An Object-Based Visual Scripting Environment

Gerti Kappel, Jan Vitek, Oscar Nierstrasz
Simon Gibbs, Betty Junod, Marc Stadelmann, Dennis Tsichritzis

Abstract

Scripting is a programming technique in which applications are constructed by composing specially designed, pre-packaged software components using a restricted set of scripting operators. Scripting simplifies programming by cutting down the number of the syntactic and semantic features found in a complete programming language, yet is inherently open-ended in that software components can be provided by a separate target language. We explore scripting models in which the basic components are written in an object-oriented target language. We introduce a *visual scripting tool* as a script development environment. Visual scripts present components and links graphically, and a visual scripting tool supports the construction of scripts through the interactive editing of scripts' graphical counterparts.

1 Introduction: What is Scripting?

We shall take as a working definition of a *scripting language:* "a compact notation for constructing applications from pre-packaged components written in a target programming language," with the understanding that scripting does in general not replace basic programming. We can characterize the problem that scripting addresses in two ways:

- to facilitate rapid application development and evolution through a compact, high-level notation that permits applications in specific domains to be constructed out of reliable, pre-packaged software components
- to reduce the details in application programming by focusing the programmer's attention on a restricted, but powerful set of operators for combining software components

We further distinguish between a scripting language, and the underlying *scripting model.* The scripting model determines what kind of software components can be scripted, and what the paradigms for combining them will be. Scripting models will typically be quite different from the more general computational model of the target language in which the software components are programmed, since their purpose is to highlight a particular, application-oriented programming paradigm, rather than general-purpose programming.

To illustrate the idea of scripting, we shall take a quick look at a couple of scripting languages. In both cases, a single scripting model provides the semantics of the scripting language. In this paper, by contrast, we shall propose a *visual scripting language* which will, ultimately, support a variety of different, application-oriented scripting models.

As a first example, let us consider the "Bourne shell" [Kernighan and Pike, 1984]. The software components of the shell's scripting model are Unix(TM) commands programmed in C or any other programming language available in the environment, including the shell itself. One of the most powerful operators provided by the shell for "glueing" together commands is the Unix *pipe*, which binds the output stream of one process running a command to the input stream of another. A command that uses both input and output streams is a *filter*. A series of filters, each of which performs a simple function, can be easily linked together to accomplish a more complex task, such as extracting records from a file, reformatting the records, sorting them, generating line numbers, and sending the output to a printer. Communication between processes can be further controlled through the use of (local) shell variables, (global) environment variables, files and signals. The combination of the shell's fairly simple scripting model and the rich collection of software tools in the environment makes it an extremely powerful tool for quickly building easily modifiable applications. A possible extension to the Unix scripting model allowing multiple pipes between processes is described in [Dami, 1989].

A second example is the temporal scripting language TEMPO [Dami, et al. 1988], which is used to specify temporal constraints amongst objects that have a notion of continuous time, i.e., duration. Temporal objects can be animations, musical fragments, or other entities with duration. A temporal script names or instantiates temporal objects, and specifies synchronization constraints over the objects (e.g., A terminates before B begins). The result is a new temporal object that is constructed from several high-level components.

Both shell scripts and temporal scripts support a scripting model which is a simplified view of the target programming model. In both cases programming of the individual components is left to another, more suitable programming language. Scripts reduce the amount of detail required in programming applications due to the restricted programming model, thereby simplifying debugging and modification, and they improve reliability and maintainability through the reuse of stable components. One can sometimes script within the target programming language itself (e.g., using macros, or overloaded operators), but a separate scripting language can be justified by its clear presentation of a suitable scripting model, and by the reduction of syntactic baggage of the target language. Mechanisms that are useful for general-purpose programming can get in the way when scripting. Although scripting is neither appropriate for nor intended to replace conventional programming, it is *open* in the sense that one can (and is encouraged to) add new components, written in the target language, to the environment as the need arises. Furthermore, it is often the case that scripts themselves end up as part of the environment of reusable components. Both shell scripts and temporal scripts, for example, can be reused as components in yet other scripts.

Not only can we distinguish between scripting and general-purpose programming, but we may also draw a distinction between a scripting language and a very high-level programming language (VHLL). In our view, scripting is fundamentally open-ended, in that it relies on a more general target computational model to supply it with software components. A VHLL, on the other hand, is a special-purpose programming language that stands on its own. A VHLL may provide "hooks" to other languages, but this is always incidental. There is a superficial similarity between scripting languages and VHLLs only in that the "primitives" provided by both operate at a high level (i.e., relative to machine instructions).

Scripting need not always take place within the confines of a textual scripting language. A *visual scripting language* presents scripts in terms of a combination of graphics and text. Instead of editing textual scripts, the programmer edits graphics. As with textual scripting languages, we can separate the purely *syntactic* aspects of the underlying scripting model (i.e., what are the components, and how may they be composed), and the *interpretation* of scripts. To provide a framework for describing scripting models, we identify the following syntactic primitives:

- there is a set of software *components*, each of which provides a certain well-defined functionality; a script may either refer to existing components or instantiate new ones
- each component has a well-defined interface consisting of a set of unbound typed *ports*
- ports may be bound via typed *links*
- scripts with unbound ports can themselves be packaged as reusable components

In the case of Bourne shell scripts, components are Unix commands, ports are arguments, input and output streams and environment variables, and links are output to input pipes, and argument bindings. With temporal scripts, the components are temporal objects, the ports are instantiation parameters and points in time, and the links are name bindings and temporal synchronization operators. Links are generally typed and may be directional. For example, one may bind an output stream to an input stream, but not vice versa. The type of a link includes the types of ports it may bind.

In both of the examples, and in general, scripting works through parameterization: a variety of applications can be constructed through scripting of software components primarily because they have unbound ports.

In addition to the purely syntactic aspects of scripting, there is typically a *script interpreter* that knows how to instantiate and bind software components, and possibly perform some global error-checking, or analysis of the configuration specified by the script to determine how the script is to be executed. For example, TEMPO's script interpreter analyses temporal constraints to determine when and how the temporal objects must be "played."

A complete scripting model therefore describes not only the kinds of components, their ports and possible links, and a mechanism for packaging scripts as new components, but also indicates how the scripts are to be interpreted. Since each scripting model addresses a particular kind of problem domain, there can be no such thing as a "general-purpose scripting model." Shell scripts would be inappropriate for temporal scripting, as would temporal scripts be for shell scripting. On the other hand, we can imagine a general-purpose scripting language, parameterized by various scripting models. Such a scripting language would provide us with a standard notation for components, ports and links, and for packaging scripts, but would also allow us to define the *types* of ports and links required by a particular scripting model and would provide a standard way for mapping scripts to their interpretation.

We propose a visual scripting language as such a general-purpose tool, and an object-based scripting model suitable for scripting object-oriented applications.

In section 2 we will examine the role of scripting in object-oriented application development, thereby motivating the need for and use of an object-based scripting model. The principal features of such a model are introduced. In section 3 and 4 the concepts of the visual scripting tool are discussed and the visual scripting language is explained via a simple example. The basic architecture of the Visual Scripting Tool is introduced. Section 5 surveys other scripting techniques and the similarities/differences to our scripting approach are outlined. We conclude with some directions for future work.

2 Scripting in an object-oriented application development environment

How does the idea of scripting apply to object-oriented application development? Why should we use another "programming" paradigm instead of staying within the object-oriented one?

Object-oriented programming (OOP) has gradually become accepted in recent years as a sound technique for building modular, maintainable and portable systems, and as a powerful technique for "raising the level of programming" through libraries or "toolkits" of pre-packaged object classes. Inheritance and type parameterization (i.e., "genericity") are the two main mechanisms exploited by OOP that enhance software reusability. Nevertheless, reusable object classes have, until recently, tended towards "basic software" (e.g., user-interface components and standard data structures) rather than application-oriented software. Construction of applications still requires substantial programming effort, and design and implementation of new object classes. With scripting, high-level functionality is provided by the pre-packaged components, which might be built up using scripting itself.

A goal in application development is to build reliable, easy to change application systems. The object-oriented approach provides us with some mechanisms in this direction. Whereas with traditional programming the application system is built as a (more or less) homogeneous block, object-oriented programming supports the view of an application as a set of objects, communicating and cooperating in a certain way. Thereby each object has *local behaviour*, represented by its operations, whereas the *global behaviour* of an application is implicitly given through operation invocation.

For providing a better understanding of the behaviour of an application system, and for providing support for the evolution of the same system, we need mechanisms for handling explicitly overall application behaviour. In this context scripting is crucial for two reasons. First, scripting isolates the coordination and the interfacing between objects in a separate structure, the script, which can be viewed and edited independently. In this way the global behaviour of an application can be understood and modified without changing the individual objects' behaviours, but by changing the script. Second, it allows users not well-versed in detailed programming to try out ideas by putting together scripts of available objects. In this way, reusability is enhanced, and there is less need for communication of applications requirements between application developers and the developers of libraries of reusable components.

We introduce a scripting model which is explored in our first prototype. We call it an object-based scripting model, since our basic software components are objects written in an object-oriented target language. We use the term "object-based" as it is defined in [Wegner, 1987]. There, a language is object-based if it supports objects as a language feature.

2.1 An object-based scripting model

The basic building blocks of a script are components, ports, and links. These syntactic constructs are interpreted in the object-based scripting model in the following way:

Components:

We distinguish three different types of components. These are objects (written in an object-oriented language), Unix executables, and pre-packaged scripts.

The visible part of an object in a script is its interface, i.e., its message selectors (operation names) with input parameters and return values. Using an object in a script is basically selecting one or several operations ("operation invocation") and binding input parameters and return values of the selected operations ("object binding"). Input parameters and return values are represented as input and output ports of objects. A special kind of object, an *object constant*, is introduced, too. The object constant facilitates the interactive specification of data of a certain type, which can be directly linked with an input port of the same type.

Ports:

The ports of a component represent unbound parameters of the component. A port is defined by specifying:

- its *type*: we distinguish between simple types (like integer, real, boolean, and char), object types (both system-defined and user-defined), and stream of {simple,object} type
- its *direction*, i.e., if it is an input or output port
- its *mode*, i.e., if the port must have received a value (mandatory) or not (optional) before execution of the component (this holds only for input ports)

Depending on the type of the component the ports have different meaning. If the component is an *object*, input and output ports of an object are arguments and return values of the operations invoked on this object within this script. Only the parameters of operations actually invoked need to be bound within the script.

If the component is a Unix executable, we distinguish between object-oriented application programs written in the target object-oriented language, and Unix commands. If the component is a *Unix command*, the input ports might be associated with arguments (flags, files to read) and the standard input stream. The output ports are the standard output and error streams.

If the component is an *object-oriented application program*, the allowed types of the ports of this component are defined by the execution model of the target language. We could imagine parameters, which allow one to specify the "entry point" of the application, i.e., the operation which is invoked when running the application. Other parameters might be bound to objects which will be involved in this application.

If the component is a *script*, input ports represent the values it needs in order to execute. As a result of the execution it produces data on its output ports.

A component can be executed only when all its mandatory ports have been linked and have received values. A port is considered to have received a value either if a new value has arrived, or if a default value had been defined for that port. Default values can only be specified for optional ports.

Links:

Links specify how ports are bound together. For object-oriented application development several types of links are appropriate, such as

- data flow links (passing values between components)
- control flow links (passing control between components)
- composition links (components represent parts of objects, which scripts combine to generate new kinds of objects)
- inheritance links (components are objects, and the links establish the inheritance hierarchy)
- instantiation links (binding of references to other objects, type parameters, and initial values of instance variables)

A more application-oriented scripting model would emphasize kinds of components, ports and links appropriate to a particular application domain, such as public administration applications, or financial applications.

The scripting model that we describe in the paper distinguishes between data flow links and control flow links. This design decision is based on two considerations: first, we want to explore the idea of integrating objects and Unix executables in the same scripting environment, which motivates a value passing model; and second, the model we adopt represents a *first* step to investigate the possibilities of scripting with objects.

A *data flow link* can combine an output port with an input port, if the type of the output port *conforms* to the type of the input port, i.e., the output must meet the input specification. In a typed object-oriented language such as Eiffel [Meyer, 1988], an output of type `Student` might be valid as input to a port of type `Person`.

Values produced by a port have to conform to its type, but are not necessarily equal to the port's type. This implies that it is not possible to guarantee in all cases that a link is valid. In linking an output port b of type B to an input port a of type A two possible cases are distinguished:

- B conforms to A: the link is valid.
- otherwise: the link will be validated at run-time.

For example, an output of type `Student` is always valid as input to a port of type `Person`, but we can only know at run-time if a `Student` may be input to a port of type `Female`.

The interpretation of the value which is passed via a link depends on its type. If the value is of simple type the passed value is taken "as such." If the value is of object type the passed value is the reference to an object instance of this type. In the former case several independent copies of the same instance (in the sense of "5 is an instance of integer") exist; in the latter case there exists no copy of an object instance.

The data-flow interpretation of scripts can lead to some ambiguities. A special kind of port for defining sequencing (i.e., control) information and a corresponding *control flow link* give the script designer the possibility of defining explicitly the order of execution of a script.

All components, except object constants, have a sequencing port (represented by a clock icon). This port is both input port and output port. When it is linked the arrow is interpreted as an "execute before" relationship. But even when its sequencing port has received a value, a component will execute only when all of its other ports have also received values. The sole difference with the other ports is that a sequencing port is always present and that its linkage is not mandatory. Unlinked sequencing ports are entirely disregarded for a script's interpretation.

This scripting model does not support conditionals and loops. This is a conscientious choice to keep the scripting model as simple as possible. Control abstractions could be implemented, for example, by specialized components which have multiple sequencing output ports.

The execution model for a script is based on the message-passing model of the underlying target object-oriented language and on a data flow model. Data transmitted through links drive the execution of the components and the invocation of operations, respectively. The implementation of links will depend on the type of components they connect. For example, a link between two objects typically implies intra-process value passing, but a link between an object and an executable implies inter-process communication.

3 Visual Scripting Tool

The visual scripting tool that we describe applies the idea of scripting discussed earlier to object-oriented application development through the medium of a visual scripting language. Visual scripts present components, scripts, ports and links graphically, and a visual scripting tool supports the development and editing of scripts through the interactive editing of scripts' graphical counterparts. The motivation for this approach is the conviction that a 2-dimensional representation of an application inherently captures the structure of the application better than a 1-dimensional, textual representation, provided that the information presented in the graphics does not exceed a certain threshold. By focusing on visual scripting rather than visual programming, we seek to keep this information threshold at a manageable level, namely at the level of the restricted computational model of the scripting language.

3.1 Generic Scripting Environment

The scripting approach presupposes a scripting environment for the development of new applications. This environment is in fact a major part of the success of established scripting systems. We use the term "scripting system" to group a number of different tools and languages that feature some scripting characteristics (see section 5).

The scripting environment should not be confused with the scripting model. It covers issues as various as user-interfaces, available components and on-line documentation. For example, HyperCard is a success thanks to its simple and intuitive user-interface even though its scripting model is limited. The Unix shell language on the other hand owes a good part of its success to the number of well documented software components available (and none to its user interface).

A generic scripting environment provides a graphical user-interface, a large number of re-usable components, and on-line help to be able to re-use components. Such an environment is generic because it is not dependent on any particular scripting model. By this we mean that the interpretation of the components and links are not "hard-wired" in the environment. Most of the necessary features of this generic environment will be provided by the visual scripting tool that we are developing. However, some features are part of the more general "application development environment." These features are outside the scope of our work and we consider them as requirements of the Visual Scripting Tool (VST).

The following features are part of the script development environment provided by the VST (those marked with an asterisk* have been selected as essential for our first prototype, described in section 3.2):

***Graphical Representation:** A graphical representation and multiple views of scripts are part of the scripting environment. A *view* of a script shows certain aspects of a script and hides others. For example, only components and links representing the control flow are shown, or only the hierarchy of components used in a script is displayed. Views also show model specific aspects of the script (e.g., certain types of ports or components only).

***User Interface:** The user interface for composing a script is a multi-window environment. The main window will be the workspace for graphically editing a script. Several other windows will be used, for example, to display the available list of components, to show additional information, and to interact with the user.

***Script Execution:** Two execution modes are provided. Either the script is interpreted as a whole, or it is interpreted "on the fly," as it is being edited. The latter is particularly useful for the user to have a direct feed-back. A completed script can also be compiled into the target object-oriented language.

Monitoring Features: The scripting environment will provide means to monitor the execution of the composed scripts and even to inspect the states of the individual components. This last feature amounts to have object monitoring features. A more complete discussion on monitoring can be found in [Kleyn and Gingrich, 1988].

On-line Assistance: At each step of the scripting process the user can require help from the scripting tool. This help will consist of textual information, examples of component re-use, and of consistency checking on the current script (e.g., conformance checking of linked ports).

***Library of Basic Components:** Some basic components will be provided to simplify the creation of new applications. For example, such a library may include translator components for linking components with non-compatible port types.

To enhance the applicability of the VST, the following requirements for a developing environment have to be taken into account:

Software Information Base: A software information base to store comprehensive information about all the available components is necessary in order to provide both on-line assistance and information for the script interpreter [Gibbs, et al. 1989].

Reusable Components: The quality of the scripts produced with our tool depends on the quality and number of available components. Also, the use of scripts will depend on the components' level of abstraction. If they are application oriented and high level, scripting might even be suited for end-users.

A Target Language Interpreter: An interpreter for the target language eases the task for the script interpreter. Some component monitoring features will also depend on the availability of such an interpreter.

3.2 Prototype Overview

The Visual Scripting Tool will implement the generic scripting environment. The design of the VST attempts to preserve a clear separation between the generic environment and features specific to a scripting model. Therefore the VST is constructed in a modular way, breaking down in two main parts, namely an user-interface area and an execution area (figure 1).

The user interface area will handle the generic aspects of the environment, i.e. the syntactic part of the VST. It is composed of a generic script editor and of an interface manager. The script editor is a purely graphical tool to edit the visual representation of scripts. The interface manager will handle the interface between the user and the VST, and it will manage the display windows, menus, requests for components, etc.

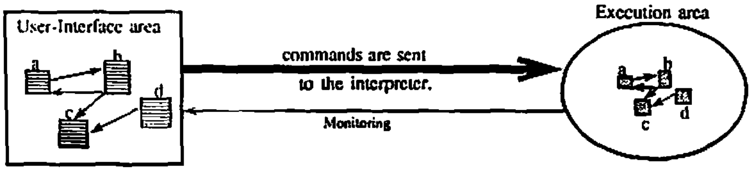

Figure 1

The execution area is composed of the actual executing software components. A script interpreter has the task to interpret the script according to a scripting model. This script interpreter will link the abstract representation of the components in the user interface area to their physical counterparts in the execution area. The script interpreter represents the semantic part of the VST. The clear distinction between the user interface area and the execution area resembles the idea of user interface toolkits which are used for building a user interface independent of an application (e.g., [Weinand and Gamma, 1988]). The Model-View-Controller paradigm of Smalltalk-80 [Krasner and Pope, 1988] motivates a similar approach.

The user will be able to edit a script in the user interface area and, at the same time, interact with the scripted application in the execution area.

For all components used in a script, an abstract description is prerequisite. This description is the source of information for the script editor to support the designer in interactive scripting and to perform consistency checks. The description is also visible in the user-interface area.

The description is implemented as a complex object composed of structural information (e.g., name and type of component, name and type of ports, execution mode, etc), textual information (e.g., documentation), and graphical representation characteristics. A basic `Component_Description` class has been built. If component descriptions with more specialized information are needed this will be achieved by subclassing. Assuming an application development environment, the component descriptions are included in the software information base.

A prototype version of a generic script editor has been implemented in Objective C [Cox, 1986] with the user interface library ICpak [PPI, 1988], running on SUN/3 workstations. One feature of the script editor is an interactive aid for building a component description. Currently, we are designing a script interpreter based on the outlined scripting model.

4 Building a Script

To introduce the visual scripting language and to provide an idea how scripting is used with an underlying object-based scripting model, we present an example that shows how a script is built. The example, adapted from [Ingalls, 1988], entails the implementation of a *FileBrowser*. For the construction of the script we assume the existence of a number of complex components whose use is understood. Note that we only use object components and script components in this example.

The goal is to build a *FileBrowser* tool that allows a user to select a file from a set of files that all have a common pattern in their name and to edit this file.

As a first step a new script is created (figure 2). This displays an empty frame in the workspace window in which components can be placed and linked. As we shall see later, ports can be added to this frame in order to parameterize the script. Figure 2 shows a script frame with the default sequencing port (the "clock" icon on the left) and the graphical representation of a component. This component is an object of the Communicator class. It is used to interact with the user.

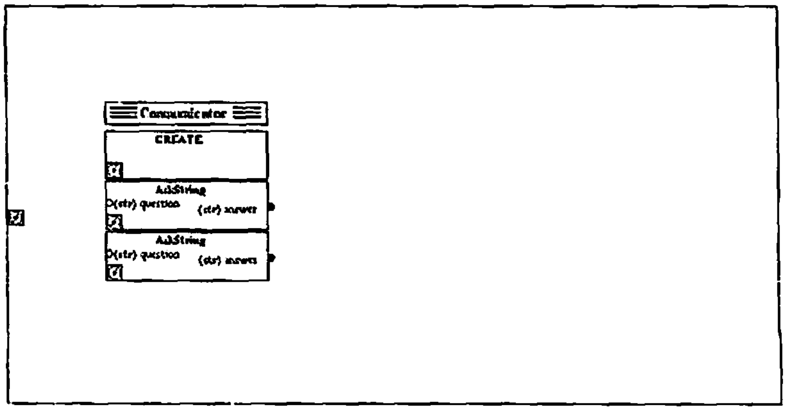

Figure 2

Object components have a graphical representation which is different from the representation of script components. This is due to multiple operation invocations: a single object can be used as the target of multiple invocations, to possibly multiple operations. Each object is visually represented by a header that indicates its type. All invoked operations will be displayed under this header, in the order chosen by the script designer. A menu attached to the object's header presents the object's interface, and can be used to select an operation. The first operation called must be one that instantiates the object, by explicitly creating a new instance, by retrieving an existing object from the object store, or by binding to the value of an output port of another object.

In figure 2 three operations of the Communicator object have been selected for execution and each of these operations have unbound ports. The ports of a component are indicated by small circles. Those inside a figure, denote input ports, i.e., they expect to receive data. Circles on the outside of a figure are output ports and will produce output as the result of the execution of the component. In this example, the type of the port is written in parentheses. A name is also given to each port for clarity.

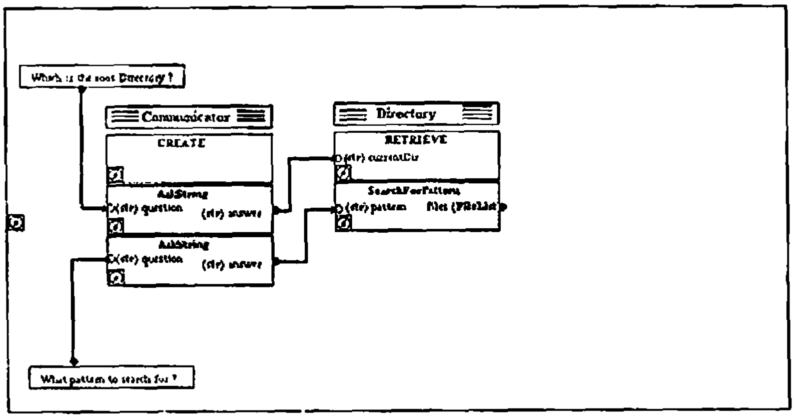

Figure 3

Links are represented by arrows connecting output to input ports. In order to link the ports of the Communicator object other components will have to be created. Figure 3 shows two string constants linked to the input ports of Communicator. Constants of any simple type can be created. They have only one output port and cannot be executed. When a script is executed, they are the first objects created.

In figure 3, a directory object has also been added, and the input ports of its operations are linked to the output ports of the AskString operations of Communicator. These operations pop-up a question window and return a string with the user's answer. The first string is used by the Create operation of Directory to set the current directory. The second string is the pattern that will be matched against all the file names of the current directory. The SearchForPattern operation will produce a list of matching file objects.

Once the set of files has been retrieved, the *FileBrowser* tool has to display the list of matching file names and let the user choose which file to edit. For this functionality a pre-defined script will be reused.

Figure 4 shows a script called `Chooser` added to the script frame. Since scripts are components with a single functionality, they have a simple graphical representation. They are displayed as "black-box" components (their internal composition is hidden) with only their name and ports visible. The `Chooser` script will return one of the objects out of the list it receives as input. This object is bound to a reference of the class `File` that will be edited with the `EditFile` operation. The link binding the `Chooser` to the `File` object is displayed as a dashed line to indicate that the ports' types are different and that a run-time check will take place to ensure that the produced object conforms to the `File` type.

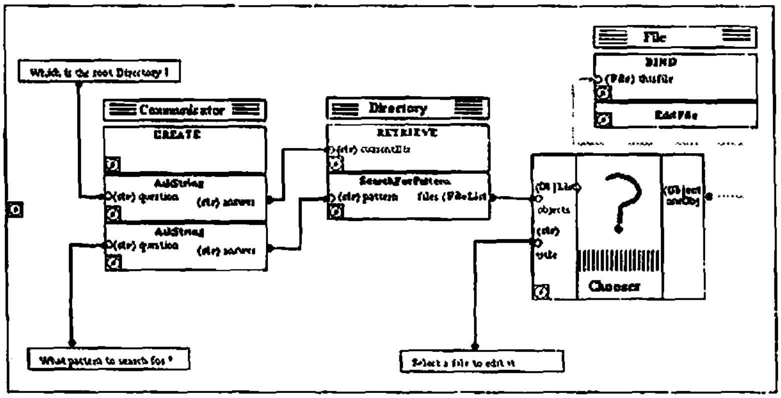

Figure 4

The script, as shown in figure 4, is executable. The script interpreter specifies the order of execution according to the following rules:

- constants are always created first,
- script components will be only executed once all their mandatory ports are bound and have received values,
- operations are invoked on objects once all their mandatory ports are bound and have received values,
- an object instantiation operation always is the first operation invoked on a particular object in a script,
- the graphical order of operations of one object determines the execution order, provided the mandatory input ports of the operations are bound and have received values.

In our example the order of execution is unambiguous. However, this does not hold true in general. Therefore, we give the script designer the possibility to specify control flow in addition to data flow.

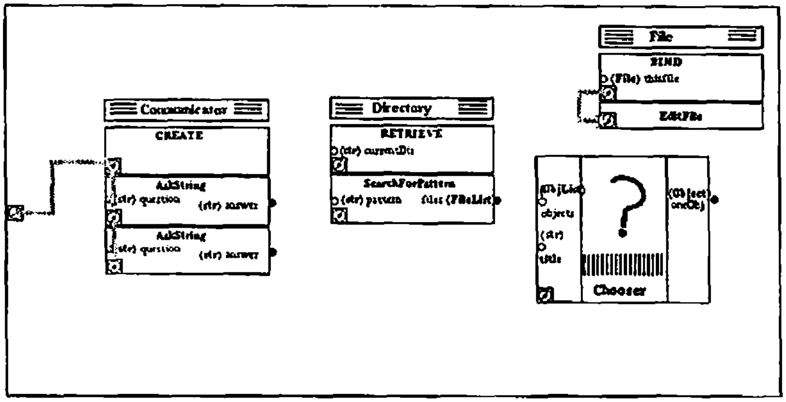

Figure 5

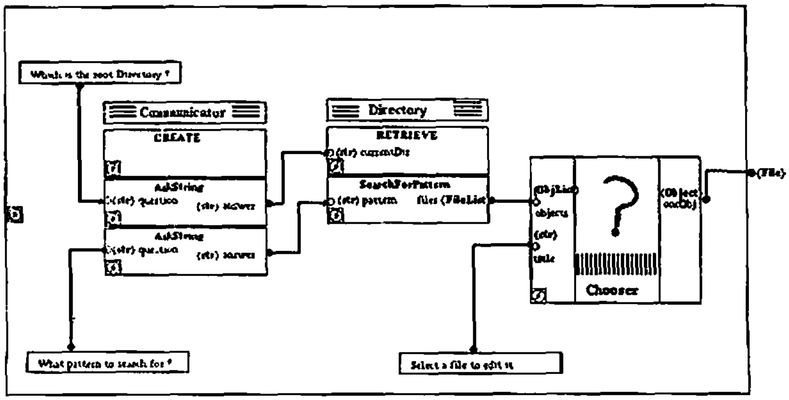

Figure 6

For the purpose of clarity, we show in figure 5 how an explicit partial order of the example script is defined. When placing the sequencing information in a script, other links and constants are hidden for readability reasons. The sequence of operation invocation on the `Communicator` object is fully specified, and the **`EditFile`** operation of the `File` object is executed after the `Bind`. The first operation of the `Communicator` object is then connected to the "clock" icon on the left of the script frame to indicate the starting point of the script.

To be able to reuse the example script as a component in another script, it has to be made available as component in the development environment. To enhance the reusability of a script input ports for input parameters and output ports for output parameters should be available.

In figure 6 a redesign of the *FileBrowser* is shown. The chosen `File` object is plugged into an new created output port of the script. After having finished the design of the script, a component description of this script is interactively built on demand.

5 Related Work

In this section we compare our scripting approach to other tools and languages which are usually referred to as scripting languages or which provide similar functionality.

Neither the list of comparison criteria nor the list of tools to compare with is an exhaustive one. It is important, however, to place the different scripting notions and related tools in a classification framework, both for supporting a technical discussion about the different features, and for pointing to open problems.

We shall compare the following systems with the Visual Scripting Tool, after providing capsule summaries of each one:

- Unix Shell,
- SERC's Visual Shell tool,
- Tempo,
- HyperCard,
- Fabrik,
- Interface Builder,
- Conic.

The *Unix Shell* [Kernighan and Pike, 1984] is the command language of the Unix operating system. It owes its flexibility mainly to two facts. First, it is possible to extend the number of usable commands by reusing Unix shell programs, so called shell scripts, and any other Unix executable as commands, and second, it provides the pipe operator for redirecting the standard output stream of a command to the standard input stream of another one. Different dialects of the shell are under investigation [Dami, 1989], which further explore useful coordination mechanisms for Unix commands.

SERC's *Visual Shell Tool* [Florijn, 1989] provides a graphical interface for building Unix shell scripts (based on the Bourne shell). It supports the visual construction of

	UnixShellScripts	Visual Shell	Tempo	HyperCard	Fabrik	InterfaceBuilder	Conic	VST
Synt. Issues:								
1. Representation	Text	Visual	Text	Visual & Text	Visual	Visual	Visual & Text	Visual & Text
2. Scripting concepts	Unix execut., pipes, files, arg., env.variables, signals, control abstractions	Unix execut., pipes, files, and arguments [3]	Objects, and temporal operators	Instance initialization, navigational links, and HyperTalk for textual, procedural scripts	Comput. and graphical components, I/O pins, links, control abstraction, and instance initialization	Instance initialization and instance binding via instance variables [2]	Software components, ports, and links	Components, ports, links
Sem. Issues:								
1. interpretation	Unix Shell	Unix Shell	Animated objects with duration	HyperCard interpreter	Bidirectional data flow model	Event paradigm (for UI) and target/action para-	Data flow model	Object binding, operation invoc., data & control flow
2. execution mode	Interpreted	Interpreted	Compiled	Interpreted	Interpreted	Not applicable	Interpreted	Interpreted
3. purpose	General purpose; not good for User Interfaces (UI)	General purpose; not good for UIs	Music, Animation (appl. specific)	Information management systems	General purpose and UIs.	UI and connection between UI objects and appl. objects	Distributed & concurrent systems	General purpose
Target Issues:								
1. target model	C + others	C + others	C++	(Hypertalk, Pascal + others)	(Smalltalk)	(Objective C)	Pascal	o-o language
2. front-end (script) self-cont. (VHLP)	front-end	front-end	front-end	self-contained	self-contained	self-contained (only descriptional)	front-end	front-end
3. openness	Open: any C program can be a component [1]	Open: any C program can be a component	Open: component can be easily added by writing C++ code.	Closed: the components are few and hard-wired.	Closed: there is a fixed number of primitives, hard-wired in the program	Closed: the components are hard-wired in the program.	Open	Open: Unix executables and any object written in the o-o language can become a component.

Table 1: Comparison of scripting systems

[1] Openness is the relative ease to add new basic software components.

[2] InterfaceBuilder is used to describe the look of a user interface and to establish the link between a UI object and an application object. The description can not be executed as such.

[3] The Visual Shell allows only the piping of processes and instantiation of process arguments.

pipelines of "filters" and their immediate execution. A filter might be a Unix command or any application program which reads from the standard input stream and writes to the standard output stream. For each filter, a description file (in a standard format) has to be created which is input to the visual shell tool. Arguments to a filter are specified by filling in an argument form, which is attached to each filter, i.e., to its iconic representation.

The temporal scripting language *TEMPO* [Dami, et al. 1988] (introduced in section 1) is for building applications which need to deal with concurrent activities and special temporal relationships between them (e.g., computer animation, computer music [Dami, 1988]). The main components of TEMPO are animated objects and temporal relationships (operators) between these objects. With temporal operators one can define, for example, sequential execution of objects A and B with A >> B. TEMPO is a textual language; however, a visual description language is under development.

HyperCard(TM) [Shafer, 1988] is a tool for building information management systems, running on MacIntosh PCs. It supports concepts for both programming in a VHLL (called HyperTalk), and scripting using a graphical interface. The scripting features enable the user to manipulate objects by creating them, changing values of their attributes, and attaching HyperTalk code to them. HyperCard predefines five object types related to each other in a fixed, non-extendible hierarchy. These are stacks (a collection of cards), backgrounds (the common parts of several cards in a stack), cards (one window that contains buttons and fields), buttons (e.g., mouse-sensible areas for specifying actions), and fields (the units that contain the information stored).

Fabrik [Ingalls, 1988] is a visual programming kit of computational and user interface components. The connections between the components are realized by linking their output pins and input pins respectively. A type is associated with each component pin. Currently only system-defined types (like arrays and records) are supported. Bidirectional data flow is the underlying model of computation, i.e., pins can serve as input and output at the same time. Most of the time bidirectionality is used for modelling multiple paths of data flow. Applications are compiled into Smalltalk.

InterfaceBuilder [Thompson, 1989] is part of the NextStep application (development) environment, in which every NeXT(TM) program "lives." InterfaceBuilder supports the interactive construction of user interfaces, and the interactive manipulation of values of instance variables of user interface objects. The second feature makes InterfaceBuilder a scripting tool, because it is possible to install/change a link between a certain interface object and an underlying application object.

The *Conic* [Kramer, et al. 1989] environment provides support for configuration programming for distributed and concurrent programs. Configuration programming stands for a way of constructing software out of software components together with their control and communication interconnections. The Conic development environment supports the description of configurations (out of existing software components), system monitoring (i.e., the monitoring of the execution of a configuration), and change management (i.e., the evolution of a system).

We have chosen as the three main criteria for characterizing and comparing the scripting systems syntactic issues, semantic issues, and environment (target) issues. The criteria reflect the interface features, the functionality, and the required / proposed environment of the different systems. The criteria can be further divided as follows:

- *Syntactic Issues:*
  - representation (Text / Visual)
  - concepts of the scripting language (e.g., components, ports, links, local constraints, compositions)
- *Semantic Issues:*
  - interpretation of scripting concepts and scripts
  - execution mode (compiled / interpreted)
  - purpose (application oriented / general-purpose)
- *Environment (Target) Issues:*
  - target language model
  - front-end to other tool/language or self-contained
  - openness

Table 1 presents an overall comparison of the different scripting systems.

6 Conclusion

The notion of scripting as an aid for object-oriented application development has been introduced. A script is a configuration of software components with unbound parameters, which are bound together via special scripting operators. In this paper concepts of an object-based scripting model, which we are going to use in the prototype implementation of the Visual Scripting Tool, have been discussed. We have described the use of the VST as a generic development environment for building scripts.

The crucial test of the VST is its acceptance by real users. There is no way that the correct functionality and the proper user interface can be totally anticipated. We plan [Pröfrock, et al. 1989] to bring the features and the first prototype VST to the attention of users already working in several areas of application development. Their immediate feedback will help us to eliminate unnecessary features, to enhance certain capabilities, and to tailor the user interface to their needs.

References

[Cox, 1986] B.J. Cox, *Object Oriented Programming - An Evolutionary Approach*, Addison-Wesley, Reading, Mass., 1986.

[Dami, et al. 1988] L. Dami, E. Fiume, O. Nierstrasz and D. Tsichritzis, "Temporal Scripts for Objects", in *Active Object Environments*, ed. D.C. Tsichritzis, pp. 144-161, Centre Universitaire d'Informatique, University of Geneva, June 1988.

[Dami, 1988] L. Dami, "Musical Scripts", in *Active Object Environments*, ed. D.C. Tsichritzis, pp. 162-171, Centre Universitaire d'Informatique, University of Geneva, June 1988.

[Dami, 1989] L. Dami, "Reusability through Horizontal Composition", in *this volume*, 1989.

[Florijn, 1989] G.H. Florijn, "Visual Shell Tool - Program Documentation", SERC (Software Engineering Research Centrum), Utrecht, The Netherlands, 1989.

[Gibbs, et al. 1989] S. Gibbs, V. Prevelakis and D. Tsichritzis, "Software Information Systems: A Software Community Perspective", in *this volume*, 1989.

[Ingalls, 1988] D. Ingalls, "Fabrik: A Visual Programming Environment", Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices, vol. 23, no. 11, pp. 176-190, Nov. 1988.

[Kernighan and Pike, 1984] B.W. Kernighan and R. Pike, *The UNIX Programming Environment*, Prentice-Hall Software Series, 1984.

[Kleyn and Gingrich, 1988] M.F. Kleyn and P.C. Gingrich, "GraphTrace - Understanding object-oriented systems using concurrently animated views", Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices, vol. 23, no. 11, pp. 191-205, Nov. 1988.

[Kramer, et al. 1989] J. Kramer, J. Magee and K. Ng, "Graphical Support for Configuration Programming", Hawaii International Conference on System Sciences, pp. 860-870, Jan. 1989.

[Krasner and Pope, 1988] G.E. Krasner and S.T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80", Journal of Object-Oriented Programming, vol. 1, no. 3, pp. 26-49, Aug. 1988.

[Meyer, 1988] B. Meyer, *Object-oriented Software Construction*, Prentice Hall, 1988.

[PPI, 1988] PPI, *ICpak 201 Reference Manual*, 1988.

[Pröfrock, et al. 1989] A.-K. Pröfrock, D. Tsichritzis, G. Müller and M. Ader, "ITHACA: An Integrated Toolkit for Highly Advanced Computer Applications", in *this volume*, 1989.

[Shafer, 1988] D. Shafer, *HyperTalk Programming*, Hayden Books, 1988.

[Thompson, 1989] T. Thompson, "The NEXT Step", Byte, vol. 14, no. 3, pp. 265-2071, March 1989.

[Wegner, 1987] P. Wegner, "Dimensions of Object-Based Language Design", ACM SIGPLAN Notices, Proceedings OOPSLA '87, vol. 22, no. 12, pp. 168-182, Dec 1987.

[Weinand and Gamma, 1988] A. Weinand and E. Gamma, "ET++ - An object-oriented application framework in C++", Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices, vol. 23, no. 11, pp. 46-57, Nov. 1988.