The Open Scripting Architecture:
Automating, Integrating, and Customizing Applications

Unpublished manuscript, 1993

William R. Cook
Apple Computer

Warren H. Harris
Apple Computer

Research Topic areas: Language design and implementation, tools and environments, components and frameworks, concurrent and distributed systems, databases and persistence.

Abstract

The Open Scripting Architecture combines aspects of object-oriented programming, distributed computation, database queries, and dynamic languages into a powerful and practical system for automation, integration, and customization of applications and system services. Applications are integrated together with distributed messaging. The messages operate upon user-level application objects, like windows and spreadsheet cells that are identified by queries over properties and document containment structure. A general-purpose scripting language automates message sending and handling, and supports persistence and mobile objects. Applications call on scripting services to customize the behavior of their objects through a generic script management API.

1 Introduction

The Open Scripting Architecture (OSA) is a comprehensive infrastructure for automating complex or repetitive tasks, integrating distributed applications and system services, and customizing application behavior. These end-user benefits are supported by a synergistic combination of technologies: distributed messaging and referencing of application objects, object-oriented scripting languages, and script development and management tools. Object-oriented concepts are used throughout the architecture, in the scripting language, the system, and the applications.

The OSA is unique in several respects. First, it decouples the clients of scripting services (applications) from the providers of those services (language processors like AppleScript™). This decoupling allows for uniform scriptability across all applications in the same way a graphics toolbox provides a uniform GUI. Furthermore, it allows scripts from different language processors to be mixed within applications. Second, it treats familiar applications as libraries of functionality that users can draw upon to create custom solutions. Third, its object-referencing model allows scripts to interact with live applications, accessing and manipulating their internal objects and invoking their methods. This differs from "shell" languages that can launch, kill and redirect I/O to and from processes, but cannot otherwise interact with their internals.

The OSA currently runs on the Macintosh, but is being ported to other platforms as part of the OpenDoc effort. Over 80 applications, from many major vendors, support the OSA.

In this paper we describe the OSA architecture. Section 2 presents a motivating example illustrating the degree to which application automation, integration and customization can be achieved. Section 3 describes the OSA inter-application referencing and object model. Section 4 discusses scripting under OSA, and presents several unique features of the AppleScript language that harmonize with the OSA architecture. Section 5 defines the OSA API, by which applications interact with scripting services. Finally, Section 6 relates the OSA to similar systems.

2 Motivation

A motivating example of the power of OSA is the automation of a catalog publishing system. An office-products company keeps all its product information in a FileMaker Pro™ database. The database includes descriptions, prices, special offer information, and a product code. The product code identifies a picture of the product in a Kudos™ image database. The final catalog is a QuarkXPress™ document that is ready for printing. Previously, the catalog was produced manually. This task took a team of twenty up to a month to complete a single catalog.

Using the OSA, a script automates the entire process. The script reads the price and descriptive text from the FileMaker Pro database and inserts it into appropriate QuarkXPress fields. The script applies special formatting: it deletes the decimal point in the prices and superscripts the cents (e.g. 34^{99}). To make the text fit precisely in the width of the enclosing box, the script computes a fractional expansion factor for the text by dividing the width of the box by the width the text (this task was previously done with a calculator). It adjusts colors and sets the first line of the description in boldface type. Finally, it adds special markers like “Buy 2 get 1 free” and “Sale price 17^{99}” where specified by the database.

Once automated, one person produces the entire catalog in under a day, a tiny fraction of the time for the manual process. It also reduces errors during copying and formatting. Of course, creating and maintaining the scripts takes time, but it is far less than the production of a single catalog.

A custom front-end to the catalog production system is created using an interface builder. The consultant draws the interface elements (buttons, fields, etc.) and then attaches scripts to them, which drive the entire process. On field specifies the percentage sale price, which the script uses to compute sale prices. The custom front-end along with its associated scripts that drive the process can be saved as a mini-application—a script stored in a document that, when double-clicked, initiates the catalog publishing process.

The automation is achieved through a single common scripting language, reducing the need to learn application-specific languages. Integration allows one to move away from the “kitchen-sink” application model. Instead, smaller applications work together, and users combine the best features of each application. Other

services like spell checking need not be provided with each application, but drawn on from the distributed environment and applied uniformly everywhere they are needed.

3 Distributed Messaging and References

The foundation of the OSA is a mechanism for distributed messaging and referencing of application data, called Apple events and Apple event objects [2].

By supporting messages and references, an application makes its functionality and objects visible to clients. Once it supports Apple events, use of the application can be automated by sending it a series of events. Given multiple applications supporting Apple events, integrated solutions can be created by sending events to coordinate processing and exchange data.

An application the supports Apple events has handlers for each event type and a mechanism for decoding references [3]. If an application does not handle an event, then it may be handled by a system event handler. System event handlers are installed on the system via "Scripting Additions" that are loaded when the system starts up.

3.1 Descriptor Format for Data and Events

All data and events to be transported between applications are stored in *descriptors*, a standard data format. Descriptors use a self-describing, structured, flattened data format. The format is designed to be easily transported or stored.

Apple event descriptors are a pair of a type code and a data handle, allowing them to store arbitrary data. Descriptors can either contain primitive data, a list of descriptors, or a record of descriptors (record elements are named by unique IDs). Primitive data types include numbers (small and large integers and floats), pictures, styled and unstyled text, process IDs, file references, aliases, etc.

An Apple event is a special kind of record consisting of an event code (event class and event id), a direct parameter, and a record of parameters with keywords IDs and values. All of the IDs in an event is stored in four-byte enumeration code. Events may be sent synchronously or asynchronously.

3.2 The Object Model

The Apple event object model specifies the structure of objects in an application or document. The objects being specified are user-level objects like documents, words, text styled, graphic objects, colors, etc. They are not necessarily related to lower-level implementation objects within the application.

The basic structure of application data is specified by a simple combination of *elements* and *properties,* as illustrated in Figure 1. Elements are a set of objects, like windows in an application or words on a page, that may be either empty or have many instances. Elements are specified by a *class*, which defines which kind of objects are to be located. A given object may have many classes of elements, and classes may have subclasses. The containment structure is not necessarily strict, in that a document object has both page and paragraph elements, neither of which

strictly contains the other. Properties, on the other hand are labels for unique aspects, like the style of a word, or the name of a window, which must always exist. They correspond to instance variables defined by classes.

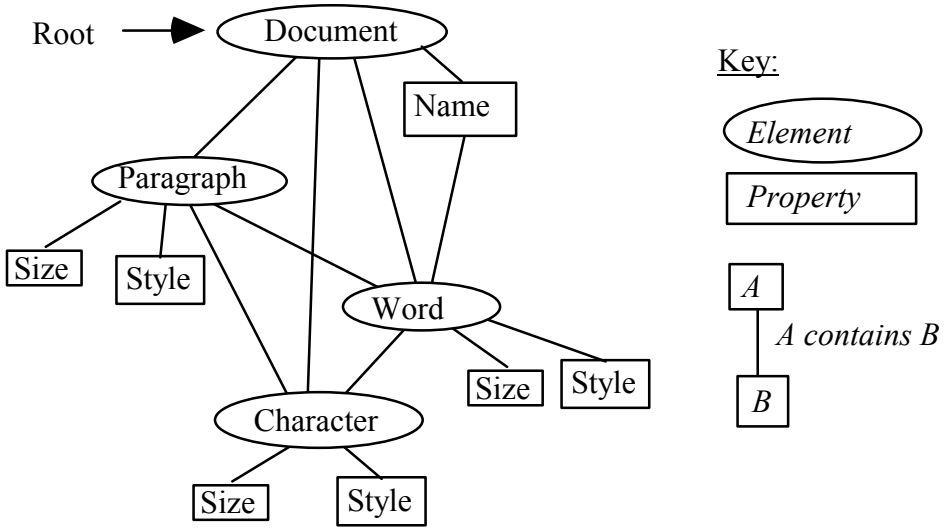

Figure 1: An example object model containment hierarchy.

3.3 Object Specifiers

An *object specifier* is a reference or query that specifies one or more objects in an application. Object specifiers locate objects in an application by their position or properties, relative to a container. There are four basic forms of object specifiers:

Property: Property object specifiers specify the name of a property to be accessed relative to a container. The name may either be an application-defined code, or a user-defined string.

Index: Index object specifiers specify a particular element of a given class. The index is either an integer or the special value middle or any. If negative, the element is taken relative to the end of the list.

Range: Range object specifiers specify a sequence of elements between two indices. The two endpoints are also object specifiers.

Test: Test object specifiers specify a set of elements of a given class that satisfy a Boolean predicate. These correspond to queries against the object containment hierarchy.

Object specifiers are represented in Apple events as nested record structures. Each object specifier form is a record with fields for the class, property name, index, and container. The container is either another object specifier record or null, representing the default or root container of the application.

3.4 Standards

Standard events and reference structures are defined in the *Apple Event Registry*. The Registry is divided into suites that apply to domains of application. Suites contain specifications for classes and their properties, and events. Currently there are suites for core operations, text manipulation, databases, graphics, collaboration, word service (spell checking, etc.). The Registry is effectively the scripter's API to applications whose routines are used as libraries.

The design and specification of the Registry was crucial to the success of the OSA in that, when correctly implemented by applications, it enables a consistent scripting interface between similar applications. In this way the Registry is analogous to the Macintosh Human Interface Guidelines. For example, the same script may be useful for manipulating text in a word processor's paragraphs, and a for fields in a database. An application framework which ensures a higher degree of consistency than is provided by the specification is much needed. Such a framework would also make the task of building scriptable applications easier.

3.5 Recording User Actions

In addition to being controlled by external events, applications can *record* events that correspond to user interactions. This allows automatic generation of scripts for repeated playback of what would otherwise be repetitive tasks. Recorded scripts can be subsequently generalized by users for more flexibility. This approach to scripting alleviates the "staring at a blank page" syndrome that can be so crippling to new scripters. Recording is also useful for learning the terminology of basic operations of an application. Recording connects the iconic and "active" levels of understanding directly to the higher symbolic layer.

Recording with high-level Apple events is very different from traditional systems that record low-level user-interface input events. Low-level recording is inherently tied to positions on the screen or in a window. But these positions are subject to whimsical change, making the recorded script useless. Apple events depend upon the contents of the document and its object model, making the recorded events more generic and robust. Of course, the document structure may change enough to obsolete recorded Apple events, but this is less likely.

One of the inherent difficulties of recording is the ambiguity of object specification. As the language of events becomes more rich, there may be many ways to describe a given user action, since the intent of the action is not known. For example, when closing a window named "Example", is the user closing the front window, or the window specifically named "Example"? Depending upon the context in which the events are played back, one or the other kind of event might be better. If the events are played back in an interactive context, then references to the front window or current selection are appropriate. But if the events are played back in a batch mode, then more explicit references are better. To avoid recording modes, the OSA guidelines favor recording for interactive contexts.

4 Scripting in OSA

Scripting is the term we use for programming in the distributed messaging environment. Scripting allows routine or complex tasks to be automated.

The term "scripting" originates from HyperCard™, and was intended to convey the idea that this particular sort of programming was easy for end-users, consultants and in-house developers—people who do not otherwise consider themselves programmers.

However, scripting connotes more than just programming. It is programming directly in terms of the applications that are to be controlled—at a level that is sufficiently abstracted from the machine, the maintenance of common data structures, and the details of memory management. Furthermore, it is "safe" in the sense that incorrect constructions will not crash the computer or operating environment. This safe and direct nature of scripting enables rapid and productive creation of "custom solutions" by a larger number of people.

Applications play a key role in OSA, serving as high-level libraries of operations and data structures. Because of the familiarity users already have with the capabilities of particular applications via their visual interface, automation of tasks involving the application can be readily accomplished by directly mapping these concepts into the verbs and nouns that the application provides to the scripting interface.

4.1 The AppleScript Language

The AppleScript language [1] provides the basic "computer science boilerplate" that works in harmony with applications' library-like functionality. This includes common data types like lists, vectors, records and numbers and simple primitive operations on them, as well as common control constructs for iteration, conditionals, functional abstraction, exception handling. A simple yet flexible object system with persistence manifests itself as first-class "script objects" in the language. Script objects facilitate the customization of applications, and enable "remote programming."

4.1.1 Language-level Application Terminology Integration

The AppleScript parser integrates the terminology of applications with its built-in language constructs. This means that when targeting Microsoft Excel™ for example, spreadsheet terms are known by the parser—nouns like "cell" and "formula," and verbs like "recalculate." This allows Apple events messages to be constructed that correspond to commands and object specifiers to be constructed which correspond to references. When a script targets an application, enough information is stored in the script to find the application if the script or application are moved to another machine or renamed.

Parser-level integration eliminates the need for libraries of "glue" routines that construct and send events for each application. The AppleScript tell statement causes the parser to read the terminology resource of an application when it can be derived from the target of the tell statement at parse time. This causes identifiers to

be created for the message, class and enumerated constant names which are scoped within the body of the tell. For example:

```
tell application "Microsoft Excel"
    quit saving yes
end tell
```

Here, the quit message is known to correspond to the particular event code that Excel accepts, and the yes constant is translated into a particular enumeration code. Furthermore, whenever a message name is encountered, the parser knows to look for particular argument keywords (in this case saving) and map them to their corresponding codes, while flagging other keywords as illegal for that message.

4.1.2 References

Certain application terms, particularly class names, allow AppleScript to parse *references* to application data as noun phrases in the grammar. These references correspond to object specifiers transported by Apple events. Here are some examples:

```
the first word of paragraph 22
name of every figure of document "taxes"
the modification date of every file whose size > 1024
```

These references can be parsed because their identifiers are known to be of particular types at parse time. Word, paragraph, figure and file are known to be class names, whereas name, modification date and size are known to be property names. The following syntax description shows a subset of what is possible:

```
<noun phrase> ::=
        <class> <expr>                          -- element class indexed by
position
        | some <class>
        | middle <class>
        | <class> named <expr>                  -- element class indexed by a
property
        | <class> id <expr>
        | every <class>                         -- ranges of elements
        | <class> from <expr> to <expr>
        | <class> <expr> thru <expr>
        | <noun phrase> whose <expr>            -- elements satisfying a test
expression
        | <noun phrase> of <expr>               -- subparts of container:
        | <expr>'s <noun phrase>
```

At runtime, when events are sent that contain references, equivalent object specifiers are constructed for references that contain the appropriate Apple event codes.

The application reference is specially recognized by AppleScript. Application references are used to determine the allowable terms at parse time, as well as the recipient of a remote message at runtime. This makes it possible not only to target a series of commands to a particular application, but also to flow data from one application to another:

```
copy the name of the first window of application "Excel" to ¬
    the end of the first paragraph of app "Scriptable Text Editor"
```

This technique as well as using local variables to hold intermediate results enables scripts to operate independently of the user's global clipboard – the normal mechanism for inter-application transfer.

4.1.3 Handlers and Properties

In addition to sending messages to remote applications, AppleScript allows functions and procedures to be defined within scripts. We call these *handlers* collectively.

Handlers may have names which are user-defined, or which come from application terminology. They may have positional parameters, or prepositional (keyword) parameters. Handlers are defined by the to or on construct:

```
to square(x)
    return x * x
end
on quit x saving s
    if ask("Really quit?") then
        continue quit x saving s
    end
end quit
```

Here, square is a user-defined handler whereas quit comes from application terminology and has the saving keyword parameter.

Handlers can be invoked by sending messages. Messages are targeted to the script itself, unless some other object or application is targeted with a tell statement. The special variables me and it (or correspondingly my and its) can be used to disambiguate whether to send a message to the current script or the remote target:

```
tell app "Microsoft Excel"
    set cell 2 to my square(cell 1)
    quit me saving no
end
```

This script will retrieve cell 1 from Excel, square its value using the local script's square handler, and put the result in cell 2. Then it will quit the current script without saving. Had the me been omitted, the script would have quit Excel instead.

Properties can be defined in scripts which may be used as persistent global variables:

```
property numberOfTimesRun : 0
```

When the program that loaded the script quits, it saves the script out to disk, saving along with it all its updated global properties.

4.1.4 Script Objects

Scripts may also be treated as first-class values within the language by bracketing a group of handlers and properties within a script construct. When executed, this construct creates a *script object*. Script objects are a natural extension of the top-level script concepts, and are easier to grasp by scripters than the more complex class/instance model.

Script objects may be targeted by tell statements which allow them to receive messages. This allows simple object-oriented programming. Script objects also allow single inheritance by delegating unhandled commands to the value in their parent property. For example:

```
script point
    property x : 0
    property y : 0
    on move...
end
script coloredPoint
    property parent: point -- inherit properties and handlers from point
    property color : red
end
```

New instances can of these scripts can be constructed by copying them. Copying recursively copies all properties, including the parent. A more class-like notion can be obtained by writing handlers that construct and return script objects.

Script objects play a central role in customizing applications. Some applications allow entire scripts to be attached to their objects thereby modifying default behavior. For example:

```
script myButtonScript
    on hilited ...
end
set the script of button 1 of window "Welcome" of app " " to myButtonScript
```

Since the destination of the above set statement is a remote application, a copy of the myButtonScript along with it's current properties will be transported over the network to the destination and reconstituted as an equivalent script object. This ability to migrate entire programs around the network has been popularized Telescript [8] is termed *remote programming*. AppleScript differs from Telescript in that scripts do not themselves "go" while retaining their execution state, but must be explicitly sent by some other script, and then executed when they arrive.

4.1.5 Dialects

More than half of Apple's market is international, and we felt that it was inappropriate to limit the script-writing populous to only the English-speaking. Dialects were introduced as a way to internationalize AppleScript.

A dialect specifies the syntax of AppleScript. The examples in this paper are presented in the English dialect of AppleScript. Other dialects exist for Japanese and French, while others are under development. For example, here are translations of scripts into other dialects:

English	**the first** character **of every** word **whose** style **is** bold
Japanese	スタイル＝ボールドであるすべての単語の最初の文字
French	**le premier** caractère **de tous les** mots **dont** style **est** gras
Professional	**{** words **|** style == bold**}**.character[1]

Dialects work by dynamically loading lexing and parsing tables, and printing routines. The parser calls constructors to create a parse tree in *Universal*

AppleScript—a dialect-independent representation of AppleScript's language constructs. The nodes also contain formatting information to specify details of the printed representation (e.g. whether to use is or =). The dictionary of events and objects read from an application's terminology resource (discussed in section 4.1.1) are tagged with bits a dialect can use to indicate plurality, masculine/feminine, etc. As a result, the syntax of an AppleScript dialect can closely approximate the structure of a natural language. A “Professional” dialect is also under development, which resembles C++.

5 Script Management

Script management is supported by an application program interface (API) that allows an application to create, execute, display, and store scripts [2]. The OSA API is a generic interface between clients of scripting services and scripting systems that support a scripting language. Each script is tagged with the scripting system that created it, so clients that are only using scripts can handle multiple kinds of script without knowing which scripting system it belongs to.

At its simplest, the script management API supports the construction of a basic script editor that can save scripts as stand-alone script applications. A second level involves attaching scripts to objects in an existing application. These scripts are triggered during normal use of the application. Finally, complete interface builders can be created that construct applications from interface parts with scripts to provide behavior. By embedding OSA into the operating system, scripting becomes a pervasive scripting service.

5.1 The OSA API

The OSA API is centered around the notion of a *script*, as shown in Figure 2. A script is either a data value or a program. Many of the routines in the API are for translating between scripts and various external formats: script text, formatted strings, storage format, and Apple event descriptors. The most important routines, however, are for executing a script or sending a message to a script.

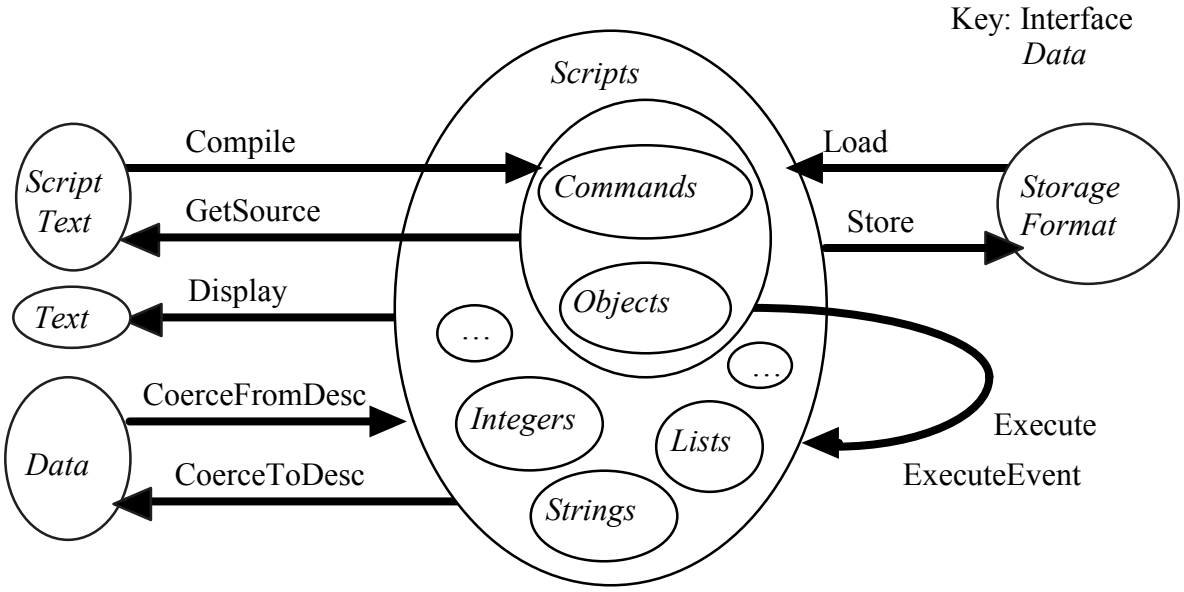

Figure 2: Schematic of the OSA script management API.

Execute/ExecuteEvent: Request that a script handle a message. Execute sends the run message to a script whereas ExecuteEvent invokes a handler in a script which corresponds to an incoming Apple event. In the course of handling the message, the script will typically send or delegate more messages (to itself or remotely), and then produce a value. Execution may be suspended and interleaved with other executions to support threaded scripts.

Compile/GetSource/Display: Convert between scripts and text. The difference between GetSource and Display is that the former prints strings as programs (with quotes and special characters), while the latter prints them in a more human-readable format.

Load/Store: Load and store support persistence. The scripts may contain properties with associated data; these properties are stored and loaded along with the rest of the script.

CoerceFromDesc/CoerceToDesc: Convert between scripts and binary data in the form of Apple event descriptors. For example, after executing a script, a numeric result may be retrieved in binary form using CoerceFromDesc.

5.2 Uses of the OSA API

5.2.1 Script Applications

A script application is a script stored in a document that runs within a simple application shell. They can be double-clicked by users, or placed into a startup folder to launch when the machine starts up. Script applications pass all events sent to the application directly to the script they contain. The script may handle the

standard Open Application, Open Documents, Print Documents, and Quit events sent by the Macintosh System 7.

5.2.2 Customizing Applications

Since scripting is a pervasive system service, applications may use it as their "macro" language rather than implement their own. By doing so they reduce the burden on the application developer to provide all the scripting power a user might want, and the burden on users to learn many special-purpose languages. The application can also use the OSA API to *attach* scripts to its existing objects. These scripts are executed during the normal operation of the application, allowing users to customize its behavior. The interesting situation is determined by the application.

Mail systems: The arrival of mail of a certain priority or with a certain sender or topic.

Speech recognition: Recognition of a certain speech element. The executions may be chained to build up a larger-scale representation of the phrase. The Macintosh AV™ speech recognition system uses the OSA, thus any scriptable application can be driven using speech.

Calendars: Triggering based on time: repeating or intermittent execution. Calendar programs already have well-developed user interfaces for dealing with time. The addition of scripting services greatly expands the capabilities of these systems.

5.2.3 Interface Builders

The most sophisticated use of the OSA API is in generic interface builders. These programs provide generic interface elements. The interface elements post messages when user interacts with them. The user arranges the elements into windows, menus, and dialogs. Scripts may be attached to any object in the interface to intercept the messages being sent by the interface elements and provide sophisticated behavior and linking between the elements. Several interface builders have been implemented with OSA support, including HyperCard™; Frontmost™, a window and dialog builder; and AgentBuilder™, which specializes in communication front-ends.

5.3 Script Systems and Packaging

Any language processor (compiler or interpreter) that supports script management operations can be registered as a scripting system within the open architecture, because the API is generic. Clients load, store, execute and display scripts using the API, without having to deal with the different scripting systems involved. Scripts are tagged with a "creator code" that is used by the generic dispatcher to locate the correct scripting component. Only script editors, which create scripts, need to select what scripting language they desire.

AppleScript is just on of many scripting systems that have already been implemented, by multiple vendors, including UserLand Frontier™, CE Software's QuicKeys™.

Scripting systems may either be packaged as system extensions, or as complete applications. The AppleScript extension is shared by all clients, and takes up between 500K and 800K of memory depending on whether it is executing or compiling scripts. It can run with as small as an 8K application script heap.

6 Related Systems

6.1 The Unix Shell

The Unix shell [5] is based on two simple and powerful concepts: streams and files of bytes. Programs are filters that take input from streams and produce output in streams. Files are stores for streams, and all system data and attributes are reified, or made concrete, as files. The need for structure is satisfied by encoding structure into streams, leading to a pervasive use of parsing. The scripting languages (shells) also tend to be text-based. Their semantics is given by text substitution rules. Since there is no distinction between programs and data, complex quoting and unquoting mechanisms are required. As a result, the languages have little referential transparency: an expression that works in one place will very likely not do the same thing if moved to a similar location. The absence of standard argument notation for programs also leads to arbitrary idiosyncrasies that a user must master.

6.2 OLE 2.0 Automation

OLE 2.0 Automation [7] is similar in goals to the distributed messaging portion of OSA. Other aspects of OLE are beyond the scope of this paper; visual embedding, for example, is comparable to the multi-vendor OpenDoc effort to define an embedding architecture. OLE lacks standards for application messages and object structures. Nor does it have an open script management API.

One technical difference between the systems is that OLE allows messages to connect directly to the low-level objects and methods in an application. Apple events in the OSA are more abstract, and must essentially be interpreted by the receiving application. There are merits on both sides of this design choice. OLE makes more assumptions about the application architecture, allowing it to define a framework for marshalling objects being sent remotely. The OSA, on the other hand, supports more high-level services like recording.

6.3 CORBA

Although the high-level goals of CORBA [6] and OSA are similar, the approach taken is significantly different. While CORBA has focused on low-level infrastructure and efficiency of messaging and transparent object migration, OSA has taken a broad approach that connects user-level scripting with basic messaging and reference constructs. The technical differences appear in the relationship between remote messaging and procedure calls. A stated goal of CORBA is full scalability, including support for small-scale implementation objects that migrate and have persistent global IDs. One might imagine splitting an application down the middle, putting half its objects on one machine, and half on another. This is not a goal of OSA, which clearly distinguishes messages from procedure calls. An OSA programmer knows when they are building in the flexibility and power of remote

messages, and they are willing to support its cost. Instead of global persistent object references, the OSA relies upon the host operating system to provide references to large-scale objects (documents and applications), whose internal objects are referenced by queries. Persistent references may or may not be supported by individual applications, but this is not mandated by the OSA.

7 Conclusion

The Open Scripting Architecture is a significant step toward a revolution in end-user computing similar to the one caused by adoption of graphical user interfaces ten years ago. That earlier revolution introduced users to the power of spontaneous computing through windows, menus, and icons. Good metaphors, simplicity and uniformity across applications were key elements. The next revolution will allow them to automate, integrate, and customize their computing processes through references, messages, and scripts. The metaphor of structured objects covers all data, preferences, system attributes on the machine and network. Again, uniformity and simplicity are key. The Open Scripting Architecture and AppleScript provide the infrastructure for this revolution.

References

1. Apple Computer. AppleScript Language Manual, English Dialect. Addison-Wesley, 1994.
2. Apple Computer. *Inside Macintosh Volume 7: Interapplication Communication*. Addison-Wesley, 1993.
3. Clark, Richard. Apple Event Objects and You. Develop, the Apple Technical Journal. Issue 10 (May) 1992.
4. Cook, William and Harris, Warren. *Designing a Modern Scripting Language*. University Video Communications, 1993.
5. *Unix User's Manual.* Berkeley, CA 1984..
6. *Common Object Request Broker: Architecture and Specification.*. Object Management Group. Document Number 91.12.1, 1991.
7. *Object Linking and Embedding 2.0*. Microsoft, 1993.
8. White, James. *Telescript: The Foundation for the Electronic Marketplace*. General Magic, Inc., November, 1993.