Efficient Specification Techniques for Software Visualization

Arne Frick
Universität Karlsruhe
Fakultät für Informatik
Vincenz-Prießnitz-Str. 1
Postfach 6980
D-76128 Karlsruhe
Federal Republic of Germany
EMail: frick@ira.uka.de

February 18, 1994

Abstract

Software Visualization is developing into an important tool for teaching algorithms and debugging complex sequential and parallel programs. However, specifying the relationship between program objects (static ones as code statements or variables and dynamic ones as data structures or object states) and their geometric counterparts turns out to be the major bottleneck in using SV systems [1, 2, 3].

My work aims at the development of a simple, universal, flexible, and modular architecture allowing novice *and* expert users to efficiently specify visualizations at arbitrary levels of abstraction (abstract data type operations, program statement or variable observation). Thus, it represents a natural extension of the concept of a *debugger* program, but is also a valuable tool for Intelligent Tutoring Systems.

My approach is best described as a combination of *visual programming* and *direct manipulation* techniques with the additional possibility of conventional programming.

A key feature of the architecture is a *Visualization Description Language (VDL)* which is interpreted and can be extended at run-time. This allows for the construction of domain-specific visualization components which are re-usable as long as the visualization targets remain in the same domain.

1 Introduction

The bottleneck for Software Visualization on the road from academia to wide-spread application is the large amount of time that has to be spent developing and describing the mapping between program objects and their visualization. A few examples showing evidence for this fact are given. BROWN and SEDGEWICK [3] pointed out that the development of a *view* for an algorithm animation in BALSA meant an effort of 15-25 hours of programming, not counting another 1–2 hours of writing a script for the animation using the view. This situation has not changed much until 1992, when BROWN reported that in the course of the DEC SRC Algorithm Animation Festival 1992, it took a group of test persons two weeks of intensive training and testing to be able to develop simple animations [2].

Besides algorithm animation, visualization techniques can be exploited in Intelligent Tutoring systems. Here, the situation is not much bet-

ter. ANDERSON [1] states that the development of a tutoring session in high school mathematics of 1 hour length takes 100 hours development time[1]. These figures show the relevance of being able to efficiently specify visualizations.

We strive to solve the problem of specifying visualizations by developing an architecture called *DynaStruct*. The name is an abbreviation for *Dyna*mic *Struct*ure Visualization. This reflects the fact that the architecture is not solely usable for software visualization, but in general for graphical simulation applications. Possible input sources in a SV environment include program traces, hand-written scripts or the output of a graphical specification tool.

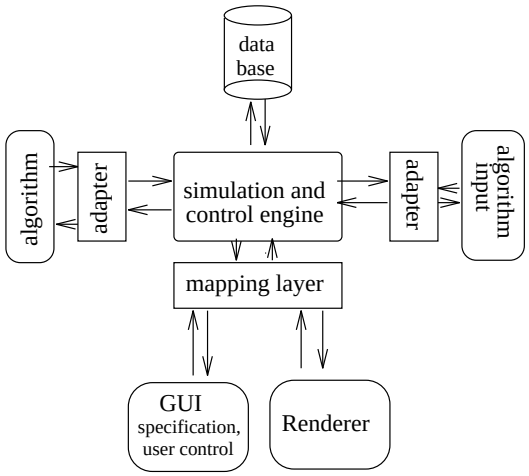

Figure 1: The DynaStruct architecture

2 Architecture

In this section, we describe the DynaStruct architecture. First, the design goals are explained. Following that, we show how to achieve them.

The major design goals for DynaStruct are *usability*, *universality*, *flexibility* and *modularity*. A system derived from the architectural specification should be simple to use. Every conceivable visualization in the visualization domain should be specifiable. A modular structure allows for components to be exchanged if either the application domain (graphs, geometry, parallel and distributed algorithms) or the visualization domain change. Modularity therefore inherently also supports *scalability* in terms of domain changes.

By flexibility we mean the possibility to extend the functionality of a system at run-time. This design goal supports the usability of the system. Although universality guarantees that the designer of a view can express every possible visualization in the visualization domain, it can be very tedious to start the design of a view from scratch. In this situation, the following observation helps: each application area has a largely unique set of constructors. Since the designer of a visualization system cannot foresee every possible application of her system, she has to allow for structuring primitive constructors into more powerful ones. Over time, the system gradually adopts more and more visual primitives for an area of application.

The main components of the DynaStruct architecture are shown in figure 1. An object-oriented data-base contains the complete history of a visualization, if desired, as well as the current mapping of program objects to visualization objects. The visualization itself is modeled as a simulation process involving events occuring in the algorithm under consideration and actions workin upon the visualization. A sophisticated control mechanism allows for navigating in the history, possibly changing algorithm data somewhere in the past of the simulation and executing from then on.

There are three well-known techniques for specifying the mapping between program objects and visualization objects [8]:

- code modification
- annotation

[1] The duration was calculated for slow students. That means that fast students might finish way ahead of time.

- declaration, constraints

The code modification method violates the software engineering principles of modularity, information hiding and code clarity. Furthermore, the source code might not even be available in modifiable form, as might be the case for library routines.

Our approach uses direct manipulation techniques on top of a graphical user interface (GUI) in order to visually specify the relationship between program objects and visualization objects. Technically speaking, the GUI drives a symbolic debugger to implant annotations and declarations into the program code. Conditional breakpoints can be used for implementing constraints. However, this cannot be considered a full-blown declarative approach since the visualization designer has to specify the location of the constraint checks.

3 Work in Progress

We currently finish the design of the VDL. A first prototype has has been used to visualize several graph algorithms. This work is being complemented by empirical studies [4].

As a result of user input, emphasis in VDL is being put on the issue of *granularity*. When visualizing complex algorithms, users want to see a visualization on an abstract level first, focusing in on details only when necessary. Consider for example an algorithm visualizing the network flow algorithm by Ford and Fulkerson [5]. At a coarse-grained level, the algorithm consists of the steps

- find an augmenting path in the residual network
- determine the flow through the augmenting path
- add the augmenting flow to the current netowrk flow to get updated maximal flow

This example illustrates a few interesting points:

- users normally do not want to see how the augmenting path is found; they are satisfied with the path and the possibility to check whether this is indeed an augmenting path
- if, however, they choose to view the path-finding part, they are no longer concerned with the higher-level details, but want to understand the abstract operation entirely in terms of the locally relevant data.

We conclude that an algorithm visualization system should provide for mechanisms to express and support granularity.

4 Directions for Future Work

The DynaStruct architecture currently works in the 2D visualization domain. Other visualization domains include 3D, sound and other possible visualization techniques included in the definition "various techniques to enhance the human understanding of computer programs" [7]. Extension of our ideas to other visualization domains should be straight-forward due to the orthogonal construction of the language, which allows for easy addition of new operations even at run-time.

It remains to investigate how distributed and parallel programs can be visualized with this architecture. We conjecture that due to observational difficulties, the best strategy in these areas is to create offline traces can then be visualized without interfering with the timing behaviour of the software systems under observation.

Dynastruct was designed to support the interactive specification of the mapping between entities (abstract or concrete) in the algorithm under consideration and visualization entities. However, the domain-specific knowledge still has to be hand-coded in domain-modules. A natural extension of our visual specification technique is to incorporate ideas from the *programming by demonstration* method [6]. This might lead to methods for the interactive specification of the domain-specific knowledge itself.

References

[1] John R. Anderson. Intelligent tutoring and high school mathematics. In *Proc. ITS'92*, volume 608 of *LNCS*, pages 1–10. Springer, 1992.

[2] Marc H. Brown. The 1992 SRC Algorithm Animation Festival. DEC SRC Technical Report 98, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, March 1993.

[3] Marc H. Brown and R. Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(1):28–39, January 1985.

[4] Delel Chaabouni, Arne Frick, Stefan Hänßgen, Christopher Hundhausen, and Günther Mossakowski. Spezifikation einer Visualisierung für den Ford-Fulkerson Algorithmus, 1994. German; instructions for a video-taped interactive experiment.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1989.

[6] A. Cypher. *Watch what I do — Programming by Demonstration*. MIT Press, 1993.

[7] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3), September 1993.

[8] Guia-Catalin Roman and Kenneth C. Cox. A taxonomy of program visualization systems. *IEEE Software*, 26(12):11–24, December 1993.