Supporting Reuse of Evolving Visual Code

Rebecca A. Walpole and Margaret M. Burnett*
Department of Computer Science
Oregon State University, Corvallis, Oregon 97331
{walpolr, burnett}@cs.orst.edu

Abstract

Although the idea of reusing code is very appealing, effective reuse has long been acknowledged as a problem. To help address the difficulties, many advocate strong management commitment to code reuse, leading to the treatment of code as an asset to be carefully managed in a well-organized repository. However, the advent of the Web may bring a change to this outlook, encouraging informal, loosely-organized code repositories. Already, for both textual and visual languages, informal repositories are beginning to emerge, featuring a high rate of change and few controls over what a producer must do to submit code. In this paper, we present techniques to address some aspects of code reuse in this kind of informal, evolving environment. These techniques build upon characteristics found in many visual programming languages. Using these techniques, our approach is able to eliminate the special work traditionally required of the producer, while still supporting the consumer's reuse efforts within these immature, evolving repositories.

1. Introduction

It has long been clear that if programmers would reuse code more often, rather than reimplementing the same logic again and again, there would be great potential for software development cost savings, including not only the cost of creating the code, but also the costs of documentation, debugging, testing, and maintenance. Unfortunately, enabling programmers to reuse code effectively has not been easy. There are many reasons why this is true, including the lack of tools for finding and composing reusable code and programmers' perceptions that writing code is easier than locating it, determining what it does, and finding out if it works [26].

Even though the problems with reuse have been significant, there have still been some assumptions upon which potential solutions could be built. These assumptions have included the notion that a repository of code was owned by an entity that could devise standards and procedures for including code in the repository; the notion that code producers would be required to conform to these standards and procedures before their code would be allowed into the repository; and the notion of a structured, mature repository, organized, for example, into a hierarchy of categories. However, the recent advent of informal, evolving, shared repositories of code for collaborative development violates all of these assumptions. The use of the Web as a repository of shared code is a prime example.

The notion of informal, evolving repositories seems to be of rising interest for visual programming languages (VPLs). For example, one of Visual AgenTalk's goals is that "Environments should allow [end-user programmers] to share, with very little effort, programs and program fragments" [19]. For VPLs with this kind of goal, we have been working on an approach to facilitate code reuse under such an informal, evolving repository, and in this paper we present our early results.

We will confine our definition of reuse to the use of an existing code component in place of creating a new component. In general, a *component* is any artifact of the software process; in this paper, we concentrate on code components, but our approach is not limited to code; it also takes into account on-line documentation, test plans, specifications, and other artifacts. We use the term *repository* to mean a collection of components. To distinguish between reuse tasks, even if they are performed by the same person, we use the term *producer* to mean the programmer who is building reusable components, and the term *consumer* to mean the programmer who is interested in using these components. Finally, *packaging* tasks are the work traditionally required of producers to prepare a component for inclusion in a formally-controlled code repository, such as conforming to standards, preparing documentation, providing test suites, etc.

In this paper, we will assume an evolving repository that consists of components written using a VPL. We imagine several possible scenarios such as Web-based repositories of end-user-produced shareware, evolving repositories of code written by novices, or informal repositories for new software projects early in their development. In these cases, it is reasonable to assume a repository numbering in the hundreds of components.

*This work was supported in part by Hewlett-Packard and by the National Science Foundation under grant CCR-9308649 and an NSF Young Investigator Award.

Because of our interest in informal, evolving repositories, these are the only two assumptions we make. In particular, we emphasize that our approach is designed to work even in the absence of:

- A managing body to enforce standards and oversee the repository.
- A set repository structure such as a hierarchy of component categories or collections.
- Component producers who meticulously provide component packaging to aid the consumer.

While each of these provides valuable assistance to the consumer, informal, evolving repositories do not usually feature this level of support.

1.1 Concrete manifestations of four fundamental reuse problems

Biggerstaff and Richter [3] identify four fundamental reuse problems a successful reusability system must address:

1. Finding components
2. Understanding components
3. Modifying components
4. Composing components

Given no requirement for classification or packaging by the code producer and a constantly evolving repository, how can a consumer deal with these problems? To focus our investigation into this question, we have devised several concrete versions of each of these problems from the consumer's perspective, in the form of an initial assumption (such as "I am not familiar with this repository") and a goal (such as "what kinds of components does this repository contain?"). This paper shows how our approach helps the consumer with some of these goals.

Finding components

- I know precisely what I want; is it here?
- I know generally what I want; which components are relevant?
- I am not familiar with this repository; what kinds of components does it contain?
- There is a component that I've used before from this repository, but I forget the name and several other details of it. Where is it?
- This component is not quite what I need; which other components are similar?

Understanding components

For this set of questions, the initial assumption is that the consumer has somehow narrowed down the search to some potentially useful components for the task at hand.

- How do I choose from among these similar components?
- What does this component do (in the most general sense)?
- Does this component do what I need?
- How does this component do what it does?
- Do I need to modify this component?
- What will I save by using this component?

Modifying components

At this point, the consumer is considering one potential component, and is deciding whether and how to customize it.

- Can I change this component to suit my needs?
- How do I make the changes?

Composing components

The consumer is considering one or a group of potential components. These questions relate to how to fit them together with each other and with the application in which the consumer is trying to place them.

- How do I use this particular component?
- Are these components compatible?
- Which version of this component should I use?

Although we have listed a number of consumer-oriented reuse questions, our work is still in the early stages, and we have not yet found ways to help answer all of them. In this paper, we present the results we have obtained for eight of these questions.

2. Related work

We have been able to locate no previous literature on reuse in evolving repositories of VPL code, and only a little on other aspects of reuse in VPLs. The previous approaches that are reported differ from ours in both the nature of the repository and the type of reuse supported. For example, reuse is facilitated in HOLON/VP [16] by allowing generic code modules to be composed by sharing objects. The emphasis is on supporting the composition of objects within a given program, however, not a general code repository. Pree's work on design books [18] discusses reuse in the context of understanding and using frameworks. This approach has a producer-intensive focus on reuse for specific domains, and is not geared toward general support for arbitrary components.

A few VPLs provide some environmental support for a subset of the reuse questions we have identified. One example is the "what kinds of components does it contain?" question. Fabrik [14] provides overview support in a simple parts bin that allows drag-and-drop access to components filed under different categories. While this is an effective strategy for easily categorized components, it does rely on the component producer to provide meaningful categorizations. National Instruments™ LabVIEW® provides similar support by placing component icons in a hierarchical menu. The producer, to add a component, must define connectors (identifying the inputs and outputs) and save the component into a library (which determines its categorization). The producer is also encouraged to give the component a custom icon. Pictorius® Prograph CPX provides a hypertext class hierarchy browser that automatically includes new classes as they are created. The browser requires no extra producer work, but includes code from only the current project.

The Jacquard editor [10] addresses the question of finding relevant components. Each user action may narrow the search space by constraining the input or output of an empty socket (component placeholder). The user can then zoom in on an empty socket to examine the compatible components. If the number of possible components is still large, they are displayed hierarchically. To use this technique effectively, the consumer must be certain of the number and types of inputs and outputs for the needed component.

Vista [21] helps the consumer figure out how to use components by making "public processors" explicit. Public processors are the (sub)components that the producer anticipates the consumer will modify via substitution. Public processors therefore specify expected reuse of the component.

In the field of software reuse outside of VPLs, recent research has emphasized systematic reuse — incorporating reuse into the business of software development — with important responsibilities expected of the producers, consumers, and management [17, 13, 22]. While these roles are appropriate for software professionals, we wish to also enable reuse in situations where institutionalized support is not available, such as for informal Web-based repositories.

3. Answering reuse questions

Our strategy in answering reuse questions for evolving repositories of VPL code is to take advantage of the integration between the language and environment in ways that remove the assumption that producers must package their code for reuse, but also give the consumer help that would not be available in a traditional textual environment. To experiment with our ideas, we have integrated a prototype into the Forms/3 [1, 4, 27, 28] language environment.

Forms/3 follows the spreadsheet paradigm in which all computation is accomplished by cells whose values are entirely dependent on their formulas. Forms are collections of cells and are the basic reusable component of Forms/3 programs. Saved forms have previously only been accessible by file name via a file dialog box. We have incorporated interactive access to a code repository in Forms/3, including capabilities aimed at answering some of the consumer questions we identified in Section 1.1.

3.1 Overview

I am not familiar with the repository;
what does it contain?

In our prototype, the consumer encountering the repository for the first time can call up an interactive overview of it by pushing the "Repository" button on the main Forms/3 window. A repository window opens (Figure 1), in which a large scrolling pane displays a component graph. Each node in the graph represents one component and edges indicate some sort of relationship between components. Edges can indicate dataflow (which includes calling and shared-data relationships) or relationships similar to object-oriented subclassing. (In the current prototype, subclass edges have not yet been implemented.)

Every form that has been saved in the Forms/3 directory hierarchy with world-readable permissions is included in this component graph—no special tasks are required to prepare the code for submission to the repository. The presence of an edge is determined from derived information, made possible by the integration of the language with the language environment. This derivation not only relieves the producer from having to document the component's relationship to other components, but also guarantees that the relationship information will not become obsolete as code evolves.

Nodes in the graph are labeled with the name of the component. More details about the origin of individual components are easily accessible in a popup window. In Figure 2 the consumer is looking at the detailed view of a node, and can see the component's location, author, date, and any packages to which it belongs.

The Forms/3 language uses packages (similar to Ada packages) to group related forms into larger units. The consumer can browse the list of packages found in the repository as another way to become familiar with the repository at a coarser granularity. The list of these packages can be pulled down from the query menu bar (see Figure 1, top left).

Recall our assumption of a repository with hundreds of components. To support a larger repository, more sophisticated visualization and browsing techniques would be required, for example [2, 7, 15, 23]. However, these systems are typically oriented toward textual data. VPL-specific features such as browsing a VPL's source code according

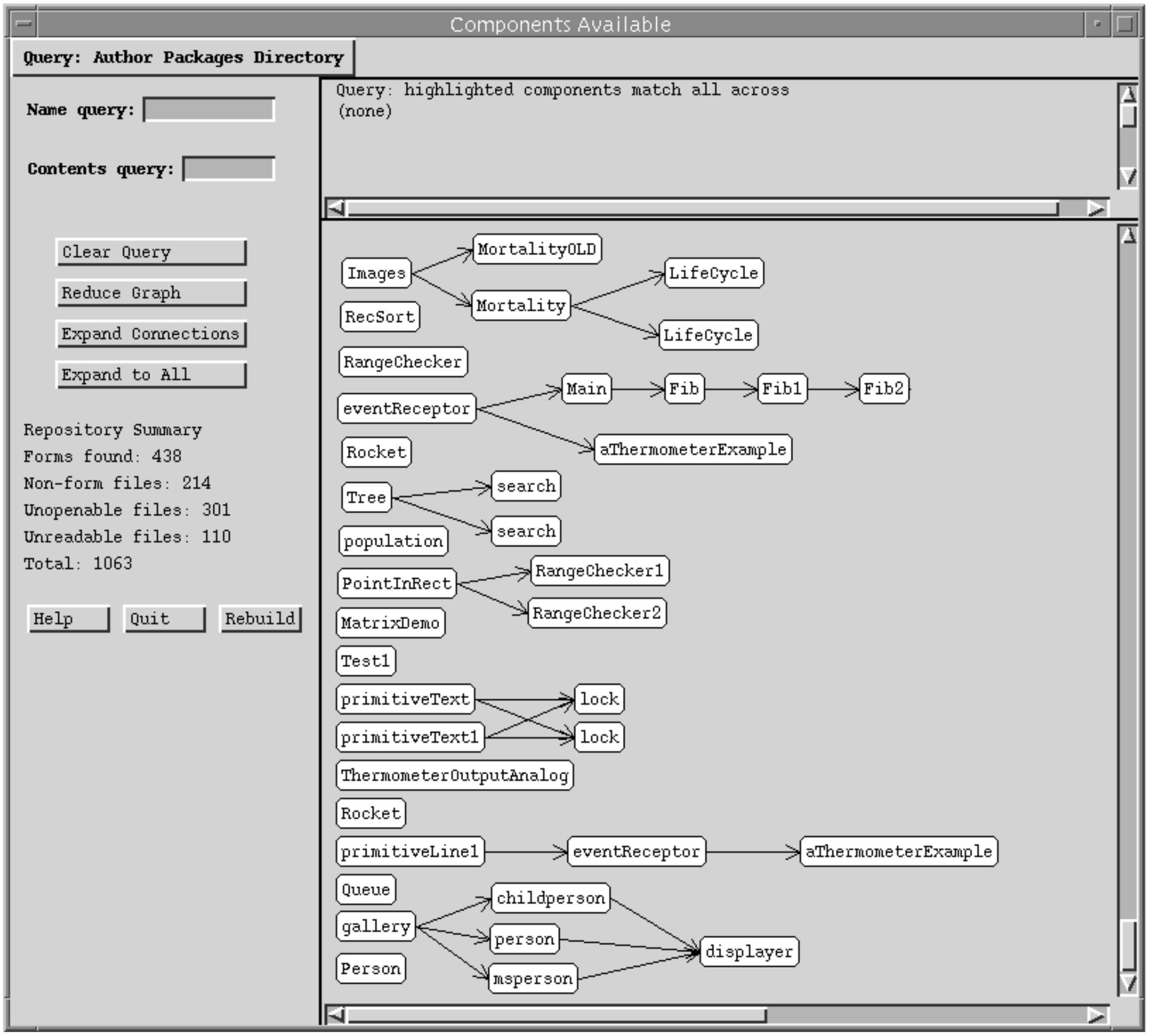


Figure 1: Repository Overview: the large scrolling pane displays components as nodes connected by dataflow edges.

to its visual characteristics might be able to draw from the work on visual databases [8, 9], but we have been unable to find reports of any research to test this idea.

In order to support the capabilities described thus far, an environment either needs full parsing capabilities or a source-level understanding of the stored executable code. This understanding is necessary to automatically derive relationships, thus freeing the producer from having to explicitly document them, a task that would otherwise be part of the packaging process. Adding parsing capabilities to a programming environment is not difficult for most textual languages, if there is an accessible grammar specification. For visual languages, visible grammars and parsing methods are still an area of active research. The strategy used in the Forms/3 prototype is to operate on the executable code, which is stored in a tokenized representation.

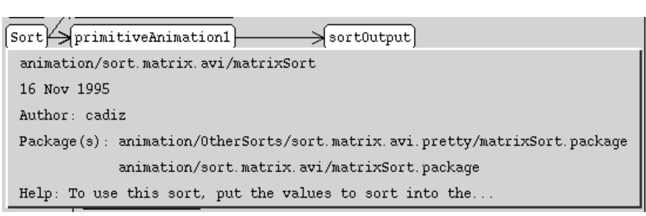


Figure 2: Detail view pulled down from Sort component (upper left), showing file name, date, author, packages, and the first line of the help file.

3.2 Search

I know generally what I want;
which components are relevant?

Suppose that the consumer needs to sort some products according to price and is looking for a component that performs sorting. The interactive query facility currently allows queries such as on name substring or any component text. For example, the consumer can type “sort” into the name field to highlight all nodes that include the substring “sort” and then press the “Reduce Graph” button to hide all nodes that are not highlighted.

The query facility also allows searching on author, package and directory, and all can be combined. Future work on our approach will allow consumers to search based on the (inferred) types of a component's inputs and outputs.

I want a component that I've used before, but I forget the name and other details. Where is it?

Suppose instead that the consumer has a particular sort in mind. There are many components with "sort" in the name in the Forms/3 repository; upon seeing this, the consumer may continue to refine the query on this subset of the repository using other criteria. For example, the consumer may remember working on the component with two other programmers, and thinks one of them may have been the producer of the sort. Adding both producers to the query (called authors in the query environment) by selecting them from the query menu turns off the highlighting on nodes not associated with those authors. The final results of this query are shown in Figure 3.

I know precisely what I want; is it here?

Finally, suppose that the consumer is specifically looking for a bubblesort. Queries can also include searches over the textual contents (comments, cell names, formulas, and even sample values) of a component and any associated on-line documents such as help files, test plans, and specifications. The consumer can type "bubblesort" into the "Contents query" text box to highlight any component that contains this term in its textual contents or associated documents.

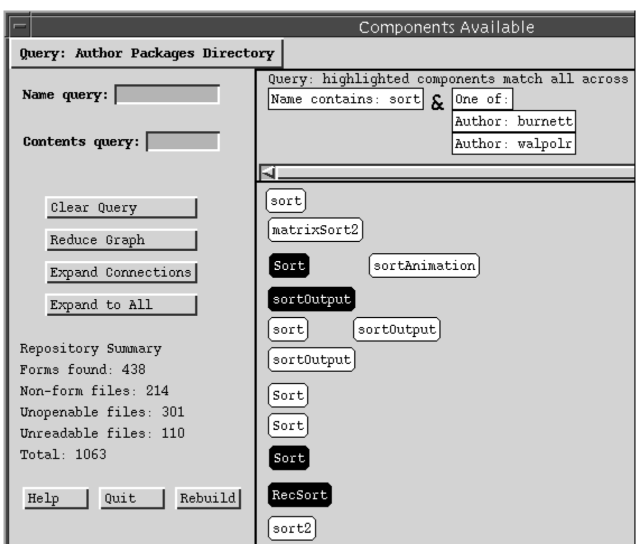


Figure 3: Query results. Before the authors were added to the query, all of the components here appeared highlighted.

Although the query facility described in this section is not unusual in the features it offers consumers, it is innovative in two ways. First the approach supports the ability to query without requiring any packaging on the producer's part; and second, its tight integration into the language environment prevents the consumer from switching between separate tools for searching for code, testing it, and changing it, as will be seen in the next section.

3.3 Exploration

What does this component do?

When the consumer is ready to investigate the purpose of a particular component, a repository browser for a traditional textual language (e.g., CodeFinder [11]) is likely to display the component's source code, but to provide no additional support for exploring the component. This is where three features found in many VPLs can be leveraged to provide support for component exploration, allowing the consumer not only to view the source code, but also to interact with executions of it.

The first two of these three features are sample values and immediate feedback. Returning to the sorting example, to explore a component of interest in the Forms/3 repository, the consumer double-clicks on its node in the graph. Without switching environments, the consumer immediately sees not only the code (formulas), but also its effects on sample values. For example, see Figure 4. (The formula tab hanging from the bottom of the "input" matrix indicates that new input can be entered here to sort values that are different from the sample.)

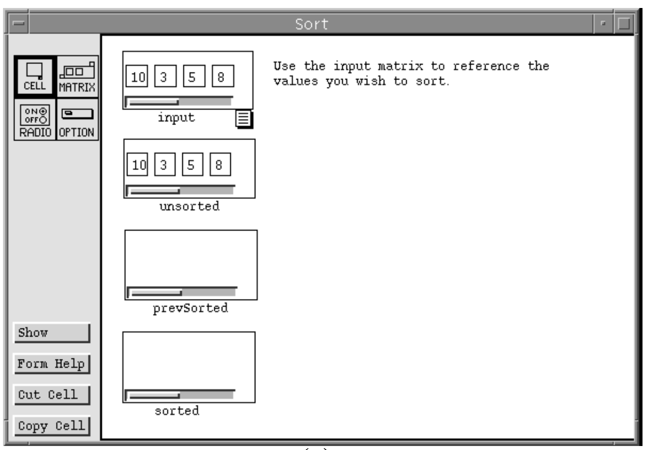
These sample values give concrete examples of the computations, illustrating the component's function. Sample values are provided by the producer of the component, but since providing sample values is the natural way to program and test in Forms/3, no special step or extra work is required of the producer. This leveraging of sample values to help answer the "What does this component do?" question can be accomplished by any of the VPLs, such as by-demonstration and spreadsheet-based languages, that support sample values.

Does this component do what I need?

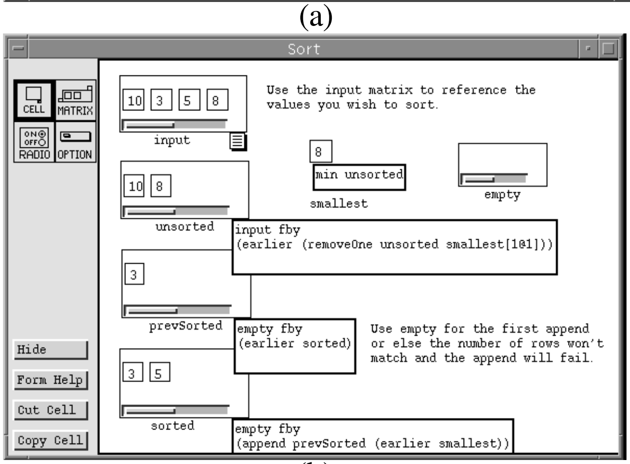
The information provided by the sample values is greatly enhanced by the presence of immediate semantic feedback (which Tanimoto termed *liveness* [25]) because the consumer can interact with the system by entering new sample values and immediately see the results. Different sample values suited to the consumer's needs, such as an input matrix that includes duplicate values, may give the consumer more confidence in the suitability of this component.

How does this component do what it does?

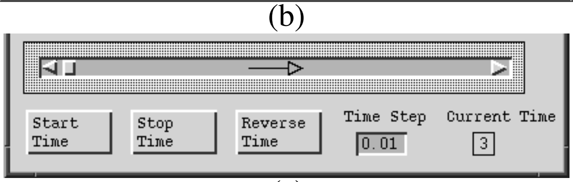
This question arises in customization situations. When the consumer displays hidden cells (local variables) and detailed comments and then allows the system to move through time as in Figure 4, the synchronized changing values of the cells resulting from liveness automatically functions as a low-level visualization of the code. In fact, this same interaction style even allows the consumer to change any formula desired and immediately see the results, thereby allowing experimentation with changes and the development of customization ideas. VPLs with somewhat less liveness can still offer a degree of interaction, even though more steps (such as pressing an



(a)



(b)



(c)

Figure 4: (a) shows the sort component as it appears when first loaded. In (b) the consumer has displayed most of the formulas and all the hidden code and comments. By manipulating the timeslider (c), the consumer can watch the sort progress.

execute button or asking for individual values) will be necessary.

While no special effort is assumed of the producer in the Forms/3 repository, if any extra work is associated with a component as it matures, the results of that work are available in the same environment. One such extra is algorithm animation, the third feature of special utility in VPLs for answering the question of what a component does. Algorithm animation is “the process of abstracting the data, operations, and semantics of computer programs, and then creating dynamic graphical views of those abstractions....[it involves] program views that conceptually portray how a program works” [24]. In contrast to the detailed visualizations discussed above, an algorithm animation allows a higher-level view of what a program does.

Forms/3 includes full support for algorithm animation [6], and whenever a producer of a component in Forms/3 has prepared an associated animation, the component graph will show an edge to an animation component. In case the consumer has reduced the graph (as in the example query above), the “Expand Connections” button adds back into the graph all components that have an edge to or from those already in the graph. Figure 5 shows a part of the graph after expansion. The consumer can click on the associated animation directly from the graph to see a high-level visualization of the original component. For example, Figure 6 shows an algorithm animation in the midst of execution. The consumer controls the animation’s speed and direction while the algorithm and the animation execute synchronously. Algorithm animation is one way to cultivate abstract understanding, which is an important factor in the ability of novice programmers to reuse code [12].

3.4 Use

How do I use this particular component?

A recent study of professional Smalltalk users [20] observed that consumers make extensive use of previous usage contexts when figuring out how to use unfamiliar components. The dataflow graphs of the overview automatically display these contexts, and the consumer can interact with any of them using the same mechanism described earlier to find algorithm animations.

For example, in Figure 5 the consumer can see that one of the highlighted sorts is connected to a component named “quiz-stats.” The consumer can load the sort and this context by double-clicking both nodes on the graph. Or, if the producer has set up a package, double-clicking on either node will produce a dialog box allowing the consumer to choose between loading the component alone or an entire package.

By running the quiz-stats program that uses the sort component, the consumer can see by example how to use the sort. In Figure 7, the quiz-stats program has run and the consumer can see that it uses a Boolean value to indicate whether the sort has finished. From this example, the consumer concludes that the same Boolean test would be useful in new programs that use this sort.

We believe that this ability to browse, retrieve and experiment with usage contexts is critical to allowing consumers to successfully reuse code that has not been packaged by its producers. This ability permits an alternative to documentation and standardization procedures that would normally be producer tasks, a feature that is especially important in immature repositories of developing code and in cooperative repositories not "owned" by any organization.

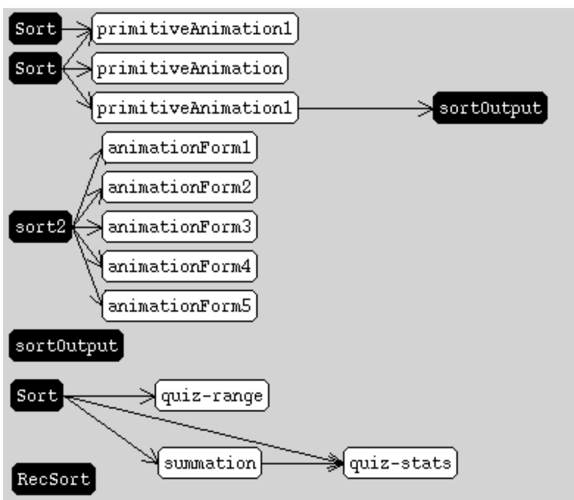


Figure 5: A portion of the graph after it has been expanded to show connections with other components.

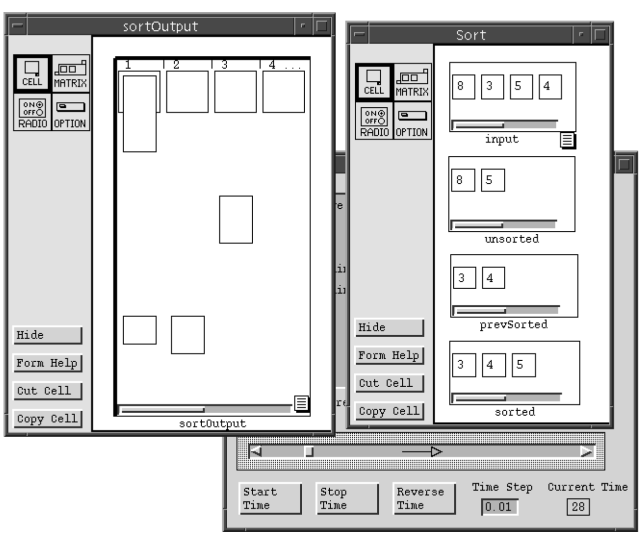


Figure 6: The consumer has loaded the animation and is controlling it with the time-slider at the lower right.

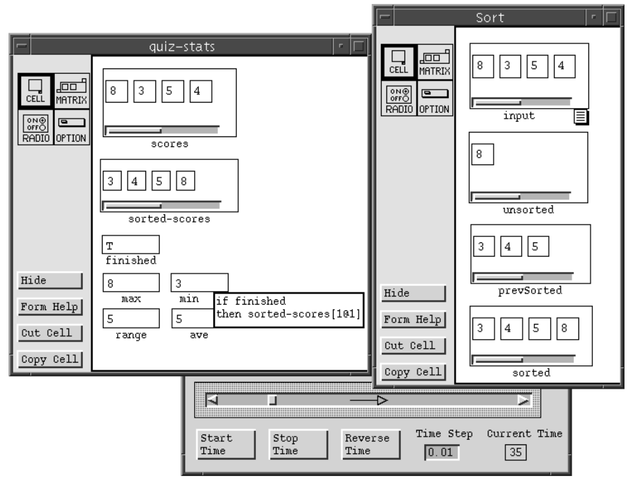


Figure 7: The quiz-stats program with time advanced far enough so that the "finished" cell has a true value. The consumer can see that the statistics are computed only if the sort has finished.

4. Conclusion

In this paper, we have presented early results of our work on an approach to facilitate reuse in evolving repositories of shared VPL code. We have presented several techniques intended for VPLs that feature a tight integration between language and environment. These techniques leverage characteristics found in these VPLs to promote code reuse features needed for these informal, evolving repositories. Among these features are:

- No special work required of component producers: Our approach supports reuse by a consumer even if the producer has not gone to the work of doing "reusability packaging."
- Saving the consumer from task separation: One of the problems traditionally present in code reuse is that it is often perceived to be easier for consumers to write new code than to find previously-written code, figure out whether it produces the desired results, and learn how to interface with it. By integrating the repository access into the language environment, we are working to address this problem by dissolving the obstacles that separate reusable code from the consumer's working environment.
- Support for informal repositories: Informal repositories do not feature the kinds of controls traditionally followed in "owned" repositories, and hence support cannot be based on the assumption that a particular classification scheme or formalized documentation standard is present. In our approach, components may be only partially-documented or standardized, but they may be explored for reuse using features supported by many VPLs, such as sample

values and automatic executions, automatically-derived dataflow views, and the low-level visualizations that are often found in VPLs.

We believe that with the advent of the Web, shared, informal repositories that do not require reusability packaging will become increasingly important. To address this need, our approach derives the kind of information formerly only available through packaging efforts by code producers and repository administrators, doing so by drawing upon the complete knowledge of language semantics inherent in tightly-integrated VPLs, along with other characteristics found in many such VPLs, such as sample values, liveness, and animation.

Acknowledgments

We are deeply grateful to Martin Griss and Gregson Siu at Hewlett-Packard for helpful discussions and feedback.

References

[1] J. Atwood, M. Burnett, R. Walpole, E. Wilcox, S. Yang. Steering programs via time travel. *Proc. IEEE Symp. on Visual Languages*, 4-11, Boulder, CO, Sept. 1996.

[2] B. Bederson, J. Hollan, K. Perlin, J. Meyer, D. Bacon, G. Furnas. Pad++: A zoomable graphical sketchpad for exploring alternate interface physics. *J. of Visual Languages and Computing*, 7(1):3-31, 1996.

[3] T. Biggerstaff and C. Richter. Reusability framework, assessment, and directions. In T. J. Biggerstaff and A. J. Perlis, eds., *Software Reusability: Concepts and Models, Volume 1*. Addison-Wesley, 1989.

[4] M. Burnett and A. Ambler. Interactive visual data abstraction in a declarative visual programming language. *J. of Visual Languages and Computing*, 5(1):29-60, Mar. 1994.

[5] M. Burnett, A. Goldberg, T. Lewis, eds. *Visual Object-Oriented Programming Concepts and Environments*. Prentice-Hall/Manning Publications Co., 1995.

[6] P. Carlson, M. Burnett, J. Cadiz. A seamless integration of algorithm animation into a visual programming language. *ACM Proc. Workshop on Advanced Visual Interfaces*, 194-202, Gubbio, Italy, 1996.

[7] M. Chalmers, R. Ingram, C. Pfranger. Adding imageability features to information displays. *ACM Symposium on User Interface Software and Technology*, 33-39, Seattle, WA, Nov. 1996.

[8] J. Corridoni, A. Del Bimbo, S. De Magistris, E. Vicario. A visual language for color-based painting retrieval. *Proc. IEEE Symp. on Visual Languages*, 68-75, Boulder, Co, Sept. 1996.

[9] M. Egenhofer. Spatial-query-by-sketch. *Proc. IEEE Symp. on Visual Languages*, 60-67, Boulder, CO, Sept. 1996.

[10] M. Gorlick and A. Quilici. Visual programming-in-the-large versus visual programming-in-the-small. Proc. *IEEE Symp. on Visual Languages*, 137-144, St. Louis, MO, Oct. 1994.

[11] S. Henninger. Using iterative refinement to find reusable software. *IEEE Software*, 48-59, Sept. 1994.

[12] C. Hoadley, et al. When, why and how do novice programmers reuse code? *Empirical Studies of Programmers, Sixth Workshop*, 1996.

[13] *IEEE Software*, Special Issue on Systematic Reuse, Sept. 1994.

[14] D. Ingalls, S. Wallace, Y. Chow, F. Ludolph, K. Doyle. Fabrik, a visual programming environment. *Object-Oriented Programming Systems, Languages, and Applications,* 76-190, San Diego, CA, Sept. 1988.

[15] B. Johnson and B. Shneiderman. Treemaps: a space-filling approach to the visualization of hierarchical information structures. *Proc. Int'l IEEE Visualization Conference*, 284-291, San Diego, CA, 1991.

[16] Y. Koike, Y. Maeda, Y. Koseki. Enhancing iconic program reusability with object sharing. *Proc. IEEE Symp. on Visual Languages*, 288-295, Boulder, CO, Sept. 1996.

[17] H. Mili, F. Mili, A. Mili. Reusing software: Issues and research directions. *IEEE Trans. on Software Engineering*, 21(6):528-61, 1995.

[18] W. Pree. Framework development and reuse support. In Burnett et al. [5].

[19] A. Repenning and J. Ambach. Tactile programming: A unified manipulation paradigm supporting program comprehension, composition and sharing. *Proc. IEEE Symp. on Visual Languages*, 102-109, Boulder, CO, Sept. 1996.

[20] M. Rosson and J. Carroll. The reuse of uses in Smalltalk programming. *ACM Trans. on Computer-Human Interaction*, 3(3):219-253, Sept. 1996.

[21] S. Schiffer and J. Fröhlich. Visual programming and software engineering with Vista. In Burnett et al. [5].

[22] M. Sitaraman, ed. *Proc. IEEE Int. Conference on Software Reuse*, Orlando, FL, April 1996.

[23] M. Spenke, C. Beilken, T. Berlage. FOCUS: The interactive table for product comparison and selection. *ACM Symp. on User Interface Software and Technology*, 41-50, Seattle, WA, Nov. 1996.

[24] J. Stasko. Simplifying algorithm animation with TANGO. *Proc. IEEE Workshop on Visual Languages*, 1-6, Skokie, IL, 1990.

[25] S. Tanimoto. VIVA: A visual language for image processing. *J. of Visual Languages and Computing*, 2(2):127-139, June 1990.

[26] W. Tracz. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Addison-Wesley, 1995.

[27] P. van Zee, M. Burnett, M. Chesire. Retire superman: Handling exceptions seamlessly in a declarative visual programming language. *Proc. IEEE Symp. on Visual Languages*, 222-230, Boulder, CO, Sept. 1996.

[28] S. Yang and M. Burnett. From concrete forms to generalized abstractions through perspective-oriented analysis of logical relationships. *Proc. IEEE Symp. on Visual Languages*, 6-14, St. Louis, MO, Oct. 1994.