCard 2.0 Handbook.* Bantam Books, 3rd edition, August 1990.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley, Palo Alto, CA, January 1983.

[Haw87] William S. Hawes. *ARexx User's Reference Manual, The REXX Language for the Amiga.* Commodore-Amiga, Inc., Maynard, MA, version 1.0 edition, 1987.

[Hel91] Martin Heller. Future Documents. *Byte*, 16(5):126–135, May 1991.

[Hoa89] C. A. R. Hoare. Proof of Correctness of Data Representations. In *Essays in Computer Science*, pages 103–115. Prentice-Hall, 1989.

[IBM92] IBM. *Using the Operating System OS/2 - Version 2.0*, 1992.

[Joh89] Robert H. Johnson. *MVS: concepts and facilities.* McGraw-Hill, 1989. QA76.6.J6576.

[Kay69] Alan C. Kay. *The Reactive Engine.* PhD thesis, University of Utah, Salt Lake City, Utah, USA, August 1969.

[LCP92] C. J. P. Lucena, D. D. Cowan, and A. B. Potengy. A Programming Model for User Interface Compositions. In *Anais do V Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens, SIBGRAPI'92*, Aguas de Lindóia, SP, Brazil, November 1992.

[Lib90] Donald Libes. expect: Curing Those Uncontrollable Fits of Interactivity. In *Proceedings of the Summer 1990 USENIX Conference, Anaheim, California*, Gaithersburg, MD 20899, June 1990. National Institute of Standards and Technology.

[Mic91a] Microsoft Corporation. *Microsoft MS-DOS Operating System Version 5.0 - User's Guide and Reference*, 1991.

[Mic91b] Microsoft Corporation. *Microsoft Word Version 5.0*, 1991.

[Mic92a] Microsoft Corporation. *Microsoft Excel Version 4.0*, 1992.

[Mic92b] Microsoft Corporation. *Microsoft Windows Version 3.1 - User's Guide*, 1992.

[Mye91] Brad A. Myers. Demonstrational Interfaces: A Step Beyond Direct Manipulation. In Dan Diaper and Nick Hammond, editors, *People and Computers VI*, pages 11–30. Cambridge, England: Cambridge University Press, 1991.

[NeX91] NeXT Computer, Inc. *The NeXTstep Advantage, Application Development with NeXTstep*, 1991.

[Par72] D.L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *CACM*, 15(12), December 1972.

[Par89] Tim Parker. Shells for UNIX: A Basic Programming Choice. *Computer Language*, 6(7):73, 1989.

[Pet90] Charles Petzold. *Programming Windows.* Microsoft Press, 2nd edition, 1990.

[Tof91] Alvin Toffler. *Powershift.* Bantam Books, December 1991.

[VRe93] *WATCOM VX·REXX for OS/2 Programmer's Guide and Reference.* Waterloo, Ontario, Canada, 1993.

[Wal92] Larry Wall. *Programming perl.* O'Reilly & Associates, 1992. QA76.73.P347W35x.

[ZS91] Chris Zamara and Nick Sullivan. *Using ARexx on the Amiga.* Abacus, Grand Rapids, MI, 1991.