Unix Tools as Visual Programming Components in a GUI-builder Environment[*][†]

Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
Patision 76, GR-104 34 Athens, Greece
email: dds@aueb.gr

September, 2001

Abstract

Development environments based on ActiveX controls and JavaBeans are marketed as "visual programming" platforms; in practice their visual dimension is limited to the design and implementation of an application's graphical user interface (GUI. The availability of sophisticated GUI development environments and visual component development frameworks is now providing viable platforms for implementing visual programming within general-purpose platforms, i.e. for the specification of non-GUI program functionality using visual representations. We describe how specially-designed reflective components can be used in an industry-standard visual programming environment to graphically specify sophisticated data transformation pipelines that interact with GUI elements. The components are based on Unix-style filters repackaged as ActiveX controls. Their visual layout on the development environment canvas is used to specify the connection topology of the resultant pipeline. The process of converting filter-style programs as visual controls is automated using a domain-specific language. We demonstrate the approach through the design and the visual implementation of a GUI-based spelling checker.

Keywords

Visual programming, components, reflection, Unix tools, pipe and filter architecture, reuse.

Introduction

A number of environments support the visual composition of graphical user interfaces (GUIs) using components with a predefined set of interfaces. In addition, technologies such as ActiveX and JavaBeans allow the development of visual components (typically GUI elements) that can be seamlessly incorporated into an integrated development environment (IDE) and subsequently used in application development. In this article we present how visual IDEs and components can be extended beyond GUI development to support visual programming for a particular domain.

A visual programming language can be informally defined as a programming language with a syntax that includes visual expressions such as diagrams, free-hand sketches, icons, or graphical manipulations [1]. Visual programming approaches aim towards easing the programming learning curve or enhancing programming productivity. Their adoption was based on a number of premises [2]:

- pictures may be able to concisely convey meaning more efficiently than words,
- pictures could help understanding and remembering,
- pictures may enhance the experience of programming by making it more interesting, and
- culturally-neutral pictures can be understandable regardless of what language people speak.

However, empirical studies did not find visual programming inherently superior to text-based programs; the extent to which a given notation is suitable for expressing a particular task depends on the context in which the language is employed [3, 4, 5]. Early work on visual programming, although promising when applied to "toy" problems, ran into difficulties when the methods were tried on problems of realistic size. Two approaches for alleviating this problem were followed: a number of researchers applied visual programming languages to limited or domain-specific parts of software development such as GUI programming, the

[*] *Software—Practice and Experiece*, 32(1):57–71, 2002.

[†] This is a machine-readable rendering of a working paper draft that led to a publication. The publication should always be cited in preference to this draft using the reference in the previous footnote.

graphic depiction of data structure behaviour, or the combination of textually programmed units to build new programs [1]. Others, proposed the design of visual programming languages based on formalisms used by existing, standard, component-based visual engineering languages such as the ITU-T MSC standard for message sequence charts, or the IEC-1131 standard for function block languages [6].

In our approach we capitalise on the strength of existing GUI-builder IDEs by crafting industry-standard software components that can be used within the GUI-designer of the IDE to perform data-flow-oriented visual programming. Although we demonstrate our method in the context of a specific domain (the visual composition of data-flow pipelines), the same underlying principles can be applied to different visual programming domains.

We define a software component as a unit of composition with contractually specified interfaces and explicit context dependencies only; one that can be deployed independently and is subject to third-party composition [7]. Components, in common with objects, encapsulate state, allow access to it through separately described interfaces, and support modular design based on separation of concerns. However, components differ from objects in a number of ways: they can be implemented in different languages, they are often packaged in binary containers, they can encapsulate multiple objects, and are typically more robustly packaged than objects [8]. Components that support visual composition implement a set of interfaces defined by the visual programming environment that supports their use. These interfaces allow the visual placement of components on forms, the handling of user-input events, and the persistent specification of their properties at program design time. Two widespread families of visual components are JavaBeans [9] and ActiveX controls [10].

Visual components are supported by a number of IDEs. Typical examples are Visual Basic (VB), Delphi, JBuilder, Latte, and Visual Café. Although ActiveX controls, JavaBeans, and their respective environments are marketed as "visual programming" technologies, in practice, their visual dimension is limited to the design and implementation of graphical user interfaces (GUIs). A number of systems such as Khoros/Cantata [11, 12] and LabVIEW [13] support component-based visual programming — also referred to as coarse-grained visual programming — using specialised components and a corresponding dedicated development environment. Now, the availability of sophisticated GUI-builder IDEs and the respective component development frameworks provides us with an alternative approach for implementing coarse-grained visual programming: one based on widely deployed, industry-standard platforms.

In the following sections we present how specially-designed reflective components can be used in an industry-standard visual programming environment to visually specify sophisticated data transformation pipelines that interact with GUI elements. The remainder of this paper is structured as follows: Section discusses our selection of components and their execution environment; Section presents how the controls we have implemented support visual programming; in Section we analyse the design and implementation of the visual controls, in Section we present how the control creation process can be automated, in Section we describe an exemplar application utilising visual programming controls, and in Section we evaluate the approach we propose. Section concludes the paper with directions for further work.

The elements of our approach: visual programming, component-based development, a graphical front-end to Unix tools, data-flow visual languages, and pipe and filter architectures have been extensively studied. See for example the references [2, 14, 1, 15, 6] (discussing visual programming and providing examples of specific approaches), [16, 17, 18, 3] (discussing graphical Unix-tool front ends), [19, 20, 21, 7, 22] (discussing component-based development), [17, 23, 24, 25, 26, 27] (discussing visual data-flow approaches), and [28, 29] (discussing pipe and filter architectures). The main contributions of this paper are: the proposal to use standard GUI builders as visual programming environments by means of specially crafted reflective components, the idea that GUI builders can be used to configure sophisticated component interactions and deployment scenarios, and the demonstration of visual programming and GUI interfacing utilising the well-known Unix text processing tools encapsulated as ActiveX components.

Component Source and Execution Environment

Most existing visual components (also known as controls or widgets) provide GUI interface elements. Typical examples include text-input boxes, buttons, list boxes, grids, radio buttons, graphs, and file selection dialogs. The few visual components that do provide computationally interesting processing (such as Microsoft's Internet Transfer control used to perform HTTP/FTP data transfers) do not utilise their visual dimension; they are presumably packaged as visual components in order to expose their properties for editing using the standard IDE property editing mechanisms. Our candidates for repackaging as visual components had to provide non-trivial computational functionality, be amenable to visual manipulation, and be available for reuse. We decided to base our work on the numerous user and system programs available under the Unix operating system implementations. Based on the Unix tool-centred philosophy, software developers have created a large collection of programs that provide a single service (e.g. compare two files, search for a pattern, deliver mail) without requiring user interaction. Many of those programs are implemented using state-of-the-art algorithms, have been stress-tested in many diverse applications for decades, and have their interface and operation standardised under efforts such as POSIX [30]. Unix filter-style tools can be combined using pipelines; an abstraction with obvious visual connotations. In addition, many of these programs are freely available in source code form through Open Source initiatives such as GNU and BSD. The initial selection of the visual programming controls builds upon our previous work on *component mining* [31, 32] where we defined a pattern language for converting such programs into (non visual) components.

A visual component (an ActiveX control or a JavaBean) is characterised by properties, events, and methods. *Properties* affect the appearance (e.g. the background colour), functionality, or state of the control or bean. *Events* are notifications sent by the controls or beans to let programs using them know that a specific action (such as a mouse click) has occurred. *Methods* are other functions that can be called from within the program that hosts the control or bean to perform some processing. By packaging Unix tools as visual components we gain a number of benefits:

- the relative placement of the tools on the environment's graphical canvas can be used to define data processing pipelines,
- the two-dimensional nature of the canvas allows the implementation of more sophisticated pipelines than the ones that can be specified using the one-dimensional Unix shell command line,
- the integration of the tools in a graphical environment allows the implementation of GUI applications where the tools can interact with a variety of standard GUI controls (widgets), and
- the Unix letter-style command-line options can be exposed as properties that can be graphically specified using the IDE's property box.

We decided to package the Unix tools as ActiveX controls, based on our previous experience with packaging them as COM (Component Object Model) objects and the availability of framework support for efficient control implementation. ActiveX controls can be used in a number of environments on Microsoft-Intel platforms such as the VB IDE, and the Internet Explorer. Although a JavaBeans-based implementation could have resulted in highly portable components, this did not apply in our case since most Unix tools are implemented in C and will not benefit from the Java binary-level portability. Our approach however, is not Microsoft-specific; the ideas we present can be employed in any environment that supports visual components with basic reflective capabilities.

Visual Programming

The visual Unix-filter components (VUFC) we implemented support the graphical specification of data-flow pipelines. The relative position of the components on the IDE design canvas determines the flow of data between them. Components, representing processes, can have input ports, which are used to read data, and output ports where data is output. The two-dimensional nature of the design canvas allows us to build sophisticated non-linear filter topologies. The default flow of data across components follows the — standard in western cultures — top-to-bottom and left-to-right direction. Ports are numbered clockwise from the bottom left edge of the control; this numbering scheme and the default port values used (0 for input, 1 for output) result in the data flow direction we described. When programmers want to override the default direction, control properties allow them to specify the port number to be used for each input or output function (e.g. assign the input to port 3). We found the port numbering scheme intuitive and convenient; however, port numbers could also have been specified in a graphical way by implementing a custom property editor (a facility available both for ActiveX controls and JavaBeans). Controls are freely drawn in arbitrary sizes and positions using the graphical design facilities of the IDE GUI editor; controls that are very close together are considered connected. Apart from setting non-default directions for data flow, the programmer needs only to specify filter-specific options (e.g. set a filter to sort in descending order). These are typically provided as component properties and can therefore be easily set, again without any code writing, from within the IDE property editor. A co-ordinating component is placed on the canvas to synchronise the actions of all visual components it contains. This component can be activated at design time to visualise the current connection status of all visual components using input and output arrows. In addition, the component provides a runtime method for starting the operation of the pipeline, activating all its members and waiting until processing is finished.

Our approach is based on three types of visual components:

Filter components are encapsulating a standard Unix filter-style process such as *sort*, *cat*, or *wc*. In addition, a generic filter component allows the realisation of arbitrary filter commands; however, this filter's operation is not determined by specific properties (e.g. *SortOrder*), but by a single property containing the customary Unix-style command-line options.

Connector components connect two components from within our family; they are implemented as a visual encapsulation of the operating system pipe abstraction.

Glue components connect our visual components with other GUI elements such as *edit boxes* and *list boxes*.

The way VUFC components are visualised on the IDE design canvas is shown in Figure 1. The T (pipe split) component and the controls on its left move data in the right-to-left direction; they are the only ones where the port assignment properties had to be manually specified.

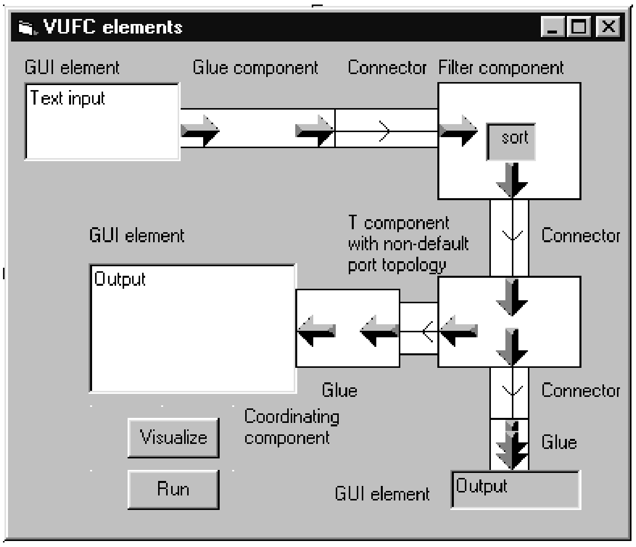


Figure 1: VUFC components and their connections.

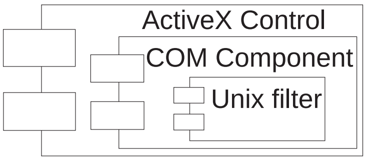


Figure 2: Visual component encapsulations.

Component Encapsulation and Operation

We created the visual components in a two-step process. We first mined and packaged Unix tools, connectors, and glue as (non visual) COM components. We then packaged the COM components as ActiveX controls. The architecture of the resulting components is illustrated in Figure 2.

Implementing COM components from scratch in C++ is not trivial. Every component, in addition to its custom functionality, must support registration, an interface for creating component instances called *IClassFactory*, object creation, reference counting, the *QueryInterface* method, and, possibly, dual interfaces for supporting its use through C++ and automation-based scripting languages. Fortunately, these tasks are supported by the *Microsoft Foundation Classes* (MFC), a large, monolithic application framework for programming in Microsoft Windows, and by the *Active Template Library* (ATL) a leaner set of template-based classes that specifically target the development of COM components. We implemented the mined components using ATL. By aggressively utilising C++ templates and multiple inheritance, ATL supports the development of COM components with brevity and minimal runtime overhead. A bare-bones ATL-based COM component can be implemented in less than 100 lines; most of them automatically generated by a “wizard”-type tool. We therefore found ATL to be ideal for implementing the large number of Unix-mined components and used it as a basis to automate the task. The full details of this operation are described in reference [32].

The second step of the visual component implementation involved the addition of visual component functionality to extend COM objects to fully fledged ActiveX controls. We initially attempted to provide this functionality by extending the ATL-based implementation of each component, but were quickly overwhelmed by the complexity of the task. We found that the addition of a single property to a control required non-trivial code and data additions to a number of C++, IDL (interface definition language), and header files. More importantly, more than 100 lines of inscrutable and unmaintainable code were needed to provide the, critical for our needs, functionality of neighbour control enumeration. We then experimented with the ActiveX control creation facilities of the VB IDE and found the operation reasonably streamlined and programmer-friendly. We retained the ATL-based components as a basis for our work, because their implementation was based on a number of system-level facilities (such as threads, pipes, non-blocking I/O, and handle cloning) that are not available in the VB environment. Our visual component architecture therefore involves wrapping each COM component within an ActiveX visual control, and, one additional co-ordinating component implemented from scratch in VB.

The stability of our pre-packaged COM components, in conjunction with the quick edit-run cycle, the easy access to component properties, and the high-level facilities for graphics programming provided by VB allowed us to implement the full functionality we required in a fraction of the time of the previous ATL-based aborted attempt.

The ability of the components to connect themselves together, depending on their location on the IDE design canvas, is based on their reflective capabilities i.e. their ability to examine their and their neighbours location.

The foundation for the concept of reflection is Smith’s reflection hypothesis [33], which states that reflective programs can access, reason about, or alter their interpretation. In our case, ActiveX controls have properties, available both at design and at run time, that can be used to access their location and enumerate the other controls that exist on the same canvas. We exploit this reflective capability to link the visual representation of the controls with their operational behaviour.

All visual components support a number of common properties and methods and some component-specific properties as shown in Figure 3. The component-specific properties are used to provide details on how the control will perform its processing (e.g. *MergeOnly, FoldCase, Reverse*) and (in exceptional cases) to specify the visual flow of data (*InPort, OutPort*). The common properties (*Height, Left, Width, Right, Enabled, Name, Visible*) are provided and implemented by the VB IDE and are needed to support the visual dimension and programmability of the controls. These properties include the location, dimension, and visibility of

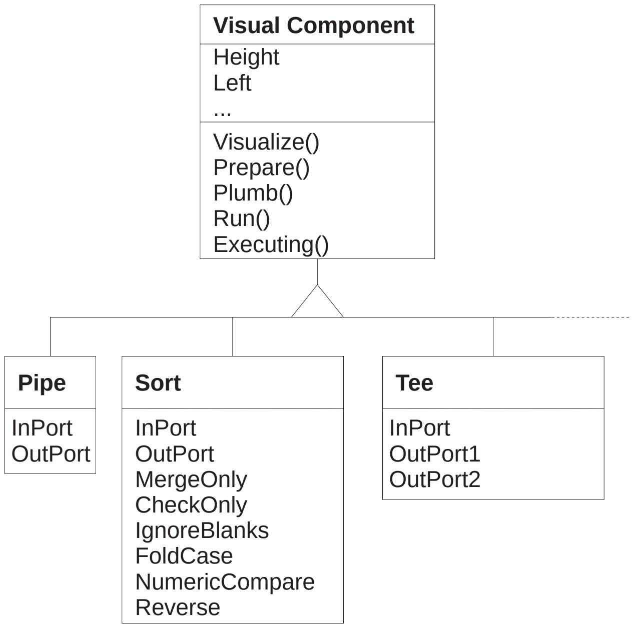


Figure 3: Visual component class diagram.

each control. The common methods (*Visualize(), Prepare(), Plumb(), Run(), Execution()*) form an interface that all visual components in our framework must provide. They are used by the co-ordinating control to orchestrate the visualisation and interconnection of the visual control setup as a network of co-operating processes. The methods of this interface work as follows:

Visualize is called during design or run time to provide the programmer with a view of the data-flow across the system. Solid arrows are used to indicate the flow of data at the input and output ports of the glue and filter components, while a single line arrow is drawn from the input to the output in all connector (pipe) components. Each component is responsible for drawing its own visual representation. It does this by enumerating all its connected neighbours, sorting them according to our defined port ordering, and assigning them to the respective ports specified by the programmer.

Prepare is the first operation performed on all controls before the configured system starts running. Controls are responsible for setting-up their internal state so that other controls can connect to them. As an example, the pipe control creates at this point the two pipe handles. The provision of a separate step used before the execution commences obviates the need of connecting the components based on their topological ordering and allows more flexible pipeline topologies.

Plumb is the second operation performed on all controls before the system runs. At this step each control connects to its neighbours, setting for example its standard output to the handle of the pipe it connects to.

```
command "sort"
options {
  MergeOnly:bool:-m:False:Merge already sorted files, do not sort
  CheckOnly:bool:-c:False:Check if given files already sorted, do not sort
  Month:bool:-M:False:Compare (unknown) < ‘JAN’ < ... < ‘DEC’, imply IgnoreBlanks
  IgnoreBlanks:bool:-b:False:Ignore leading blanks in sort fields or keys
  TmpDirectory:string:-T:"":Use specified directory for temporary files, not $TMPDIR or /tmp
  FoldCase:bool:-f:False:Fold lower case to upper case characters in keys
  Alphanumeric:bool:-d:False:Consider only [a-zA-Z0-9 ] characters in keys
  ASCII:bool:-i:False:Consider only [\040-\0176] characters in keys
  NumericCompare:bool:-n:False:Compare according to string numerical value, imply IgnoreBlanks
  OutputFile:string:-o:"":Write result on file instead of standard output
  Reverse:bool:-r:False:Reverse the result of comparisons
  StableSort:bool:-s:False:Stabilize sort by disabling last resort comparison
  FieldSeparator:string:-t:"":Use separator instead of non- to whitespace transition
  Unique:bool:-u:False:Only output the first of an equal sequence (with CheckOnly check for strict
}
```

Figure 4: Declarative description of the *sort* command.

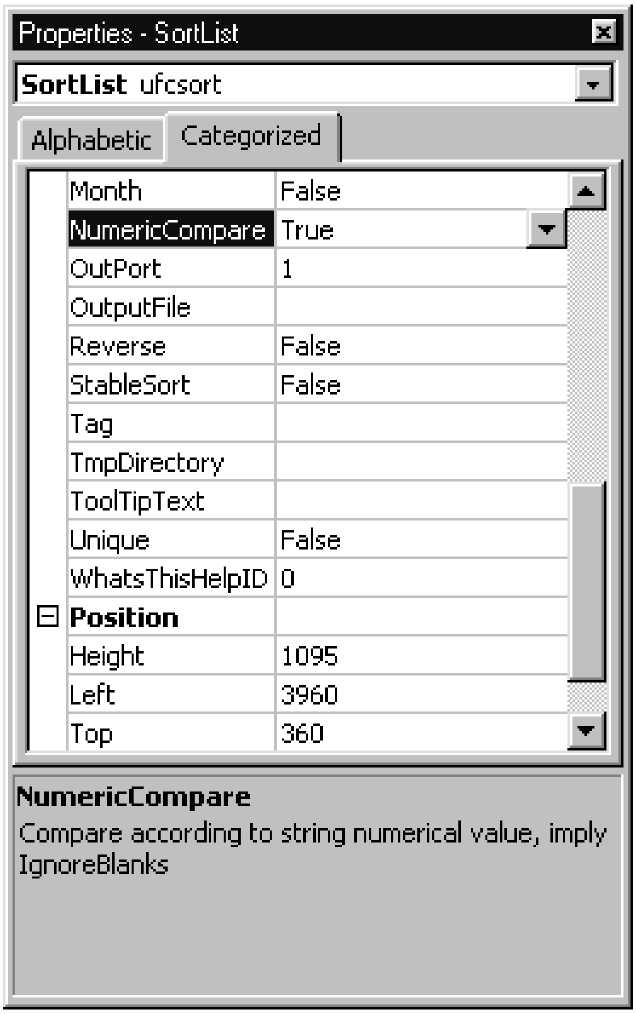
Run is a method that makes components begin processing their data. It is a do-nothing operation for passive components implemented as operating system abstractions (connectors, implemented as pipes); active components (filters and glue) at this step start-up a separate process or thread. This step is the last operation for bringing up a running pipeline.

Executing is a method that returns *true* if a control is internally processing data. In order to avoid deadlocks filter and glue controls process their data asynchronously as independent threads or processes. The co-ordinating control waits for the completion of the pipeline operation by monitoring the *executing* status of all controls on the canvas.

Automating Control Creation

Although the plethora of available filter-style tools provided us with a rich selection of candidates for encapsulation as visual controls, it made us realise that the act of encapsulation is a labour intensive and time consuming task. Following an earlier experience in automating packaging of Unix tools as COM objects [32] we defined a process and implemented support tools for automating the creation of filter-style VUFC controls. Specifically, for every filter that is to be converted into a control, one has to define the syntax and semantics of the tool's command-line options using a small domain-specific language [34]. An example of this description for the *sort* filter is depicted in Figure 4. For every filter command line option (e.g. -r) one specifies:

- a meaningful name that is to be used as the respective control property,
- the option's type (*bool* for stand-alone options; *string*, *int*, or *double* for options that take arguments of the respective type),
- the option's default value,
- a descriptive text that appears as help for the given property in component object browsers and the IDE property editor, and



Properties - SortList

SortList ufcsort

Alphabetic | Categorized

Month	False
NumericCompare	True
OutPort	1
OutputFile	
Reverse	False
StableSort	False
Tag	
TmpDirectory	
ToolTipText	
Unique	False
WhatsThisHelpID	0
⊟ **Position**	
Height	1095
Left	3960
Top	360

NumericCompare
Compare according to string numerical value, imply IgnoreBlanks

Figure 5: Presentation of the *sort* properties in the VB IDE.

- the respective letter code expected by the filter as a command-line argument.

A small compiler, implemented in Perl [35], compiles the declarative description of the filter interface into a VB control definition source file that implements the respective component (e.g. *VUFCsort*). The code contains:

- a member variable for every filter command-line option,
- methods for getting and setting the member variable value (thus exposing the command-line option as a "property" of the component),
- methods for loading and saving the property values,
- a method for initialising the properties to a known state, and
- a method for executing the filter with a command line constructed dynamically to match the values of the component's properties.

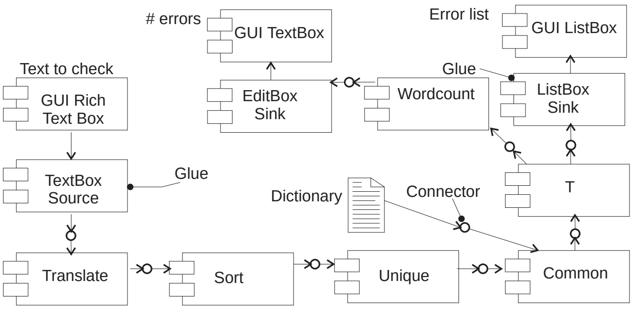


Figure 6: A spell checker with a GUI.

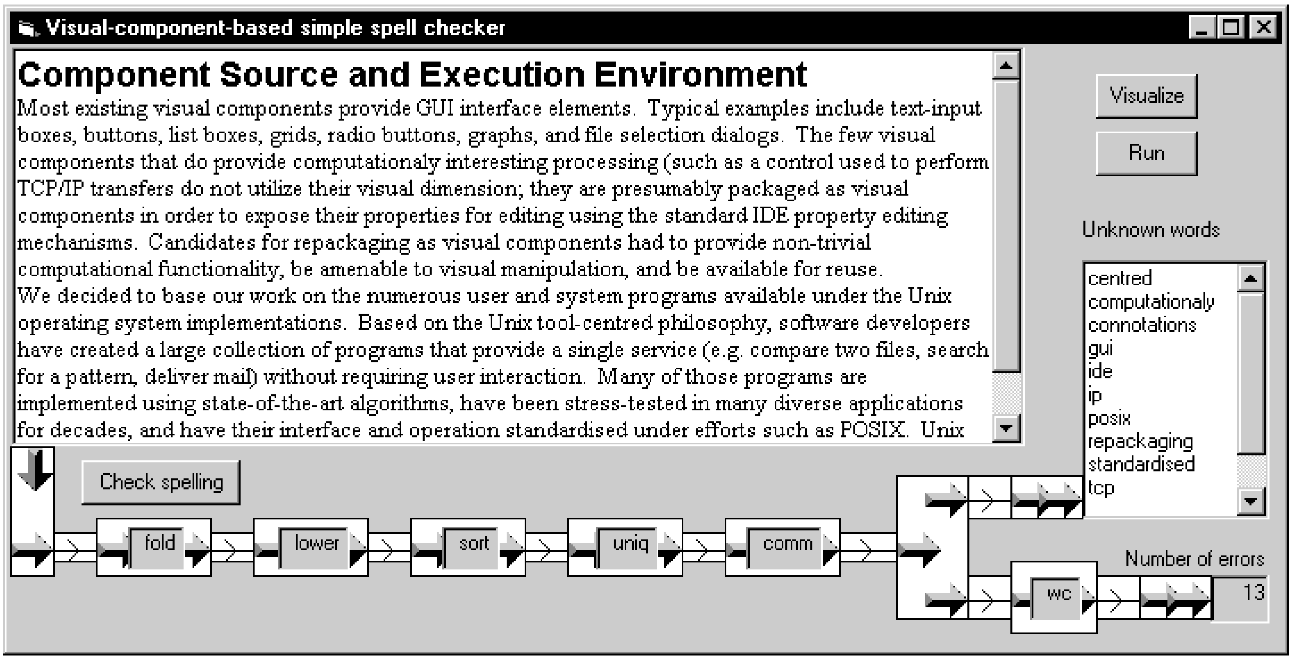
The component also inherits (due to VB limitations, by source code reuse) and exposes as properties the common methods and properties of the standard VUFC interface. An example of how the properties of the automatically created *VUFCsort* component appear in the VB IDE can be seen in Figure 5. Connector and glue-type components still need to be written by hand, but the effort required to implement them is only a small part of the effort that would be required to repackage the large number of filter-style programs without an automated process.

Exemplar Application

To demonstrate the viability of our approach we have designed and implemented a simple GUI-based spell-checker based on a pipeline of ActiveX controls. Figure 6 depicts the UML component interaction diagram of the spell-checker. The text to be spell-checked is retrieved from the GUI edit box using a *text box* source glue component. It is transformed into a list of words using the *translate* component that is a direct equivalent of the Unix *tr* command. The word list is then transformed into a sorted list of unique words using the *sort* and *unique* components that correspond to the Unix *sort* and *uniq* commands. Finally, the sorted stream of words to be spell-checked and a dictionary of all acceptable words are processed by *common* — derived from the Unix *comm* command — that outputs the errors: a list of words contained in the first stream and not contained in the second one. This stream of misspelled words is cloned into two streams using *T* the equivalent of the Unix *tee* command. One stream is sent, using the *list box sink* glue component, to a GUI list box. The other is passed to the *wordcount* component to count the number of words contained in it and then, via an *edit box sink*, to a GUI text box. It is important to note that the integration of GUI elements as parts of the pipeline and the cloning of a data stream can not be implemented using the standard Unix linear pipeline system.

We implemented the GUI-based spelling checker using VUFC following the design we outlined. The implementation consisted of:

- drawing the controls to form the spell check pipeline,



Component Source and Execution Environment

Most existing visual components provide GUI interface elements. Typical examples include text-input boxes, buttons, list boxes, grids, radio buttons, graphs, and file selection dialogs. The few visual components that do provide computationaly interesting processing (such as a control used to perform TCP/IP transfers do not utilize their visual dimension; they are presumably packaged as visual components in order to expose their properties for editing using the standard IDE property editing mechanisms. Candidates for repackaging as visual components had to provide non-trivial computational functionality, be amenable to visual manipulation, and be available for reuse.

We decided to base our work on the numerous user and system programs available under the Unix operating system implementations. Based on the Unix tool-centred philosophy, software developers have created a large collection of programs that provide a single service (e.g. compare two files, search for a pattern, deliver mail) without requiring user interaction. Many of those programs are implemented using state-of-the-art algorithms, have been stress-tested in many diverse applications for decades, and have their interface and operation standardised under efforts such as POSIX. Unix

Unknown words

centred
computationaly
connotations
gui
ide
ip
posix
repackaging
standardised
tcp

Number of errors 13

Figure 7: A GUI-based spelling checker built using VUFC.

- specifying (using the property editor of the IDE) the operational parameters of each filter component (e.g. that the *wordcount* filter should count lines),
- adding two lines of code: one for connecting the "Check spelling" button to the co-ordinating control, and one to specify an option to the *fold* filter that contained an embedded newline and could not be specified from the property editor.

The implementation and resultant user-interface are depicted in Figure 7. The visual components and their interconnections need not be visible to the end-user; they are included in the figure to demonstrate the pronounced similarity between the design of the spelling checker and its visual implementation.

Compared to a spelling checker implemented using a linear pipeline in the Unix environment, our component-based implementation offers the following enhancements:

- it provides a graphical user-interface,
- it counts the number of errors detected utilising VUFC's ability to implement non-linear pipeline topologies, and
- it can check formatted text.

In addition, the application was implemented using a typed and modular language in a rich integrated development environment offering a purely graphical method for interconnecting the components, a syntax-aware editor, a sophisticated debug facility, a graphical interface builder, integrated help facilities, and source-code management. Third-party tools also provide support for profiling, automated source code examination, and browsing facilities. This level of support is not existent in Unix-based shell-programming approaches and very difficult to obtain in visual programming environments implemented from scratch.

Approach Evaluation

The use of a GUI-builder IDE for visual programming compared with the use of a dedicated visual programming editor is a choice that depends on the balance of specific advantages, disadvantages, and limitations of each approach.

Modern IDEs provide a mature development environment that enhances programmer productivity in a number of ways. As an example, Visual Basic provides a sophisticated graphical editor, a rich mechanism for specifying component properties (categorised by function, alphabetically, or grouped on a form; all with two levels of help material), a syntax-aware program editor (with keyword colouring, real-time function argument prompting and checking, and multiple levels of undo), a debugger, and an integrated object browser. In addition, programmers (including those relying on visual programming components) profit from the facilities provided for interacting with the environment. Specifically, environments based on Java 2 or Microsoft's .net architecture offer classes supporting most common GUI elements, database connectivity, major networking protocols, email and HTTP transactions, multimedia data, localisation and internationalisation, object serialisation, directory services, and distributed computations.

Furthermore, major existing IDEs benefit from a large installed user base that in turn results in programmer familiarity with the environment, wide choice and availability of supplementary documentation, professional magazines, conference and exhibitions, consulting services, and training courses. The large user base has also resulted in the blossoming of an add-on industry providing components and tools. An illustrative case is Component Source — a major commercial vendor of ActiveX components — that provides over 2300 components in its January 2001 CD. In addition many design, source code control, and testing tools can be seamlessly integrated into mainstream IDEs.

Finally, visual components targeting industry-standard IDEs can be crafted using existing rich component frameworks and technologies. In our case we were able to use Unix components already encapsulated in a COM framework and experiment with two different technologies for creating ActiveX controls, one based on the C++ Active Template Library (ATL), and one based on the ability of Visual Basic to create ActiveX controls.

However, the approach we propose is not without problems. The most important potential problem comes from the clash with the underlying programming model supported by the IDE (e.g. Visual Basic in our case) that can make it difficult to express the semantics of the combined system or reason about the resulting artefact. In the specific application we described, the data-flow paradigm of the visual language is orthogonal to the native imperative / object-oriented paradigm supported by Visual Basic. Had we developed visual programming components for expressing imperative constructs such as assignments and loops the interactions between the two could potentially become formidable sources of confusion.

In addition, the editor used for the GUI-design is not optimised for visual programming. As an example, components with common semantic (as opposed to topological) properties can not be selected and manipulated as a group while maintaining their visual connections, nor does the editor provide a way to draw arbitrary lines between components. At a deeper level, as the IDE is not designed for visual programming notions such as modularity, encapsulation, and abstraction are not inherently available in a visual form. They can sometimes be accommodated by utilising GUI elements such as forms and frames, forcing however an unnatural expression style. Similarly, the connection between visual elements located on different forms can not be expressed in a visual way, but has to rely on textual representations such as object name bindings.

Finally, the specific implementation we have demonstrated (the visual design of Unix-style pipelines) has some unique restrictions, that are not however limitations of the approach we propose. The data passed between the components consists of strings, while other visual programming systems like Khoros/Cantata, AVS/Express, and LabVIEW allow for complex structures to be passed between components. Although in theory pipelines can be used to pass binary data and complex structures, Unix systems customarily pass records delimited by newlines and delimit fields by whitespace or another special character. This practice is not overly restrictive: it has served admirably many data processing tasks [36]; furthermore, more complex data structures can always be expressed as character strings using XML. One additional restriction of our implementation might appear to be the absence of feedback loops, again a feature of the visual programming systems mentioned above. While the topologies supported by our implementation can express loops, most Unix filters only accept a single input source — a design choice influenced by the more restrictive linear topology of the pipelines that can be expressed in the common Unix shells — and can not therefore be easily connected into a loop structure.

Conclusions

We have demonstrated that standard GUI component technologies and environments can be used to support a visual programming approach. We feel that this is a significant result, because the relative slow-moving acceptance of visual programming techniques can be partly attributed to the quality and functionality of the environments that support them. As many visual programming environments are research-oriented, proof-of-concept projects, they can not easily compete with the commercially-developed, polished, and supported commercial IDEs. In contrast, visual programming based on the reuse of proven existing components in a widely adopted IDE levels the playing field and allows research to focus on program representations and methodologies that can increase productivity rather than supporting infrastructure.

In the form implemented VUFC can be used for production work, but can also be extended and improved in a number of ways. Currently the user is responsible for the correct layout of the visual components. An interesting enhancement would be the provision of a verification step after the pipeline plumbing phase to detect obvious errors in port connections. This verification could be based on a type system for component ports [37, 38] preferably providing edit-time feedback using a static type inference system [39]. A related improvement concerns the topologies that can be implemented. Some existing filters accept only files for input and output and not arbitrary data streams. We are experimenting with the use of named pipes to equip such filters with connectable streams.

The most significant future work will however involve the exploration and extension of the visual part of the functionality. As an example, the operation of the pipeline at runtime can be visualised by colouring the components to indicate their level of activity. Such a facility can be used for debugging or diagnostic purposes. In addition, more sophisticated connector components can be designed to provide greater expressive freedom to the visual designer. Passive connectors could be used to propagate port assignments across non-rectangular regions, while split connectors could be used for functionally dividing a complex design into multiple forms in a manner analogous to the current practice in multi-sheet electronic circuit diagrams. Finally, the application domain of our approach can be extended in a number of directions; pipelines are not the only programming artefact that can be efficiently expressed in a visual way. Component deployment in distributed applications is one obvious target; others surely exist and wait to be explored.

Acknowledgments

The author wishes to thank the anonymous reviewers for their detailed and insighfull comments on earlier versions of this paper. Part of the research described here was performed on board of the *HS Limnos*; the tolerance and support of the ship's officers and crew is gratefully acknowledged.

References

[1] Margaret M. Burnett and David W. McIntyre. Visual programming. *IEEE Computer*, 28(3):10–19, March 1995. Special issue on visual programming.

[2] N. C. Shu. Visual programming: Perspectives and approaches. *IBM Systems Journal*, 28(4):525–547, 1989. Reprinted in IBM Systems Journal 38(2&3):199-221 1999.

[3] Francesmary Modugno, Albert T. Corbett, and Brad A. Myers. Graphical representation of programs in a demonstrational visual shell—an empirical evaluation. *ACM Transactions on Computer-Human Interaction*, 4(3):276–308, 1997.

[4] M. Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.

[5] K. N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, 5(1):109–142, 1997.

[6] M. Münch and A. Schürr. Leaving the visual language ghetto. In *1999 IEEE Symposium on Visual Languages (VL '99)*, Tokyo, Japan, September 1999. IEEE Computer Society.

[7] Clemens Szyperski. *Component Software: Behind Object-Oriented Programming*. Addison-Wesley, 1998.

[8] Alan Cameron Wills. Designing component kits and architectures. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures: Advances and Applications*. Springer Verlag, 1999.

[9] Patrick Chan and Rosanna Lee. *The Java Class Libraries : Java.Applet, Java.Awt, Java.Beans*, volume 2. Addison-Wesley, 1996.

[10] Stephen Jakab, Darren Gill, Alex Homer, and Dave Jewell. *Visual Basic 5.0 ActiveX Control Creation*. Microsoft Press, Redmond, Washington, USA, 1996.

[11] Mark Young, Danielle Argiro, and Jeremy Worley. An object oriented visual programming language toolkit. *Computer Graphics*, 29(2):25–28, 1995.

[12] Mark Young, Danielle Argiro, and Steven Kubica. Cantata: Visual programming environment for the khoros system. *Computer Graphics*, 29(2):22–24, 1995.

[13] Jörgen Jehander. Graphical object-oriented programming in LabVIEW. Online. http://www.ni.com/pdf/instrupd/appnotes/an143.pdf. 9 May 2001, October 1999. National Instruments Corporation. Application Note 143.

[14] D. Smith, A. Cypher, and J. Spohrer. Kidsim: Programming agents without a programming language. *Communications of the ACM*, 37(7):54–67, July 1994.

[15] Wayne Citrin. Strategic directions in visual languages research. *ACM Computing Surveys*, 28(4es):132–132, December 1996. Available online http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/a132-citrin/.

[16] T. R. Henry and S. E. Hudson. Squish: A graphical shell for Unix. *Graphics Interface*, pages 43–49, 1988.

[17] P. E. Haeberli. ConMan: A visual programming language for interactive graphics. *Computer Graphics*, 22(4):103–111, August 1988.

[18] K. Borg. Ishell: A visual Unix shell. In *Conference on Human Factors in Computing Systems (CHI '90)*, pages 201–207, 1990.

[19] Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, September 1992.

[20] Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, September/October 1998.

[21] Desmond Francis D' Souza and Alan Cameron Wills. *Objects, Components, and Frameworks With UML : The Catalysis Approach*. Addison-Wesley, 1998.

[22] Peter M. Maurer. Components: What if they gave a revolution and nobody came. *IEEE Software*, 33(6):28–34, June 2000.

[23] Susan Eisenbach, Lee McLoughlin, and Chris Sadler. Data-flow design as a visual programming language. In *Proceedings of the Fifth International Workshop on Software Specification and Design*, pages 281–283. ACM, 1989.

[24] D. D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing*, 3(1):69–101, March 1992.

[25] T. Smedley, P. Cox, and S. Byrne. Expanding the utility of spreadsheets through the integration of visual programming and user interface objects. In *Proceedings of Advanced Visual Interfaces '96*, pages 148–155, Gubbio, Italy, May 1997. ACM, ACM Press.

[26] M. Toyoda, B. Shizuki, S. Takahashi, S. Matsuoka, and E. Shibayama. Supporting design patterns in a visual parallel data-flow programming environment. In *1997 IEEE Symposium on Visual Languages (VL '97)*, Isle of Capri, Italy, September 1997. IEEE Computer Society.

[27] E. Ghittori, M. Mosconi, and M. Porta. Designing new programming constructs in a data flow VL. In *1998 IEEE Symposium on Visual Languages (VL '98)*, Nova Scotia, Canada, September 1998. IEEE Computer Society.

[28] Regine Meunier. The pipes and filters architecture. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, chapter 22, pages 427–440. Addison-Wesley, 1995.

[29] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[30] International Organization for Standardization, Geneva, Switzerland. *Information technology — Portable operating system interface (POSIX) — Part 2: Shell and Utilities*, 1993. ISO/IEC 9945-2:1993 (IEEE/ANSI Std 1003.2-1992 & IEEE/ANSI 1003.2a-1992).

[31] Diomidis Spinellis. Explore, excogitate, exploit: Component mining. *IEEE Computer*, 32(9):114–116, September 1999.

[32] Diomidis Spinellis and Konstantinos Raptis. Component mining: A process and its pattern language. *Information and Software Technology*, 42(9):609–617, June 2000.

[33] Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, January 1982.

[34] Diomidis Spinellis and V. Guruprasad. Lightweight languages as software engineering tools. In J. Christopher Ramming, editor, *USENIX Conference on Domain-Specific Languages*, pages 67–76, Santa Monica, CA, USA, October 1997. Usenix Association.

[35] Larry Wall, Tom Christiansen, Randal L. Schwartz, and Stephen Potter. *Programming Perl*. O'Reilly and Associates, Sebastopol, CA, USA, second edition, 1996.

[36] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, 1984.

[37] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. Fabrik: A visual programming environment. *ACM SIGPLAN Notices*, 23(11):176–190, November 1988. Proceedings of the 1988 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '88), San Diago, CA, USA, September 1988.

[38] Marc A. Najork and Eric J. Golin. Enhancing show-and-tell with a polymoprphic type system and higher-order functions. In *1990 IEEE Workshop on Visual Languages*, pages 215–220, Skokie, IL, USA, October 1990. IEEE Computer Society.

[39] Rebecca Djang, Margaret Burnett, and Roger Chen. Static type inference for a first-order declarative visual programming language with inheritance. *Journal of Visual Languages and Computing*, 11:191–235, April 2000.