Proceedings of the Third Program Visualization Workshop

The University of Warwick, UK, July 1–2, 2004

Ari Korhonen, editor

Research Report CS-RR-407
Department of Computer Science
The University of Warwick
Coventry CV4 7AL
UK

Distribution:
University of Warwick
Department of Computer Science
Coventry CV4 7AL
UK

Tel: +44 24 7652 3193
Fax: +44 24 7657 3024
Email: pvw04@dcs.warwick.ac.uk
URL: http://www.dcs.warwick.ac.uk/

Printed by WarwickPrint, 2004.

ISBN 0 902683 74 8

Foreword

This is the proceedings of the Third Program Visualization Workshop, PVW'04, organized at the University of Warwick, UK, on July 1–2, 2004.

Program Visualization Workshop has been organized in Europe every second year since year 2000. Previous workshops have been organized in Porvoo, Finland (2000) and at HornstrupCentret, Denmark (2002). All workshops have also been organized in co-operation with ACM SIGCSE and in conjunction with the ITICSE conference, to promote participation in both conferences.

The aim of the workshop is to bring together researchers who design and construct visualizations and visualization systems for computer science, especially for programs, data structures and algorithms, and, above all, educators who use and evaluate visualizations in their teaching. Due to the limited number of participants and that most participants are actively working in the field of software visualization, the workshop provides an excellent opportunity to exchange ideas and experiences, as well as disseminate novel systems.

This year twenty-two participants from ten countries attended the workshop. The invited lecture by Ben du Boulay from the University of Sussex and twenty paper presentations gave us an excellent overview of current lines of research in program visualization and algorithm animation, as well as in several other interesting topics in the visualization field. In addition, the workshop allowed us to get to know many people working in the field, and thus give seeds to fruitful co-operation in the future.

This proceedings includes the revised and extended versions of the papers accepted to the workshop. Each paper was initially reviewed by two members of the program committee, and the first revised version was included in the pre-proceedings delivered to the participants in the workshop. All papers were presented in the workshop and many fruitful discussions followed the presentations. After the workshop at least one member of the program committee critically reviewed each paper to give opportunities to improve the papers even more. In addition, the authors were allowed to extend the papers to include more details, examples and arguments. The final versions are now included in this proceedings. The papers themselves are available online at: http://www.cs.hut.fi/Research/PVW04/.

I wish to thank the program committee members for their critical and encouraging comments on all of the papers. The program committee was

- Lauri Malmi, Helsinki University of Technology, Finland (chair)
- Mordechai Ben-Ari, Weizmann Institute of Science, Israel
- Mike Joy, University of Warwick, UK
- Ari Korhonen, Helsinki University of Technology, Finland
- Guido Rößling, Technische Universität Darmstadt, Germany
- Rocky Ross, Montana State University, USA
- Ángel Velázquez-Iturbide, Universidad Rey Juan Carlos, Spain

Mike Joy was also responsible for the local arrangements, and I wish to thank him for his excellent work. The proceedings was edited by Ari Korhonen.

Espoo, Finland, October 2004

Lauri Malmi

Contents

Representation in Learning Computer Science: Black Boxes in Glass Boxes Revisited

Ben du Boulay, University of Sussex

Central issues in the teaching of computer science and other subjects such as mathematics are those of the nature and use of representations. In Open Learning Environments a crucial role of representation is to encode and then offer students ways of thinking about the entities, relationships and processes of the domain in question. For example, in our work on learning Prolog, much effort centred on trying to make program trace information at once both meaningful yet explicit about the complex, and normally implicit, processes such as unification and search underpinning that computer language. In our work on the Discover system, an even more explicit model of the functioning of the virtual machine was provided to aid both program construction as well as debugging. In our work on an advice system for novice Unix file system users, the representation issue was centred not just on the semantics of the domain, but also on novices' misunderstandings of that domain. The system needed to make plausible inferences about what the user might have meant to achieve when they typed the peculiar command that did. In our current work with Java we have been exploring the way that programmers exploit the multiple representations typically available in a program development and debugging environment to build a coherent understanding of the program as a complex multi-faceted entity. Indeed like Prolog, Java provides complex, but normally implicit process that learners need to understand if they are to be successful programmers.

This paper describes a number of projects undertaken by our research group in the area of learning and teaching university level computer science. In each case it focuses on the various representations explicitly and implicitly available to the learners and discusses how those representations assisted (or impeded) learning and problem-solving.

A Survey of Program Visualizations for the Functional Paradigm

Jaime Urquiza-Fuentes, J. Ángel Velázquez-Iturbide
Universidad Rey Juan Carlos, Madrid, Spain

{j.urquiza,a.velazquez}@escet.urjc.es

1 Introduction

One of the definitions for visualization is to give a visible appearance to something, making it easier to understand. In Price et al. (1998), *software visualization* is defined as "the use of crafts of typography, graphic design, animation and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software". *Program visualization* is defined as "the visualization of actual program code or data structures in either static or dynamic form".

We focus on the functional programming paradigm here. We study crafts used to visualize functional program code and data structures. The study has been done over sixteen systems. These systems can be categorized in multiple ways (Price et al., 1993; Myers, 1986; Brown, 1998). Although we do not want to make a new taxonomy, we differentiate among integrated development environmens, debuggers, teaching systems and visualization systems.

We have made a compilation of information about functional visualization systems (this information is very dispersed). In general, most systems are partial solutions to the main problem; the visualization of functional programs. Our ultimate aim is to find a more general solution.

In section 2 particular aspects of the functional paradigm are introduced. Section 3 briefly describes the systems we studied. The visualization of each particular aspect identified is presented in section 4. In section 5 the evaluation of some systems is described. Finally we draw our conclusions in section 6.

2 Features of the Functional Paradigm

The functional programming paradigm has some particular features that are needed to be visualized to understand the execution of a program. In functional programming the source code of a program is formed of bodies of functions. Each function is a set of rules. In the following, a program to compute the addition of elements in a list is shown:

```
fun sumlist list(int) -> int
    | sumlist([]) = 0
    | sunlist(head::rest) = head + sumlist(rest);
```

The execution of a functional program begins with an expression in which some of the functions of the program are called. Each execution step is a rewriting step applied on an expression, and its result will be another expression. The following are all the rewriting steps of the execution of `sumlist([3,5,2])`.

`sumlist([3,5,2])` ⇒ `sumlist(3::[5,2])`
`3 +` `sumlist([5,2])` ⇒ `sumlist(5::[2])`
`3 + 5 +` `sumlist([2])` ⇒ `sumlist(2::[])`
`3 + 5 + 2 +` `sumlist([])` ⇒ `sumlist([])`
`3 + 5 +` `2 + 0`
`3 +` `5 + 2`
`3 + 7`
10

As shown in the previous example, the rewriting steps are applied to parts of the whole expression (framed code in the example). For each step, the next subexpression to rewrite (or reduce) is called the *redex*. Each rewriting step is related to the evaluation of a (sub)expression, and each evaluation gives (sub)results. Therefore, important aspects to visualize are the evaluation of (sub)expressions, its corresponding redexes and the (sub)results obtained.

Another feature to visualize is the order in which function calls are executed. Two important details are the evaluation of parameters and how pattern matching is used to select the appropiate rule in the body of the function to be applied.

The environment of variables (also called contour) fixes their values, so it will be important to clearly visualize those environments. Moreover, if complex data structures are used in a program, as lists or trees, it will be desirable to work with special visualizations for them.

There are two ways (also called strategies) of executing a functional program. The previous example shows eager execution. Alternatively, lazy execution evaluates an expression only when necessary. For example, the expression `fact(4+2)` is reduced to `if((4+2)=1) then 1 else (4+2)*fact((4+2)-1)`. The function `fact` is applied before evaluating the argument `4+2`. In order to avoid inefficiency, the subexpression `4+2` does not appear 3 times, but it is unique and shared among the three places. Therefore, expression sharing is an important feature of lazy evaluation to visualize.

3 Systems Studied

Normally, the features to be visualized and the way this is achieved depend on the class of the system being used. We have therefore classified systems into four categories: integrated development environments, debuggers, teaching systems and visualizing systems.

Integrated development environments use to integrate a number of tools under the same interface. *CIDER* (Hanus and Koj, 2001) uses the lazy language Curry. It integrates edition, program analysis tools, a graphical debugger, and a dependency graph drawing. Execution data are collected in a trail. Its debugger supports breakpoints and changing the execution direction.

WinHIPE (Naharro-Berrocal et al., 2002) uses the language Hope. Programs are executed under the eager stategy. It shows the set of expressions resulting during an evaluation. Its debugger provides general options such as executing one or n steps, evaluating to the next breakpoint, evaluating the redex or backtracking to a previous expression. It shows graphically lists and trees and supports a wide range of customizations, including graphical format, typographic characteristics and subexpression visibility. From the static visualizations generated, it allows building animations that can be saved and loaded for educational use.

ZStep (Lieberman and Fry, 1998) is a Lisp integrated environment. Its debugger supports execution in both directions, evaluating the selected expression and executing until the end. Speed execution control and a tree function calls are also provided. Execution data can again be found in a trail. *ZStep* simultaneously shows the source code and the execution code. Execution errors are located in the same place where the correct values should be located.

The last environment is called *TERSE* (Kawaguchi et al., 1994). Properly speaking, it is not a functional program environment, but rather a term rewriting system (which is the basis of functional program execution). It has been developed with Standard ML/NJ, and allows transforming *TERSE* programs into Standard ML programs. During execution it allows selecting, among all redexes available, the one that will be reduced. Also, it permits to choose the rule to apply and the execution strategy. It shows a global vision of the expression, represented as a tree, and a zoomed vision of a particular area of it, and also generates rewriting sequences.

Debuggers adapted to functional programming are commonly called *steppers*. We have studied six debuggers. *Freja* (Nilsson and Sparud, 1997) and *Buddha* (Pope, 1998) use subsets of the language Haskell. They are algorithmic debuggers, and use the dependency reduction

graph to guide the user while debugging.

Hat (Sparud and Runciman, 1997) also supports a subset of Haskell. It generates a trail of reduced redexes, allowing to browse it in a graphical way.

Hood (Gill, 2000) uses the whole Haskell language. To visualize the execution, the source code must be modified, inserting calls to the visualization system where a visualization is needed (either a function or a data structure). The visualization is obtained as a result of the execution of the program.

Prospero (Taylor, 1995) and *Hint* (Foubister and Runciman, 1995) are very similar systems. *Prospero* uses the language Miranda and *Hint* uses a subset of Haskell. Both generate and use a trail. While debugging, they allow using breakpoints, but do not allow changing the direction of the execution.

Teaching systems usually focus the user's attention on particular aspects of programming languages in order to gain understanding. *Evaltrace* (Touretzky and Lee, 1992) uses Lisp. Its visualizations are documents generated with LaTeX. This system is focused on differentiating between applying and evaluating actions. It also visualizes macros and side effects. It is integrated into a programming environment.

KIEL (Berghammer and Milanese, 2001) works with a subset of the language Standard ML, were only first order functions are allowed. It allows changing the execution strategy and executing a number of rewriting steps.

DrScheme (Findler, 2002) uses the language Scheme. It allows using four subsets of the language. When a error is produced, *DrScheme* locates the function call that produced it. It has an static debugger which, using type inference, can predict potential errors.

KAESTLE & FooScape (Boecker et al., 1986) work with Lisp too, but they only visualize data structures and function call graphs. They can generate snapshots of each visualization and sequences of them. They use trails generated by the *FranzLisp* system and are also integrated in a programming environment.

We have studied two *visualization systems*. *GHood* (Reinke, 2001), which graphically shows the observations made by *Hood*. It has typical VCR controls and possible EPS output of its graphs. It generates animations where the speed can be controlled. *Visual Miranda* (Auguston and Reinfelds, 1994) uses the language Miranda. It generates a textual trail, but it can be shown in a graphical way.

4 Partial Visualizations of Functional Programs

In this section, we describe how the systems cited above support the visualization of the different aspects (partial visualizations) of functional programming mentioned in the second section. Four partial visualizations and some existing combinations of them are considered.

4.1 (Sub)expressions, Redexes and (Sub)results

A functional expression has a tree structure, so all systems work internally with expressions represented as trees (or directed graphs in lazy functional languages). Many systems also visualize expressions as trees (see Fig. 1). These are *CIDER*, *KIEL* (which allows interacting directly with the abstract syntax tree), *Prospero*, *Hint* and *TERSE* (which gives a different representation to constructors, variables and functions).

When an expression is large, its visualization can be confusing. Therefore, some tools make a compact version of the expression (see Fig. 2). *Prospero* and *Hint* allow applying filters (spatial and temporal ones). Moreover, *Hint* provides a metalanguage to define those filters. *TERSE* transforms subtrees into tree nodes. *WinHIPE* elides the visualization of less important subexpressions by applying fish-eye views. *ZStep* allows filtering expressions by defining conditions. *Evaltrace* compacts trivial evaluation steps; for instance, in the evaluation of `sumlist([])`, the evaluation of the parameter `[]` into itself is trivial.

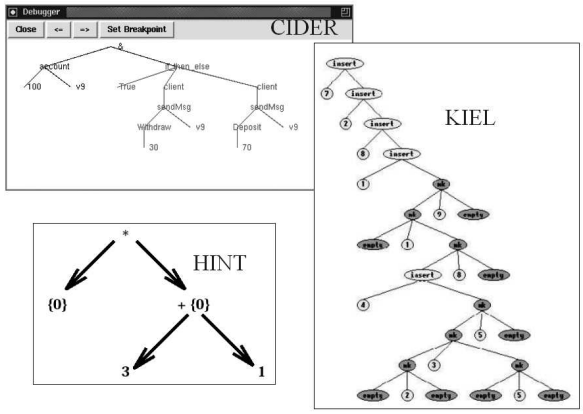


Figure 1: Expressions represented as trees

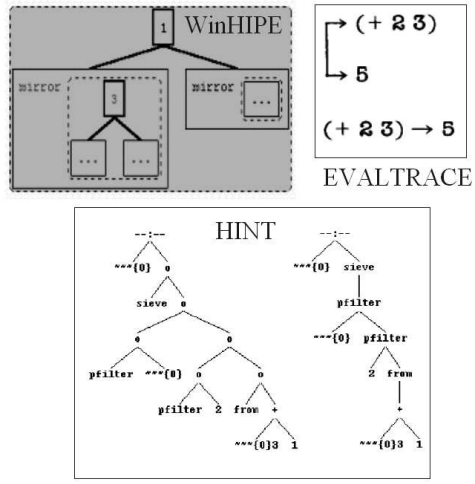


Figure 2: Compression of large expressions

All systems except *Hood & GHood* and *Evaltrace* highlight the redex (see Fig. 3). *Hood & GHood* only show the value of a variable marked as observable.

Hat, *Freja*, *Buddha* and *Evaltrace* connect each subexpresion and the result of its evaluation. *DrScheme* shows simultaneously the current expression, its reduction and the function definition used in the rewriting step. *Visual Miranda* connects the expression with its subexpressions and finally with its result (see Fig. 4).

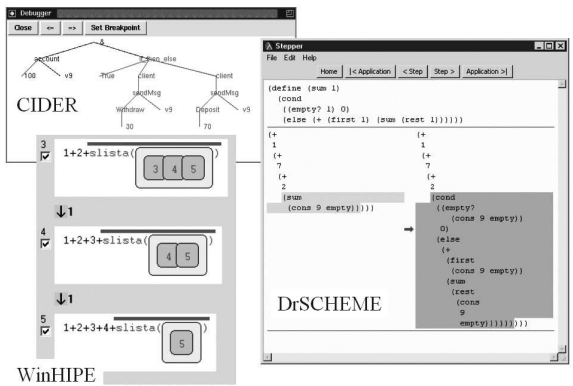


Figure 3: Redex highlighting

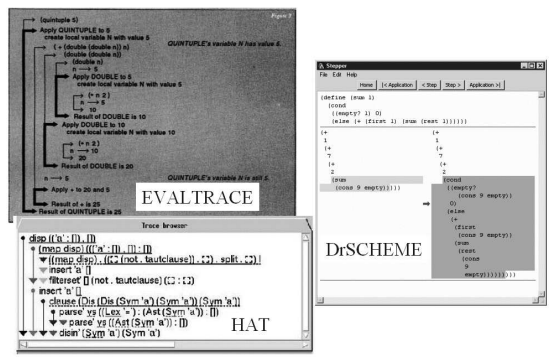


Figure 4: Connecting expressions & results

4.2 Function Calls, Function Application and Pattern Matching

The visualization of function calls is carried out by *KAESTLE & FooScape* by drawing a static flow diagram where functions are represented as ellipses, and function calls are represented as arcs from the caller to the callee (see Fig. 5). The user can choose to hide calls from a function, compacting the diagram. It allows a dynamic visualization too, by highlighting functions that have not finished their execution. *Evaltrace* focuses on differentiating evaluation and application of functions to their arguments. This is done by drawing different lines while evaluating the parameters of a function call or while applying the function: a thin line and a thick line respectively. Also, lines connect the evaluation of parameters with the function application and the result (see Fig. 4). *Visual Miranda* shows the full pattern matching process, trying to match the rule and detailing if the match fails or succeeds (see Fig. 4). Finally, *Hood* allows showing function calls and their result if the observation is located in the definition of the function (see Fig. 6).

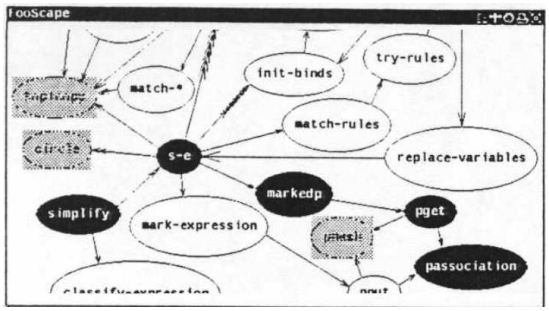


Figure 5: A function call graph in FooScape

```
last = observe "last" last'
last' (x:xs) = last xs
last' [x] = x
---last
    { \ (_ : _ : []) -> trhow <Exception>
    , \  (_ : []) -> trhow <Exception>
    , \  [] -> trhow <Exception>
    }
```

Figure 6: A *Hood* observation

Function calls may also be used as an auxiliary element, even though they do not play an important element in visualizations. Thus, *WinHIPE* uses function calls as breakpoints, but the visualization displays the current expression as a whole.

4.3 Variables and Data Structures

The contour of variables and their values is visualized in several ways. All the systems show variable values. *Hood* is a special case, because it shows values in a particular location of the source code, so in the body of a function, the user can choose to visualize a variable in a rule and not in others. *Evaltrace* identifies a contour by connecting the begining and the end with a thick line. If this line is solid, then the global contour is the parent of the present contour. Otherwise, there is a local variable definition and the parent contour is the closest enclosing one. *DrScheme* connects variables and their occurences with lines (see Fig. 7). *Visual Miranda* shows the value for each variable before evaluating a (sub)expression (see Fig. 4).

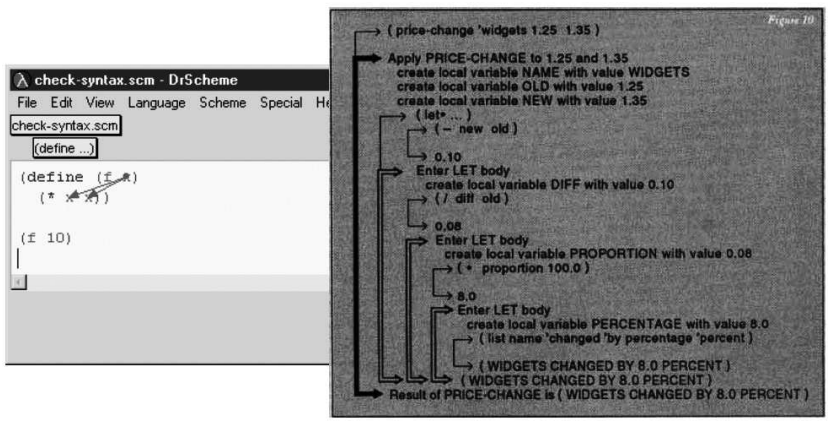


Figure 7: Contour visualization with *DrScheme* and *Evaltrace*

Only two systems allow alternative visualizations of complex structures (see Fig. 8). *WinHIPE* permits to customize the visualization of tree and lists, by identifying constructors used (the predefined constructors of the language for lists; `Node` and `Empty` for binary trees), and assigning to it the corresponding shapes, line styles, background and foreground colours and dimensions defined by the user configuration. *KAESTLE & FooScape* visualize lists by drawing their elements into squares, putting one after another or connecting them with arcs as needed. It allows modifying the layout of each list visualized and its contents.

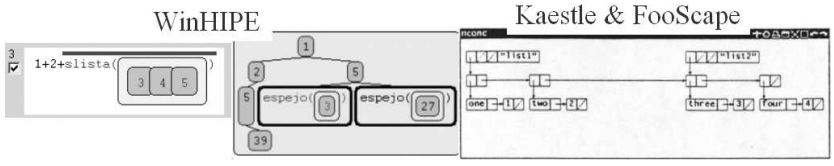


Figure 8: Alternative representations for complex data structures

4.4 Subexpression Sharing in Lazy Evaluation

Systems supporting lazy evaluation should visualize shared subexpressions. *CIDER* and *Hat* do not visualize shared subexpressions until they are reduced, then they highlight all occurences of the shared subexpression. *Prospero* and *Hint* (see Fig. 9) only visualize once a shared subexpression, being connected the rest of occurrences to the first one by arcs or labels.

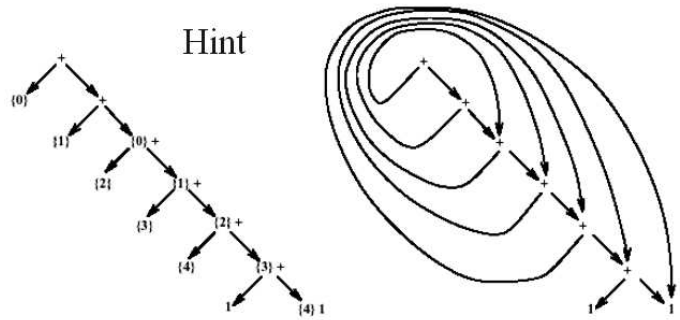


Figure 9: Visualization of shared subexpressions in lazy evaluation

4.5 Combining Partial Visualizations

Some systems combine some of the previous partial visualizations. While *Hood & GHood* are able to show values of variables and results of function calls, and *KAESTLE & FooScape* display a function call graph and current state of lists in the program, the rest of systems tend to blur the separation of code and data in functional programs. *WinHIPE*, *DrScheme* and *Visual Miranda* display values and data structures integrated into expressions. *Evaltrace* does it too but in a different way, by integrating values of variables into a pretty-printed textual description of the execution of the program. In addition *DrScheme* and *Evaltrace* show the contour of variables, and *Visual Miranda* is able to display the full pattern matching process.

5 Systems Evaluation

We have only found two documented experimental evaluations of systems. The first (Chitil et al., 2001) is a comparative study of three systems: *Freja*, *Hat* and *Hood*. The study is focused on their tracing and debugging facilities. A number of criteria are evaluated for each system: readability of expressions, the process of locating an error, redexes and language constructs, and modification of the program. This study identifies strengths and weakness of each system and then suggest how the systems can be improved.

The second documented evaluation (Medina-Sánchez et al., 2004) is of the *WinHIPE* environment. This evaluation is focused on effortlessness and usability of animations and their construction process. The experiment was done with students and its results show that animations are easy to use, its construction process is easy to learn, and are understood as a help to complete other tasks, such as debugging or program understanding.

There are two more evaluations but they are documented in a rather informal way. In the section 6 of Findler (2002) some experience with *DrScheme* is briefly described and in section 5 of Reinke (2001) some details are shown about experience with *GHood*.

6 Conclusions

We have given a survey of visualizations of functional programs provided by sixteen different systems. In order to make the exposition more meaningful, we have given a two dimension classification. On the one hand, we have classified the systems into four categories (programming environments, debuggers, teaching systems, and visualization systems). On the other

hand, we have considered the most important partial views provided about functional programs (expression evaluation, function calls, values, and subexpression sharing). Our selection of these dimensions has been pragmatical: we do not pretend these dimensions to be the most important ones, but we found them especially clarifying to us. In Price et al.'s taxonomy, the corresponding categories are program (B.1) and purpose (F.1).

In spite of this variety of visualizations, none provides comprehensive visualizations, with multiple views of all the aspects. There are several systems covering several features of functional programming, more comprehensive visualizations are still lacking. We advocate for a comprehensive approach that would make use of solutions given by current systems. A comprehensive approach could offer current partial views, but it would also offer more powerful and flexible visualizations on the different features of functional programs.

Such a comprehensive visualization still has to be designed. However, notice that the first identified feature, namely expression evaluation, is the basic element of the functional paradigm. Consequently, it should be the basis if the new visualization. Two other features (function calls and values) are partial views that mimic our understanding of program execution derived from the imperative paradigm. Therefore, they should be integrated in the expression model. Finally, subexpression sharing is a particular and important aspect of lazy evaluation.

Acknowledgements

This work has been supported by projects GCO-2003-11 of the Universidad Rey Juan Carlos and TIN2004-07568 of the Ministerio de Educación y Ciencia.

References

M. Auguston and J. Reinfelds. A Visual Miranda Machine. In *Proceedings of the Software Education Conference (SRIT-ET'94)*, pages 198–203. IEEE Computer Society Press, 1994.

R. Berghammer and U. Milanese. Kiel - a computer system for visualizing the execution of functional programs. In *Functional and (Constraint) Logic Programming, WFLP 2001*, pages 365–368, Christian-Albrechts-Universitt zu Kiel, 2001. Report No. 2017.

H.D. Boecker, G. Fisher, and H. Nieper. The enhancement of understanding through visual representations. In *Proceedings of the ACM SIGCHI'86 Conference on Human Factors in Computing*, pages 44–50. ACM Press, 1986.

M.H. Brown. A taxonomy of algorithm animation displays. In J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price, editors, *Software Visualization. Programming as a Multimedia Experience*, pages 35–42. MIT Press, 1998.

O. Chitil, C. Ruciman, and M. Wallace. Freja, Hat and Hood - A comparative evaluation of three systems for tracing and debugging lazy functional programs. In *Implementation of Functional Languages, 12th International Workshop, IFL 2000, Selected Papers*, volume 2011 of *LNCS*, pages 176–193. Springer, 2001.

R.B. Findler. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

S.P. Foubister and C. Runciman. Techniques for simplifying the visualization of graph reduction. In K. Hammond, D.N. Turner, and P.M. Sansom, editors, *Functional Programming*, pages 65–77. Springer, 1995.

A. Gill. Debugging Haskell by observing intermediate data structures. In G. Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41 of *ENTCS*. Elsevier, 2000.

M. Hanus and J. Koj. Cider: An integrated development environment for Curry. In *Functional and (Constraint) Logic Programming, WFPL 2001*, pages 369–373, Christian-Albrechts-Universitt zu Kiel, 2001. Report No. 2017.

N. Kawaguchi, T. Sakabe, and Y. Inagaki. Terse: Term rewriting support environment. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Standard ML and its Applications*, pages 91–100. ACM Press, 1994.

H. Lieberman and C. Fry. ZStep95: A reversible, animated source code stepper. In J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization. Programming as a Multimedia Experience*, pages 277–292. MIT Press, 1998.

M.Á. Medina-Sánchez, C.A. Lázaro-Carrascosa, C. Pareja-Flores, J. Urquiza-Fuentes, and J.Á. Velázquez Iturbide. Empirical evaluation of usability of animations in a functional programming environment. Technical report, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Madrid, Spain, 2004. Ref. 141/04.

B. Myers. Visual programming, programming by example, and program visualization: A taxonomy. In *Proceeding of the ACM SIGCHI'86 Conference on Human Factors on Computing Systems*, pages 59–66. ACM Press, 1986.

F. Naharro-Berrocal, C. Pareja-Flores, J. Urquiza-Fuentes, J.Á. Velázquez-Iturbide, and F. Gortázar-Bellas. Redesigning the animation capabilities of a functional programming environment under an educational framework. In M. Ben-Ari, editor, *Proceedings of the Second Program Visualization Workshop*, pages 60–69, University of Aarhus, Department of Computer Science, 2002. DAIMI PB - 567.

H. Nilsson and J. Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4(2):121–150, April 1997.

B. Pope. *Buddha - A Declarative Debbuger for Haskell.* PhD thesis, Department of Computer Science, The University of Melbourne, Australia, June 1998.

B.A. Price, R. Baecker, and I. Small. An introduction to software visualization. In J. T. Stasko, J. Domingue, M. H. Brown, and B.A. Price, editors, *Software Visualization. Programming as a Multimedia Experience*, pages 3–27. MIT Press, 1998.

B.A. Price, R.M. Baecker, and I.S. Small. A principled taxonomy of software visualisation. *Journal of Visual Languages and Computing*, 4(3):211–266, September 1993.

C. Reinke. GHood: Graphical visualisation and animation of Haskell object observations. In R. Hinze, editor, *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, volume 59 of *ENTCS*, pages 121–149. Elsevier Science, 2001.

J. Sparud and C. Runciman. Tracing lazy functionals computations using redex trails. In H. Glasser, P. Hartel, and H. Kuchen, editors, *Proc. 9th Intl. Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, volume 1292 of *LNCS*, pages 291–308. Springer, 1997.

J.P. Taylor. *Presenting the evaluation of lazy functions.* PhD thesis, Department of Computer Science, Queen Mary University of London and Westfield College, London, UK, 1995.

D.S. Touretzky and P. Lee. Visualizing evaluation in applicative languages. *Communications of the ACM*, 35(10):49–59, October 1992.

Enhanced Expressiveness in Scripting Using AnimalScript 2

Guido Rößling, Felix Gliesche, Thomas Jajeh, Thomas Widjaja
Department of Computer Science / Dept. of Business Administration and Computer Science
Darmstadt University of Technology, Darmstadt, Germany

{guido, gliesche, jajeh, widjaja}@rbg.informatik.tu-darmstadt.de

Abstract

AnimalScript 2 is a new implementation of the visualization language AnimalScript used in the Animal system. The new implementation adds important features, especially conditional and loop statements. It also prepares the ground for further advanced components, such as methods or object templates. Several examples illustrate the expressiveness and ease of use of AnimalScript 2.

1 Introduction

One of the many different approaches for generating algorithm or program visualization content (abbreviated "AV" for the rest of this paper) is *scripting*. Here, the user provides a simple ASCII file containing commands that steer the visualization. The commands are usually held in plain English to make using the underlying scripting language easier. Typical examples for scripting-driven AV systems include JAWAA (Akingbade et al., 2003), JSamba (Stasko, 1998), and the *JHAVÉ* visualization environment with its visualization front-ends GAIGS, JSamba (Naps et al., 2000), and Animal (Rößling and Freisleben, 2002).

Scripting files are normally very easy to create manually. The user requires only a text editor and a certain familiarity with the scripting notation to become productive. Even better, it is relatively easy to modify existing code so that it generates scripting commands for visualization purposes while running the underlying program. Generating *some* working scripting code is normally rather easy. Writing a scripting code that presents a "good" visualization is more difficult. However, the same is true for *any* AV system that allows or forces the user to explicitly layout the visual components.

In this paper, we focus on the added capabilities to the original version of the scripting language AnimalScript provided by the Animal system (Rößling and Freisleben, 2001). To avoid confusion, we will always refer to the new implementation as AnimalScript 2, and use AnimalScript for the original implementation. We first review the main features of interest in the original scripting language and motivate why a new implementation was needed. The added features are then described in detail. The paper concludes with a short overview of the current implementation status and the goals we have set for the final version.

2 A Quick Overview of AnimalScript

Each AnimalScript animation consists of a single file with a set of lines. Each line can contain exactly one command or comment. To make parsing the files easier, each operation starts with a unique keyword. The parser can therefore determine the appropriate action by parsing the first keyword, although later parameters usually determine the actual action taken.

AnimalScript is parsed line-by-line. This means that once a given line is parsed, the appropriate animation commands are added to an Animal animation. Normally, each operation – whether declaration of a new object or animation effect – takes place in a separate animation step. If multiple operations shall take place in the same animation step, the user has to surround them with curly braces { } to indicate a block. Similarly to programming languages, the animation treats this block as a unit placed in the same step.

AnimalScript comes with built-in support for the graphical primitives `point`, `polyline` / `polygon`, `text` and `arc`. There are also specific commands for generating subtypes, such as

`squares, lines,` or `circles`. To enhance the use of ANIMAL for computer science education, the following common complex objects are also supported: `list elements` with an arbitrary number of points, `arrays` in either horizontal or vertical orientation, and `source / pseudo code` including indentation and highlighting.

Most commands have a set of optional parameters for setting specific properties. This includes simple settings, such as color or display depth. Arrows may be added at the beginning or end of a polyline. The user can also switch between polylines and polygons using the boolean *closed* property, using *closed=true* for polygons, and *closed=false* for polyline objects.

ANIMAL offers only a small selection of animation effects at first glance, limiting the operations to `show / hide, move, rotate` and `change color`. Each animation effect can work on an arbitrary set of animation objects at the same time. The expressive power of the scripting language becomes obvious when the set of options for the commands is reviewed. For example, a `move` can be made `to` a certain location, `along` an object defined inside the command, or `via` a previously defined object. The latter supports easy reuse of common move paths inside an animation, for example for sorting problems.

To further enhance the expressiveness of the scripting language, each object type can offer specific subtypes of a given animation effect. These are passed as an optional parameter to the standard animation effect (Rößling, 2001). For example, a polygon may offer the user the following `move` types:

- move the whole object,
- move a single node,
- move an arbitrary set of nodes,
- move the whole object except for a single node,
- move the whole object except for an arbitrary subset of nodes.

In this way, it is very easy to reach rather complex behavior based on a still simple notation. To further support animation authors, the computer science-based primitives also have their own set of commands. This especially concerns the following operations:

- Generating a group of source or pseudo code with user-specified font and color settings. As an exception to the general rule, each code line or line fragment is added as a separate component. This avoids exceedingly cluttered scripting code with several hundred characters in one line, and thus makes the script far easier to read;
- highlighting or unhighlighting a single line of code or a fragment thereof - for example, the boolean condition of a `for` loop;
- generating an array with user-defined font and color settings, either in horizontal or vertical orientation;
- installing an "array index pointer" with an optional label, useful for example to indicate an array position in sorting algorithms;
- putting values into the array and swapping array elements. The latter operation is animated automatically if a positive effect duration is specified;
- creating list elements with an arbitrary number of pointers at either the top, bottom, left or right side;
- resetting or setting a given list pointer. Here, the user can specify either a position or a target object. ANIMAL then figures out the appropriate way to handle the pointer based on the relative positions of the two objects.

Each AnimalScript object has a unique ID. Once the current line is parsed, the animation author can retrieve the current *bounding box* of the defined object, yielding the smallest rectangle that covers the whole object.

All AnimalScript coordinates can be specified in a number of different ways:

absolute coordinates give an explicit pair of (x, y) coordinates on the screen. To yield a visible object, x and y should be positive and within the display window borders;

locations can be defined once and reused as often as necessary;

relative coordinates are the most expressive and powerful option. Here, the location of a given object is determined based on other visible or hidden objects. Typically, the position is determined based on the *bounding box* of a given object by giving one of the eight compass directions or "center" and an (x, y) offset. Polyline or polygon objects also allow placement relative to a given *node*. Components can also be aligned to the *base line* of a text component. Finally, the location can also be defined as an offset from the `previous` coordinate.

A special `echo` command can be used for user feedback. Apart from simply printing a certain text to the command line or main window, the actual bounding box of a given object or set of objects can be retrieved, as well as individual objects and their IDs. In this way, if the layout on the screen does not match the author's expectations, some debug commands using `echo` can be integrated to figure out exactly *what* went wrong. Finally, objects can be grouped or ungrouped to save repeating the objects IDs for objects that are animated in the same way over several operations. It is important to note that a component inside a group can still be animated individually, *without* effect on the other group elements.

The syntax of AnimalScript, JAWAA and JSamba is roughly similar. AnimalScript uses String identifiers for objects instead of integers. Additionally, AnimalScript offers a fine-grained timing, compared to the "instant" or "animated" modes in JAWAA and JSamba. AnimalScript also boasts a far greater flexibility in animating and placing components, as outlined above.

In both JAWAA and JSamba, all parameters have to be given for all commands without an introducing keyword. Thus, commands start with a descriptive keyword, followed by a set of seemingly arbitrary values, typically of type integer. AnimalScript strictly requires keywords between most parameters, such as `color` or `depth`. At the same time, most parameters including their associated keyword are optional. This combination makes the underlying script easier to read, but also somewhat longer, than scripts for JAWAA or JSamba.

As can be seen from this overview, AnimalScript is rather powerful and expressive. However, there is one crucial drawback. As stated before, each line is parsed separately, as the context of the previous lines is retrieved from the Animal-internal animation object. Thus, many of the standard parsing concepts, such as *abstract syntax trees*, are not needed to parse AnimalScript animations. To make the implementation easier and more efficient, we took the ultimately unfortunate implementation decision to stay at a "single line parser". This brings one severe limitation: interesting components such as *loops* or *conditionals* can not be supported by the original AnimalScript parser.

To address this problem, we decided to re-implement the whole parser from scratch. This was also a good opportunity to clean up some the messier parts of the source. Compared to the former implementation with about 7500 lines of code, a team of three students of Business Administration and Computer Science was formed for this task.

3 Added Features in AnimalScript2

AnimalScript 2 is downward compatible to AnimalScript. That is, all commands which worked in AnimalScript will ultimately work in its successor. "Ultimately", because the

extent of the scripting language means that we had to restrict the amount of work we could tackle at one time.

The main goal of developing AnimalScript 2 is changing the line-based parsing approach to one based on abstract syntax trees. Apart from allowing components such as loops and conditionals, this will ultimately allow us to support method invocations. Currently, the most striking additions are the `while` and `for` loops and the `if` conditional with an optional `else` part. The additional `loop` construct iterates the loop body for a number of repetitions specified as an arithmetic expression. Apart from its use as a "shorthand notation", the `loop` construct is also helpful for beginners in programming. The trinary conditional operator `(booleanExpression) ? expression : expression` is not supported. The syntax for the entities follows the syntax used in Java and is shown in Listing 1. `intVarDecl` in the `for` construct stands for the initialization or declaration of a variable.

```
while (booleanExpression) #execute as long as expression is true
{
  command
}

for (intVarDecl; booleanExpression; arithmeticExpression) # as in Java
{
  command
}

loop (arithmeticExpression) # iterate exactly "expression" times
{
  command
}

if (booleanExpression) {
  command
}

if (booleanExpression) {
  command
}
else {
  command
}
```

Listing 1: Loops and conditional constructs

The body of each loop may also contain sub-blocks, just as in a "real" programming language. As shown in Listing 1, the curly braces can appear either on a new line or at the end of the current line. In contrast to C, C++ and Java, the curly braces *must* appear, even if the command body consists of only one command. Users familiar with a C-like syntax should find it easy to learn and effectively use the notation.

To support the loops appropriately, AnimalScript 2 introduces commands for handling arithmetic, boolean and String-based expressions. Arithmetic expressions currently cover the base operators `+, -, *, /` and `%` (modulo). This includes the precedence of multiplication and division over addition and subtraction, as well as parentheses.

AnimalScript 2 also supports integer variables, defined in the same way as in Java: `int nrIterations = 10`. Note that a semicolon can optionally be used to be even closer to the

notation employed in Java, C and C++. Integer variables can be assigned arbitrary (integer) expressions using the assignment operator, e.g. `nrIterations = 5 * i`.

Boolean variables are defined as in Java. They can be assigned either one of the two literals `true / false`, another boolean variable or an arithmetic expression with C semantics (0 is `false`, all other values are `true`). The boolean operators cover conjunction && and disjunction ||, the boolean comparison operators `==` and `!=`, and integer comparisons yielding a boolean result (using $<, <=, ==, >=, >$, and $!=$).

AnimalScript 2 also offers String variables, declared as `string myString = "Hello"` and assigned a new value in the usual way. Strings can be concatenated using the *Perl/PHP*-notation with a point in the middle. Thus, `myString . " world"` yields the String `Hello world`. The concatenation works on String, boolean and integer variables and literal Strings.

Due to the way String variables are expanded, even the names of variables can be generated dynamically. This is mainly useful in loops that generate individual objects with a unique variable name, for example `a1, a2, a3, ...`. A similar effect can be achieved in some scripting languages for programming, notable PHP and Perl, with operators such as `$$` (Lerdorf and Tatroe, 2002).

4 Example Use of AnimalScript 2

The source code shown in Listing 2 swaps the first element of an array with the minimum array value. It therefore constitutes a part of the *Straight Selection* sorting algorithm. We assume the presence of a method `swap` that can swap two elements on a given array.

```
int[] values = new int[]{3, 2, 4, 1, 7};
int pos = 1;
int minIndex = 0;
while (pos < values.length) {
  if (values[pos] < values[minIndex])
    minIndex = pos;
  pos++;
}
swap(values, 0, min);
```

Listing 2: Java code for swapping the minimal array element with the first array element

The original AnimalScript does not provide any loop support. Therefore, the structure has to be "flattened", resulting in something like Listing 3.

```
%Animal 1.4
array "values" (10, 10) length 5 "3" "2" "4" "1" "7"
arrayMarker "pos" on "values" atIndex 1 label "pos"
arrayMarker "minIndex" on "values" atIndex 0 label "minIndex"
moveMarker "minIndex" to position 1 within 5 ticks
moveMarker "pos" to position 2 within 5 ticks
moveMarker "pos" to position 3 within 5 ticks
moveMarker "minIndex" to position 3 within 5 ticks
moveMarker "pos" to position 4 within 5 ticks
moveMarker "pos" to outside within 5 ticks
arraySwap on "values" position 0 with 3 within 10 ticks
```

Listing 3: Example animation code in the original AnimalScript

The script in Listing 3 is easy to read but very hard to understand, as the semantics of the array operations are hidden. In essence, the reader has to build his own hypothesis why the markers change, and what the actual underlying algorithm is. Listing 4 shows the same code, implemented in AnimalScript 2.

```
%Animal 2.0
array "values" (10, 10) length 5 int {3, 2, 4, 1, 7}
int pos = 1
int minIndex = 0
arrayMarker "pos" on "values" atIndex pos label "pos"
arrayMarker "minIndex" on "values" atIndex minIndex label "minIndex"
while (pos < 5) {
  if (values[pos] < values[minIndex]) {
    minIndex = pos;
    moveMarker "minIndex" to position pos within 5 ticks
  }
  pos = pos + 1
  moveMarker "pos" to position pos within 5 ticks
}
arraySwap on "values" position 0 with minIndex within 10 ticks
```

Listing 4: Example animation code in AnimalScript 2

At first glance, the code shown in Listing 4 is longer than the code in Listing 3 (15 lines versus 11 lines of code). It is easy to see that this depends on the actual array: the number of code lines are fixed for both the Java and the AnimalScript 2 listing. For AnimalScript, the number of code lines depends on the array length and the ordering of the elements.

There is a strong similarity between the Java code in Listing 2 and AnimalScript 2. The script contains twelve lines of effective AnimalScript 2 code, if we ignore lines 1, 11 and 14. Six lines of assignments, conditional and loop are identical or almost identical. The mapping from the Java array declaration to AnimalScript 2 is also easy. The main changes concern the commands for installing visible array position markers and moving them in concert with the value assignments. The complete animation code for *Selection Sort* in AnimalScript 2 is shown in Listing 5. The equivalent AnimalScript notation contains 34 lines of code.

```
%Animal 2.0
array "values" (10, 10) length 5 int {3, 2, 4, 1, 7}
int pos = 0
int spos = 0
int minIndex = 0
arrayMarker "pos" on "values" atIndex pos label "pos"
arrayMarker "spos" on "values" atIndex spos label "pos"
arrayMarker "minIndex" on "values" atIndex minIndex label "minIndex"
while (pos < 4) {
  minIndex = pos
  for (spos = pos + 1; spos < 5; spos = spos + 1) {
    if (values[spos] < values[minIndex]) {
      minIndex = spos;
      moveMarker "minIndex" to position spos within 5 ticks
    }
  }
  arraySwap on "values" position pos with minIndex within 10 ticks
  pos = pos + 1
  moveMarker "pos" to position pos within 5 ticks
}
```

Listing 5: Example selection sort animation code in AnimalScript 2

As can be seen, the new features are very helpful for array-based algorithms, such as sorting or searching. They significantly reduce the cognitive effort of coding "programs" into visualizations. Loops and conditional can of course also be used for other programs.

Compared to the original implementation of AnimalScript, the new implementation can offers significant run-time performance advantages. This is especially true for programs that use the new conditional or loop statements. For example, the implementation of *Selection Sort* shown in Listing 5 only has to parse 20 lines from the file. The iterative version implemented for AnimalScript has to parse 34 lines of code. Any reduction of file I/O, especially concerning parsing operations, can greatly improve the run-time, even when buffered streams are employed.

Additionally, the look-up mechanism for object position determination in AnimalScript 2 is faster than in the original AnimalScript. In the original version, the animation had to be fast-forwarded to the "current point in time" to accurately determine the bounding box of a given element. Relative placement commands used in defining new objects or within animation effects were therefore very time-consuming to evaluate. In the new version, the look-up is significantly faster thanks to the (hidden) tree structure used for storing and evaluating the animation.

At the moment, we cannot provide conclusive run-time measurements, as the implementation of the array operation visualizations has not been fully implemented. We plan to do a more extensive evaluation once the implementation of the parsing and execution are completed.

5 Summary and Further Work

AnimalScript 2 is a re-implementation of the scripting language AnimalScript (Rößling and Freisleben, 2001). The previous line-based parser is replaced by a parse tree. The main change that is visible to users are the important base operations for simplifying animation creation: loops, conditionals, variables and expressions.

The additions considerably increase the expressiveness of AnimalScript, pushing it closer to a full-fledged programming language with visualization. This simplifies manual generation, for example of sorting algorithms. We plan to evaluate the effects on (semi-)automatic generation once the implementation is finished. As the user is not required to use the new commands, AnimalScript 2 is at least not "more difficult" to learn and use than the original release.

All new components can be parsed and evaluated. Some of the older (and not very well documented) advanced features of the original scripting language are missing and placed on hold for more important content. This includes importing several scripting files into a single animation and internationalization aspects. Additionally, the support for pre-defined *locations* is still under development.

The team is currently working on getting all object generation commands set up. While this task is per se relatively simple, the size of the Animal system with 216 classes and about 45000 lines of code has to be taken into account. Becoming familiar with all components and their interplay is hardly trivial, as can be seen when studying the reference work (Rößling, 2002). Currently, all additional features can be parsed, evaluated and executed, apart from the occasional bugs to be expected in any significant software project.

Due to the complete redesign and reimplementation of the parsing process, the new version of the scripting language is ready for other advanced extensions. This includes method definitions and blocks that define author-specific objects based on a set of primitives. Due to the size of the implementation team and the other demands on their time, not all goals are realistic - this is only a one-year project without payment!

Note that Animal itself still offers the graphical drag-and-drop user interface. Thus, Animal still supports beginners, but has extended its support for "expert" programmers using AnimalScript 2.

Once the implementation is finished, the new release of AnimalScript 2 will be available online under `http://www.animal.ahrgr.de`. A set of examples will also be available there, as there was just too little space in this paper for more.

References

Ayonike Akingbade, Thomas Finley, Diana Jackson, Pretesh Patel, and Susan H. Rodger. JAWAA: Easy Web-Based Animation from CS 0 to Advanced CS Courses. In *Proceedings of the* 34^{th} *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2003), Reno, Nevada*, pages 162–166. ACM Press, New York, 2003.

Rasmus Lerdorf and Kevin Tatroe. *Programming PHP*. O'Reilly & Associates, Sebastopol, CA, 2002. ISBN 1-56592-610-2.

Thomas Naps, James Eagan, and Laura Norton. JHAVÉ: An Environment to Actively Engage Students in Web-based Algorithm Visualizations. *Proceedings of the* 31^{st} *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 109–113, March 2000.

Guido Rößling. Algorithm Animation Repository. Available online at `http://www.animal.ahrgr.de/` (seen August 14, 2004), 2001.

Guido Rößling. Animal-Farm*: An Extensible Framework for Algorithm Visualization*. PhD thesis, University of Siegen, Germany, 2002. Available online at `http://www.ub.uni-siegen.de/epub/diss/roessling.htm`.

Guido Rößling and Bernd Freisleben. AnimalScript: An Extensible Scripting Language for Algorithm Animation. *Proceedings of the* 32^{nd} *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2001), Charlotte, North Carolina*, pages 70–74, February 2001.

Guido Rößling and Bernd Freisleben. Animal: A System for Supporting Multiple Roles in Algorithm Animation. *Journal of Visual Languages and Computing*, 13(2):341–354, 2002.

John Stasko. Smooth Continuous Animation for Portraying Algorithms and Processes. In John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors, *Software Visualization*, chapter 8, pages 103–118. MIT Press, 1998.

THORR: A Focus + Context Method for Visualising Large Software Systems

Eoin McCarthy, Chris Exton
SVCR Group
Department of Computer Science and Information Systems, University of Limerick

Eoin.D.McCarthy@ul.ie, Chris.Exton@ul.ie

Abstract

Many attempts have been made to construct tools that aid a programmers understanding of source code and system structure. Syntactic highlighting and tabbed interfaces can only be useful to a degree; greater steps need to be taken to accelerate programmer comprehension and in reducing programming and maintenance times. This paper presents a new programming environment for software engineers, THORR. The programming environment hopes to overcome some of the shortcomings currently associated with similar visualisation projects, in terms of balance between level of detail and context, by using 'Degree-of-Interest' information visualisation techniques to implement an attention reactive user interface. An overview of the prototype tool and its basic functionality is also given.

1 Introduction

Large scale software systems, like production-sized legacy programs can be incredibly difficult to maintain or update. Much of the time expended in performing these tasks is taken up by refreshing a programmer's knowledge of the system or training a new engineer into understanding the system(Knight, 2001). Under these circumstances, a programmer's productivity can decrease as the work can be laborious and tiresome. As a result errors can occur or projects can be late.

Software visualisation was brought about as a means of easing the maintenance section of the software lifecycle. It is believed that by visualising a software system graphically, be it program or algorithm animation, the knowledge decay that programmers experience can be slowed, and the length of time it takes to remember or discover code can be reduced.(Ball and Eick, 1996), (Grundy and Hosking, 2000).

During the design phase of software development, programmers will use diagrams to illustrate the software design in the form of UML diagrams, using industrially recognised tools such as Rational Rose, and Select Enterprise. However, it is difficult to find equally prominent tools that aid programmer understanding of an implemented system. One of the reasons for this is maybe that many visualisation systems are tested using small software examples and as a result may not scale well to industrial-sized software systems.

In this paper, we present a new software visualisation tool for software engineers called THORR, in the hope of reducing programmer effort, and shortening maintenance times. We will attempt to overcome the problems of scalability, and navigating large data sets that have prevented other visualisations from being widely accepted.

Section 2 will present the methods of visualisation that THORR utilises, describing the innovative animation techniques they have employed. Section 3 describes the architecture of the visualisation tool we have created based on the DOI technique, how it generates and processes valuable architectural information. Section 4 identifies particular needs that user may have when using a visualisation system and discussed how THORR aids those needs. The fifth section deals with the implementation of the tool including some metaphors and methods used in developing and presenting the visualisation. Section 6 discusses an evaluation performed on the tool, and section 7 provides some concluding remarks.

1.1 Software Visualisation

While it has been described in many different ways for the purpose of this paper software visualisation will be defined as:

> *"...the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both human understanding and effective use of computer software."*(B Price, 1992).

2 Visualisation Techniques

2.1 Fish-Eye Lens

The Fish-Eye lens distortion technique was original described by Furnas (Furnas, 1986). It was based on a technique best described as 'thresholding'. It was primarily used to visualise hierarchical structures, in which each element was assigned 2 numbers. The first number was based on its relevance and the second number was based on the distance between the information element in question and the element regarded to be the focused node. A threshold value was then selected that was compared with a function of these two values to determine which information would be presented and suppressed.

2.2 Focus + Context

One of the biggest difficulties in visualizing large quantities of information is the lack of screen space in which to visualize it. Users can become disoriented or even 'lost' in a visualisation where complex navigation is required to obtain information.

Focus + context techniques were introduced as means of displaying huge quantities of information on one screen. A distortion algorithm is used to magnify a certain part of the screen while de-emphasizing the rest. As a result, the user can have a constant overview of the information while focusing in on one particular area.

Card et al(Card and Nation, 2002) use the terminology *focus + context* when referring to visual techniques that provide simultaneous access to both overview and detailed information. According to Card, *focus + context* is based on the following three premises:

- The user requires both the overall picture and a detailed view.
- There may be different requirements for the information in the detailed view than in the overall picture.
- Both displays may be combined into a single dynamic view.

2.3 Degree-Of-Interest Animation

Developed at XEROX-PARC(Card and Nation, 2002) the DOI (degree-of-interest) tree is an example of an attention reactive interface, used for visualising hierarchical information. The *focus + context* method, which it utilises, differentiates it from other similar visualisations. While other *focus + context* visualisations apply distortion algorithms to the screen itself, fisheye views and perspective walls for example, the DOI tree applies transformations and magnifications to the tree to illustrate interest in specific nodes.

The researchers at XEROX PARC expanded on Furnas's work. In Furnas's graphs, all nodes were given the same intrinsic values, hence when one node is focused on; all other nodes are treated in the exact same manner. The DOI tree assigns different values to varying nodes depending on their relationship with the focused node. There calculation treats the children of the focused node as ordered and assigns a fractional degree-of-interest offset to the children, then a smaller fraction to the focused nodes' grandchildren and so on. Each time a new node is selected as the focused node, the DOI offsets for all nodes must be recalculated.

3 Support for User Needs

3.1 Scalability

The problem in creating an effective scalable visualisation tool is multi-faceted. Designers have to consider massive amounts of input to the tool; huge amounts of information processing and then must also have efficient algorithms, which display the information in an insightful manner. When these considerations are undertaken, then there is a chance for creating a truly scalable system.

There are many tools that have been implemented to show off one particular animation or visualisation technique, which they deem to be highly important(Robertson et al., 1998), (Sanjaniemi and Kuittinen, 2003) to name but a few. When these tools are evaluated, it is usually done with a sample set of miniature or 'toy' software systems. There is not enough emphasis placed on whether or not these tools will actually work in industry. They have not considered what will happen if the change the sample size from 200 LOC to 2million LOC, and more often than not unless the metaphor for visualisation has been design with huge programs in mind, it will not scale well.

The presentation of large amounts of information is often spread across an area that spans multiple screens, but THORR will attempt to visualise a meaningful overview of a system in a single display, which will result in a need for novel and complex layout algorithms.

3.2 Interaction

The difficulty in implementation of the interaction mechanism required in a visualisation is directly proportional to the amount of information it is required to visualise. When dealing with small sample sets, very simple interaction techniques will be adequate both in terms of user-interface and navigation of visualisation. It is difficult for a user to get confused when dealing with small amounts of information and as a result even badly designed interaction techniques could be sufficient.

When the amount of information increases to a very large number of information elements, there must be more advanced techniques put in place. Users navigation must be stringently designed and aid the user, by only constraining movement to what is required. When dealing with huge graphs, complete freedom of movement can result in major difficulties. (Cockburn and McKenzie, 2000), (Wood et al., 1995)

THORR will be designed with all of these interaction difficulties in mind. It must allow users to retain overall context at all times. It must also give users the ability to move from high-level information, which has very little detailed information, to lower level more comprehensive information with ease. The most successful interaction mechanisms constrain its users in some way, so THORR will be designed to limit the movements that can be carried out by the users. THORR will also implement a 3D interaction technique, which when combined with the other interaction requirements will result in complex implementation but will enhance the users visualisation experience.

3.3 Interoperability

For a tool to be truly accepted in industry it must allow for interfacing with other tools that software maintainers have at their disposal, whether they be other visualisation, design, or re-engineering tools. It must be as open as possible, in terms of what platforms it can run on, what languages it can accept as input, and how extensible it is. With so many tools only in prototypical stages, it is difficult to find ones that accept multiple languages as input, as they are generally focused on proving the concept of their own specific visualisation technique. These 'toy tools' also create their own informal definition of their visualisation layout resulting in an output formation that is only useful to that specific tool.

A common interchange format would be useful to allow interoperability between tools. Such as interchange would act as a 'buffer' or 'interface' between different tools and it would make it easier to judge the benefits of individual tools. Identical information sources could be used when performing comparisons between tools, resulting in better overall qualitative results. o input or export formats have been described.

The THORR tool will attempt to emulate some of the better design decisions that other visualisation systems have implemented. It will implement an input system, which will accept a common interchange. This results in a tool that will be language independent and also encourage interoperability with tools that accept similar formats. It will be designed to be somewhat extensible and also be implement in a language that is not operating system specific.

3.4 Automation

The amount of automation required in visualisation systems is inversely proportional to the simplicity of the interaction controls used in the system. In other words, the easier it is to traverse large amounts of information in a tool, the harder it should be to design and implement to the navigational controls. THORR will be required to automate some if not all of the more time-consuming visualisation tasks. It should be able to automatically set-up projects, and extract meaningful information from software systems without requiring an overt amount of user input. The visualisation should assist the user's cognitive processes by implementing a navigational system that will allow the user quick and easy access to all nodes in the graph, and it should be able to automatically decide which nodes to display in detail and which nodes should be filtered. There is, however, a fine line between helping users through the automation of tasks and hindering users by applying excessive constraints, so a balance between the two must be struck.

4 THORR Visualisation

Studies have shown that up to 70% of lifecycle costs are consumed within the maintenance phase, with 50% of these costs relating to comprehension alone (de Lucia and Fasolino, 1996),(Rajlich, 1994). Hence, according to these references, up to 35% of the total lifecycle costs can be directly associated with simply understanding the code.

The overall aim of the THORR visualisation then, is to decrease both the time and effort expended by programmers in trying to understand software systems. Visualisation systems have shown results in improving the speed at which users understand, remember, and find information.(Robertson et al., 1998),(Robertson et al., 1991). *Focus + context* methods have been proven increase the speed of use of visualisation systems by retaining overall context(Mitchell and Cockburn, 2003). We hope that by integrating these attributes, users will be able to quickly navigate through large-quantities of information extracted from software systems.

The visualisation itself is a high-level hierarchical visualisation employing degree-of-interest animation. The graph consists of a hierarchical tree of coloured cubes corresponding to various components of the software system, such as classes, superclasses, subclasses and interfaces. Edges between nodes correspond to inheritance parent-child relationships.

Its major advantage over similar tools is the fact that it can visualise thousands of nodes on one screen. It can be navigated without ever having to zoom in or out on a particular section of the tree, thus retaining context at all times. It also circumvents the need for scrolling as all the information is displayed on the one screen.

4.1 Degree-of-Interest Computation

When the tree is first laid out, each node is assigned a DOI value depending on its relationship to the root node.

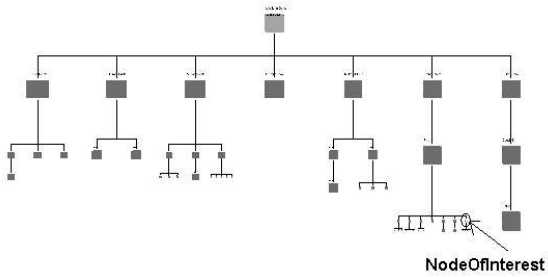


Figure 1: THORR graph in intitial state

When a user selects a node to be *interested* in the DOI for each node is then calculated according to the relationship it has with the focused node. All DOI values are between 1 and 0, where 1 is assigned to the focused node and 0 is assigned to the nodes that have little or no relationship with the focused node. To demonstrate this *focus + context* computation, consider the screen captures of the same graph taken from THORR at two different states in Figures 1 and 2. Figure 1 is a screen capture at the initial layout of the graph and Figure 2 captured the graph after*NodeOfInterest* has been selected by the user.

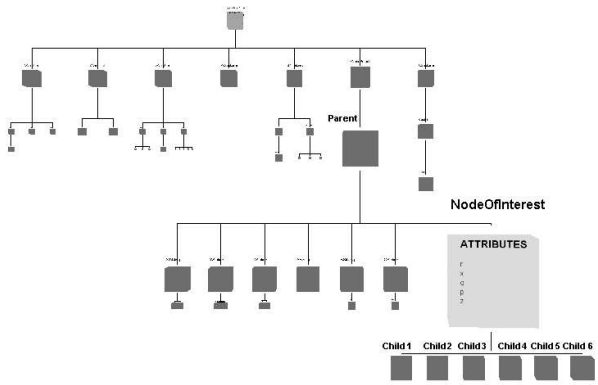


Figure 2: THORR graph in focussed state

When a user chooses a node to become the focused node, as in Figure 2, THORR firstly assigns that node a DOI of 1, then assigns a lesser offset to its children, and parents, then a smaller offset again to their respective children and parents and so on until the least related nodes have offsets of 0. Positions are then calculated, taking into account the changes of size that will occur, and a smooth interpolation between the two states is animated.

4.2 THORR Functionality

Text will always be an integral part of visualizing software, so the option of viewing the source code of individual methods has been included.A searching functionality has been incorporated into the tool to allow for even faster access to components of the visualisation. Searching the visualised information for specific or partial-phrases can yield in multiple results. The ability to capture both screen shots and the state of the graph for later analysis has also been integrated into the THORR visualisation. Simply clicking on the relevant screen capture will restore the graph to that state.

4.3 Node Characteristics

Each node is presented on screen in the form of a 3D cube. A user can interact with these cubes not only by double-clicking to expand them, but also by rotating them around the

Y-axis. This added ability allows information to be presented not solely on the front of the cube, but on 4 of the 6 sides.

This functionality is exploited by presenting different facets of Java classes. Constructors, operations and attributes were chosen as the relations to be presented on each of the sides. These divisions are similar to those implemented in the 'Javadocs' documentation tool of the Java API. Scrolling has been implemented to allow text to be presented in a sizeable manner, without reducing the amount of text that can be displayed on each side.

5 THORR Framework and Implementation

The THORR visualisation system focuses primarily on visualizing Java software systems and UML software designs in an efficient and comprehensible way.

5.1 Java

Java was selected as it is an excellent sample language to use in this prototype as it offers many different class relations such as inheritance, associations, inner-classes, super or sub-classes, local classes and packages, many of which are worth visualising. Java consists of a number of APIs, which are intended to be used or extended through inheritance with the aim of aiding productive programming by reducing programmer effort and increasing portability(Taivalsaari, 1996). While this ability to extend source code is useful, often to fully understand a class one must understand its superclasses and the classes it inherits from as well.

5.2 XMI

XMI(OMG, 1999) is the XML Metadata Interchange format standardized by W3C and the OMG. It is this standardisation that allows THORR to import both Java source code and UML diagrams. In the case of this tool, the information from the visualisation can be used in other tools and across platforms allowing for the combined use of many tools in a heterogeneous environment. A public domain tool, JavaRE(Andersson, 2001), analyses Java source and outputs information in the form of XMI documents.

6 Evaluation

An evaluation of Thorr was implemented in order to discern if the proposed methods of visualisation provide additional benefit to programmers. A simple tool A versus tool B evaluation was set up, between Thorr and the IDE Eclipse(Eclipse-Foundation, 2004). Eclipse was chosen because it is a widely used development environment when developing industrial sized Java software systems. It also has the ability to visualise inheritance hierarchies at a basic level and provides users with advanced search capabilities.

A pilot study was carried out to determine if changes to the evaluation design were required. There were a few small problems with the tool and the phrasing of the tasks that were identified and corrected for the actual evaluation.

The evaluation assigned 2 sets of tasks to 2 different groups. Group 1 carried out the Task Set A in Thorr and Task Set B in Eclipse, whereas Group 2 carried out Task Set B in Thorr and Task Set A in Eclipse. This swapping of task was employed as an attempt to negate dissimilarities in the programming experience between participants, the thinking being that participants that had difficulty performing task in Eclipse should also have some problems performing similar tasks in Thorr.

The tasks in each set were designed be exploratory and searching task that got more difficult as they proceeded through them. Participants were asked to locate certain classes, then attributes of classes, and also questions about the inheritance ancestry of classes. The length of time it took to finish each task was recorded and compiled.

The evaluation was carried out with 6 working professionals with varying levels of Java experience. 2 were classed themselves as being very experienced, whereas the other 4 described themselves as having intermediate knowledge of Java. The participants in each group had an even mix of average and experienced programmers. The results indicated that participants finished tasks on average 56% faster when using Thorr, and also finished more tasks correctly.

The users were also given a debriefing session in which they gave some qualitative information regarding the 2 tools. It was noted that there was a slight preference for using Thorr as an exploratory tool, as the participants preferred Thorr's navigation and interaction techniques. Some commented on its "..ease of use.." over Eclipse and stated that the "..layout was beneficial.." when exploring the graph.

7 Conclusions

Focus + context, and degree of interest techniques have proven effective in the field of information visualisation. The THORR visualisation tool aims to bring these helpful methods to the field of software maintenance. Its ability to visualise extremely large data sets, and also be able to focus on particular areas quickly and easily without losing context can be especially useful to software engineers. Further evaluation of the tool may help pinpoint what areas of the tool benefit users to most, in order to improve upon weaker areas of the tool and exploit its potential qualities.

References

Marcus Andersson. Javare - java roundtrip engineering, 2001. URL `http://javare.sourceforge.net/index.php`. Honours Reports of the University of Canterbury.

I S Small B Price, R M Baecker. A taxonomy of software visualisation. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume II, pages 597–606, 1992.

Thomas Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer*, 29 (4):33–43, 1996. URL `citeseer.ist.psu.edu/ball96software.html`.

Stuart K. Card and David Nation. Degree-of-interest trees: A component of an attention-reactive user interface. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, page 231246, 2002.

Andy Cockburn and Bruce McKenzie. An evaluation of cone trees, 2000. Department of Computer Science University of Canterbury Christchurch.

Andrea de Lucia and Anna Rita Fasolino. Understanding function behaviors through program slicing. In *Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*, page 9. IEEE Computer Society, 1996. ISBN 0-8186-7283-8.

Eclipse-Foundation. Eclipse ide, 2004. URL `http://www.eclipse.org`.

G.W. Furnas. Generalized fisheye views. In *Proceedings of CHI '86, New York*, pages 16–23. ACM, 1986.

John Grundy and John Hosking. High-level static and dynamic visualization of software architectures. In *Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)*, page 5. IEEE Computer Society, 2000. ISBN 0-7695-0840-5.

Claire Knight. System and software visualisation, 2001. URL `citeseer.ist.psu.edu/knight00system.html`. Handbook of Software Engineering and Knowledge Engineering.

David Mitchell and Andy Cockburn. Focus+context screens: A study and evaluation, 2003. Honours Report from the University of Canterbury.

OMG. Xmi - xml metadata interchange, 1999. URL `http://www.omg.org/technology/xml/index.htm`.

V. Rajlich. Program reading and comprehension. In *Proceedings of Summer School on Engineering of Existing Software*, pages 161–178, 1994.

G. Robertson, M. Czerwinski, K. Larson, D. Robbins, D. Thiel, and M. van Dantzich. Data mountain: Using spatial memory for document management. In *ACM Symposium on User Interface Software and Technology*, pages 153–162, 1998. URL `citeseer.ist.psu.edu/robertson98data.html`.

George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone trees: animated 3d visualizations of hierarchical information. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 189–194. ACM Press, 1991. ISBN 0-89791-383-3.

J. Sanjaniemi and M. Kuittinen. Program animation based on the roles of variables. In *In Processding of the ACM 2003 Symposium of Software Visualsiation (Softvis 2003)*, pages 7–16. ACM Press, 2003.

Antero Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996. ISSN 0360-0300.

A.M. Wood, N.S. Drew, R. Beale, and R.J. Hendley. Hyperspace: Web browsing with visualisation. In *In Third International World-Wide Web Conference Poster Proceedings*, pages 21–25, 1995.

MatrixPro – A Tool for On-The-Fly Demonstration of Data Structures and Algorithms

Ville Karavirta, Ari Korhonen, Lauri Malmi, and Kimmo Stålnacke
Helsinki University of Technology
Department of Computer Science and Engineering
Finland

{vkaravir, archie, lma, kstalnac}@cs.hut.fi

Abstract

In this paper, we introduce a new tool, MatrixPro, intended for illustrating algorithms in action. One can produce algorithm animations in terms of direct manipulation of the library data structures, the process we call visual algorithm simulation. The user does not need to code anything to build animations. Instead, he or she can graphically invoke ready-made operations available in the library data structures to simulate the working of real algorithms. Since the system understands the semantics of the operations, teachers can demonstrate the execution of algorithms on-the-fly with different input sets, or work with "what-if" questions students ask in lectures. Such an approach lowers considerably the step to adopt algorithm visualization for regular lecture practice.

1 Introduction

A wide range of visualization systems has been developed to demonstrate various computer science core topics in the past decade. The major problem, for example, in teaching data structures and algorithms has been the difficulty of capturing the dynamic nature of the material. A proper tool for classroom demonstration would provide an ideal way to teach these kinds of concepts. Such a tool would support custom input data sets (Brown, 1988), provide multiple views of the same data structure possibly with different levels of abstraction (Hansen et al., 2000; Stern et al., 1999), include execution history (Hung and Rodger, 2000), and allow flexible execution control (Boroni et al., 1996; Rößling and Freisleben, 2002; Stasko et al., 1993). However, even though there exist many very sophisticated tools to visualize and animate these topics, the systems do not support, in general, easy on-the-fly usage. The illustrative material is typically very laborious to create. Or, the material must be prepared beforehand – at least to some extent. Thus, very few systems support the demonstration *on-the-fly*.

In this paper, we describe how to apply *visual algorithm simulation* (Korhonen, 2003) to aid the development of illustrative material for a data structures and algorithms course. Visual algorithm simulation is a generalization of algorithm animation in the sense that it allows real interaction between the user and the underlying data structures. The user is not only able to watch an animation with different input sets but he or she can also change the same data structures the algorithm modifies during the animation process. Actually, the user is able to simulate the algorithm step by step if required. The simulation consists of the very same changes in the data structures as any implemented algorithm would do. Moreover, we extend the scope of direct manipulation (see, *e.g.*, Dance (Stasko, 1991) or Animal (Rößling, 2000)) in which the user can change the visualization on the screen: in visual algorithm simulation these changes are delivered to the underlying data structures that are updated as well. Thus, there is always an actual implementation for each data structure involved in the simulation process and the corresponding visualization appears automatically on the screen after the structure is changed by the algorithm or the user. In the rest of the paper, we use, for simplicity, the term algorithm simulation instead of visual algorithm simulation.

Previously, we have employed the concept of algorithm simulation in our TRAKLA2 system (Korhonen and Malmi, 2000; Korhonen et al., 2003) that delivers *algorithm simulation exercises*. The system can give immediate feedback on learners' performance by evaluating the

correctness of the algorithm simulation. The simulation concept allows the system to compare the user made simulation sequence with the sequence generated by an actual algorithm. In addition, the system can create a model solution as an animation for each individual exercise by executing the algorithm. Similar to this, we have developed an application that allows the instructor to use the same framework for illustrating several concepts in algorithmics. The new tool, called MatrixPro, is even more powerful because we do not have to settle for simulating the ready-made exercises and follow the strict algorithm involved, but we can also allow the instructor to interact with any data structure already implemented in our library. Moreover, the simulation may also follow an algorithm that has not been implemented yet.

MatrixPro is based on the Matrix *algorithm simulation application framework* (Korhonen and Malmi, 2002; Korhonen et al., 2002, 2001). The framework provides the basic visualization and animation functionalities that are employed in this application. MatrixPro allows an easy usage of these functionalities through the graphical user interface that is designed to support lecture use and easy demonstration requirements. The motivation is to build a general purpose tool that requires no prior preparation at all to illustrate several algorithms and data structures regularly taught in computer science classes.

The rest of the paper is organized as follows. We start by introducing some basic concepts of algorithm simulation in Section 2. In Section 3 we describe the MatrixPro tool that we have built to test our ideas in practice. Finally, Section 4 concludes the discussion and reports some preliminary evaluation results and our future development ideas.

2 Algorithm Simulation

In *algorithm simulation*, the user manipulates graphical objects according to the modifications allowed for the underlying structure (like an array or a binary tree) in question and creates a sequence of simulation steps. These steps include basic variable assignments, reference manipulation, and operation invocations such as insertions and deletions.

In *automatic algorithm animation*, the user may watch the display in which the changes in each data structure representation are based on the execution of a predefined algorithm. Thus, it is the algorithm that modifies the data structure in similar steps as above, and the visualizations representing the structure are generated automatically. In Matrix framework, both of these methods to create animation sequences are supported and the system allows them to be combined seamlessly.

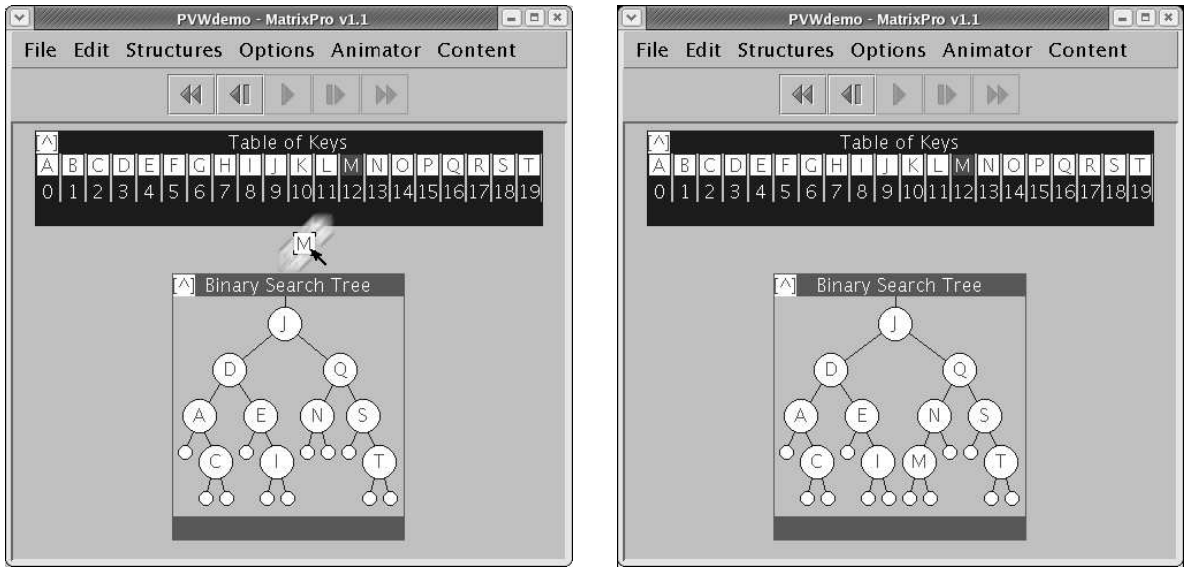
From the user point of view, algorithm simulation allows operations on a number of *visual concepts*. These include representations for arrays, linked lists, binary trees, common trees, and graphs. A data structure, such as a heap, can be visualized using several visual concepts like the binary tree or the array representation. Each visual concept may have several *layouts*, which control its visual appearance in detail. Moreover, the basic visual concepts can be nested to arbitrary complexity in order to generate more complex structures like adjacency lists or B-trees. The first one can be seen as a composite of an array and a number of lists, and the second one as a composite of a tree that has arrays in its nodes.

One of the key issues in algorithm simulation is that the user works on the *conceptual level* instead of the *code level*. The simulation environment automatically creates conceptual displays for data structures and allows the user to view and interact with the structures on different abstraction levels. Moreover, the actual underlying data structures are completely separated from their visual representations, and therefore one data structure can have different visual appearances. For example, a heap can be visualized either as an array or a binary tree, but the user can still invoke heap operations using both representations, even simultaneously.

In order to aid students to understand better the hierarchy of different concepts, we have introduced two separate groups of structures, Fundamental Data Types (FDT) and Conceptual Data Types (CDT). FDT structures behave like skeletons of data structures without any semantic information on the data stored in them (*e.g.*, lists, trees, and graphs). The struc-

ture is changed by performing primitive operations such as renaming keys or by reference manipulation. CDT structures are implementations for abstract data types, and they typically retain more constraints and are changed only by operation invocations that change the structure in terms of predefined rules. For example, Binary Search Tree (BST) is a structure that maintains the specific order of the keys stored in it.

An example of an algorithm simulation operation is represented in Figure 1. The example shows a simple operation of inserting one key into a binary search tree. In the operation, the user drags the key M from the Table of Keys and drops it on the BST (see Figure 1a). The system automatically calls the insert–routine of the underlying implementation of the BST, inserts the key to its correct position, and updates the visualization (see Figure 1b). The result is a valid binary search tree and its representation on the screen.



(a) (b)

Figure 1: Algorithm simulation example. (a) The user drags the key M from the Table of Keys and drops it on the binary search tree. (b) The key is inserted into the binary search tree.

3 System

MatrixPro is a tool for instructors to create algorithm animations in terms of algorithm simulation. The animations can be prepared prior to the lecture or on-the-fly during the lecture in order to demonstrate different algorithms and data structures at hand. The tool allows the instructor to ask, for example, *what-if* type of questions in a lecture situation, and thus make the lecture more interactive. Moreover, there is an option to introduce exercises for students (Korhonen et al., 2003). The details of these, however, are left out of the scope of this paper.

3.1 User Interface Objects

The main view of the program consists of a menubar, toolbar, and the area of the visualizations as depicted in Figure 2.

The menubar and the toolbar share the main functionality incorporated into the GUI. The menubar has the traditional File, Edit, and Options menus together with the special purpose Animator menu. There are also two other menus, Structures and Content menus, that are used to construct and insert *library data structures* on the screen, and to solve *algorithm simulation exercises*, respectively. The toolbar is an essential user interface component which

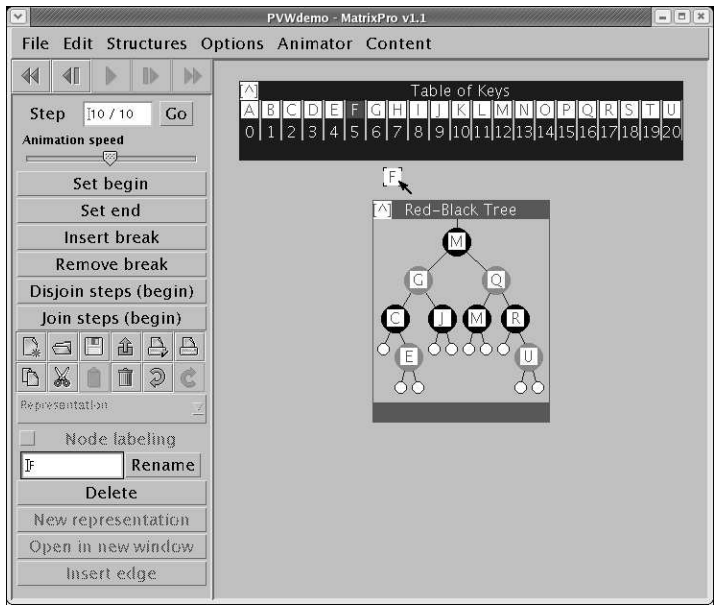


Figure 2: The MatrixPro main window. The user is currently dragging the key F to the Red-Black Tree.

enables users to handle the created animations. Through the toolbar the user can modify the animation easily by adjusting the animation speed, renaming representations, adding and removing breaks in an animation sequence, or changing the granularity of the animation sequence (join/disjoin steps). In addition, the toolbar contains functions for undoing and redoing operations as well as saving and opening animations.

Both – the toolbar and the menubar – contain the operational interface of the system called the *animator*. All built-in structures are implemented in a way that each update operation, such as a method invocation that changes the structure, is stored step by step in an additional queue and can be revoked later on. This technique allows the animator to rewind the queue of steps and replay the sequence. Naturally, the steps can also be invoked and revoked one at a time.

The area of visualizations contains the *visual entities* that the user can interact with in terms of algorithm simulation. Each visual entity is composed of at most four different types of interactive components. A *hierarchy* refers to a set of *nodes* connected with *binary relations*, and a node can hold a *key*. A key can be a primitive or some more complex structure. If the key is a primitive, it has no internal structure that could be examined further. If the key is some more complex structure, it is treated recursively as a hierarchy.

All these four entities can be moved around the display. The simulation consists of *drag and drop operations* which can be carried out by picking up the *source* entity and moving it onto the *target* entity. However, only the end point of a reference (binary relation) moves. Each single operation performs the *proper* action for the corresponding underlying data structure the entity is representing. An action is proper if the underlying data structure object accepts the change (*e.g.*, change of a key value in a node or change of a reference target).

A collection of hierarchies can be combined into a single window. Within a window it is possible to move entities from one hierarchy to another as described above. By using the GUI clipboard functionality, entities can also be moved between windows. In addition, the GUI provides functionality for manipulating hierarchies such as opening a (sub)hierarchy in other window, disposing of a hierarchy, minimizing or hiding an entity or some part of it, (re)naming or resizing an entity, changing the orientation of a hierarchy (flip, rotate), and changing the representation (concept or layout) for a hierarchy.

3.2 Features

On-the-fly usage. One of the most important features is the ability to use the system on-the-fly basis. MatrixPro offers an easy way to create algorithm animations by either applying the automatic animation of CDT structures or by simulating an algorithm by hand. All the ready-made CDT structures can be animated on-the-fly by invoking operations on them. Moreover, the user can freely modify the data structures also as an FDT by making primitive changes.

The system aids the algorithm simulation process by making it easy to invoke operations for structures also from the toolbar. By implementing a simple interface any method of the underlying structure can have a corresponding interface functionality (*e.g.*, push button) that appears both in the toolbar and in the pop-up menu of the structure. For example, for AVL trees the automatic balancing after insertion can be turned off and the rotations simulated by push buttons when appropriate during the simulation process.

In addition, the nodes in a structure can be automatically labelled, *i.e.*, a unique number appears beside each node. With this feature turned on, one can explicitly refer to a node while explaining or asking something. This is useful especially in a lecture situation as the instructor can ask questions concerning the view.

Customized animations. The system supports customization of animation in two ways. First, the user can build animations with custom input data sets by simply dragging elements from one structure, *e.g.* an array, to another structure. Even the whole array can be inserted into another structure, in which case the items are inserted in consecutive order to the target, if it supports the insertion operation. Second, the system allows controlling the granularity of the visualized execution history, *i.e.*, how large steps are shown when browsing the animation sequence. For example, primitive FDT modifications usually correspond to a couple of micro steps that form a single animator step. CDT operations, however, may consist of many animator steps. Thus, a number of animator steps can be combined to a macro step that is performed with a single GUI operation. In general, one animator step can have any number of nested steps.

Storing and Retrieving Animations. Although the automatically created animations may serve as visualizations to illustrate an algorithm without any modifications, some instructors certainly want to customize and save examples for later use. The *underlying data structures* can be saved and loaded as serialized Java objects. Thus, the corresponding animations can be recreated from this file format. Moreover, the *animations* can also be exported in Scalable Vector Graphics (SVG) format. The generated SVG animations also include a control panel that enables to move the animation backward and forward in a similar way as in MatrixPro itself.

In some cases teachers need to prepare simple figures of data structure, *e.g.*, to illustrate textual material. Of course, general purpose drawing tools and formats can be applied, but they lack the ability to automate the process of creating data structure representations by directly executing the corresponding algorithms that produce the result. For example, the TEXdrawmacros can be considered to be such a format that the user or a tool can produce. However, creating a complex conceptual visualization (for example, a red-black tree with dozens of nodes and edges or even a directed graph) is a very time consuming process without a tool that can be automated (*i.e.*, programmed) to produce valid displays. With MatrixPro this process can be easily automated and the produced visualizations can be exported directly in TEXdraw format to be included in LATEXdocuments.

Finally, we note that data structures can also be loaded from and saved into ASCII files which is practical, for example, for graphs, because the adjacency list representation is easy to write using any text editor. However, in this case, only the FDT information is stored.

Figure 3 summarizes the working process when preparing examples.

Customizable user-interface. The user interface can be customized to fit the needs of

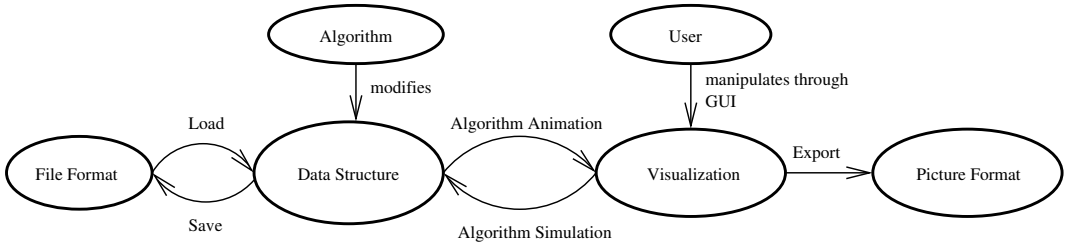


Figure 3: The process of creating algorithm animations in terms of algorithm simulation.

various users as the initial set of toolbar objects can easily be modified.

In addition, by modifying the configuration file, the contents of the menubar and pop-up menu can be changed. For example, the default font, font size, and background color can be set. Moreover, the default representation (layout) for each data structure can be changed.

Library. MatrixPro includes a library of data structures, which the user can operate on.

4 Conclusions

Algorithm animation has been an active research topic since the early 1980's and a multitude of visualization systems has been developed during this time. However, there has been no breakthrough in the field so that visualization systems would have become standard tools for instructors. One of the key reasons for this has been that creating animations is too laborious. Simply, installing the systems, learning to use them, and finally preparing examples take much more time than, for example, simple drawing on a whiteboard.

In this paper, we have introduced a system, MatrixPro, which overcomes the preparation barrier by applying algorithm simulation and a simple user interface. The power of algorithm simulation is based on the following issues. First, data structures are presented and manipulated on a conceptual level, which gives the teacher an opportunity to concentrate on the key concepts and principles of the algorithm. No coding is needed to create dynamic examples of algorithms. In addition, code level details which complicate student's learning process are totally suppressed. All this supports students' learning process and helps them to build viable mental models about the topics at hand. Moreover, the classification of structures to fundamental data types, conceptual data types and abstract data types helps them to better understand the hierarchies among different structures. Actually, most text books do not support this since they classify algorithms solely on the application point of view, *i.e.*, to basic structures, sorting, searching and graph algorithms.

Second, algorithm simulation is a far more powerful interaction method than that allowed by simple drawing tools. In algorithm simulation, the system interprets the simulation operations in the appropriate context. One can perform either simple low level operations such as assigning a new key value to a node or changing a reference, or higher level operations on abstract data types. Because no coding is required, it is straightforward to demonstrate on-the-fly what happens for a structure when operations are performed on it. This allows easy exploration of the general behavior of the algorithm with different sets of input values. Moreover, the teacher (or a student) can easily ask what-if type questions and get an immediate response from the system. Finally, algorithm simulation allows creation of exercises with automatic feedback. We have prepared an extensive set of such exercises in a system called TRAKLA2 (Korhonen et al., 2003) and these exercises are also available in MatrixPro.

Because generation of examples remains a natural task for teachers, MatrixPro supports tuning the presentation output and dynamics in many ways. The resulting ready animations or figures can be stored in formats which apply well to publishing them on papers or web pages. This is an important practical point of view, and neglecting it would diminish the

advantages of using any visualization system considerably.

Our own experience from the system has been very positive. For example, demonstrating complicated dictionaries such as red-black trees or B-trees, has previously been very clumsy using drawing tools. With MatrixPro it is easy to show the basic principles and describe the general behavior of a data structure in a very obvious way. Based on these experiences we believe the approach demonstrated in the MatrixPro tool is the direction that finally breaks the barrier of using algorithm visualization tools widely.

However, there are still many possibilities for improvement. First, although we already cover the most important dictionaries, we need to widen the current selection of priority queues, sorting methods and graph algorithms as well as other application fields. Second, some sort of history view showing several steps simultaneously is needed to make it easier to visually compare the states of the structure. Third, in many cases, especially with graph algorithms, it is important to allow free positioning mode for the user to create visually appealing examples.

Finally, we note that an obvious barrier remains: it is hard or impossible to transfer pictures or animations from one visualization system to another. People prefer to use different systems and these should not stay isolated from each other. Thus, we should be able to continue processing of an existing animation in a system that better supports the features we require. However, this is an issue that should be discussed on a wider forum among the developers of the visualization systems.

References

Christopher M. Boroni, Torlief J. Eneboe, Frances W. Goosey, Jason A. Ross, and Rockford J. Ross. Dancing with Dynalab. In *27th SIGCSE Technical Symposium on Computer Science Education*, pages 135–139. ACM, 1996.

Marc H. Brown. *Algorithm Animation*. MIT Press, Cambridge, Massachussets, 1988.

Steven R. Hansen, N. Hari Narayanan, and Dan Schrimpsher. Helping learners visualize and comprehend algorithms. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*, 2(1), May 2000.

Ted Hung and Susan H. Rodger. Increasing visualization and interaction in the automata theory course. In *The proceedings of the 31st ACM SIGCSE Technical Symposium on Computer Science Education*, pages 6–10, Austin, Texas, USA, 2000. ACM.

Ari Korhonen. *Visual Algorithm Simulation*. Doctoral thesis, Helsinki University of Technology, 2003.

Ari Korhonen and Lauri Malmi. Algorithm simulation with automatic assessment. In *Proceedings of The 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, pages 160–163, Helsinki, Finland, 2000. ACM.

Ari Korhonen and Lauri Malmi. Matrix — Concept animation and algorithm simulation system. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 109–114, Trento, Italy, May 2002. ACM.

Ari Korhonen, Lauri Malmi, Jussi Nikander, and Panu Silvasti. Algorithm simulation – a novel way to specify algorithm animations. In Mordechai Ben-Ari, editor, *Proceedings of the Second Program Visualization Workshop*, pages 28–36, HorstrupCentret, Denmark, June 2002.

Ari Korhonen, Lauri Malmi, and Panu Silvasti. TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. In *Proceedings of Kolin Kolistelut / Koli*

Calling – Third Annual Baltic Conference on Computer Science Education, pages 48–56, Joensuu, Finland, 2003.

Ari Korhonen, Jussi Nikander, Riku Saikkonen, and Petri Tenhunen. Matrix – algorithm simulation and animation tool. http://www.cs.hut.fi/Research/Matrix/, Nov 2001.

Guido Rößling. The ANIMAL algorithm animation tool. In *Proceedings of the 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'00*, pages 37–40, Helsinki, Finland, 2000. ACM.

Guido Rößling and Bernd Freisleben. ANIMAL: A system for supporting multiple roles in algorithm animation. *Journal of Visual Languages and Computing*, 13(3):341–354, 2002.

John Stasko, Albert Badre, and Clayton Lewis. Do algorithm animations assist learning? An empirical study and analysis. In *Proceedings of the INTERCHI'93 Conference on Human Factors on Computing Systems*, pages 61–66, Amsterdam, Netherlands, 1993. ACM.

John T. Stasko. Using direct manipulation to build algorithm animations by demonstration. In *Proceedings of Conference on Human Factors and Computing Systems*, pages 307–314, New Orleans, Louisiana, USA, 1991. ACM, New York.

Linda Stern, Harald Søndergaard, and Lee Naish. A strategy for managing content complexity in algorithm animation. In *Proceedings of the 4th annual SIGCSE/SIGCUE on Innovation and technology in computer science education, ITiCSE'99*, pages 127–130, Kracow, Poland, 1999. ACM Press. ISBN 1-58113-087-2.

Multi-Lingual End-User Programming with XML

Rob Hague and Peter Robinson
University of Cambridge Computer Laboratory
Cambridge, UK

{Rob.Hague,Peter.Robinson}@cl.cam.ac.uk

Abstract

There is no ideal programing language. Each is better suited to some tasks rather than others. This suitability varies not only with the overall goal of the finished program, but also with different stages of development such as architectural design, detailed implementation, and maintenance. The situation is even more acute in the case of end-user programming languages, which cater for a much more varied user population. It would therefore be advantageous to allow the same program to be viewed, and edited, in a number of different languages. We have developed a system, Lingua Franca, that provides this facility for end-user programming languages in the setting of ubiquitous computing in the home.

Introduction

Since the invention of FORTRAN in 1957, high level programming languages have proliferated to the extent that it is no longer feasible to count them. These range from special-purpose, niche languages to the almost ubiquitous C and C++, and include approaches such as visual programming and programming by demonstration in addition to conventional textual languages.. This diversity would not survive if all languages were equal. It exists because any given language is better at some tasks than others. Of course, commercial concerns also play a part. While there is a significant degree of overlap and duplication amongst languages, the selection of the proper language for a given task renders the programmer's task significantly easier (and, conversely, the selection of the wrong language may render it difficult or impossible).

A similar disparity exists between different operations that a programmer may wish to perform on a program. The Cognitive Dimensions of Notation framework (Green, 1989) categorizes such tasks as *incrementation* (adding new information), *transcription* (from one notation to another), *modification* or *exploratory design* ("combining incrementation and modification, with the further characteristic that the desired end state may not be known in advance"). These four types of activity make different demands of languages, and so it would be reasonable to suppose that different languages are useful for different manipulations (and at different points in the development process).

This conjecture is borne out by numerous studies into the applicability to different representations of the same data to different problems based on that data (Whitley, 1997). Cox (1996) goes further, developing an environment that specifically supports the use of multiple notations for the representation of data relating to logic problems of the type found in GRE tests (used to assess applicants to graduate schools in the US). We have created an XML-based system that applies a similar idea to programming. A program may be created, viewed or edited in one of several languages, and changes are automatically visible in all representations.

Context

This work arises from observations made while designing an end-user programming language for use in the context of *ubiquitous computing* (technology that "disappears into the background" (Weiser, 1991)). Our work focuses on the networked home, in which domestic devices are augmented with communication and computation. Such a network allows devices to work in concert to do more than they can individually, but this is of little use if there is no way for

Figure 1: Prototype Media Cubes

the user to control this enhanced functionality. We believe that end-user programming may provide a means for such control.

One of the language designs considered was the *Media Cubes* (Blackwell and Hague, 2001), a tangible programming language in which programs are constructed by manipulating physical blocks as opposed to text or graphics (Figure 1). This language has numerous positive characteristics; in particular, it allows a smooth progression from direct manipulation to programming, and it provides a concrete representation for abstract concepts. This makes it particularly suited to users unfamiliar with programming. However, it has one major drawback: there is no external representation of the program once it is created. This makes it impossible to edit the program after it is created, or even to examine it to ensure it is correct.

Instead of redesigning the language to avoid this deficiency (and consequently losing many of its positive qualities), we sought to design an environment in which such languages may be usefully employed. By allowing a program created using the Media Cubes to be viewed and edited using another language, the deficiency becomes far less problematic.

Lingua Franca - Scripting in Many Languages

The system produced is based around *Lingua Franca*, an XML-based intermediate form common to all scripting languages in a system. Common intermediate forms have been used extensively to integrate components written in different languages, the most recent example being Microsoft's .NET Framework (Microsoft, 2004). Lingua Franca goes beyond conventional multiple-language systems in that it not only allows translation from various source languages to the intermediate form, but also allows translation from the intermediate form to the source languages. By combining these two processes, it is possible to effect translations between different source languages. This, in turn, allows a single component to be manipulated in multiple languaes at different times. In contrast, previous systems have allowed different language to be used for different components, but each component was always represented in the same language.

While it is possible to translate from traditional intermediate forms (which are almost exclusively designed for the purpose of being executed directly, or for further transformation into a form that may be exectuted), the results are generally imperfect reconstructions of the original source, and require significant effort to comprehend. Aside from the problems associated with recreating the structure of the code, secondary notation – elements that have meaning to the programmer, but do not affect execution, such as variable names, layout and comments – cannot be reproduced, as they were discarded during the translation into intermediate form. In contrast, Lingua Franca explicitly encodes all structure and secondary notation, allowing the original source code to be reconstructed exactly.

The two types of secondary notation that are most commonly discarded when translating a script from one form to another are point annotations, such as comments, and higher level structure, such as loops. Both of these may vary greatly from language to language. Lingua Franca allows multiple point annotation elements to be associated with a part of a script; each

Figure 2: A Simple Lingua Franca Program

```
<xax:group name="Shopping">
  <receive event="0">
    <nt>OutOf</nt>
    <receiveonce>
      <nt>ReorderFood</nt>
      <dispatch>
        <nt>Order</nt>
        <nts>
          <param event="0"
                 name="nts"/>
        </nts>
      </dispatch>
    </receiveonce>
  </receive>

  <receive>
    <nt>Day</nt>
    <nts>Sunday</nts>
    <dispatch>
      <nt>ReorderFood</nt>
    </dispatch>
  </receive>
</xax:group>
```

annotation is tagged with a notation type, to allow language environments to determine which (if any) to display. Higher level structure is represented by grouping; again, each group is tagged with a type (such as while loop), which may imply a particular structure, and language environments may use this to determine how to display the group's members. Unlike point annotation, any environment that can display Lingua Franca can display any group, as in the worst case it can simply display it as a grouped collection of primitive operations.

Different source languages are supported by the Lingua Franca architecture via "language environments" that translate between the source language and Lingua Franca. The most general class of language environments perform translation in both directions; these may be used to edit a script, by first translating from Lingua Franca to a "source" notation, modifying that representation, then finally translating it back to Lingua Franca.

Not all environments allow translation in both directions. Some language environments only translate from the source notation to Lingua Franca (and may only be used to create scripts). The Media Cubes are a prime example of such an environment. Others only translate from Lingua Franca to some other notation (and may only be used to display scripts).

Programs in Lingua Franca are held on a central database. All or part of the "corpus" of programs may be examined or modified over the network, via the HTTP protocol (Fielding et al., 1999). Lingua Franca is represented as XML. The use of standards representations and protocols allows implementors to take advantage of widely available tools and libraries.

The structure of Lingua Franca is broadly similar to that of the π-calculus (Milner, 1999), but uses asynchronous as opposed to synchronous events (several formal calculi based on asynchronous events exist, but non match the semantics of Lingua Franca). Asynchronous events were considered to be a more suitable for ubiquitous computingSaif and Greaves (2001), and are implemented by the GENA protocol (Cohen et al., 2000).

Figure 2 illustrates a simple Lingua Franca program, expressed using the XML representation. The overall effect of this program is to construct a shopping list of items as they run out, and then send out orders for them on Sunday. The program consists of a group element named "Shopping". Within this group are two receive elements; these elements represent listening processes.

Events are specified using a notification type (nt) and a notification subtype (nts), and are written in the form "nt/nts", with "nt/*" being shorthand for an event with the given notification type, but any notification subtype. receive and receiveonce elements react to any event with the appropriate notification type and optional subtype, specified by the nt and nts elements, by dispatching events specified by any dispatch subelements, and creating new processes corresponding to any receive or receiveonce subelements. receiveonce elements act in the same way, but as opposed to continually listening for events and reacting to each one, they react to a single event and are then removed. The event attribute may be used to bind an event to a name, allowing data from the event to be accessed using the param element; in the example above, this is used to produce an "Order" event with the same notification subtype as the corresponding "OutOf" event.

Whenever an OutOf/* event occurs, the first receive element creates a new process, which consists of a receiveonce that reacts to a ReorderFood/* event (the subtype is not significant), and dispatches an event Order/O.nts, where O.nts is the notification subtype of the original OutOf/* event. On receipt of a Day/Sunday event, the second receive element dispatches a ReorderFood/- event, which causes those processes to dispatch the appropriate Order events, after which they are removed (as they correspond to receiveonce elements, as opposed to receive.)

A Menagerie of Programming Languages

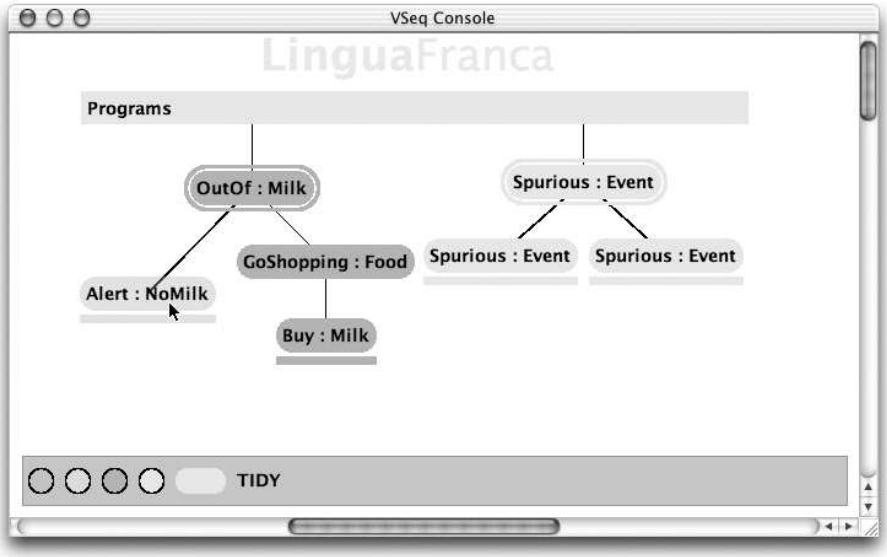
A variety of scripting languages are being developed in order to demonstrate the flexibility and range of the Lingua Franca architecture. These languages are designed to complement each other, in that they may be used to perform different manipulations on the same script with ease.

It is of course possible to manipulate Lingua Franca directly in XML form. However, this is needlessly difficult, as the programmer only has access to primitive operations, and would have to build up higher level operations from scratch. More importantly, manipulating the code directly in this way bypasses most of the integrity checks that would be performed by a bona fide language environment, and hence creates the risk of introducing malformed code into the database, or accidentally removing or modifying data associated with another language.

Hence, a textual language, LFScript, has been developed to provide an interface familiar to those with experience of conventional scripting languages, allowing access to low-level features of the Lingua Franca system without the problems associated with manipulating the XML directly. It is envisaged that this will be mostly useful for creating and editing substantial scripts, a task most likely to be undertaken by someone with at least some programming background. It would be possible to compile an existing imperative language such as C into Lingua Franca, as the event mechanism is sufficiently flexible to model imperative execution. However, programs encoded in this way would have a very different structure from those created using event-based languages, and as such it would be difficult to take full advantage of the facilities for providing two-way translation. Instead, the textual language is patterned more on declarative languages, and makes the Lingua Franca primitives available to the user, in addition to providing higher level constructions such as synchronous functions and sequencing.

The VSeq language environment allows the user to manipulate Lingua Franca programs in the form of event diagrams. These diagrams represent causal relationships via a tree of nodes.

Figure 3: A program rendered in the VSeq console, and a fragment of the corresponding Lingua Franca source



It is likely to be the main environment for the manipulation of mid-sized scripts, as it offers visual access to the full range of Lingua Franca functionality, but may become unmanagable for large programs.

A "console" application, shown in Figure 3, allows the user to edit the diagram interactively. Leaf nodes in the diagram (further distinguished by being underlined) represent events that will be emitted, and correspond to dispatch nodes. Unadorned nodes correspond to receiveonce nodes, while nodes with an outer border represent receive nodes. Rectangluar strips, such as the one labelled "Programs" in Figure 3, are used to represent groups.

The ordering implied by the tree corresponds to causality; the effect of a node is conditional on the event described by its parent. Nodes may be moved by dragging with the mouse. Horizontal ordering of siblings is reflected in the sequential ordering of elements in the corresponding Lingua Franca program. Dragging one node over another connects adds it as a child of that node, provided that the resulting tree would correspond to a valid Lingua Franca program.

The console ensures that the vertical ordering remains consistent with the parent-child relationship; if a child is moved to a position above its parent, it is detached from that parent and added to the root (there is no visible representation of the root node; children of the root appear to be disconnected). Preserving this invariant means that vertical ordering of connected elements in a Vseq diagram reflects the causal ordering embodied by the corresponding program.

The console also includes a *toolbox* (the blue area towards the bottom of the window), enabling a wider range of manipulations of the diagram. These tools are modeless; a tool is dragged from the toolbox and dropped on the diagram element that it is to be applied to. The four coloured circles allow the user to colour parts of the tree. This is purely secondary notation, and has no effect on execution. The oval allows new nodes to be created. The "Tidy" tool lays out a node and all of its children in an orderly fashion, without changing the ordering or containment properties.

Secondary notation in VSeq consists of visual properties specific to VSeq, such as the size and position of an element's representation, and more general properties, chiefly background colour. These are encoded separately, to allow other language environments to express generic properties whilst ignoring those useful only in the context of VSeq.

We are also considering the feasibility of a purely presentational form, and cannot be used

to create or edit scripts, but only to display them. This allows it to be specialized in order to facilitate searching and comprehension of scripts. This form may be interactive, in the sense that it may include user-initiated animation to aid navigation, but the user cannot make any changes that are propagated back to the Lingua Franca corpus.

Perhaps the most unusual of the language environments being developed for use with Lingua Franca is the Media Cubes language, mentioned previously. This is a "tangible" programming language. In other words, it is a language where programs are constructed by manipulating physical objects—in this case, cubes augmented such that they can determine when they are close to one another.

The faces of the cube signify a variety of concepts, and the user creates a script by placing appropriate faces together; for example, to construct a simple radio alarm clock, the "Do" face of a cube representing a conditional expression would be placed against a representation of the act of switching on a radio, and the "When" face against a representation of the desired time. In an appropriately instrumented house, the representation can often be an existing, familiar item, or even the object itself. In the above example, a time could be represented using an instrumented clock face, and turning the radio on could be represented by the radio itself, or its on switch. One significant difference between the Media Cubes language and other tangible computing interfaces is that the user does not construct a physical model of the program, but performs actions with the cubes that cause parts of the program to be created or combined.

The Media Cubes language is intended to be easy for those unfamiliar to programming, and as such would provide a low-impact path from direct manipulation to programming. However, as mentioned, the language as it stands is unusual in one very significant respect - scripts do not have any external representation. This means that it is only feasible to construct small scripts, and that, once created, scripts may not be viewed, and hence may not be modified. In most circumstances, this would render the language all but useless, but the Lingua Franca makes it feasible to include niche languages such as the Media Cubes in a system without sacrificing functionality.

Status and Further Work

A working prototype of the Lingua Franca execution engine has been implemented. This comprises of an interpreter for the intermediate form, an HTTP server to allow clients to add, remove and replace code in the corpus over the network, and an interface to the GENA protocol to allow scripts to interact with the outside world.

The VSeq language environment has been implemented, and the other language environments mentioned are under development. Working prototypes of the Media Cubes have been developed, but currently use infra-red to communicate with a base station. The requirement for line-of-sight that this imposes makes the current prototypes unsuited to user testing; a version based on radio communication has been planned. The Media Cubes language environment has been developed as a simulator, in the using a simulator, combined with "Wizard of Oz" techniques for user studies (Wilson and Rosenberg, 1988).

A core requirement of Lingua Franca language environments that implement bidirectional translation is that that translation must be reversible. For example, if a program is translated from VSeq into Lingua Franca, and then back into VSeq, the result should be identical to the original program. This goal has been achieved in the language environments to date. As components of the system may run on devices with limited processing power, runtime efficiency is a factor. Consequently, we have moved from using a scripting language (Python) and a generic tree representation (DOM) to a systems language (Java) and a custom representation in more recent versions.

Once the prototype language environments have been produced, a user study in which participants apply the languages to various tasks will be conducted. This will allow us to establish

the relationships between tasks and languages, and will inform the subsequent refinement of those languages.

An important point about the work is that it limits its scope to a specific domain, that of end-user programming in the domestic environment. This is intentional, as it reduces the problem of translating between arbitrary, general purpose programming languages to the far more tractable problem of translating between closely related languages. This is not only useful in its own right, but may well produce results that will inform the design of more generally applicable systems.

References

Alan F Blackwell and Rob Hague. Autohan: An architecture for programming the home. In *the IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 150–157, 2001.

J Cohen, S Aggarwal, and Y Y Goland. General event notification architecture base: Client to arbiter, 2000. Internet draft `http://www.upnp.org/download/draft-cohen-gena-client-01.txt`.

Richard Cox. *Analytical Reasoning with multiple external representations*. PhD thesis, University of Edinburgh, 1996.

R Fielding, J Gettys, J Mogul, H Frystyk, L Masinter, P Leach, and T Berners-Lee. Hypertext transfer protocol - http/1.1, 1999. Request For Comments (RFC) 2616.

T RG Green. Cognitive dimensions of notation. *People and Computers V*, pages 443–460, 1989.

Microsoft. *Microsoft .NET Framework*. 2004. `http://www.msdn.microsoft.com/netframework/`.

Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

U Saif and D Greaves. Communication primitives for ubiquitous systems or rpc considered harmful. In *ICDCS International Workshop on Smart Appliances and Wearable Computing*, 2001.

Mark Weiser. The computer for the 21st century. *Scientific American*, pages 94–110, 1991.

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

J Wilson and D Rosenberg. *Prototyping for User Interface Design*. North-Holland, New York, 1988.

Multimodal Modelling Languages to Reduce Visual Overload in UML Diagrams

Kirstin Lyon, Peter J. Nürnberg
Department of Computer Science, Aalborg University Esbjerg
Denmark

(kirstin, pnuern)@cs.aue.auc.dk

Abstract

It is normally easier to build a program when the design issues are fully understood. One way of doing this is to form a "mental map" or visualisation of the problem space. Visualisation is an increasingly important method for people to understand complex information, to navigate around structured information and to aid in software design. Software visualisation techniques are usually tackled from a graphical point of view. Some similarities exist between spatial hypertext applications and software visualisation techniques. A limitation of both of these is that when the amount of information becomes large enough, users may experience visual overload. This paper suggests using multimodal cues to reduce this.

1 Introduction

Software visualisation deals with animation of algorithms as well as the visualisation of data structures. One of its main goals is to achieve a better understanding of complex programs, processes, and data structures through diagrams. Graphical representations allow for a quick overview of information. In such representations multidimensional conceptual structures are reduced into a plane. This can cause several design issues. Users may also experience difficulties in understanding a diagram that they themselves created previously due to its complexity.

Organising large amounts of unstructured information has similar difficulties to that of software design modelling. A difficulty with present modelling systems is that the structures they provide are often too rigid, and are often fixed for/by the user before the problem space is fully understood. Even when a problem space is fully understood, conditions may change through time. Entities may have various visualisations depending on the contexts in which they are being used. Changes in visualisation may force users to rethink structures. Some software and knowledge visualisation tools share similar principles, and therefore similar benefits and limitations.

When the amount of information to be organised becomes large enough, users may experience information overload. For example, a user has 10 emails to organise. This is a relatively easy task as many of the details from each page can be remembered. If this number is increased to 10,000, remembering each email becomes unrealistic. Now the emails need to be organised in some way, for example by category, date, size, sender etc. A user is able to remember a collection of categories rather than individual mails. Without this structuring of information, users become overloaded with information. Visual overload occurs when users are presented with many graphical elements simultaneously, for example, in a UML diagram.

Some hypertext tools allow users to organise their information spatially without knowing the structure beforehand, and allow users to model changing spaces. With these tools users create work areas and modify objects within this space with the organisation of information occuring over time. However, difficulties occur when trying to recreate highly complicated structures in a linear/planar representation e.g. serialising a data structure is known to be simpler than parsing it. Some suggestions as to how to deal with visual overload exist, e.g. increasing dimensions (Robertson et al., 1998), using different types of visual cues or increasing modalities. This paper focuses on the latter option.

The remainder of this paper is organised as follows: Section 2 compares spatial hypertext and software visualisation tools. Section 3 outlines some possibilities of modality use.

Section 4 describes a scenario. Section 5 provides some future work and Section 6 some conclusions.

2 Spatial Hypertext and Software Visualisation

In this section, we look at how spatial hypertext could improve software visualisation tools. We first consider some tools are available at present for software designers. We then consider some spatial hypertext systems.

2.1 Software Visualisation Tools

The Object Oriented Analysis and Design (OOA&D) methodology considers programs to be simulations of reality. In order to build an effective program an efficient model must be created first. The tools used for this are graphical in nature e.g. rich pictures, class diagrams, use case diagrams etc.

Tools such as BlueJ(Sanders et al. (2001)) converts users' class diagrams into classes automatically. A more complete suite, Rational Rose(Rose), offers a more sophisticated implementation of UML diagrams that also include re-factoring and full cycle capabilities. JBuilder(JBuilder) and similar products provide a fast and visual method of creating a user interface prototype.

2.1.1 Modelling Languages

Modelling languages allow users to model the real world graphically. One such language is UML. UML provides the user with a set of graphical elements, and allow the user to structure them in a way that is appropriate for the task. Modelling languages organise a subset of knowledge that is important to a particular software design problem. UML is currently the most widely accepted standard for object-oriented software diagrams. However the notation set is large, and can be customised (Wieringa, 1998).

Other modelling languages include Organisational Requirements Definition for Information Technology Systems (ORDIT) and SCIPIO.

ORDIT is based on the idea of role, responsability and conversations, making it possible to specify, analyse and validate organisational and information systems supporting organisational change(Dobson and Strens (1994)). The ORDIT architecture can be used to express, explore and reason about both the problem and the solution aspects in both the social and technical domains. From the simple building blocks and modelling language a set of more complex and structures models and prefabrications can be constructed and reasoned about. Alternative models are constructed, allowing the exploration of possible futures.

The SCIPIO method identifies three ways of modelling a situation, to support strategic planning decisions as well as tactical design decisions(scipio).

- Business relationship modelling provides a view of the network of collaboration agents and their oblications to one another.
- Business rule modelling provides a view of the rules, policies, regulations and other constraints imposed on the business.
- Business asset modelling provides a static view of the resources of the organisation.

Both languages are graphical in nature. They have similar problems to other graphical modelling languages.

2.2 Spatial Hypertext Tools

Hypertext research looks at how to present and represent material electronically that is too complex to represent on paper (Nelson, 1965). Spatial hypertext is a structure discussed in the hypertext domain. It takes advantage of our visual and spatial intelligence. These tools allow users to organise information spatially, and allow structures to emerge over time.

In general, spatial hypertext tools are similar to Visual Knowledge Builder (VKB) (Shipman et al., 2001). These tools are good for information-intense activities, such as analysis or research. How information is presented is decided upon by the user, with no specific structure being decided for them beforehand. As new files are added, structures will inevitably change. Files are given various attributes which allow the user to see what items are related i.e. colour, proximity, shape and size.

2.2.1 Modelling Language

Spatial hypertext tools provide the user with a modelling language inside the application that performs a similar task to that of Unified Modelling Language (UML) for software designers. The visual layout of these tools and UML may be viewed as visual languages. UML diagrams can be seen as similar to spatial hypertext. Both interfaces share visual properties.

2.3 Comparison

When comparing spatial hypertext and UML diagrams, it is noticable that in both, some form of spatial organisation exists. Shapes of objects are also important. UML has a predefined set of shapes that have specific functions and semantics, whereas the language used in VKB is decided upon by the user.

Some limitations for both languages exist e.g. diagrams in both languages can become complex and difficult to interpret and remember through time. It is also difficult to categorise information. There is also a difficulty when the amount of information becomes large, users become inundated with information. If we consider a UML class diagram to be similar to a spatial hypertext diagram, we can use some solutions suggested by that community to improve this.

3 Multimodal Modelling Language

This section first of all describes a possible language, and then considers what devices could be used.

3.1 Modelling Language

Multi-modal interfaces use more than one modality(sense) to convey meaning i.e.vision, audio, haptic etc. It is possible to mix various modalities e.g. visual and audio, tactic and audio etc. Research has considered various combinations of modalities and how to create an interface. Guidelines exist on how to create multimodal interfaces Heller and Dianne Martin (1995). One method is to use them in a way you would naturally. This approach has several benefits. Each sense has a particular set of characteristics. If this is exploited correctly, user interfaces become easier to use. This paper considers increasing the number of modalities to include visual, audio and haptic (sense of touch).

When considering what information to transfer to what modality, it is necessary to understand the strengths of each sense. This is summerised in tab. 1.

Vision is best suited to high-detail work. It can attend to anything in sight at that time. However, when the amount of information increases it becomes difficult to maintain and overview and focus at the same time. Visual attributes include proximity, colour, shape and size. Audio is better at maintaining an overview. It has less detail than visual cues, but

Quality/Ability	Audio	Visual	Haptic
Transmission	Broadcast, linear	Broadcast	Narrowcast
Information content	High, but less detail than complex images.	High and flexible	Surface properties

Table 1: Comparison of audio,visual and haptic cues.

may be useful in providing spatial or density information. It may also be useful to describing relationships between objects. This is difficult to do with visual cues, as diagrams can become complex when all relationships are described. Its attributes include pitch, rhythm, volume, etc. Tactile (touch) information is limited in the amount of information it can hold. You are limited to how much you can touch at the same time. This could be useful for highlighting areas of diagrams. It's attributes include roughness, temperature etc. Sutcliffe (2003)

This paper suggests using visual cues for high detailed areas, audio for maintaining an overview of the relationships between objects, and tactile for highlighting objects that are of interest to the user.

3.2 Devices

Difference modalities require different devices, for example, as most information presented by a computer is through visual means, all computers are provided with a monitor. Audio cues require either headphones or speakers. Both of these come as standard with many computers. It is not so easy to use tactile cues.

There are two main types of haptic devices:

- Glove or pen-type devices that allow the user to "touch" and manipulate 3D virtual objects.
- Devices that allow users to "feel" textures of 2D objects with a pen or mouse-type interface

WingMan Force Feedback Mouse and the iFeel mouse are examples of 2D devices(ifeel and wingman). Examples of 3D devices include the Phantom(phantom) and CyberGrasp(cyberglove), a glove style interface.

Before specifying what tool should be used, it is necessary to consider exactly what kind of cues the user will receive, and if the user should input tactile cues themselves.

4 Scenario

In this section, we consider a scenario of a UML class diagram use. We base this scenario on traditional software design methodology, i.e. proto-type is designed then implemented, testing and evaluation is carried out and the prototype is modified with information from the evaluation.

Introduction. *David and Grace are programmers. They have been newly employed by a university to modify the payroll system for staff. First they look at the original designs of the program. Then they decide what needs to be modified through checking the new tax system, and how the salaries of staff will change. Then they create a new design. Once this is established they implement the modifications and check that all classes are changed if neccessary.*

Diagram familiarisation. *Initial challenge is to learn how UML is being used in this case so that standards can be maintained.*

David and Grace would consult either other programmers at the university who are familiar with UML, or consult the documentation that describes how UML is being used in this case. Some time is required to become familiar with the university's usage of UML

Design familiarisation. *Next they must become familiar with the original design.*

They consult the program's diagrams. As this is a large program, the diagrams are large and complex. Some time is required to form an overall understanding of how the program works.

Program Design. *Modifications need to be designed. This may involve classes being added, deleted or modified in some way. To do this, they need to understand how the classes interact, and the constraints on each class.*

These relationships are not shown in UML class diagrams at present. They would have to use other diagrams to find out what parts of objects related to what other parts of objects. They need to know the constraints on each object too.

Update documentation. *As an ongoing task, documentation is updated so that when the program needs to be modified again, future employees may understand how the program works.*

At present, diagrams would be modified with the new changes. Difficulties still occur when it is not clear where an object belongs. A particular visualisation that best suits may be chosen. In future when the program needs to be modified once more, it may be different people that work on it, so documentation needs to be clear.

4.1 Multimodal suggestions

This paper suggests transfering some of the visual information into audio or haptic output. Different relationships can be represented by different modalities. At the moment it is difficult to represent many-to-many relationships. Audio could be used to represent all relationships between objects. Haptic output could be used to highlight specific objects, e.g. when an object is touched, the 'feel of the object changes in some way.

Increasing the number of modalities used can increase the amount of information the user can take in at one time. Each modality could contain information for a different layer of the diagram. The main difficulty of UML diagrams is that they are trying to represent information that is multi-dimensional, so using only 2D graphics is perhaps too limiting.

5 Future Work

Our next step is to build a spatial hypertext prototype that uses a multimodal modelling language. Guidelines for multimodal interface design already exist, but it is still difficult to find a useful balance between the modalities. It is also not clear what information is best represented in audio or haptic instead of vision. To resolve this, usability tests will be carried out at various stages of the analysis to find this balance. With those results fed into a prototype.

6 Conclusions

At present the main focus for modelling languages has been on creating them visually. This has limitations, for example, how to represent mutli-dimensional information in a 2D space. Users can become overloaded visually due to complex diagrams, and it can take time to become familiar with unknown designs. Similarities between software visualisation and spatial hypertext exist. Both are useful for information intense tasks, and have a visual modelling language. Some possible suggestions for reducing visual overload have been suggested from the spatial hypertext community, and should also be applicable to the software visualisation community.

References

Keith Andrews, Wolfgang Kienreich, Vedran Sabol, Jutta Becker, Georg Droschl, Frank Kappe, Michael Granitzer, Peter Auer, and Klaus Tochtermann. The infosky visual explorer: exploiting hierarchical structure and document similarities. *Information Visualization*, 1(3/4):166–181, 2002. ISSN 1473-8716.

Nelson Baloian and Wolfram Luther. Visualization for the mind's eye. In *Software Visualization*. Springer, 2001.

cyberglove. URL `http://www.immersion.com/3d/products/cyber_glove.php`.

J. E. Dobson and M. R. Strens. Organizational requirements definition for information technology systems. In *IEEE International Coference on Requirements Engineering*. IEEE, 1994.

John Domingue, Blaine A. Price, and Marc Eisenstadt. A framework for describing and implementing software visualization systems. In *Proceedings of Graphics interface*, 1992.

Jing Dong. Uml extensions for design pattern compositions. In *Journal of Object Technology, vol. 1, no. 5, pages 149–161*, 2002.

Rachelle S. Heller and C. Dianne Martin. A media taxonomy. *IEEE MultiMedia*, 2(4):36–45, 1995. ISSN 1070-986X.

ifeel and wingman. URL `http://www.logitech.com/`.

JBuilder. URL `http://www.borland.com/jbuilder`.

Ted Nelson. Complex information processing: A file structure for the complex, the changing and the indertimate. In *Proceedings of the 1965 20th national conference*, 1965.

phantom. URL `http://www.sensable.com`.

George Robertson, Mary Czerwinski, Kevin Larson, Daniel C. Robbins, David Thiel, and Maarten van Dantzich. Data mountain: using spatial memory for document management. In *Proceedings of the 11th annual ACM symposium on User interface software and technology*, pages 153–162. ACM Press, 1998.

Rational Rose. URL `http://www-306.ibm.com/software/rational`.

Dean Sanders, Phillip Heeler, and Carol Spradling. Introduction to bluej: a java development environment. In *Proceedings of the seventh annual consortium for computing in small colleges central plains conference on The journal of computing in small colleges*, pages 257–258. The Consortium for Computing in Small Colleges, 2001.

scipio. URL `http://www.scipio.org`.

Frank M. Shipman, Haowei Hsieh, Preetam Maloor, and J. Michael Moore. The visual knowledge builder: a second generation spatial hypertext. In *Proceedings of the twelfth ACM conference on Hypertext and Hypermedia*, pages 113–122. ACM Press, 2001. ISBN 1-59113-420-7.

Alistair Sutcliffe. *Multimedia and Virtual Reality: Designing Multisensory User Interfaces.* Lawrence Erlbaum Associates, 2003.

Roel Wieringa. A survey of structured and object-oriented software specification methods and techniques. In *ACM Computing Surveys, Vol.30, No.4.* ACM, 1998.

Concretization and animation of Finite Automata with c-cards

Andrea Valente
Aalborg University in Esbjerg
Denmark

av@aue.auc.dk

Abstract

The paper presents a new way of introducing *finite automata* to classes of 10 to 12 years old, and in general to students with a limited mathematical background. Our approach builds on *computational cards* (or c-cards), a project aiming at scaling-down the learning complexity of computer science core contents, by presenting symbol manipulation via a tangible, physical metaphor. C-cards are still at a very early stage of development as an educational tool, yet we believe they represent a nice, intuitive mind-tool, that can be consistently applied to a variety of (theoretical) computer science concepts. Here an algorithm for mapping finite automata on c-cards is given and its educational implication discussed: the mapping provides a mean to concretize and animate automata. The algorithm works with both deterministic and non-deterministic finite automata.

1 Introduction: motivation and background

In this paper we present a new way of introducing *finite automata* to classes of 10 to 12 years old, or older students with a limited mathematical background. Finite state machines are central in (theoretical) computer science and represent a powerful mind tool for understanding a great variety of computing machines that we (and our children) face in everyday life: mobile phones, video-recorders, vending machines and even more traditional devices like cars and lifters can be easily described and discussed by formalizing them as finite automata (FA for short). However these concepts are seldom learned outside university courses and even in that environment they are usually considered tough by students (Hämäläinen, 2003); in this context a state-of-the-art tool is JFLAP (Cavalcante et al., 2004), an instructional software covering very many topics in the theory formal languages. However fast and robust, visual and interactive in nature, this tool (like others of its kind) does not make any attempt to unify the various notations used in the field, resulting in a collage of highly integrated ad-hoc tools, each adopting a different, specific formalism; the suggested target-group of JFLAP is university-level students.

Our approach builds on *computational cards* (or c-cards), a project aiming at scaling-down the learning complexity of computer science core contents, by presenting symbol manipulation via a tangible metaphor: physical objects represent computational elements and objectified symbols (Valente, 2003). Although c-cards are still at a very early stage of development as an educational tool, we already showed that small-sized card systems (called *card circuits*) can implement interesting and quite complex behaviors. Here we propose to concretize and animate FA with c-cards. In this way students can play and learn about automata theory *before* actually being exposed to its formal notation, so to ground the meaning of abstract symbols in concrete examples (a discussion on *symbol grounding* can be found in Harnad (1990)); our approach is strongly related to *learning-by-doing*, advocated by Papert and constructivism in general (Papert, 1993).

In the rest of the paper we will recall the definition of deterministic FA (or DFA) and the basic characteristics of c-cards as an educational tool. After that an automaton will be defined and implemented by a card circuit. The discussion of this example will guide us to define a general mapping between DFA and card circuits. Some considerations about the way of introducing DFA to the students will conclude the paper.

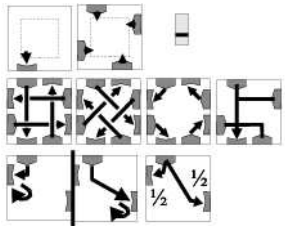


Figure 1: A deck of c-cards.

2 Finite Automata

A DFA is defined as a 5-tuple: $< S, \Sigma, T, s, F >$ where S is the set of all states of the automaton, Σ is its alphabeth, T is the transition function, defined as a mapping from $S \times \Sigma$ onto S; there are also a single state $s \in S$, that is the initial state for the automaton, and the set of all final states (also called acceptance states), F.

An automaton can be used in two ways: as a language (or string) recognizer and as a generator. In the first case the user provides a string (in the alphabeth of the automaton); the automaton will read the input string one symbol at the time, and change state accordingly, until the string is completely consumed. At this time the user can check if the DFA is in one of its final states, and if so, the string is considered as *accepted* by the machine. The set of all the strings accepted by a DFA is the *language* recognized by the automaton. The other use of a DFA is to generate strings of its language. In this case the user starts with the automaton in the initial state and an empty output string. At each step the user picks (randomly) a transition of the form $T(currentState, symbol_i) = newState$, where $currentState$ is the actual state of the state machine, and applies it, changing state to $newState$ and appending the symbol $symbol_i$ to the output string. The *game* can be stopped when the DFA is in a final state, so that the output string is part of the language of the automaton.

Another kind of finite automata is the non-deterministic finite automata, or NFA, where the transition function is defined from $S \times \Sigma$ onto S^* (sets of states). After reading a symbol from the input string the state of an NFA can change in a non-deterministic way, into one of a number of new states.

3 C-cards

C-cards are presented in (Valente, 2003), where we give the 8 basic types of cards (together they form a deck, figure 1); some of these types represent pipes, connecting elements, while others are the active computational elements. The standard set of c-cards is called a deck.

To play a game at c-cards, we need to print some copies of the deck, then we cut free our cards and some pegs (small rectangles of paper, representing symbols). At this point we are ready to compose a card circuit, by connecting cards together; in every circuit there is at least one source (a starting point for pegs) and one pit card (where pegs will stop and accumulate). To execute a computation, a number of (identical) pegs will be placed on the source cards of a circuit; then we will move them, one at the time, to represent the flow of information traversing the circuit. Pegs are *objectified symbols*, that run through the cards, following a precise semantics for each card type, until they reach a pit card, where they must stop.

C-cards semantics is defined by a graph-rewriting system (Valente, 2003), where rewriting card circuits corresponds to manually animating them by moving pegs around. This semantics fits well with the concrete and interactive nature of c-cards; even very young students can easily manage games with formal rules (see Kharma et al. (2001), McNerney (2000) and Papert (1993)).

Sometimes, in problem solving, top-down reasoning is crucial, and in these cases it is

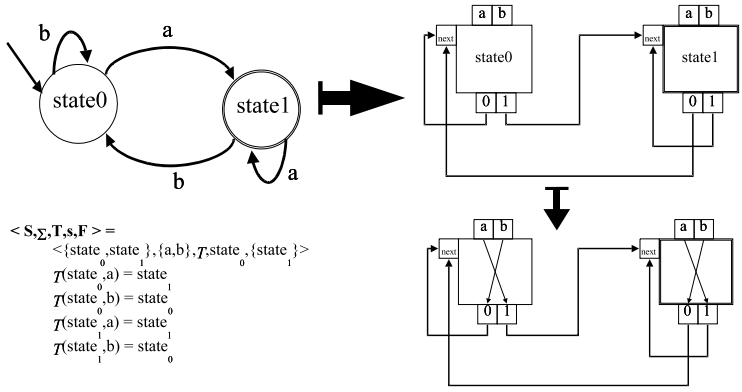


Figure 2: A DFA and its shape in c-cards

possible to sketch a card-circuit functional specification, called *shape*. In a shape, source and pit-cards are visible, but the rest of the circuit is blank; arrows can be drawn freely on blank parts to show pegs paths. Shapes are used later, to implement state machines with our cards.

Many abstract concepts, like deterministic state machines, probability and information transmission can be discussed on a concrete basis with c-cards, that, for what we know, is the only tool where both computing elements and information itself are reified. A distinctive feature of c-cards is that they show the rather abstract concept of *state* of a computing system in a very explicit way: the state of a card circuits coincides with the visual layout of its cards. To change the state, you have to change the layout (for example, by flipping one card on its the other face).

We believe c-cards have many of the features needed to encourage explorations of (theoretical) computer science concepts; moreover it is a very cheap and extendible tool. In fact we designed c-cards to have both a paper (tangible) and an html-javascript implementation; using the E-Si (pronounced easy) editor and simulator, teachers can design exercises for their courses based on traditional lectures (to visualize and animate), lab-based activities, and also for individual learning.

4 Implementing finite automata in c-cards

As an example consider the deterministic finite automaton in figure 2 on the left; its recognized language is $b^*aa^*(bb^*aa^*)^*$, and an example of accepted string is $bbaaa$. This automaton can be represented as graph with two states, connected by arrows, labelled with symbols from the alphabeth.

To implement the DFA in c-cards we can start by designing some shapes, corresponding to the automaton's states, that we later will refine into circuits. Our idea is based on the fact that a state of a DFA can be seen as a black box, waiting for an input (the next symbol provided by the user) to compute the automaton next state; such computation is directed by the DFA's transition function. Then the next state is activated and it will wait for its input.

Figure 2 in the upper-right corner, shows the card circuit that results from this idea of *state-as-black-boxes*; there are infact two shapes, labelled $state_0$ and $state_1$, each with two source cards on the top, representing the possible inputs that the state can receive (a or b). Each shape also has two possible outputs, at the bottom, labelled 0 and 1, representing the next state of the automaton; finally there is a pit-card labelled *next* at the side of each shape, that is used during the animation of the circuit as the unique entry point in the shape, something like a handle to access that state. Special colors can be used to indicate which circuit-shapes correspond to the intial state (and final states) of the automaton. The final things we need to do after defining these two shapes, is to connect them in the right way and

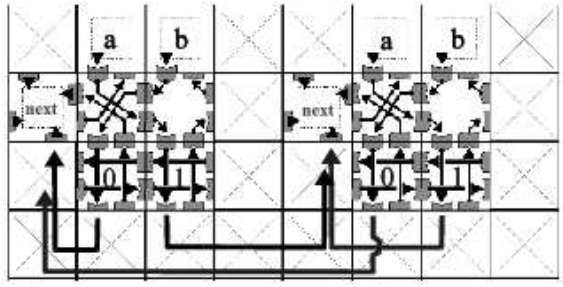


Figure 3: A possible implementation of the DFA-shape

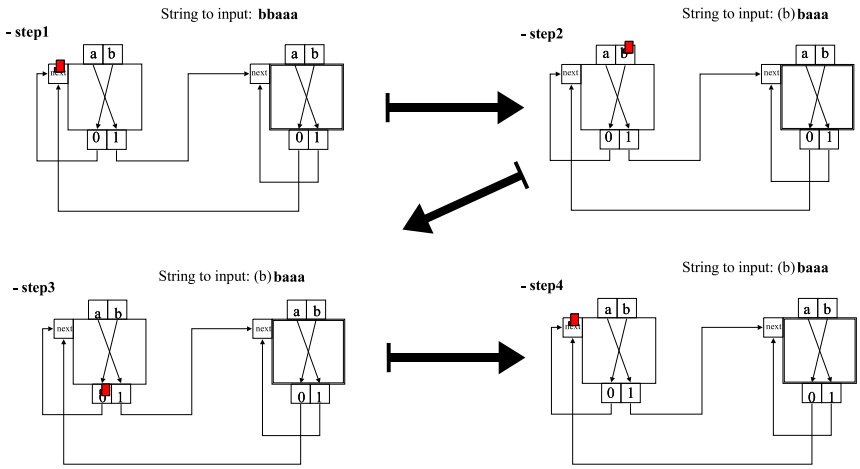


Figure 4: Initial steps of the animation of a DFA-shape

refine their internal structure, so to match the transition function of the original automaton.

A partial refinement of the states is visible in figure 2, lower-right corner, while the (almost) complete implementation of our circuit is depicted in figure 3 (some connections are still sketched, to simplify the readability of the circuit).

The implemented card circuit can then be used recognize the input string *bbaaa*; figure 4 shows the first steps of this recognition: at the beginning a peg (the only one needed to animate this circuit) is set into the *next* pit-card at the side of $state_0$, because it is the initial state for this automaton. Then the user has to manually set the peg in the source-card labelled b of the $state_0$-shape of the circuit. The shape will then answer by sending the peg out through the channel labelled 0, from where it will reach again the *next* pit-card at the side of $state_0$. At this point the user have to set the peg in the b source-card again, to simulate the reading of the second symbol of the input string. This process will go on until the string is consumed, and if the current state will then be $state_1$, then the string belongs to the language of the automaton.

The general algorithm for implementing a DFA $< S, \Sigma, T, s, F >$ in c-cards is then:

- for each $state_i$ in S, design a shape with:
  - a pit-card at the side, labelled *next*
  - a source-card on the top for each symbol of Σ
  - a cross-card at the bottom for each state in S
  - an arrow connecting the source-card $symbol_j$ to the cross-card $state_k$, where $T(state_i, symbol_j) = state_k$

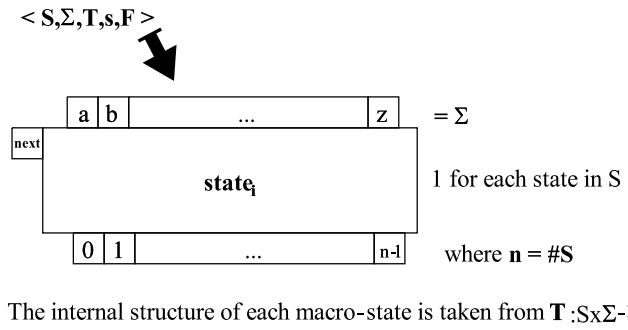


Figure 5: The general recipe for mapping DFA into c-cards circuits

- for each shape $state_i$
  - for each cross-card $state_l$ at its bottom,
    - draw an arrow connecting it to the *next* pit-card at the side of the $state_l$ shape

This mapping (visually summarized in figure 5) is easily extended to NFA. Suppose that for the state $state_i$ and a symbol $symbol_j$, the transition function of the NFA is defined like: $T(state_i, symbol_j) = \{state_a, state_b\}$. Then we just need to use a random-card to implement the non-deterministic choice between states $state_a$ and $state_b$.

5 How to present the subject to the students

A possible way of introducing students to FA (with c-cards) is then to start with remarking the importance of top-down analysis in problem solving. Then we could give the students some shapes already defined, that implement a particular automaton, maybe taken from some known contexts. For example a DFA with three states, called *home*, *school* and *holiday*, and a transition function representing the life of a student commuting among the three places. Animations could then be shown, to demonstrate how to use automata and how to *read* their results. After two or three examples, generalization can take place, and we can discuss what is needed to describe a generic FA. In this way we can lead students towards the standard definition of this class of state machines: the 5-tuple.

As for the exercises, there are a number of possibilities for presenting automata theory in an *enjoyable* way (Hämäläinen, 2003). For example students can be asked to design their own *bike lock*, with code: the lock will be modeled as an automaton and the accepted language will be the code (or a set of possible codes). In another kind of exercises we can exploit a finite automaton as string generator: the string produced can then be mapped on *colored scarfs*. Another source of inspiration could be some simplified version of ethograms (see *www.ethograms.org*), where an ethogram is a (probabilistic) automaton, used in ethology to provide an inventory of the behaviors of a species; this should strengthen the idea that automata exist *in the wild*, in the real world. Non-determinstic automata could be introduced by discussing errors and random behaviors (see the way information theory and communication are expressed in c-cards, in Valente (2003)).

6 Conclusion

In this paper we show that FA can be implemented and discussed within c-cards, thus enlarging the area of applicability of our educational tool: not only it can be used to introduce young students to the idea of computational devices, and to the tasks involved in designing, implementing and testing symbol-manipulating machines; it is also possible to build on top of that and eventually structure a course of automata theory, suitable for slightly older children.

C-cards offer in fact a plain and tangible metaphor about computation, and this results in a scalable mind-tool, that can be consistently applied from simple computer science concepts to more formal and complex issues (such as finite automata). This september, in a workshop (ICALT 2004 at Joensuu, Finland) c-cards will be tested with a large class of children (6 to 12 years old); we anticipate that feedback and data collected in that occasion will help to improve both the design and effectivness of our tool. Our long-term goal is to base a teaching methodology on c-cards, and for this reason we are currently exploring an extension to concurrency and more complex state machines, such as (approximations of) Turing machines; the latter is quite challenging, given that standard c-cards do not have enough expressive power to implement that class of language-recognizers.

References

R. Cavalcante, T. Finley, and S. H. Rodger. A visual and interactive automata theory course with jflap 4.0 . *Proc. of SGICSE'04, March 3-7, 2004*, 2004.

W. Hämäläinen. Problem-based learning of theoretical computer science. *Proc. of the Kolin Kolistelut - Koli Calling 2003, Koli, Finland*, 2003.

S. Harnad. The symbol grounding problem. *Physica*, D 42:335–346, 1990.

N. Kharma, L. Caro, and V. Venkatesh. Magicblocks: a construction kit for learning digital logic. *Computer Applications in Engineering Education*, august, 2001.

T. McNerney. *Tangible Programming Bricks: An Approach to Making Programming Accessible to Everyone*. M.S. thesis, MIT Media Laboratory, Cambridge, MA (2000), 2000.

S. Papert. *The children's machine. Rethinking School in the Age of the Computer*. Basic Books, New York, 1993.

A. Valente. C-cards: using paper and scissors to understand computer science. *Proc. of the Kolin Kolistelut - Koli Calling 2003, Koli, Finland*, 2003.

A Lightweight Visualizer for Java

John Hamer
Department of Computer Science
University of Auckland
New Zealand

J.Hamer@cs.auckland.ac.nz

1 Introduction

We report on a lightweight tool (LJV) that provides high-quality visualizations of Java data structures. The key distinguishing characteristic of the tool is its *effortlessness*. For instructors, LJV can be easily adopted without requiring any change to the teaching environment; i.e., you can keep the same editor, Java environment, code examples, etc. For users, the interface to LJV is truly simple: a call to a single (static) method that takes a single (Object) argument.

LJV works by using Java reflection to traverse the fields of an object (and the fields of the fields, etc.), generating a textual description of the connectivity as it goes. The description is then passed to the graph drawing program GraphViz (North and Koutsofios, 1994)[1]. GraphViz automatically produces an image of the graph, in a choice of bitmap or vector formats.

Each visualizations depicts a single Java object. A sequence of visualizations constitutes an animation that the user can step through, forwards and backwards, at their leisure.

LJV includes a configuration mechanism that allows broad and precise elision of detail. Users can choose to hide certain fields or groups of fields, or show entire objects in summary form.

This paper discusses the design of LJV and its intended learning objectives. While no formal evaluation of the tool has been undertaken, some classroom experiences are reported. A conclusion reflects on tools and pedagogy.

2 An illustration, and commentary

Examples of the visualizations produced by LJV are shown in Figure 1. The first diagram shows a binary (red-black) tree containing four entries. The second is a circular doubly-linked list with three entries. Both examples happen to be from the standard Java library, but this is not necessary; LJV has no built-in knowledge of any particular data structures, and will work just as well with data structures written by instructor or student.

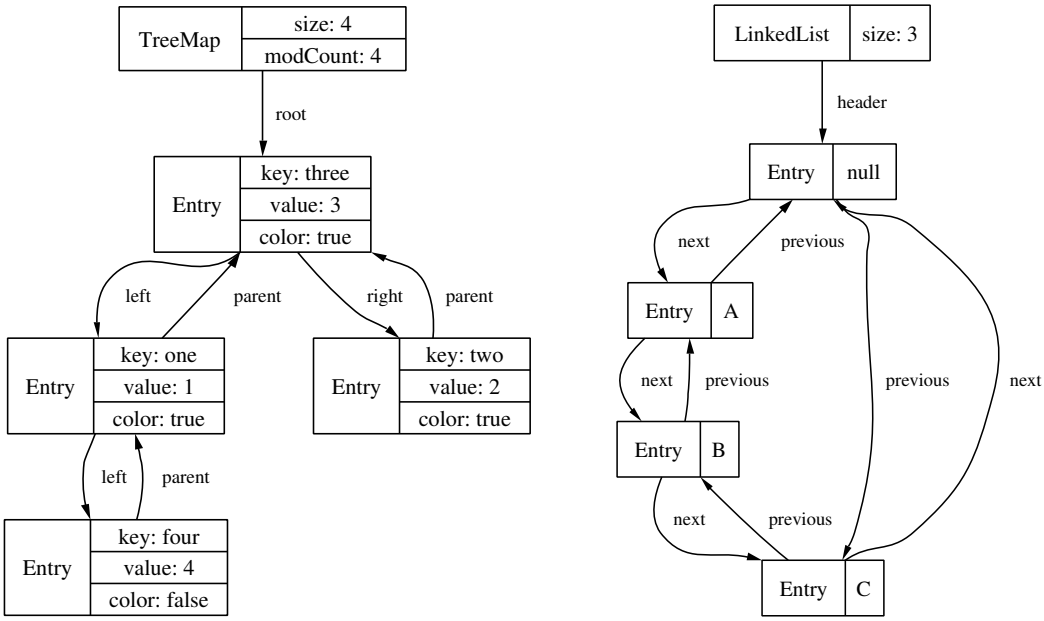
Two further observations are in order:

æsthetics Despite the logical symmetry of the data structures, the diagrams exhibit a certain amount of arbitrary variation:

- the entries in the linked list are not evenly spaced;
- some edges are straight while others are curved;
- the curved edges "wobble" in some places.

We feel that the *organic* nature of the diagrams makes them more interesting, and therefore likely to retain a viewer's attention longer than would a rigid layout. This consideration was one of the factors in choosing GraphViz as the drawing tool.

[1] GraphViz is a free, open-source program, available for all major platforms from `www.research.att.com/sw/tools/graphviz`



(a) A binary tree

(b) A linked list

Figure 1: Some example visualizations

shallow semantics The left subtrees appear to the left in Figure 1(a) by good luck. They could just as easily have appeared on the right. This "luck" ran out in the linked list diagram, where the list elements appear in an unconventional anti-clockwise order.

Our visualizer does not provide any means of hinting at a deeper semantic meaning in the data structures depicted. At times, this can lead to momentary confusion by the reader. However, the tradeoff is one of greatly simplified usage. LJV is able to draw objects without any prior knowledge of their data type, and it generally does a good job.

3 Design

In designing LJV, we have adopted the following principles:

- students must be engaged in active learning (Hundhausen and Douglas, 2000);
- the tool must be simple to use. As a guideline, setting the Java CLASSPATH has proven to be "too hard" for a significant number of students;
- avoid unnecessary features that could distract students from substantive course material (Naps, 1998);
- minimise the effort required by instructors to integrate tools into the curriculum (Naps et al., 2003);
- software must be reliable.

LJV has a number of features that distinguish it from more traditional "heavy-weight" visualizers:

- Setup is straightforward. Once an administrator has installed the GraphViz "dot" utility, a user needs only to copy a single, small (600-line) Java source file into her working directory to be immediately able to generate visualizations.

- The tool is easy to use. A user inserts calls to a static method, passing the object he wishes to display. As these methods are executed, a numbered sequence of pictures is written to disk. He can then view the animation frame-by-frame using any standard picture viewer.
- The visualizer works on any Java program. No specific programming conventions need to be followed.
- Active learning is supported, as the user must decide which objects to display, where to place the calls to draw these objects, and what to elide.
- "Wrong" data structures can be viewed (as well as correct ones). The visualizations faithfully depict actual code behavior.
- Feedback is *not* immediate. The visualizations are not seen until the program completes.

  We believe that a delay between "doing" and "seeing the result" can be important, as it provides a time for anticipation and reflection on the expected outcome[2].
- Visualizations can be easily incorporated in reports, www pages, and presentations.

4 Configuration

A *drawing context* is used to control the appearance of graph nodes and edges, to elide classes and fields, and to control the format and naming of output files.

A default drawing context can be configured to give reasonable defaults, such as ignoring private fields or treating all system classes as primitive types.

Configuration of the drawing context is ongoing. The operations currently supported include:

treatAsPrimitive The specified class is treated as a primitive value; i.e., the result of calling **toString** on the object is displayed in-line, rather than showing the object as a separate node.

This is an effective mechanism for reducing the amount of clutter in a visualisation. Most Java classes provide a **toString** method that conveys the content of the object as a string. This includes all the Java collection types (lists, maps, etc.). Ellipsis ("...") may be inserted to replace the middle of excessively long strings.

ignoreField Suppresses display of the given field; i.e., LJV pretends the field does not exist.

ignoreClass, ignorePackage Suppresses display of any field of a given class, or from a given package.

ignorePrivateFields A boolean value. If set (the default), fields that are not normally accessible are not displayed. This includes private and protected fields in other classes, and package-visible fields from other packages.

setClassAttribute, setFieldAttribute Attributes include border and font colour, font size, fill style, etc. For example, a binary tree node can be made bright pink and different colours given to the left and right links as follows:

```
Context ctx = getDefaultContext( );
ctx.setClassAttribute( Node.class, "color=pink" );
ctx.setFieldAttribute( "left", "color=red"  );
ctx.setFieldAttribute( "right", "color=blue" );
```

[2] In support, we note anecdotally that a similar sentiment can be heard from "old timers," who often claim that the delays inherent in punch-card programming led to more a more thoughtful coding process than that observed in the instantaneous environments of today.

The available attributes are determined by the GraphViz tool.

Fields can be set by name (as above) or by specific reference.

showFieldNamesInLabels Determines whether field names should be displayed in nodes or not.

qualifyNestedClassNames Nested classes are given compound names by the Java compiler; e.g., a class **Entry** nested in a class **LinkedList** will be named **LinkedList$Entry**. Normally, only the last part of these names is displayed. Setting this option results in the full name being used.

outputFormat This string field determines the output format. The default is `png`, a widely used image format. The `ps` (encapsulated postscript) format can also be used for generating high-resolution scalable graphs suitable for including in reports.

baseFileName A numeric suffix is added to give a unique name for each output graph (e.g., `graph-0.png`, `graph-1.png`, etc.) The set of graphs so generated can then be viewed as a slide show using any standard image viewer.

5 Implementations

In addition to a static Java method, LJV has been integrated into a modified debugger (Fiedler, 2004) and a Java interpreter (Niemeyer, 2004). These integrated environments provide immediate display of the visualization, and may prove useful in some learning contexts.

We note that these variations sacrifice some of our design principles. Setup is not as simple, and various constraints are imposed by the environment. We feel it is best to regard these versions as "experimental."

5.1 BeanShell integration

The BeanShell is a Java interpreter[3]. As well as being used as a stand-alone interpreter, the BeanShell has also been embedded in several popular Java development tools, including GNU Emacs (Kinnucan, 2004), Sun Forte (Microsystems, 2004), BEA WebLogic (Inc., 2004), and Apache Ant (Foundation, 2004). More information on the BeanShell can be found at `www.beanshell.org`.

One of our students (McCall, 2003) integrated LJV into the BeanShell, providing a **display** command that opens a window containing the visualization of an object. A screen dump of the system in action is shown in Figure 2. The white-on-black window in the bottom right is the BeanShell console, in which the user has typed a number of Java statements. Two **display** commands have been issued, resulting in two image frames (one for a linked list, and the other for an array list). The frames are sized to fit the image, and are refreshed after each console command.

5.2 JSwat integration

JSwat is a stand-alone, graphical Java debugger written by Nathan Fiedler (Fiedler, 2004).

We have developed an experimental integration of LJV into JSwat. The interface is through a "graph" menu command that is available when browsing values in a stack frame. Images are refreshed each time the debugger steps or stops at a breakpoint. The integration uses a viewer called Grappa (Mocenigo, 2004), which allows the visualization to be scrolled and zoomed.

[3]Technically, a Java-compatible scripting language. The language is an extended version of Java that supports dynamic typing.

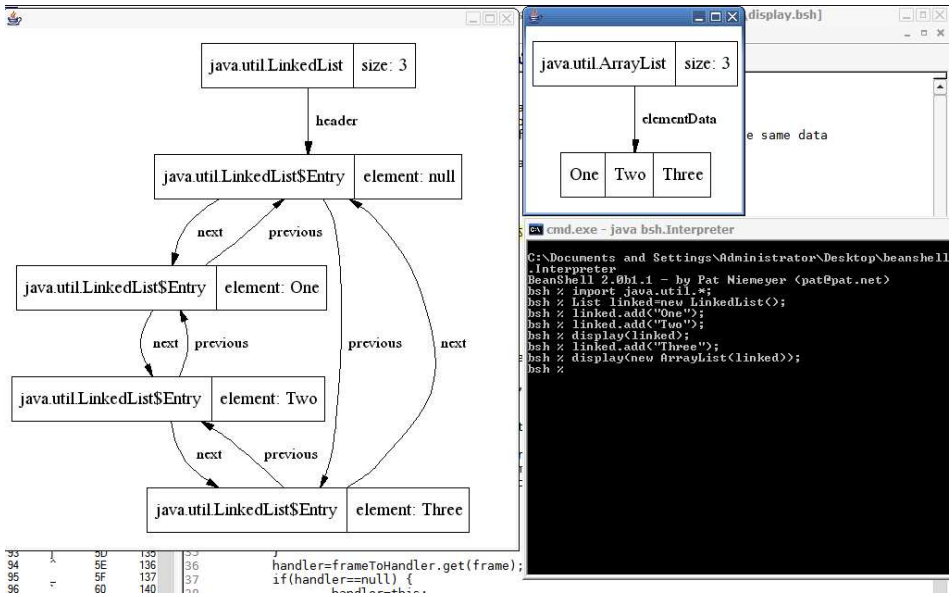


Figure 2: A session using the LJV-BeanShell integration

A screen dump is shown in Figure 3. The stack frame appears on the top left of the window, and a single Graph Viewer window holds all the images. Not shown is a configuration screen that provides access to the LJV elision settings.

Work is in progress in extending the debugger to support to direct manipulation of the graph. This includes updating data values, eliding or expanding nodes, manual adjustment to node positions, setting colour and other node and edge attributes, etc.

6 Evaluation

LJV has been deployed in three classes to date. Two of these courses were CS2-level data structures, and the other a CS1-level introductory programming course. No formal evaluations have yet been undertaken, but some anecdotal evidence has been gathered.

The tool has been used in two educational settings. First, a structured laboratory session was used to both introduce the tool and to identify some common misconceptions. Later, students were left to use or ignore the tool as they saw fit during the remaining course-work.

The laboratory presented a series of Java code fragments, for which students were asked:

> *. . . make a sketch in your engineering notebook of the graph you expect to see* before *you run the program. If you can think of more than one plausible output, sketch them all.*
>
> *If the output differs from your expectation, write down a concise summary of the difference and think about why your prediction was wrong.*

The code fragments explored various **String**s, **StringBuffer**s, one and two-dimensional arrays, and parameter passing. For example,

```
String x = "Hello";
String y = new String(x);
Dot.drawGraph( new Object[]{ x, y } );
```

Feedback from students has been positive, and no problems with the use of the system were reported. A number of students made regular use of the tool later in the course, with one even undertaking a significant extension (see Section 5.1). We intend to survey students in subsequent courses to determine whether they continue using the tool.

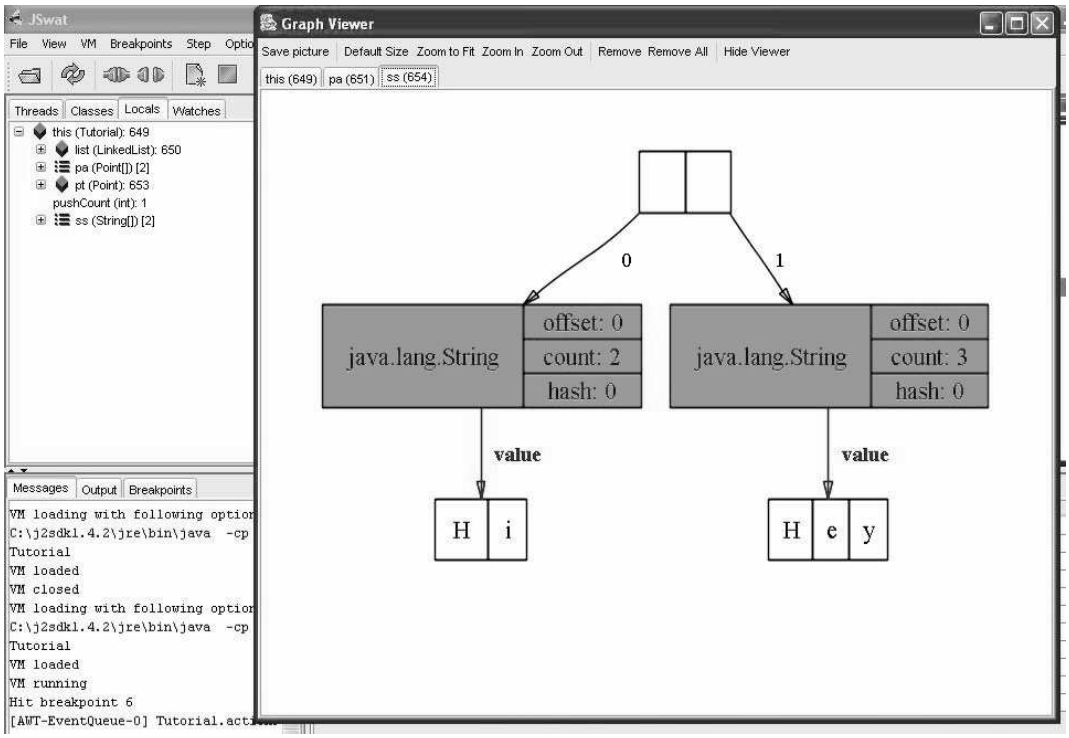


Figure 3: Exploring with the LJV-JSwat integration

7 Misconceptions

The primary benefit of the tool has been in overcoming misconceptions concerning the Java data model. We are compiling an "inventory" of common mistakes held by novices. Such misconceptions are often stubbornly persistent, and are regularly observed in students in advanced courses (Holland et al., 1997). The "inventory" includes the following:

Reference semantics Java assigns object types by reference, but primitive types by value. Many students fail to notice there is a difference, and even when told often fail to register the significance.

Primitive string Strings in Java are objects, but string constants *look* like primitive values. The "primitive string" misconception is the assumption that strings have value semantics.

Object arrays Object arrays hold references, not values. Students sometimes assume that because arrays of primitive have value semantics, all arrays do so.

2D arrays 2-dimensional arrays are constructed from 1-dimensional arrays. Arrays thus have several uninitialised states: a null reference, or a reference to a null array block. Arrays can also be jagged, so the second dimension must be determined on a row-by-row basis. These subtleties are often missed.

Static field Static fields are not part of any object, yet they share the same scope rules as object fields. This manifests in students sprinkling the `static` keyword through their code seemingly at random. The misconception is reinforced when programs compile and "appear" to work.

Inheritance A broad category that needs further refinement. The central issue is that inheritance means objects are often not the same as their declared types. The misconception leads students to use concrete types (**LinkedList**, **Vector**, etc.) in preference to an interface type (say, **List**).

Identity/attribute confusion This is another misconception category, from (Holland et al., 1997). It manifests in various false beliefs, such as "only one variable can reference to a given object at a given time" and "if you have two different variables, they must refer to two different objects."

LJV proved helpful in alerting many students to the fact that they held misconceptions. By turning off all elision, students are presented with visualizations that accurately reflect the Java data model, including deep sharing.

8 Related work

Pedagogically, Naps's `Visualizer` class (Naps, 1998) shares much in common with the work presented here, with both approaches emphasising the need for a tool simple enough for students to use without unnecessary distraction. Our visualizer is more general, does not require any special configuration to handle new data structures, and is easier to use. Naps's approach supports abstract presentations of data (e.g., bar charts for integer arrays) and supports arbitrary customisation.

Another approach to supporting abstract presentations was taken by Korn and Appel (1998), who developed a tool for specifying many kinds of visualizations of Java programs in a declarative pattern-action notation. For example, the tool can produce a visualization for a parse tree that uses bitmap images for the internal nodes (e.g., a large "plus" symbol for an addition node). The notation is expressive enough to construct quite sophisticated diagrams. However, it does not resemble Java code, and would require a considerable effort for students to master. To hide the notation from novice users, a "visual pattern constructor" was provided. The authors report some success in using the system in a compiler course.

JavaViz (Oechsle and Schmitt, 2002) can display UML object and sequence diagrams of running Java programs. The object diagram layout is simplistic, and looks similar to DDD. Some support is provided for visualizing concurrent threads in the sequence diagrams. No elision support is provided.

DDD is a widely used graphical debugger (Zeller and Lütkehaus, 1996) that runs on Unix platforms. The system supports a notation for specifying how to display the contents of a node, including nested layout of aggregate structures. References are expanded manually, and the system supports a simplistic automatic layout that will redisplay the current graph. Shared references are not recognized, which can lead to diagrams in which a single object appears in multiple places.

9 Conclusions and future research

Effort has been identified as a major impediment to the uptake of visualization tools for teaching. Many tools come attached to large or hermetic environments, and cannot be easily incorporated into an existing teaching setting. LJV's primary distinction is the very low effort required to install and run, and its ability to integrate easily into virtually any Java environment.

Most of the tools that operate on arbitrary (student) Java programs provide limited support for controlling the amount of detail displayed. The elision controls provided by LJV offer a powerful solution to this problem, and should prove adequate for most introductory courses.

The usefulness or otherwise of a visualization tool is ultimately determined by how widely it is adopted. Seeing students continue to use LJV after the end of formal instruction is evidence that the tool has real educational value. However, more formal research is needed to drive further development. For example, we have no evidence as to whether the delayed feedback model contributes to improved learning over a debugger model that provides instant feedback. A formal study comparing plain LJV to the JSwat integration may shed some light on this question. We also note a perceived demand from educators for tools that can provide

abstract presentations of data structures. It remains to be seen whether LJV can be extended to support the necessary transformations without sacrificing its essential simplicity.

References

Nathan Fiedler. JSwat, a Java debugger, April 2004. URL `www.bluemarsh.com`.

Apache Software Foundation. Apache Ant project, August 2004. URL `ant.apache.org`.

Simon Holland, Robert Griffiths, and Mark Woodman. Avoiding object misconceptions. In *SIGCSE'97 Twenty-Eighth Technical Symposium on Computer Science Education*, pages 131–134, San Jose, 1997. ACM Press. ISBN 0-89791-889-4.

Christopher Hundhausen and Sarah Douglas. Using visualizations to learn algorithms: Should students construct their own, or view an expert's? In *IEEE International Symposium on Visual Languages*, pages 21–30, September 2000.

BEA Systems Inc. BEA WebLogic platform, August 2004. URL `www.bea.com`.

Paul Kinnucan. Java development environment for Emacs (JDEE), August 2004. URL `jdee.sunsite.dk/`.

Jeffrey L. Korn and Andrew W. Appel. Traversal-based visualization of data structures. In *Proceedings IEEE Symposium on Information Visualization*, pages 11–18. IEEE Computer Society, 1998. ISBN 0-8186-9093-3.

Sam McCall. private communication, October 2003.

Sun Microsystems. Sun Java studio, August 2004. URL `www.sun.com/forte/`.

John Mocenigo. Grappa: A Java graph package, August 2004. URL `www.research.att.com/~john/Grappa`.

Thomas Naps. A Java visualizer class: Incorporating algorithm visualizations into students' programs. In *ITiCSE'98 Innovation and Technology in Computer Science Education*, pages 181–184, Dublin, Ireland, August 1998.

Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *ACM SIGCSE Bulletin*, 35(2):131–152, 2003. ISSN 0097-8418.

Pat Niemeyer. BeanShell: Lightweight scripting for Java, April 2004. URL `www.beanshell.org`.

Stephen C. North and Eleftherios Koutsofios. Application of graph visualization. In *GI'94 Graphics Interface*, pages 235–245, Banff, Alberta, Canada, 1994. URL `citeseer.nj.nec.com/221206.html`.

Rainer Oechsle and Thomas Schmitt. JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java debug interface (JDI). In S. Diehl, editor, *Revised Lectures on Software Visualization, International Seminar*, volume 2269 of *LNCS*, pages 176–190. Springer-Verlag, 2002. ISBN 3-540-43323-6.

Andreas Zeller and Dorothea Lütkehaus. DDD — a free graphical frontend for Unix debuggers. *SIGPLAN Notices*, 31(1):22–27, January 1996.

Program state visualization tool for teaching CS1

Otto Seppälä
Helsinki University of Technology
Department of Computer Science and Engineering
Helsinki, Finland

oseppala@cs.hut.fi

1 Introduction

To understand a natural language one must first understand the vocabulary, the meaning of each word. When the vocabulary is combined with grammar, it becomes possible to explain more complex concepts. However, this is usually not enough. To understand the whole meaning of any piece of text or speech, it is crucial to know the frame of reference. The same words carry a different message at different times and in different places. The same applies to computer programs, but with a fundamental difference: the frame of reference is different for the same line of code each time that single piece of code is executed. The programmer thus has to understand how the runtime state of the program changes when the program is executed.

In imperative or procedural programming this frame of reference is easier to understand, as using only local and global variables is equal to object-oriented programming within a single object. In object-oriented programming we have multiple objects and thus the visible variables can change as a result of each method call. Before the learner fully adopts the object-oriented way of thinking, following the program state can require a lot of attention. When we conducted a questionnaire about programming habits on our main programming course, 43 percent of the students claimed that they spent most of their time trying to make their programs conform to exercise specifications or trying to fix runtime errors. This implies that understanding of the dynamic state of the program should be given more attention. This view is supported by Mulholland and Eisenstadt (1998), who found that in most of the cases the best way to help the students with their programming problems was just to show them how their programs actually worked. This is essentially something that can be done with a debugger.

Holland et al. (1997) propose using counterexamples to avoid object misconceptions. We believe that such examples will be more powerful using a program state visualization tool to support the process.

2 Background

Using a debugger for lecture demonstrations or as a supportive tool when making exercises has been found to be an effective way of explaining how a program executes. Cross et al. (2002) used the *jGRASP* debugger as an integral part of their CS1 course and reported being surprised by the positive response from the students. Their hypothesis is that the added complexity in object–oriented programs compared to procedural programming impedes the students' understanding. This complexity is lowered by using a debugger to step through the programs, revealing layer by layer how the program state changes.

The debugger was used on lectures to explain programs. It was reported that using the debugger both motivated students and made lecturing easier. jGRASP has all the functionality of a typical debugger and also creates control structure diagrams to help in program understanding. Cross et al. (2002) conclude that any modern debugger could be used on lectures to better explain programming concepts.

Different debuggers and learning environments have their specific strong and weak points. The educational debugger/animator, *Jeliot 2000* (Levy et al., 2001), is well suited to teach the basics of procedural programming, but rejects Java programs that use any object-oriented features. With the growing tendency to teach object-oriented programming objects-first, this is a bit troublesome.

Kölling and Rosenberg (2002, 2001) describe how the *BlueJ* environment can be used to effectively teach object-oriented programming. BlueJ has been designed to lower the introductory level to programming by eliminating the use of many relatively complex language constructs that have to be written to try out a single method. BlueJ also draws an UML-like diagram of the class hierarchy of the program.

One crucial problem with BlueJ is that it does not display the references between objects. Another is that it assigns names to objects. This easily leads to misunderstanding, as students are known to conflate variable names with object identities (Holland et al., 1997). We believe that a diagrammatic notation without any artificial naming is less likely to create such misconceptions.

Such diagrams are created at runtime by the free *Data Display Debugger (DDD)* (Zeller and Lütkehaus, 1996) which makes beautiful visualizations of references between data structures. It also has a huge amount of different visualizations for displaying how data evolves during the program. DDD can be used for lecture demonstrations and student use the same way as Cross et al. (2002) used jGRASP. We feel that the diagrams explaining the references between the data structures would make it even more useful in explaining the concept of reference variables and showing how the state of each instance evolves during program execution. DDD and jGRASP use a separate window for displaying the execution stack. In a procedural programming environment, this is usually enough, but in object-oriented programming the stack is bound to have a lot of references to objects. A special case is the *this*-object on which the method call was made. The *VizBug* system (Jerding and Stasko, 1994) uses arrows to represent method calls. These arrows emanate from and end on visual elements representing both instances and global (static) methods. Doing so, the arrows essentially reveal all the "this"-objects currently in the stack. A similar notation, the collaboration diagram, which can be used to describe the order of method calls between instances is found in UML (The Object Management Group, 2003).

Vizbug does not visualize references between objects. However, the system visualizes the inheritance hierarchy in the same diagram. This was done to better display ideas of polymorphism and inheritance when methods and constructors are invoked.

One system still worth mentioning is the JAVAVIS (Oechsle and Schmitt, 2002). JAVAVIS visualizes the program state by showing in separate windows the contents of each of the stack frames currently in the stack. This information is shown with an object-diagram-like notation, essentially revealing which information could be referenced through the objects in the stack. JAVAVIS also uses sequence diagrams to better show the control flow during the execution.

As told before, our goal was to find a system that would visualize both the method invocations and references simultaneously in the same graph. While many systems came close, such a tool was not be found. The author therefore constructed a program visualizer/debugger for testing this concept in computer science education. This debugger uses a diagram notation we call the *program state diagram*. This notation combines some features found in visualizations constructed by the VizBug and DDD debuggers. Our visualization closely resembles UML, but with intentional differences where we do not want the learner to confuse the two.

The interface of the visualization tool is intentionally simple, containing only the basic stepping and running operations for traversing the program execution. There are two views to the debugged program, one showing the diagram and the other displaying the source code.

3 Program State Diagrams

Combining features from both the object diagram and collaboration diagram, we come up with a new notational scheme: the program state diagram. This notation attempts to show most of the runtime state of the program in a single diagram. Essentially, this means displaying all relevant instances, all references to them and some of the contents of the runtime stack together.

The diagram displays instances as white rounded rectangles and references between them as yellow arrows. The name of the referring variable is shown next to the arrow.

Method calls, their parameters and local variables are shown in the same diagram using a visualization style that sets them apart from the instances and references. In our approach, the method calls are depicted in the same diagram using red curved arrows to better separate them from the reference arrows. Arrows have been used for visualizing method invocations for example in VizBug and in the UML collaboration diagram.

Static and local reference variables are also visualized. As these cannot be seen to belong with any specific instance, they are shown with separate ellipsoids. Local references have a blue background and static references purple. The objects these variables reference are shown with reference arrows in the same style as used to show references between instances. Static method calls use a notation similar to that used in VizBug. A static method is shown as a separate rounded rectangle with a green background color to distinguish it from instances. All of these code elements (with the exception of a static variable) are used in the example given in the next section and thus the corresponding visual elements are present in Figures 1 and 2.

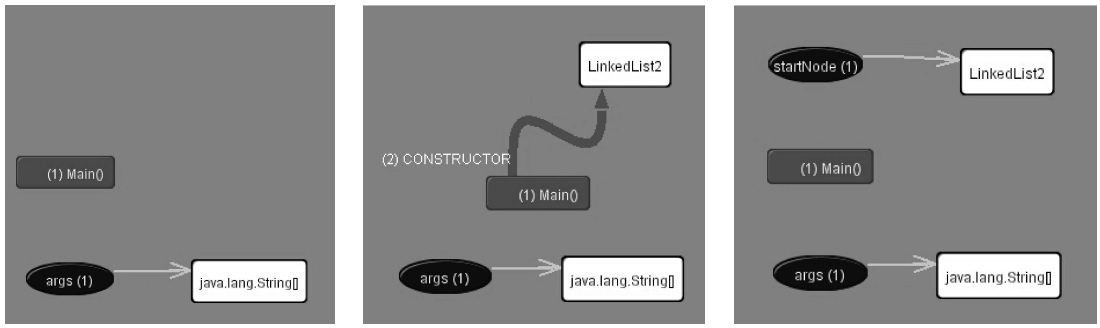
4 Usage

Recursion and reference variables are the source of many elementary misconceptions, because they are somewhat hard to explain verbally. Visualizing recursion is quite easy using the program state diagram. Three diagrams created by the debugger are shown in Figure 1. We will now explain how to interpret the notation by studying these figures and the example program they were generated from. The example program we are debugging will construct a doubly linked list with four nodes. The program code is shown in Tables 1 and 2.

As with any Java application, on the bottom of the runtime stack lies the method *main*. In our example this method creates the linked list by adding new nodes one by one and linking them to the end of the list using a recursive method call. In the leftmost image, we see the beginning state with only the *main*–method and its command line parameter *args*. The middle image shows the *main*–method invoking the *LinkedList2* constructor. Constructor calls are visualized with red arrows, similar to method calls. On the right we see the program state just before the loop which adds the next three nodes. This image shows another example of a reference by a local variable.

We now skip a number of steps to reach Figure 2, which shows the program state when adding the last node to the linked list is nearly finished. The recursion progressing through the list is clearly visible, as are the parameters given to the method invocations. The new node is given as parameter *link* for all the other calls but the last. The latest method call made was *setPrevious*, invoked on the *LinkedList* instance on the far left. The parameters and local variables for this call can be found by reading the stack height, which is shown both next to method arrows and local variable names. After this method call ends, its arrow retracts to its starting position. In the image we can now also see references between the instances that create the linked list from the separate nodes.

The debugger also has the typical prettyprinted code window where the currently executing location in the code is shown with emphasized background color. These two views should support each other, both aiding in understanding the runtime behavior of computer programs.



(a) Initial state (b) Constructor call (c) Before the loop

Figure 1: The three first steps of our debugging session

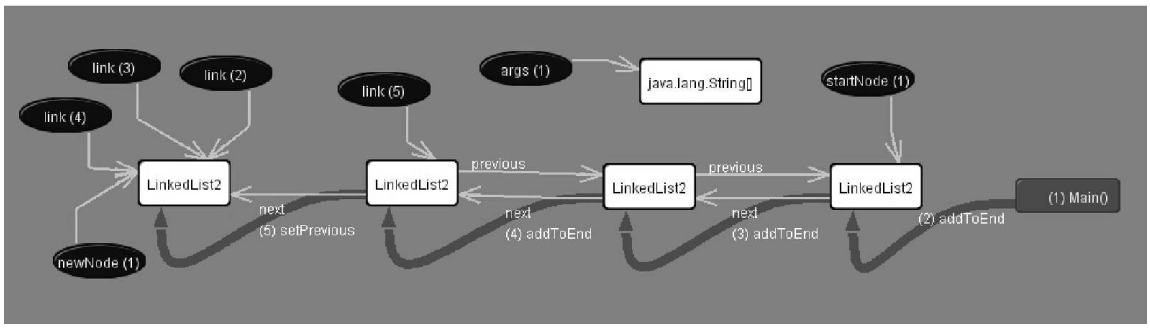


Figure 2: Program state after a number of steps

```
public class LinkedList2 {
   private LinkedList2 next;
   private LinkedList2 previous;
   private int value;

   public LinkedList2(int i) {
      next = null;
      previous = null;
      value = i;
   }

   public void addToEnd(LinkedList2 link) {
      if (this.next != null)
         next.addToEnd(link);
      else{
         this.next = link;
         this.next.setPrevious(this);
      }

      value++;
   }

   public void setPrevious(LinkedList2 link) {
      this.previous = link;
   }
}
```

Table 1: LinkedList2.java

```
public class Example{

   public static void main(String[] args) {
      LinkedList2 startNode = new LinkedList2(0);

      for (int i=1; i<4; i++) {
         LinkedList2 newNode = new LinkedList2(i);
         startNode.addToEnd(newNode);
      }
   }
}
```

Table 2: Example.java

5 Discussion

We have created a new tool for programming education for novices. This tool can be used to visualize some otherwise complex concepts such as recursion, references, stack etc. We plan to use the tool during lectures to better explain program examples. Experiences and examples of using a visualization tool for explaining program examples during lectures can for example be found in Cross et al. (2002).

The true challenge lies with the students using the program on their own to explore both lecture examples and their own programs. We have a hypothesis that seeing how variables change their values and how program execution proceeds through the code from line to line should help in building a correct mental model of program execution. Nevertheless, we can never be certain what students think caused the behavior they saw. One possible approach to evaluation of the tool in student use is to use a questionnaire that is filled while observing the execution of simple program. The questions probe typical misconceptions related to objects and classes as well as those related to references between objects. Our hypothesis of the results is that the comparison between students that try the exercise with or without the tool should show differences in the train of thought.

Evaluating the effect of using the tool on lectures is also not straightforward. As the tool essentially changes how examples are presented and code examined, it is hard to say which part of the observed effect is due to the tool and which is due to other changes in the style of presenting subjects or changes in how the time is divided between topics. While we cannot directly evaluate learning results, we can still collect student impressions using anonymous surveys as suggested in (Naps et al., 2003).

To what extent the program supports understanding the runtime state of the program remains to be seen. We aim to introduce the tool on our programming course in Fall 2004.

References

James H. Cross, T. Dean Hendrix, and Larry A. Barowski. Using the debugger as an integral part of teaching CS1. In *32nd ASEE/IEEE Frontiers in Education Conference*, volume 2, pages F1G–1–F1G–6. IEEE, November 2002.

Simon Holland, Robert Griffiths, and Mark Woodman. Avoiding object misconceptions. *ACM SIGCSE Bulletin*, 29(1):131–134, 1997.

Dean F. Jerding and John T. Stasko. Using visualization to foster object-oriented program understanding. Technical Report GIT-GVU-94-33, Graphics, Visualization and Usability Center, College of Computing, Georgia University of Technology, Atlanta, 1994.

Michael Kölling and John Rosenberg. Guidelines for teaching object orientation with java. In *The Proceedings of the 6th conference on Information Technology in Computer Science Education*, pages 33–36. ACM, 2001.

Michael Kölling and John Rosenberg. BlueJ - the Hitch-Hikers Guide to Object Orientation. Technical Report 2, The Maersk Mc-Kinney Moller Institute for Production Technology, University of Southern Denmark, September 2002.

Ronit Ben-Passat Levy, Mordechai Ben-Ari, and Pekka Uronen. An Extended Experiment with Jeliot 2000. In *Proceedings of the First Program Visualization Workshop*, pages 131–140. University of Joensuu, 2001.

Paul Mulholland and Marc Eisenstadt. Using software to teach computer programming: Past, present and future. In John Stasko, John Domingue, Marc Brown, and Blaine Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 10, pages 399–408. MIT Press, 1998.

Thomas Naps, Guido Rössling, Jay Anderson, Stephen Cooper, Wanda Dann, Rudolf Fleisher, Boris Koldehofe, Ari Korhonen, Marja Kuittinen, Charles Leska, Lauri Malmi, Myles McNally, Jarmo Rantakokko, and Rockford Ross. Evaluating the educational impact of visualization. *SIGCSE Bulletin*, 35(4):124–136, December 2003.

Rainer Oechsle and Thomas Schmitt. Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface(jdi). In Stephan Diehl, editor, *Software Visualization, State-of-the-art survey*, pages 176–190. Springer, 2002.

The Object Management Group. *Unified Modeling Language (UML) version 1.5*, 2003. Available on a webpage : `http://www.omg.org/technology/documents/formal/uml.htm` (last checked September 29^{th} 2004).

Andreas Zeller and Dorothea Lütkehaus. DDD—A Free Graphical Front-End for UNIX Debuggers. *ACM SIGPLAN Notices*, 31(1):22–27, 1996.

Application of Helix Cone Tree Visualizations to Dynamic Call Graph Illustration

Jyoti Joshi, Brendan Cleary, Chris Exton
Department of Computer Science and Information Systems
University of Limerick
Ireland

Brendan.Cleary@ul.ie

Abstract

We describe a tool that enables users to record and visualise runtime behaviour of software applications developed in Java. The execution trace, stored in the form of an XML file is visualized using 3D call graphs that are an extension of the Cone Tree information visualisation technique. This tool gives the user the ability to create several call graph views from a program's execution trace, providing additional representations of the program execution to both novice and expert programmers for the purposes of program execution analysis.

1 Introduction

As Java programs get larger and more complex, they become more difficult to understand. The maintainer, who tries to understand a program, reads some code, asks questions about this code, conjectures answers, and searches the code and documentation for a confirmation of the conjectures. In developing this prototype tool our primary goal is to help maintainers to easily verify conjectures on the runtime behaviour of the program with respect to the program structure. Our project focuses on tracing the dynamic behaviour of Java programs and using innovative 3D graph visualizations to achieve these goals. Our tool is functionally divided into two components namely: the Java Trace Generator (JTG) and the Call Graph Viewer (CGV). The JTG records program execution events in an XML format. The CGV viewer provides a 3D visualization media to which a program's execution trace, derived using the JTG, can be rendered. The next section presents related work; section 3 presents work on the JTG and its implementation details. Section 4 then presents the CGV and its implementation.

2 Related Work

A number of visualization tools exist that focus on program execution and a few of the more notable recent examples are Jinsight (IBM Research), IsVis (Jerding and Rugaber, 1997), JavaVis (Oechsle, 2001).

Jinsight is a tool for visualising the execution of Java Programs developed by IBM. Jinsight offers functionality for collecting Java program trace and also provides an environment for visualising program execution. It includes various views for examining several different aspects of run-time behaviour.

The IsVis visualization tool supports the browsing and analysis of execution scenarios. A source code instrumentation technique is used to produce the execution scenarios. In IsVis, the dynamic event trace can be analysed using a variation of Message Sequence Charts called Scenario View.

JavaVis monitors a running Java program and visualizes its behaviour with two types of UML diagrams for describing dynamic aspects of the program, namely object and sequence diagrams. It is suitable for small sized programs.

VisiVue (VisiComp Inc) is a runtime visualisation tool that dynamically displays the operations of any Java program at runtime, creating animated diagrams of the data structures and providing a complete trace of the programs execution. This tool is particularly effective in the Java educational environment and for smaller applications.

Reiss (Reiss, 2003) has developed a dynamic Java visualizer that provides a Box display view of the program and attempts to provide high- level program-specific information in real time.

Our tool is different from the above in that it makes use of a variation of the Cone Tree visualisation technique called the Helix Cone Tree for visualising program execution, which we feel provides a better technique for viewing a large amount of hierarchical, sequential data simultaneously.

3 Java Trace Generator

Program execution monitoring can be divided broadly into two categories: time driven monitoring and event driven monitoring (Jain, 1991). Time driven monitoring also known as sampling observes the state of the monitoring system at certain time intervals. Sampling however provides only summary statistical information about program execution. Event driven monitoring reveals the dynamic behaviour of program activities by events (Oechsle, 2001) as we require behavioural information not just snapshots, the JTG implements an event driven monitoring approach. The Java Trace Generator architecture (Figure 1) comprises the following elements:

- Debugger - The Debugger represents the starting point of the Java Trace Generator, it launches a primary Virtual Machine (VM) then launches a secondary VM which executes the Debugee whilst maintaining a connection to the primary VM. The Debugger also reads the event filter configuration file and sets the filter options for event requests. Events can be of one of the following types: Class Prepare Event, Class Unload Event, Method Entry Event, Method Exit Event, Thread Start Event, Thread Death Event and Exception Event.

- Debugee - This component represents the target application to be monitored. This is written in standard Java and compiled with '-g' option to facilitate the generation of all debugging information.

- EventObserver - The functionality of this element is to act as a main monitoring entity. It is event driven in operation and continuously monitors the EventQueue from the secondary VM; listeners are activated once a new event is received.

- Text Listener - This component transforms event information received from EventObserver according to an XML format and outputs it to the associated file. This XML debug trace is the intermediate medium of information exchange between the JTG and CGV.

The JTG is implemented using the JDI layer of the JPDA platform (Sun Microsystems, b). JPDA was developed to provide an infrastructure to build end-user debugger applications for the Java Platform, the JDI provides introspective access to a running virtual machine's state, class, array, interface, and primitive types, and instances of those types. The JTG using the JDI provides us in an XML format the behaviour of a java program, it's the task of the CGV to visualise and present that behavioural information to the user.

4 Call Graph Viewer

In theory, every problem can be encoded as a graph problem (Collberg et al., 2003). Call graphs represent execution relationships between discrete sections of software and are commonly used in aiding programmers understanding of code design and execution behaviour. There are two types of call graphs namely, Static Call Graphs and Dynamic Call Graphs. Call graphs generated by our tool fall into the second category as they are computed from

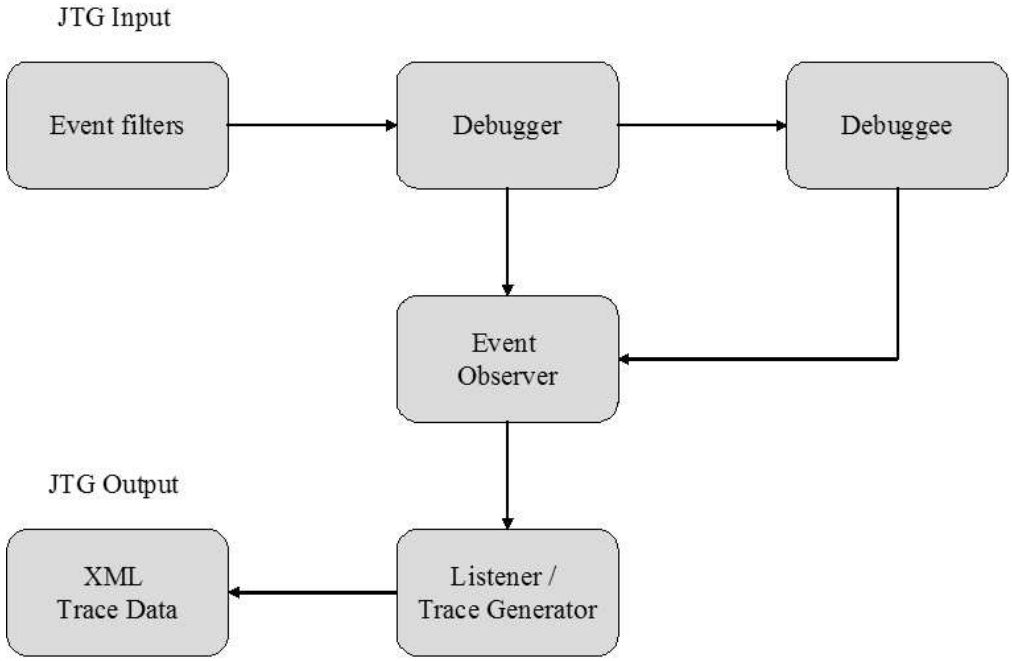


Figure 1: Java Trace Generator Architecture

the trace containing runtime execution details of the Java program. The Call Graph Viewer (CGV) provides the user with the ability to visualise these runtime execution details using a Helix Cone Tree.

4.1 The Helix Cone Tree

2-Dimensional call graph representations, while being an extremely useful and intuitive graph for the representation of small execution traces; are highly restricted in their ability to efficiently represent large data sets in finite display space. The Cone Tree (Robertson et al., 2003) graph visualization technique uses a 3-Dimensional rendering space to display hierarchies of information. The interactive nature of Cone Trees and this expansion of the traditional tree layout into the third dimension provide a significant improvement in the quantity of data displayable in a finite display space. Cone Trees however are not without their problems (Cockburn and McKenzie, 2000) specifically the occlusion of nodes and text labels within sub graphs.

The Helix Cone Tree (Figure 3) in an effort to alleviate these problems cuts and stretches the base of a cone along the y axis into a constrained helix thus reducing the amount of occlusion of nodes and text labels in a particular sub graph. This altering of the vertical positioning of nodes within a sub graph in a predictable order introduces an additional dimension of information to the traditional Cone Tree. This additional dimension can be mapped to various attributes of a dataset but in the CGV is used to represent the implicit chronological element present in program execution data.

4.2 Call Graph Viewer Technology

The CGV is an implementation of the CHIVE program source visualisation framework (Cleary and Exton, 2004). The CHIVE has at its core a customisable visualization framework that caters for the 3D visualization and manipulation of generic hierarchical data structures. In designing the CHIVE to satisfy the requirements of an adaptable, reusable visualisation framework it was partitioned into 3 subsystems each responsible for an aspect of the visualisation task (Figure 2).

- The Data Model provides a platform for presenting independent datasets in terms of nodes and relationships to the graph drawing and layout algorithm subsystem. The separation of the data model from problem domain specifics allows the user the latitude to quickly develop and modify algorithms for extracting the data and relationships independent of the data model implementation. The data model itself is an extension of the DefaultTreeModel class, taken from Java Swing, which is a common, highly documented generic tree data structure.

- The Graph Drawing and Layout Subsystem takes as input a data model and outputs a visualisation based on the application of a graph layout to that data model. This separation of the graph layout algorithm from the data model allows multiple layouts to be applied to a given data model resulting in many different visualisations, one such layout algorithm as used in the CGV is the Helix Cone Tree. Along with the relationships defined by the layout, essential to any graph visualisation is the information expressed by the nodes of that graph; this comprises not only the user data associated with the node but also the visual representation of that node. In the CHIVE this appearance information of a node is defined by a glyph (Telea et al., 2002). Glyphs encapsulate all the visual aspects (geometry, appearance and text components) of a node including any interactive behaviour.

- The Graph visualisation, Interaction and Manipulation Environment is responsible for providing an anchor for visualisations in a 3D rendering environment and for mediating user interaction with visualisations in that environment. To accomplish these goals it provides a Java 3D (Sun Microsystems, a) universe to which the graph layout subsystem renders its visualisations and also various interaction behaviour components. These interaction behaviour components can be used locally within visualisation and also globally within the environment. When used locally behaviours such as manual and animated rotation, selection and pruning effect only the select node and the sub graph rooted at that node. When used in a global context they effect the visualisation as a whole.

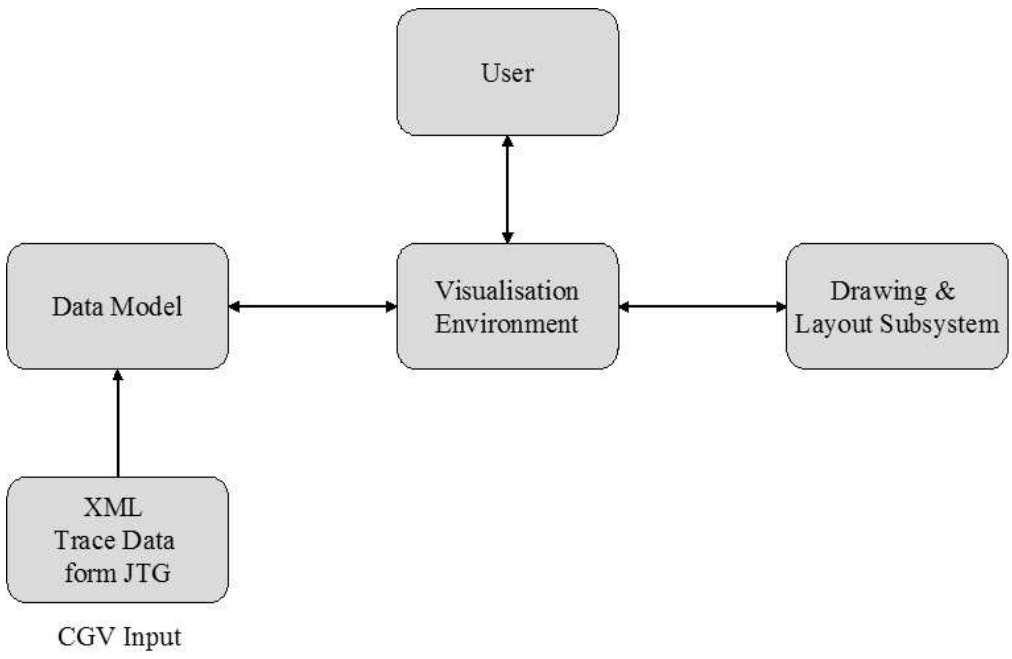


Figure 2: CHIVE Architecture

4.3 3D Call Graph Viewer Implementation

For an execution trace to be visualised that trace data needs to be transformed into the data model acceptable by graph drawing and layout subsystem. This data model serves as the interface between the CGV and the JTG. The different sets of relations that can be defined in the data model specify the different types of views (described next) in the form of call graph trees that can be displayed to the user. Call graphs however generally contain cycles. Traditional 2-Dimensional call graphs represent cycles using additional links between nodes; this however would violate the simplicity of the Cone Tree layout and render its advantages in terms of graph clarity obsolete. Cone Trees are at their most effective when displaying simple hierarchies without cycles, so in our implementation we chose to repeat nodes where a cycle occurs. This solution however, due to the size of tree produced, is not viable when very deep cycles are encountered.

4.4 Call Graph Views

The CGV lets you work with the trace information through various views, each depicting different aspects of your program's objects and execution sequence.

- The CGV's Main View displays a complete call graph tree with all execution details. Here the root node represents start of process, the first level of child nodes represent all the thread start events from the program and subsequent levels indicate start of method and end of method events. Different types of event nodes are represented by different glyphs. On selection of a node, the information pane of the user interface gives detailed information about that node. For example if a node for a method start event is selected, the selected method details displayed in the information pane will include; method name, class name, thread name, method arguments and time at which method was called. Figure 3 shows the main view of the CGV.

- The Thread View allows the user to select one or more threads from the list of threads executed through out the program execution, this view then provides a Thread Execution Tree for the selected thread or threads.

- The Object Instantiation View allows the user to select a root object, the system then provides the user with a view of the object instantiation sequence derived from that root object. Figure 4 shows the thread and object views of the CGV.

Finally the user can interact with these views in a variety of different ways; for example the user can select any of the above views for visualisation of trace data, when selecting a node the node details are displayed in the information pane, the user can select a node and get a highlighted view of sub tree starting from the selected node and the system allows the user to browse through previously visited views.

5 Conclusions and Future Work

In this paper we present a program visualisation tool that is based on the combination of technologies we have developed for capturing the runtime behaviour of java programs and the visualisation of hierarchical, sequenced data.

The Java Trace Generator (JTG) captures the runtime behaviour of executing java programs by implementing an event driven monitoring approach and exports that behavioural information as an XML execution trace. The Call Graph Viewer (CGV) represents the execution traces generated by the JTG as Helix Cone Trees and provides an environment for the user to interact with and inspect several views of the subject programs execution behaviour.

Significant future work includes the synchronization and tighter integration of the JTG and CGV. This involves the concurrent generation of call graph views as the monitored program is

being executed and changing monitoring program parameters from the CGV to alter the future execution sequence of the monitoring program for different purposes like better performance, memory management and exception handling. Other important areas of future work include the development of new prototypes that expand on the number of views and representations used, evaluation of these prototypes and the integration of the JTG and CGV into a popular Integrated Development Environment (IDE).

Given the system's modular design it is hoped that new visualisation techniques can be easily applied to the existing trace mechanism to quickly develop future prototypes. As for the evaluation of these prototypes several issues need to be overcome, including the ability of the technologies to scale to larger program sizes, the validity of the experiments and the environment in which they are performed and the selection of participants to the experiments.

Work is currently underway on the development of a CHIVE eclipse plugin, we hope to capitalise on this work in integrating the CGV and JTG with the eclipse IDE (The Eclipse Project). We see integration of our tool with a popular and modern IDE as being key both to the successful adoption of our tool and also in providing a realistic environment in which to perform experiments. Tools succeed in being adopted by conveying a perception of usefulness and increased efficiency (Zayour and Lethbridge, 2001). By integrating our tool with a successful IDE such as eclipse we hope to reduce the burden of switching between applications on the user and capitalise on user and participant familiarity with the IDE encouraging the use of our tool and also providing a more realistic context for conducting experiments.

References

B. Cleary and C. Exton. Chive - a program source visualisation framework. In *Proceedings of the 12th International Workshop on Program Comprehension (IWPC 04)*, pages 268–269, Bari, Italy, June 24-26 2004. IEEE Computer Society Press.

A. Cockburn and B. McKenzie. An evaluation of cone trees. In *Proceedings of the 2000 British Computer Society Conference on Human Computer Interaction*, University of Sunderland, 2000.

C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph based visualization of the evaluation of software. In *Proceedings of the ACM symposium on Software Visualization*, pages 77–86, San Diego, CA, USA, June 2003. ACM Press, New York, NY.

IBM Research. Jinsight visualizing the execution of java programs, 2001. `http://www.research.ibm.com/jinsight/`.

R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design.* John Wiley & Sons, 1991.

D. Jerding and S. Rugaber. Using visualization for architectural localization and extraction. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE)*, pages 56–65, Amsterdam, The Netherlands, October 6-8 1997. IEEE Computer Society Press.

R. Oechsle. Automatic visualization with object and sequence diagrams using the java debug interface (jdi). In *Proceedings of the Software Visualization: International Seminar*, pages 176–190, Dagstuhl Castle, Germany, May 20-25 2001. Springer-Verlag Heidelberg.

S. Reiss. Bee/hive: A software visualization back end. In *Proceedings of the ACM symposium on Software visualization*, pages 57–65, San Diego, CA, USA, June 2003. ACM Press, New York, NY.

G. G. Robertson, J. D. Mackinlay, and S. K. Card. Cone trees: animated 3d visualizations of hierarchical information. In *Proceedings of the SIGCHI conference on human factors in computing systems: Reaching through technology*, pages 189–194, New Orleans, Lousiana, USA, 2003. ACM Press, New York, NY.

Sun Microsystems. Java3d, 2003a. `http://java.sun.com/products/java-media/3D/`.

Sun Microsystems. Java platform debugger architecture documentation, 2004b. `http://java.sun.com/j2se/1.4.2/docs/guide/jpda/`.

A. Telea, A. Maccari, and A. Riva. An open visualization toolkit for reverse architecting. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 02)*, pages 3–10, Paris, France, June 27-29 2002. IEEE Computer Society Press.

The Eclipse Project. Eclipse, 2004. `http://www.eclipse.org/`.

VisiComp Inc. Visivue java software visualization tool, 2004. `http://www.visicomp.com/product/visivue.html`.

L. Zayour and T. C. Lethbridge. Adoption of reverse engineering tools a cognitive perspective and methodology. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC 01)*, pages 245–255, Toronto, Canada, June 12-13 2001. IEEE Computer Society Press.

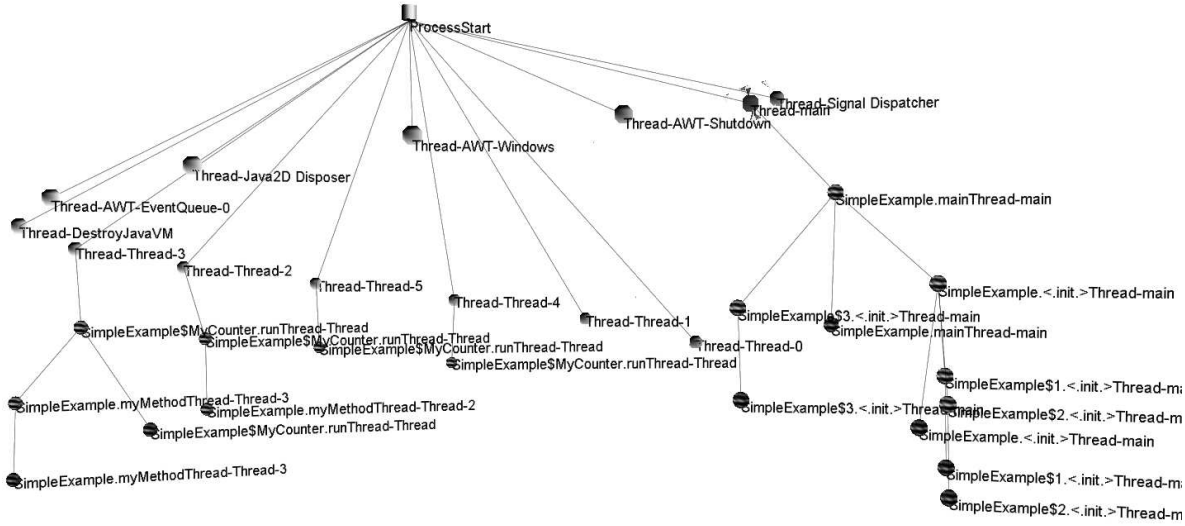


Figure 3: Call Graph Viewer - Main View

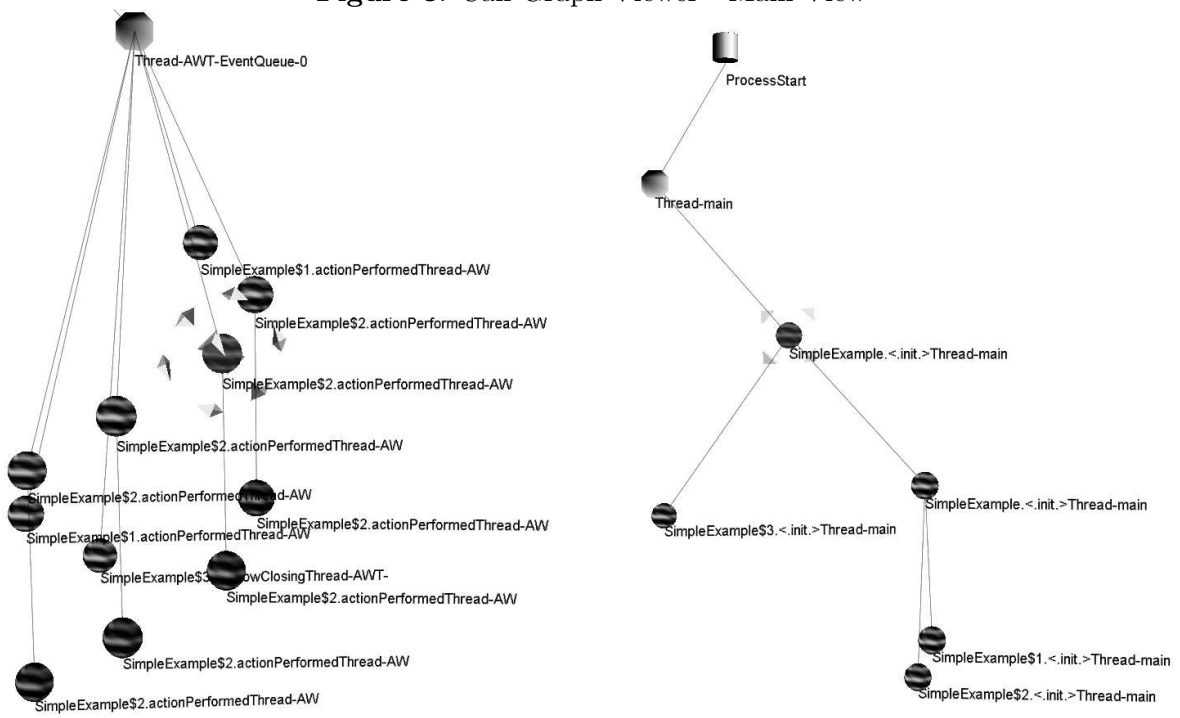


Figure 4: Call Graph Viewer - Thread View and Object View

Algorithm Visualization through Animation and Role Plays

Jarmo Rantakokko
Uppsala University, Department of Information Technology
Uppsala, Sweden

jarmo@it.uu.se

1 Introduction

The motivation and expectations for using visualization of algorithms in the education is that the students will grasp and understand the abstract algorithms and concepts used in computer science courses easier, i.e. the algorithms will be more concrete for the students. Visualization is a useful technique for learning in any computer science course. In this paper, we focus on parallel computing and in particular on parallel sorting algorithms (Grama et al., 2003). Parallel computing is intrinsically more difficult than sequential computing, as it requires coordination of the parallel processes which can be very problematic to conceptualize. We want the students to understand how the processors work together in an algorithm and how they interact. As the students can *see* how the processors run simultaneously in parallel this will illustrate important concepts such as processor load balance, synchronization, and communication.

The use of pictures and visualizations as educational aids is accepted practice. Textbooks are filled with pictures and teachers often diagram concepts on the blackboard to assist an explanation. Animation includes one more dimension, explaining and illustrating the dynamic properties of an algorithm. Role plays go even further by actively involving the learners within the algorithm. Furthermore, animations and role plays bring a variety into teaching that is usually missing. There are different learning styles, i.e. Visual (V), Aural (A), Read/Write (R), and Kinesthetic (K) (Fleming, 2001). While traditional teaching and lecturing are more directed towards the Aural and Read/Write learners, animations are more directed towards the Visual learners and role plays towards Kinesthetic learners. Moreover, computer systems students have some preference to the visual learning style (Fleming, 2001).

In this paper, we discuss how students perceive these two visualization techniques, animations and role plays, for understanding abstract algorithms in parallel computing. We have used both techniques in the same course and then let students answer questions anonymously in writing. The students have also taken the VARK-test (Fleming, 2001) to correlate the answers with their learning style.

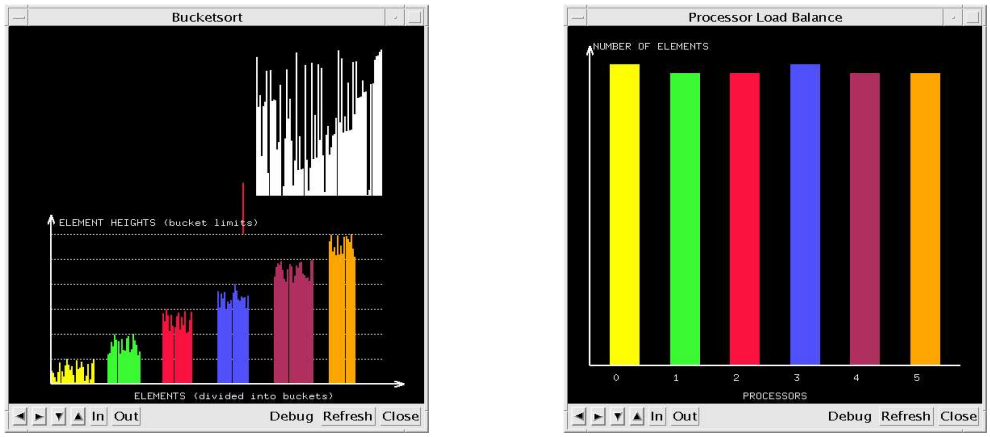
Computer animations are commonly used in traditional computer science courses, see e.g. (Naps et al., 2003a,b; Stasko et al., 1998), but little efforts have been made in parallel computing. Most visualization techniques are used by researchers for analyzing the parallel performance of their programs (Pancake, 1998). For educational purposes a group at the Georgia Institute of Technology led by John Stasko has developed the software package Polka for creating animations of parallel algorithms (Stasko et al., 1991, 1998). We have used this package to develop our animations. Another system that supports visualization of concurrent events is the Pavane system (Roman et al., 1992). A historically interesting note of using role plays is the experiment set up by Lewis F. Richardson in 1922 for computing a weather prediction (Richardson, 1922), long before the existence of parallel computers. Richardson simulates a parallel algorithm by letting individual workers do partial numerical calculations in parallel, as in todays parallel computers.

2 Computer Animations

We have developed two different animations for parallel sorting using the Polka package (Stasko et al., 1991, 1998). The animations are based on two parallelization strategies that are completely different and with different properties regarding communication, load balance, memory requirements, etc. The purpose is to let the students compare the algorithms, learn how they work and understand their inherent problems, i.e. the communication pattern, computation complexities and load balance. The students can work individually with the animations in a computer room exercise.

2.1 Parallel bucket sort

The first animation shows a parallel bucket sort algorithm for sorting a sequence of integer numbers in the interval [1, 100]. The numbers are represented with bars of corresponding height. We define k buckets by dividing the interval length uniformly into k subintervals, e.g. [1,25], [26,50], [51,75], [76,100] using four buckets on the interval [1,100]. The elements (numbers) are filtered in a sequential step into the equally spaced buckets. The buckets are then assigned to different processors and sorted in parallel with Quicksort. When running the animation, the user is prompted for the number of buckets, the number of processors, and how to distribute the buckets to processors. In addition, the animation is independent of the random number sequence. The user can choose from different random number sequences as input or even provide her own data file. In the animation we have two windows, one dynamic view of how the elements are sorted (Figure 1a), and one static view of the processor load balance (Figure 1b).



(a) Physical sorting view

(b) Processor load balance

Figure 1: The *bucketsort* approach for parallel sorting.

In a first step, the elements are moved from a single stack to the different buckets. The buckets are coded with different colors depending on which processors they belong. Next, the elements are sorted simultaneously in the different buckets creating an illusion of parallelism. The processor with the highest work load will continue sorting the longest time, illustrating the effect of the load imbalance. Different random number sequences will give different work loads in the buckets. The workload can then be balanced with, e.g., using more buckets than processors and distributing the buckets cyclicly to the processors or by providing a file with pre-described non-uniform bucket widths.

2.2 Parallel Quicksort

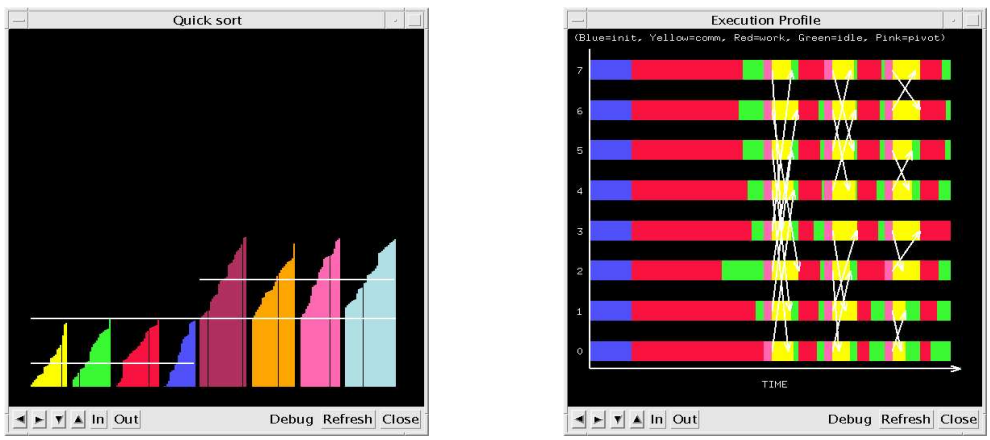
The second animation illustrates a memory efficient and fully parallel algorithm. The data is kept distributed all the time and there are no large serial sections as in the filtering operation above. In this approach, we divide the elements equally among the processors and perform a local sort within each processor. Then we exchange data pairwise between processors to get a global sorting order. The algorithm is outlined in Figure 2.

In this animation, the user can also choose the number of processors and the random number sequence as input. Here, the processor load balance depends on how the pivot elements are chosen in step *3.1*, Figure 2. There are different pivot selections strategies available for the user to choose as input. The animation supports two different views of the algorithm. The dynamic view shows how

Parallel quick-sort

1. Divide the data into p equal parts, one per processor.
2. Sort the data locally for each processor.
3. Perform global sort.
   - 3.1 Select pivot element within each processor set.
   - 3.2 Locally in each processor, divide the data into two sets according to the pivot ('$\leq$' or '$>$').
   - 3.3 Split the processors into two groups and exchange data pairwise between them so that all processors in one group get data less than the pivot and the others get data larger than the pivot.
   - 3.4 Merge the two lists in each processor into one sorted list.
4. Repeat 3.1 - 3.4 recursively for each half ($\log_2 p$ steps).

Figure 2: Parallel Quicksort. The elements are divided equally among the processors and sorted locally. Then, in a number of steps, the processors split their data into two parts according to a pivot and exchange data pairwise with a merging step to get a global sorting order.



(a) Snapshot during global sorting

(b) Execution profile

Figure 3: Views of the parallel quicksort animation.

the elements are physically sorted and moved between different processors (Figure 3a). All parallel activities are animated simultaneously, creating an illusion of parallelism. In the second view, an execution profile is grown simultaneously as the animation continues (Figure 3b). Here, different activities are color coded in bars for respective processor. Arrows represent the communication. The arrows are drawn from the sender to the receiver processor and their stopping point indicates how much data is communicated. From the execution profile it is possible to extract both computation and communication load imbalances.

3 Role Plays

Role plays are well suited for visualization of parallel algorithms. Each student can take the role of a processor in the algorithm. All bottlenecks, such as communication and load imbalance, in an algorithm become very obvious and concrete. The role plays can be performed within class.

Again, we have used different sorting algorithms for demonstration. The first algorithm is a pipelined version of *bubblesort*. The students are given different parts of an unsorted deck of cards. They lay out their cards on the table. The first student starts its first iteration moving the largest card by pairwise compare-exchange to her right while the other are idle. When she comes to the end of

her cards in the first iteration she must compare and exchange cards with her neighbor and can then start the next iteration. The other students proceed in the same way but have to wait awhile in the first iteration before they can start. Similarly, the processors/students to the right will finish sorting first and will then be idle. This algorithm requires frequent communication and synchronization and it also has a large load imbalance.

Next, we let the students perform the *odd-even transposition* algorithm. Here the students are also given parts of the deck of cards but now they first sort their own cards (locally). Then, they exchange all their cards alternately with their left and right neighbor. In each step, one student in a pair merges his cards with his neighbor's cards and gives back half of the cards, keeping the other half. The algorithm proceeds in *p*-steps (where p is the number of processors/students). This algorithm is much faster and requires less communication than the pipelined bubblesort algorithm. Still, there is some load imbalance, with half of the processors doing the merging while the other half of the processors are idle.

Finally, we let the students simulate the parallel *quick-sort* algorithm, as shown in Figure 2. This algorithm minimizes the communication compared to the two previous ones and terminates in $log_2(p)$ steps. Any load imbalance comes obvious from bad pivot choices giving some students more cards than the others.

4 Students

The target course, *Programming of parallel computers*, is a C-level course given in the third year. The students come from different programs, Master of Science, Computer Science, Natural Science, and exchange students. The students are mostly male (85%). The pre-requirements for the course are that they have at least taken two programming courses. This is their first course in concurrent programming techniques. The course is given with traditional lectures and has mandatory computer room exercises.

5 Results

In addition to a course evaluation we have asked the students to fill out a survey with the following questions: Grade 1-5 (where 1=not useful and 5=very useful) traditional lectures (L), textbook (B), animations (A) and role plays (R) with respect to:

1. Understanding parallelism and algorithms
2. Motivation for learning
3. Help for programming assignments
4. Which combination of teaching methods would you benefit most of, e.g. (L)+(B)+(A)

The results from the survey are summarized in Figure 4. The scores for the fourth question are calculated by counting the number of (L), (B), (A), and (R), respectively, then dividing these numbers by the number of students and finally multiplying with five to normalize with the other scores.

The results show that traditional lectures outperform the other teaching methods in all aspects. But it is interesting to see that the students feel that both animations and role plays give better understanding for the algorithms than the textbook and that the students prefer animations before the textbook for learning (in combination with lectures). Comparing the two visualization techniques clearly shows that computer animations are experienced by the students as having higher impact on learning than role plays. The results of the survey depend very much on factors such as quality of lectures, book, animations, enthusiasm of lecturer, and preference to learning style.

The students also took a VARK-test (Fleming, 2001). The results from the test, Figure 5, show that this particular group of students had some preference to the Read/Write and Kinesthetic learning styles. This does not correlate with the results of the questionnaire where the textbook got low scores while lectures got high scores.

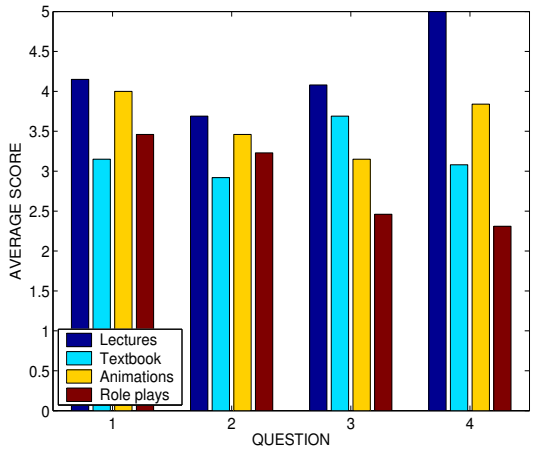


Figure 4: Average scores from the survey, 13 students answered the questionnaire.

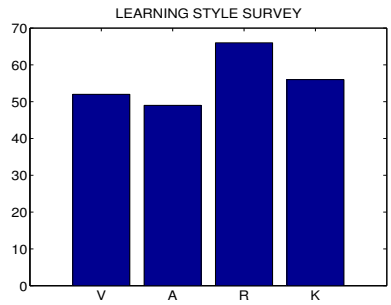


Figure 5: Total scores from the VARK-test. 15 students answered the questionnaire.

The course evaluation, which the students also answered anonymously, was very positive giving an average score 4.3 (out of 5.0) for the course as whole, 4.6 for the lectures and 4.1 for the laboratory exercises including the animations. The textbook got an average score 3.6 which can still be considered good.

6 Discussion

We have used both computer animations and student role plays to visualize abstract algorithms in parallel programming. Both techniques increase and facilitate learning and understanding of the specific algorithms. They also illustrate general concepts, such as load imbalance, synchronization and communication, in a very concrete way. In this particular study, they are experienced by the students as giving better understanding for the parallel algorithms than the textbook.

There are some fundamental differences between these two visualization techniques. Computer animations allow the students to work in their own pace and let them re-run the animations as often as they want to. They can also experiment with different input parameters and data testing different scenarios and behavior of the algorithms. Role plays on the other hand make the inherent concepts more concrete and real. For example, a load imbalance forces one participant to work more than the others. Role plays do not require any technical equipment or software development, maintenance, etc. Developing computer animations requires a lot of work and it can be difficult and time consuming to learn how to do this (Naps et al., 2003a). A problem with role plays is that they can be difficult to perform in a large group of students. It can be impossible to let all participate and then some of the students may not be able to follow or feel left out and be unengaged. Also, while role plays promote understanding of details it can on the other hand be difficult to see the big picture of the algorithms.

In the student survey, animations got better scores than role plays. But the best results were given to the lectures. One explanation is that the impact of the other factors, e.g. quality and enthusiasm,

has higher impact than the individual learning styles. Moreover, the lectures are a mixture of different teaching methods, including discussions, writings and visualization with pictures on the blackboard.

References

Niel Fleming. Vark, a guide to learning styles. `http://www.vark-learn.com/`, 2001.

Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Introduction to parallel computing, second edition. *Pearson Education Limited*, 2003.

Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudoplh Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and Angel Velazquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *inroads 35:2*, pages 131–152, 2003a.

Thomas L. Naps, Guido Rößling, Jay Andersson, Stephen Cooper, Wanda Dann, Rudolph Fleischer, Boris Koldehofe, Ari Korhonen, Marja Kuittinen, Charles Leska, Lauri Malmi, Myles McNally, Jarmo Rantakokko, and Rockford J. Ross. Evaluating the Educational Impact of Visualization. *inroads 35:4*, pages 124–146, 2003b.

Cherri M. Pancake. Exploiting visualization and direct manipulation to make parallel tools more communicative. *Lecture Notes in Computer Science 1541*, 1998.

Lewis Richardson. Weather prediction by numerical process. *Cambridge University*, 1922.

Gruia-Catalin Roman, Kenneth C. Cox, Donald Wilcox, and Jerome Y. Plune.G. Pavane: A system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing, Vol 3, No 1*, pages 161–193, 1992.

John Stasko, William F Appelbe, and Eileen Kraemer. Utilizing program visualization techniques to aid parallel and distributed program development. *Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, Technical report GIT-GVU-91/08*, 1991.

John Stasko, John Domingue, Marc H Brown, and Blaine A Price (editors). Software visualization: Programming as a multimedia experience. *MIT Press, Cambridge, MA*, 1998.

Inside the Computer: Visualization and Mental Models

Cecile Yehezkel, Mordechai Ben-Ari, Tommy Dreyfus
Department of Science Teaching, Weizmann Institute of Science, Israel

ntcecile@wisemail.weizmann.ac.il

1 Introduction

The EasyCPU environment was developed for teaching computer architecture to high school students of computer science. As a result of experience during the development, it was decided to perform research on the *mental models* that arise in the students as they learn. These mental models come from attempts by the students to make sense of the *conceptual models* presented to them by the software tool and their learning materials (textbook and exercises). These conceptual models concern the various components of the computer and their interconnections. We wished to investigate the relationship between the conceptual models that were presented and the mental models of the students. We conjectured that experience of the students as end-users of computers would be the basis of an initial non-viable mental model that the conceptual model would have to overcome.

2 Theoretical background

The first researcher to use the concept of mental models was Craik (1943), but the development of the theory of mental models was advanced by the later birth of cognitive science. The renewal of interest in mental models dates from the publication of Johnson-Laird (1983) and Gentner and Stevens (1983). Johnson-Laird (1983) saw mental models as a way of describing the thought processes of people as they solved deductive problems. Gentner and Stevens (1983) collected the work of several researchers on this topic, and claimed that mental models supply people with a means of understanding the functioning of physical systems. Norman's paper in this collection (Norman, 1983) defines a mental model M(T) as a conceptualization of a target system T by a *user*. According to him, this conceptualization defines the manner in which the user will carry out an operation on T. The mental model M(T) develops over time as a result of the interaction of the user with the target system T. M(T) is distinct from C(T), the conceptual model, which is the conceptualization of T by the *developer*.

Norman (1983) claimed that mental models do not have to be accurate, but they must be functional. The factors which affect the development of a mental model include: the technical background of the user, previous experience with similar systems and the structure of the user's own thought patterns. The user has a tendency to make up general rules that seem to fit all target systems. The mental model serves to guide decisions: the user can "run" the model to predict the outcome of an operation on the system. In effect, the model enables the user to mentally exercise the system.

When studying computer architecture, a student may not interact directly with a target system T, but rather with a *learning model* of the system L(T) that is presented by the visualization system on a computer platform. Therefore, it is possible that the resultant mental model M(T) may be different that it would have been had the student worked directly with T.

3 The EasyCPU environment

The EasyCPU environment is based on a simplified model of an 8-bit version of the Intel 80X86 microprocessor family (Yehezkel et al., 2001). EasyCPU has two modes of operation: a basic mode and an advanced mode. The basic-mode enables the novice student to learn the syntax and semantics of individual assembly language instructions (Figure 1). The visualization

shows the main units of the computer: CPU, memory segments, and elementary I/O. The units are connected by control, address, and data busses, which are animated during the execution of an instruction. The student can simulate a single instruction's execution and follow the data transfer between the CPU's registers and the data flow between the units.

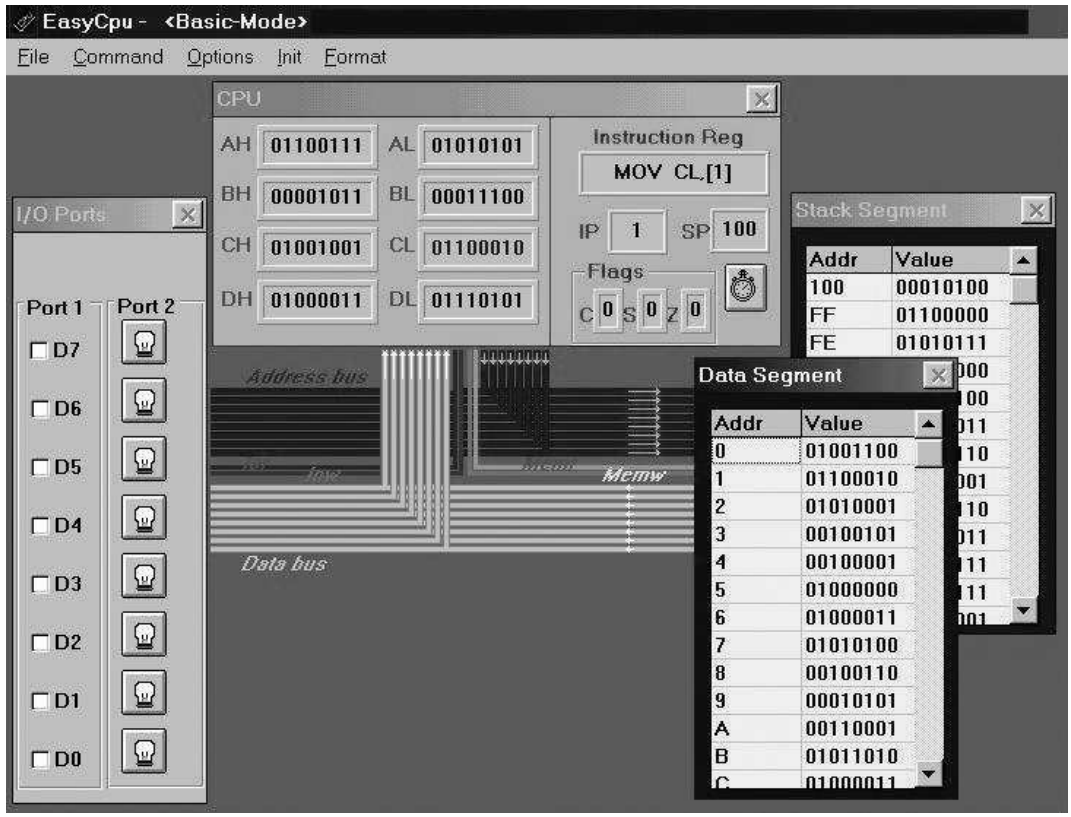


Figure 1: A screenshot the EasyCPU Basic Mode during the execution of `mov cl, [1]`

4 How can we describe the computer model?

A computer can be described both from a *static* viewpoint—the different units and a map of their interconnections, and from a *dynamic* viewpoint which describes the interaction and data transfer between the units. The conceptual model is based on four units (CPU, memory, input and output), and is composed of:

- The layout of the units and their interconnections.
- The method of transferring data along the connections.
- Standard interaction scenarios based on bus cycles.

The first two form the static viewpoint and the third describes the dynamic viewpoint. From the specific instruction being executed and its operands, you can predict what operations will be performed and describe them in a scenario.

5 Research

The research was carried out on a class of eleven tenth-grade students. They studied a one-year 90-hour course in computer architecture and assembly language programming. The students were evaluated on the basis of a project that they developed.

The mental models of the students were investigated at two points of time: first, after studying the material theoretically from the textbook but before they were exposed to the EasyCPU environment, and second, after carrying out tasks using the environment. The research tools were two tasks, a pretest and a posttest, in which the students were asked to describe both the static viewpoint (the topology of the interconnections between the units) and the dynamic viewpoint (six scenarios describing data transfer as a result of executing specific instructions). They were asked to describe the system model both graphically and in written text alongside the graphs.

The results were summarized in a table, with a row for each student, showing the six dynamic models that the student drew. This enabled us to identify the characteristic topologies of the drawings and to assign to each one a category defining the mental model associated with the topology. This gave us a mental model for each student at the pretest and then again at the posttest. In order to validate the results, the written text was also analyzed, searching for textual evidence that would support or contradict the categorization of the graph that was drawn.

We found that the students develop mental models that can be categorized into the four different topologies shown in Figure 2. For this target system, the CC model is the correct

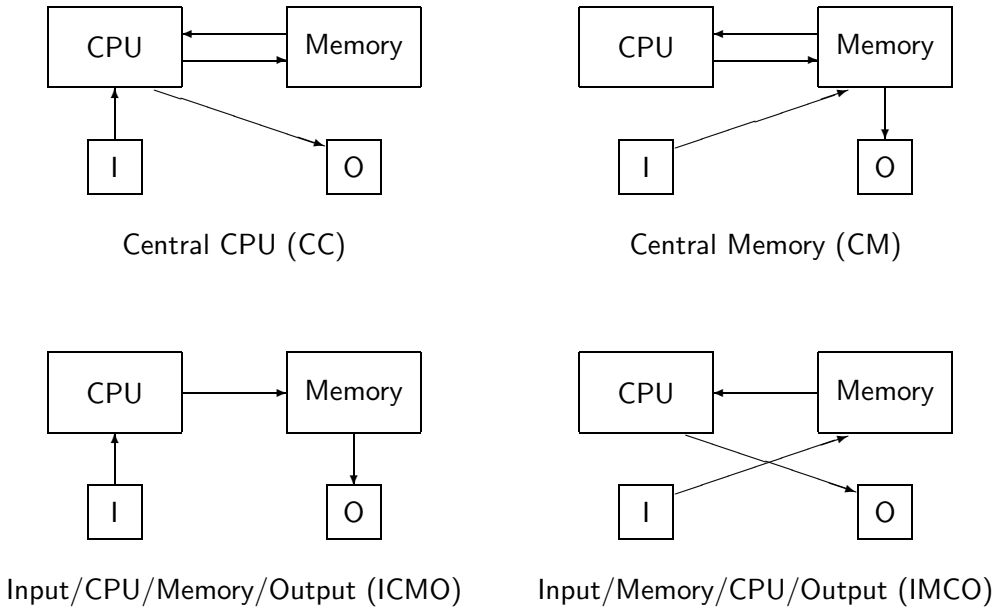


Figure 2: The four topologies

one, since all memory accesses and input-output are performed via the CPU. The CM model incorrectly assumes that I/O is done directly to memory. We will use the term ICO model for the two variants, one with the memory inserted before the CPU (IMCO) and one with the memory inserted after the CPU (ICMO). These models stresses input and output operations, downplaying the role of the main memory.

On the pretest, five of the eleven students chose to draw systems consistent with the ICO model, two more drew CM models, and only four drew the correct CC model. We interpret this as occurring because the students are used to interacting with the computer as end-users, which would tend to encourage the construction of a mental model M(T) consistent with the ICO model. The end-user interacts with the computer through I/O and experiences the computer as a processor of information, streaming from input to the CPU for processing and then to the output. As one student said:

> The data and instructions are entered through the input to the CPU, during this process the data is translated to the binary base and the instructions are translated

> to machine code. Then the CPU executes the operation. The result is transferred to the memory and then displayed onto the output.

Students who described CM models were influenced by what they learned about the stored-program architecture: A typical explanation by these students is:

> The instructions are entered through the input to the memory. ... The memory executes the instruction and controls the information flow.

On the posttest, after the students had interacted with the L(T) model presented by the visualization, all the students drew models consistent with the correct CC model. This supports the view that the visualization was critical in enabling the construction of a viable mental model, a process that did not occur from textbook learning alone.

6 Conclusions

After studying theoretical materials, the majority of the students still held mental models that had been influenced by their experience as end-users. The visualization feature of the basic mode of the EasyCPU environment enabled the students to develop a viable model of the computer architecture by visualizing dynamic interaction between the units during scenarios of instruction execution. This paper describes part of a larger research project that points out the influence on the learning process of conceptual models presented to the student by a visualization environment. The conceptual models presented in a visualization environment have to be designed through an evaluation procedure to ensure that the model is correctly interpreted by the students.

References

Kenneth Craik. *The Nature of Explanation.* Cambridge University Press, Cambridge, 1943.

Dedre Gentner and Albert L. Stevens, editors. *Mental Models.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

Philip N. Johnson-Laird. *Mental Models: Towards a Cognitive Science of Language, Inference and Consciousness.* Harvard University Press, Cambridge, MA, 1983.

Donald A. Norman. Some observations on mental models. In Dedre Gentner and Albert L. Stevens, editors, *Mental Models*, pages 7–14. Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

Cecile Yehezkel, William Yurcik, Murray Pearson, and Dean Armstrong. Three simulator tools for teaching computer architecture: EasyCPU, Little Man Computer, and RTLSim. *Journal on Educational Resources in Computing*, 1(4):60–80, 2001.

An Approach to Automatic Detection of Variable Roles in Program Animation

Petri Gerdt, Jorma Sajaniemi
University of Joensuu, Finland

{Petri.Gerdt|Jorma.Sajaniemi}@cs.joensuu.fi

1 Introduction

Many students have difficulties in learning to program computers. One reason that makes computer programming a difficult skill to learn is that programs deal with abstract entities, that are unrelated to everyday things. Visualization and animation is a way to make both programming language constructs and program constructs more comprehensible (Hundhausen and Stasko, 2002; Mulholland, 1998). Petre and Blackwell (1999) note that visualizations should not work in the programming language level because within-paradigm visualizations, i.e., those dealing with programming language constructs, are uninformative. Hence students learning to program benefit more from visualization of higher-level program constructs than visualization of language-level constructs.

Sajaniemi (2002) has introduced the concept of the *roles of variables* which he obtained as a result of a search for a comprehensive, yet compact, set of characterizations of variables that can be used, e.g., for teaching programming and analyzing large-scale programs. A role characterizes the dynamic nature of a variable embodied by the sequence of its successive values as related to other variables and external events. A role is not a unique task in some specific program but a more general concept occurring in programs over and over again. Table 1 gives ten roles covering 99 % of variables in novice-level, procedural programs.

Roles can be utilized in program animation to provide automatic animation of concepts that are at a higher level than simple programming language concepts. In a classroom experiment (Sajaniemi and Kuittinen, in press), traditional teaching of elementary programming was compared with role-based teaching and animation. The results suggest that the introduction of roles provides students with a new conceptual framework that enables them to mentally process program information in a way similar to that of good code comprehenders. Moreover, the use of role-based animation seems to assist in the adoption of role knowledge and expert-like programming strategies.

In this paper, we will present the role concept and the role-based program animator PlanAni, and discuss possibilities to implement automatic role analysis that is needed in order to automatically animate programs provided by students.

2 The Role Concept

Variables are not used in programs in a random way but there are several standard use patterns that occur over and over again. In programming textbooks, two patterns are typically described: the counter and the temporary. The *role* of a variable (Sajaniemi, 2002) captures this kind of behavior by characterizing the dynamic nature of a variable. The way the value of a variable is used has no effect on the role, e.g., a variable whose value does not change is considered to be a *fixed value* whether it is used to limit the number of rounds in a loop or as a divisor in a single assignment. Roles are close to what Ehrlich and Soloway (1984) call variable plans; however, Ehrlich and Soloway give only three examples and have no intention to form an complete set.

Table 1 gives informal definitions for ten roles that cover 99 % of variables in novice-level procedural programs (Sajaniemi, 2002). Each variable has a single role at any specific time, but the role may change during the execution. If the final value of the variable in the first role is used as the initial value for the next role, the role change is called *proper*. On the other

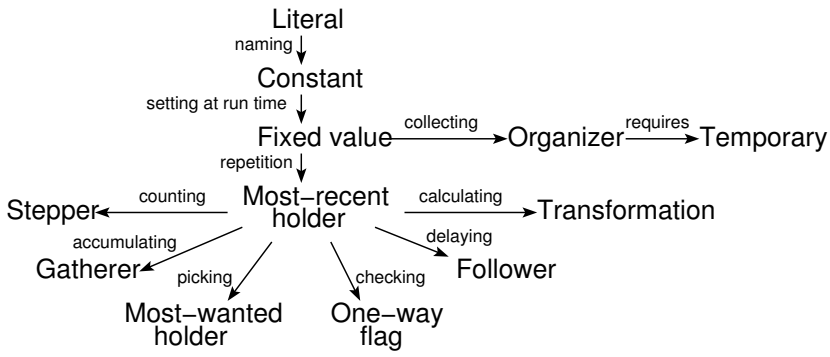


Figure 1: Role relationships. Literal and constant are programming language constructs; other nodes are the roles.

hand, if the variable is re-initialized with a totally new value at the beginning of the new role phase, the role change is said to be *sporadic*.

Roles can be used to describe the deep meaning of programs: what is the purpose of each variable and how do the variables interact with each other. For example, calculating the average of input values requires a *most-recent holder* for the input values, a *gatherer* for the running total, and a *stepper* for counting the numer of input items. Figure 1 describes the connections between roles that can be used as a basis for introducing the roles in elementary programming classes.

Table 1: Roles of variables in novice-level procedural programming.

Role	Informal description
Fixed value	A variable initialized without any calculation and not changed thereafter.
Stepper	A variable stepping through a systematic, predictable succession of values.
Follower	A variable that gets its new value always from the old value of some other variable.
Most-recent holder	A variable holding the latest value encountered in going through a succession of values, or simply the latest value obtained as input.
Most-wanted holder	A variable holding the best or otherwise most appropriate value encountered so far.
Gatherer	A variable accumulating the effect of individual values.
Transformation	A variable that always gets its new value with the same calculation from values of other variables.
One-way flag	A two-valued variable that cannot get its initial value once its value has been changed.
Temporary	A variable holding some value for a very short time only.
Organizer	An array used for rearranging its elements.

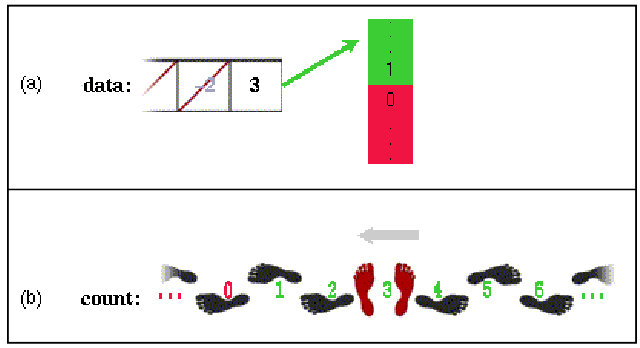


Figure 2: Visualizations of the same operation for different roles: comparing whether a *most-recent holder* (a) or a *stepper* (b) is positive.

3 The PlanAni Program Animator

We have implemented a program animator, PlanAni, that is based on the role concept (Sajaniemi and Kuittinen, 2003). In PlanAni, each role has a visualization—role image—that is used for all variables of the role. Role images give clues as to how the successive values of the variable relate to each other and to other variables. For example, a *fixed value* is depicted by a stone giving the impression of a value that is hard to change, and a *most-wanted holder* by flowers of different colors: a bright one for the current value, i.e., the best one found so far, and a gray one for the previous, i.e., the next best, value. A *most-recent holder* shows its current and previous values, also, but this time they are known to be unrelated and this fact is depicted by using two squares of a neutral color.

In addition to role images, PlanAni utilizes role information for role-based animation of operations. As the deep meaning of operations is different for different roles, PlanAni uses different animations. For example, Figure 2 gives visualizations for two syntactically similar comparisons "`some_variable > 0`". In case (a), the variable is a *most-recent holder* and the comparison just checks whether the value is in the allowed range. In the visualization, the set of possible values emerges, allowed values with a green background and disallowed values with red. The arrow that points to that part of the values where the current value of the variable lays, appears as green or red depending on the values it points to. The arrow flashes to indicate the result of the comparison.

In Figure 2(b) the variable is a *stepper* and, again, the allowed and disallowed values are colored. However, these values are now part of the variable visualization and no new values do appear. The values flash and the user can see the result by the color of the current value. In both visualizations, if the border value used in the comparison is an expression (as opposed to a literal value), the expression is shown next to the value.

Figure 3 is an actual screenshot of the PlanAni user interface. The left pane shows the animated program with a color enhancement showing the current action. The upper part of the right pane is reserved for variables, and below it there is the input/output area consisting of a paper for output and a plate for input. The currently active action in the program pane on the left is connected with an arrow to the corresponding variables on the right. frequent pop-ups explain what is going on in the program, e.g., "*creating a gatherer called sum*". Users can change animation speed and the font used in the panes. Animation can proceed continuously (with pauses because the frequent pop-ups require clicking "Ok" button, or pressing "Enter") or stepwise. Animation can be restarted at any time but backward animation is not possible.

PlanAni is implemented using Tcl/Tk and it has been tested both on Linux/Unix and Windows NT. The architecture consists of four levels as depicted in Figure 4. The lowest level takes care of primitive graphics and animation, and implements the user interface. The next level knows how to animate the smallest actions that are meaningful to viewers of the animation. This level is language independent in the sense that it can be used to animate

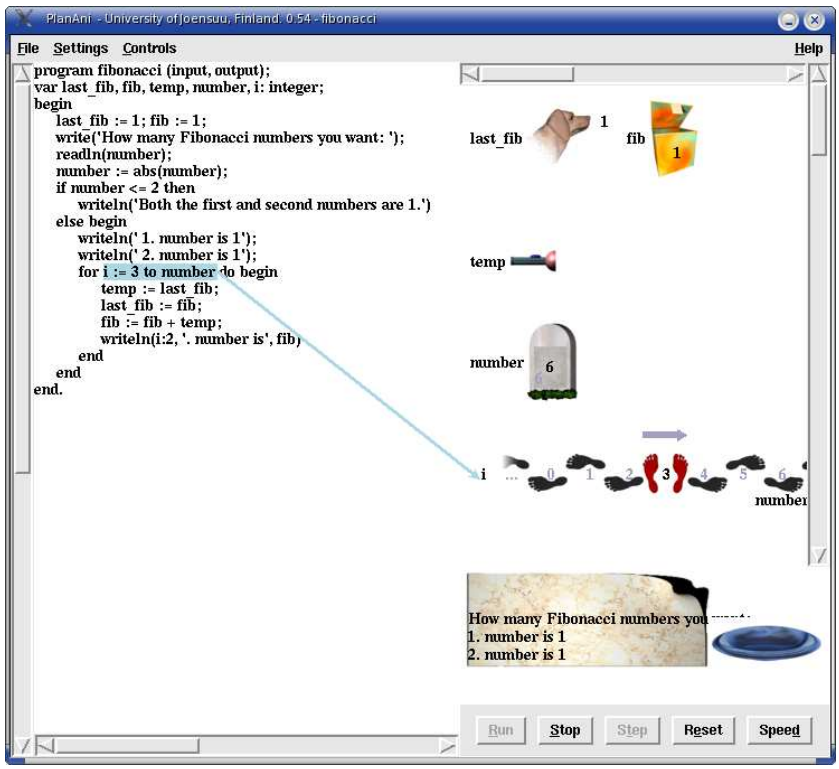


Figure 3: A screenshot of the PlanAni user interface.

Automatic role analysis
Program–level animation
Statement–level animation
Primitives for graphics

Figure 4: Architectural levels of PlanAni.

programs written in various languages, e.g., Pascal, C, and Java.

The third level takes as input a program to be animated, annotated with the roles of variables and possible role changes, and animates the program automatically. Finally, the uppermost level does not need role information because it finds roles automatically. Currently, the two uppermost levels are not implemented. As a consequence, animation commands must be authored by hand for each program to be animated. Typically 5 animation lines are required for each line in an animated program.

The uppermost level of PlanAni will accept programs without role annotations. It should make a dataflow analysis and assign roles and role changes based on this analysis. Its output—the input to the third level—looks like the program in Figure 5 with two variables that are initially *most-recent holders*. When the second loop begins, the role of the variable `count` changes to *stepper*. Since the last value of the old role is used as the first value in the new role, the role change is marked as proper.

The implementation of the fourth level is challenging for two reasons. First, roles are cognitive concepts which means that different people may assign different roles to the same variable. There is also the possibility of repeating phases of the same behavior making automatic detection of, e.g., *one-way flag* behavior hard. Second, the recognition of roles requires extensive dataflow analysis which is a research area of its own.

```
program doubles (input, output);
var count{MRH}, value{MRH}: integer;
begin
    repeat
        write('Enter count: '); readln(count)
    until count > 0;
    while count{proper:STEP} > 0 do begin
        write('Enter value: '); readln(value);
        writeln('Two times ', value, ' is ', 2*value);
        count := count - 1
    end
end.
```

Figure 5: A Pascal program with role annotations.

4 Automatic Detection of Roles

Roles are cognitive constructs and represent human programming knowledge. An automatic analysis of roles takes as input computer programs and tries to assign roles to the variables. The automatic analysis must find a connection between the program code and the cognitive structures of the programmer. The fact that human cognition is not exact whereas program code is exact makes this task a challenging one.

A related example of extracting programming related information automatically is the PARE (Plan Analysis by Reverse Engineering) system (Rist, 1994). PARE extracts the plan structure of an arbitrary Pascal program that it gets as input. A plan is a series of actions that achieve a goal, which PARE defines as a sequence of linked lines of code. Lines of code that either use or control another lines are linked, and define a sequence of lines, which constitute a plan. A program may have many plans, which together build up the plan structure of the program. PARE deals with larger constructs when compared to automatic role analysis: the plans detected by PARE represents a view to the solution of a problem, whereas automatic role analysis essentially searches program code for clues of stereotypical usage of variables.

The primary objective of automatically assigning roles to variables can be divided into two subgoals: to automatically find characteristics of a variable from a source program ("dataflow analysis" of the identification phase in Figure 6); and to map these characteristics to a certain human understood role ("Matching" in Figure 6).

The first subgoal requires customization of compiler and dataflow analysis methods and techniques (Aho et al., 1988; Nielson et al., 1998) in order to extract information about how data flows through a variable; this information is compressed into a flow characteristics description (FC). The second subgoal presupposes the creation of a database that contains mapping information between human defined variable roles and flow characteristics. The creation of this database is called the learning phase in Figure 6. The database is obtained by taking a set of existing programs with roles annotated in the style of Figure 5, and applying the same dataflow analysis technique as in the identification phase. With this method, a set of flow characteristics can be attached to each role, and some machine learning mechanism ("Learning" in Figure 6) can then be used to generalize the results into a role–FC database.

Role definitions may differ from person to person and it is probable that programmers with different backgrounds and experience levels produce different mappings between the role concept and actual program code. By asking programmers with different backgrounds to assign roles to the programs used in the learning phase, it becomes possible to analyze the differences in the resulting databases, i.e., differences of the programmers' mental models.

The complexity of the extraction of FCs differ, some can be determined with a relative

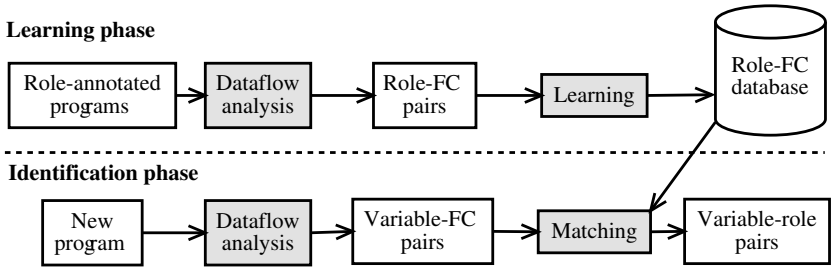


Figure 6: An overview of the learning and matching phases in automatized variable role detection.

simple syntax analysis, others need more complicated analyses of the data flow. The whole process of program analysis begins with the scanning of the program, during which the program is partitioned into tokens, such as reserved words and strings. These tokens are then processed through syntax analysis, in which the tokens are matched to the syntax of the programming language. The variables of the program are located during the scanning process and basic information about their type and assignments are found during syntax analysis.

Some examples of FCs are *related variables* and *value scaling*. The related variables charasteristic indicates how many variables affect the value of the variable under examination. The value scaling charasteristic tells that the value of a variable is scaled by some constant or variable in some context during its lifetime. The related variables FC can be determined during syntax analysis by simply recording all variables that appear on the right hand side of assignments. Syntax analysis also produces data that can be used to find out whether the value of a variable is scaled. The simplest case of value scaling is the modification of the assigned value by some constant, such as in "`x := x + 1`". In a more complex case value scaling may include two variables on the right hand side of an assignment, where the other represents the scaling factor. Both related variables and value scaling may happen in the same assignment; in general FCs are not exclusive, but rather conditions which may apply in association with each other.

The extraction of some FCs require the examining the execution order of a program. For example the FC called *change frequency* records how many times the value of a variable changes during its lifetime. This information requires that program execution is simulated in order to find out how many assignment are made to a certain variable. The lifetime of a variable is an important source of information for determining FCs. An example of a dataflow analysis technique, which is needed when examining the lifetime of a variable is the live variables analysis (Nielson et al., 1998). A variable is said to be *live* at a certain point of program execution if the value assigned to it will be used later as the execution proceeds. The live variables analysis may show that a value assigned to a variable in the beginning of a program is not used at all, in fact the next use of the variable is independent of the previous value. In this case the variable may have two different roles during its lifetime. If a variable is not live at a certain point of a program, then it may not have an active role at that time. A variable may in fact have many different roles during its lifetime, thus the lifetime of a variable and the lifetime of a role are two different things.

Another reason for examining the execution order is the fact that the sequence in which the FCs of a variable appear are important. Consider for example the variable `count` in the program of Figure 5. The change frequency of `count` is determined to be frequent, as it is assigned values repeatedly inside two loops. On the other hand the value scaling FC apply to the variable `count` too, as its value is modified with the statement "`count := count - 1`" in the while loop. If the appearance order of the FCs are not considered, then the role of the variable `count` looks like a *stepper*. A *stepper* can be identified by the combination of a frequent change frequency and the value scaling FCs.

If the sequence in which the FCs of the variable `count` appear is considered, then we can

identify two groups of FCs. The first group includes only the frequent change frequency FC (in the repeat loop) and the second group includes a pair of FCs: frequent change frequency and value scaling (in the while loop). The fact that the FCs of `count` can be grouped into two distinct groups suggests that `count` might have two different roles during its lifetime. The latter group identifies the role of *stepper* as discussed above. The former group including only the change frequency FC needs an additional FC to indentify a role: *user input*, which indicates that the value of the variable is dependant on user input. Thus we get a grouping of the FCs frequent change frequency and user input, which suggests that the role of the variable `count` is *most-recent holder* during the first loop of the program in Figure 5.

Certain roles appear in pairs, which adds a new dimension to the automatic detection of roles. For example a variable with the role of an *organizer* appear often with an another variable with the role of a *temporary*. The *temporary* variable is used to facilitate the reorganizing of the *organizer* variable. An another example of a role pair is a *gatherer*, whose gathering activities are controlled by a *stepper*.

Our initial prototype will analyze Pascal programs and deal with only three roles: *fixed value*, *stepper*, and *most-recent holder*. These three roles covered the majority of variables (81.7–90.3 %) in a study of three Pascal programming textbooks (Sajaniemi, 2002). The implementation is done using Tcl/Tk and it is based on Yeti (Pilhofer, 2002), a Yacc-like compiler-compiler (Aho et al., 1988).

5 Conclusion

In this paper we have presented the concept of variable roles, which are a set of characterizations of variables. The concept explicitly embodies programming knowledge in a compact way that is easy to use in teaching programming to novices. The PlanAni program animator visualizes the roles in a program by attaching a role image to each variable and animating operations on the variables according to their roles. This way the program visualizations that the PlanAni animator produces goes further than the mere surface structure of the program.

Automatic role detection is needed in order to make automatic role-based animation of arbitrary programs possible. The combination of a non-exact cognitive concept and the exact nature of programming languages makes this task a non-trivial one. We suggest that automatic role analysis is possible by performing dataflow analysis combined with a machine learning strategy. Automatized role analysis also makes it possible to deal with the roles of variables in large-scale programs, which may provide interesting opportunities for large-scale program comprehension.

Acknowledgments

This work was supported by the Academy of Finland under grant number 206574.

References

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1988.

K. Ehrlich and E. Soloway. An Empirical Investigation of the Tacit Plan Knowledge in Programming. In J. C. Thomas and M. L. Schneider, editors, *Human Factors in Computer Systems*, pages 113–133. Ablex Publishing Co, 1984.

S. A. Hundhausen, C. D. Douglas and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13:259–290, 2002.

P. Mulholland. A Principled Approach to the Evaluation of SV: A Case Study in Prolog. In J. Stasko, J. Dominique, M. H. Brown, and B. A. Price, editors, *Software Visualization – Programming as a Multimedia Experience*, pages 439–451. The MIT Press, 1998.

F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, Heidelberg, 1998.

M. Petre and A. F. Blackwell. Mental Imagery in Program Design and Visual Programming. *International Journal of Human-Computer Studies*, 51(1):7–30, 1999.

F. Pilhofer. YETI - Yet another Tcl Interpreter. Internet WWW-page, URL: http://www.fpx.de/fp/Software/Yeti/, 2002. (March, 2004).

R. S. Rist. Search Through Multiple Representations. In D. J. Gilmore, R. L. Winder, and F. Detienne, editors, *User-Centred Requirements For Software Engineering Environments*. Springer-Verlag, New York, 1994.

J. Sajaniemi. An Empirical Analysis of Roles of Variables in Novice-Level Procedural Programs. In *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, pages 37–39. IEEE Computer Society, 2002.

J. Sajaniemi and M. Kuittinen. Program Animation Based on the Roles of Variables. In *Proceedings of the ACM 2003 Symposium on Software Visualization (SoftVis 2003)*, pages 7–16. Association for Computing Machinery, 2003.

J. Sajaniemi and M. Kuittinen. An Experiment on Using Roles of Variables in Teaching Introductory Programming. *Computer Science Education*, in press.

TeeJay - A Tool for the Interactive Definition and Execution of Function-oriented Tests on Java Objects

Ralph Weires, Rainer Oechsle
University of Applied Sciences, Trier, Germany

weiresr@fh-trier.de, oechsle@informatik.fh-trier.de

Abstract

This paper describes the testing tool *TeeJay* whose main purpose is the function-oriented testing of Java objects with a particular support for remote RMI objects. TeeJay offers the possibility of defining and executing tests in an interactive manner. A test case consists of method calls and checks. TeeJay is a capture/replay tool comparable to many test tools used for testing graphical user interfaces. A test is defined by recording all interactively executed method calls and checks. The method calls can be selected for execution in a way similar to the way method calls are executed in BlueJ.

Whereas BlueJ is an educational development environment with a focus on the design aspect, TeeJay is a corresponding tool for testing.

1 Introduction

An important area of computer science education is software development. Teachers should emphasize right from the beginning that programming is not equal to software development, but only a part of it. Other important activities of the software development process are analysis and design on one hand and program testing on the other.

BlueJ (Barnes and Kölling, 2004) is an educational development environment with a focus on the design aspect. TeeJay is a corresponding tool for testing which can be used to interactively create and execute tests for Java programs with the help of a graphical user interface and without the need of writing any test code.

The available building blocks that can be used in a test are classes, existing objects, existing primitive variables, and existing remote resources. All these building blocks are visualized by trees. These trees contain the methods that can be applied to each existing class and object. Tests in TeeJay can simply be composed by interactively selecting methods or constructors that should be invoked within a test. It is also possible to specify conditions for the attributes of the objects or for the return value of a called method. To realize this functionality, TeeJay makes extensive use of the *Java Reflection API*.

TeeJay requires a Java Runtime Environment (JRE) of version 1.4 or higher. It can be freely used and copied under the conditions of the GNU General Public License (GPL). The whole underlying diploma thesis (Weires, 2003) as well as the software itself are available on one of the author's web pages (at `http://www.ralph-weires.de/stud-da.php`).

2 Usage Overview

TeeJay offers two working perspectives to the user, one for the definition of tests and the other one for the execution of existing tests. In this paper we will only show some sample parts of the user interface in order to give a brief overview over the main usage and functionality of the program.

2.1 Definition View

The definition view shows the current test environment which consists of classes, objects, primitive variables, and remote resources. Remote resources are represented e.g. by RMI registries and are used to obtain references to remote RMI objects. Once a reference to a remote object has been obtained, remote objects can be used like any other local object.

However, operations on RMI objects are automatically called with a timeout. An RMI call is cancelled after a certain (user-defined) time has elapsed without the method call having returned. Thus, we take into account possible disturbances of the network connection to the remote service.

The views for classes and objects show all members (attributes and methods) which can be used in a test. The classes view will now serve as an example to explain how the methods can be used. Figure 1 shows the view of some sample classes in the environment. The classes are displayed in a tree structure, sorted by their packages. The child nodes of a class

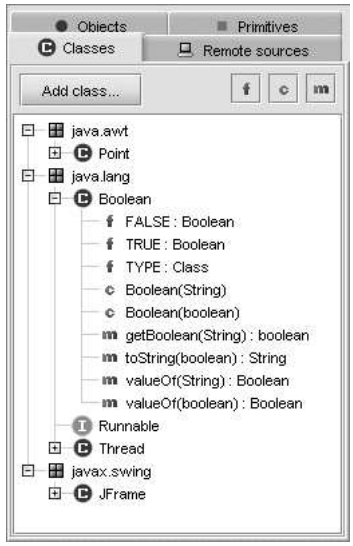


Figure 1: Classes view of the test environment

node represent constructors, class attributes (static attributes in Java), and class methods (static methods in Java). Figure 1 depicts these members for the class `Boolean`, because the node representing this class is currently expanded. TeeJay can, however, only access members with the modifier `public` - so bypassing the defined visibility for the members of a class is not possible in the current version of TeeJay (although this might change in a future version). The available members are visualized by their respective declaration. They can be used in tests. In the example it would thus be possible to call e.g. the static method `getBoolean(String):boolean`.

To actually call a method, the required parameters have to be specified. Any objects and primitive variables, which are currently present in the test environment and which were gained by the execution of previous operations, can be used. They can be referred to by their identifier. Besides such arguments coming from the test environment it is also possible to define fixed values for a call. In the case of a primitive parameter every literal of the respective data type can be used. In the case of an object parameter, `null` can be used or a literal string, if the parameter type is compatible with the class String. If the return type of a called method is not `void`, it is additionally possible to assign the return value of the called method to a variable of the test environment. The identifier of the variable has then to be indicated.

Instead of assigning the return value of a method call to a variable, it is also possible to check certain conditions for a return value. This option is mainly given for primitive values. Such return values can be compared with a type compatible reference value (which can e.g. be again a value out of the test environment), using one of the usual comparison operators like `==` oder `<=`.

After the complete specification of a method call, the method is actually called using the Java Reflection API. It is thus possible to compose the desired test environment piece by piece and to optionally check certain conditions. The result of every method call is displayed in

a logging window. In case of a thrown exception during the execution of a method call, the respective exception object is displayed in the logging window together with its stack trace to support debugging.

If a class is needed in a test, the class has to be added to the test environment. This is done by indicating the fully qualified class name. The classes view can so be filled as shown in Figure 1 for several sample classes and interfaces. The user is thus enabled to test class methods, or to construct her test environment by creating objects which will later be used in tests.

In order to allow TeeJay to access the required classes, the user has to specify the directories and libraries (jar or zip archives) that will be searched. Only compiled class files are needed, there is no source code required. For RMI objects it is also possible to enable dynamic class loading, thus making the local presence of the class files optional. To make use of this option, it is however needed to start the program with some special options which are not further described here.

The objects view which is shown in Figure 2 can be used in a way quite similar as the classes view. For every object, all public instance attributes and methods (non-static attributes and

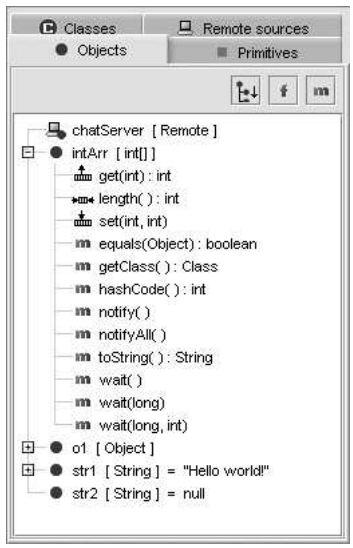


Figure 2: Objects view with pseudo array methods

methods in Java) are displayed. To get a better overview it is possible to hide the attributes, the methods, or all inherited members.

The type of a reference to an object gained by a method call depends on the formal return type of that method. The fields and methods of an object that are displayed, and are thus available, depend on the current reference type. The type of a reference can be interactively changed by an explicit cast operation.

If an object reference represents an array, TeeJay offers some pseudo methods which do actually not exist. These methods represent the reading and writing of an element of the array as well as reading the array length. The types of the parameters and return values of these methods depend on the array type. As an example, Figure 2 shows the methods get, set, and length for an array of int.

For testing purposes, TeeJay provides a kind of capture/replay function which can be used for the definition and execution of tests. To define a test, the program can be switched into a recording mode, in which all operations interactively executed by the user within the test environment are recorded. After having finished the recording, the recorded sequence of operations is available as a test case which can later be replayed.

It is additionally possible to define test suites which are composed of recorded test cases as described above or other test suites. Complex test trees consisting of many test cases can

thus be created. A test suite can be executed later by a single mouse click. For every test case within a test suite, it is possible to specify the number of times the test has to be repeated.

All defined tests can be saved persistently. This is why they can be reused in future sessions. To run one of the available tests, TeeJay provides a separate view (besides the definition view) which will be described in the next subsection.

2.2 Test Run View

The second perspective of TeeJay's user interface is used to select one of the available tests and run it. Every defined test can thus be executed again after the source code of the tested classes has been modified. Therefore TeeJay is suited for regression testing. A regression test checks whether all the functions that have been successfully tested in the past, still work after the code has been modified.

The execution speed of a test can be adjusted by the user. The progress of a started test can be observed during its execution. The results are displayed in two ways:

1. In the detailed view the result of every single step is displayed with all the details like thrown exceptions and stack traces in case of an error.

2. The summary of the results of the test run so far displays some statistics to give a rough overview over the current progress. Figure 3 shows an example of such a summary after completion of a test run. These statistics contain information about the number of executed tests, operations and checks (checks are operations, too). For the checks it is also displayed how many of them were executed successfully and how many failed. After a test run has been terminated, there is a single summarizing message being displayed to inform the user about the result of the test run. The message shown in figure 3 appears only if the test run was completed successfully (it was not aborted neither by the user nor by a failed operation), and if additionally all checks have passed.

3 Related Work

We know a few tools which have a functionality that is comparable to TeeJay. This section takes a look at the differences to these tools and the consequential pros and cons in comparison with TeeJay.

3.1 JUnit

JUnit (Beck and Gamma, 1998) is certainly one of the most common tools for testing Java programs. Like TeeJay, JUnit is used for the creation of function-oriented tests. It has a simple structure and is very versatile. It requires the definition of tests by directly writing test code. This has the advantage that there are no restrictions for the definition of tests, whereas TeeJay does not allow the definition of control structures, e.g., in tests.

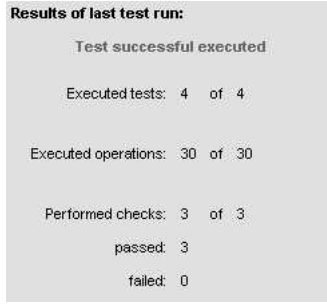


Figure 3: Summary of a test run

The fundamental difference (besides the graphical user interface) between these two tools is that in JUnit tests are defined *statically*, whereas in TeeJay they are defined *dynamically*. Dynamic test definition means that all test steps are executed immediatly after their definition.

Hence in TeeJay it is possible to observe the results and effects of every step already at definition time. The user is thus provided with a test environment which is always up to date, making it possible to analyse the progress and the effects already during the definition of a test. Errors can therefore be detected immediately. We believe that especially for programmer novices this feature can be very helpful.

Another difference between the tools is that the single test cases in JUnit are independent from each other and can therefore be executed in any order without any problems. Thus every test case represents a self-contained test.

Tests in TeeJay may on the contrary have dependencies. If an object is e.g. added to the test environment in one test, it is usually still available after that test has finished. In this way tests can be created which are based on each other, making it possible, e.g., to reuse a basic test in more than one test scenario.

However, this feature of TeeJay has the disadvantage, that, during the execution of a test suite, a failure of a single test case may cause the failure of the whole test suite. This is different in JUnit because of the independency of the test cases.

3.2 Exacum

We found the commercially available testing tool Exacum (`http://www.ist-dresden.de/products/Exacum/`) only when the development of TeeJay was nearly finished. Even though TeeJay was not influenced by Exacum, there are some similarities between these two tools. But there are also some important differences.

Exacum is not only comparable to TeeJay, but also to JUnit. In contrast to JUnit, the definition of tests in Exacum is also done using a graphical user interface. Like TeeJay, Exacum offers the possilitiy to the user to define simple test steps like constructor or method calls as well as checks as part of a test case. However, this definition is done statically in Exacum, whereas it is done dynamically in TeeJay, as already mentioned in the previous subsection. The test steps are actually executed only if the user starts a test run. This happens after a test has been completely defined. So there is no execution of any test step during the definition of a test.

In this regard Exacum is similar to JUnit. Hence, the pros and cons mentioned in the previous subsection do not have to be repeated here. Futhermore, like in JUnit, different test cases are independent from each other.

To sum up, Exacum is like JUnit but has a graphical user interface for the definition of tests like TeeJay. JUnit offers more flexibility because the test steps have to be programmed in Java. TeeJay supports especially the testing of remote RMI objects, whereas Exacum mainly focusses on servlets and EJBs (Enterprise Java Beans).

3.3 BlueJ

BlueJ (Barnes and Kölling, 2004) is originally not a testing tool, but a development environment for Java whose main purpose is the support of teaching object-oriented principles to programming beginners. TeeJay adapts the philosophy of BlueJ to the world of program testing. Like in BlueJ, TeeJay users work on classes and objects, e.g., by interactively selecting methods to be called. Also like BlueJ, TeeJay visualizes the available classes and objects as well as their methods and attributes.

However, during the development time of TeeJay, BlueJ has been enhanced with components that provide testing support. This has been achieved by integrating JUnit into BlueJ (Patterson et al., 2003). Hence, it is meanwhile possible to define unit tests in BlueJ in a similar manner as in TeeJay. Besides the possibility of explicitly writing JUnit test code, it is

possible to interactively create a corresponding test class for a class represented in the UML class diagram view of BlueJ. For such test classes, test methods can be defined by recording a sequence of interactively executed calls of constructors and methods in the environment. Also, similar to TeeJay, it is possible to define certain conditions for the return values of called methods while a test is being recorded. These conditions are tested when the created test methods are run afterwards. The graphical user interface for executing tests is similar to the one known from JUnit.

Although the testing features of BlueJ and TeeJay are based on the same idea and are thus similar in many ways, there are also some significant differences. BlueJ supports test fixtures. A test fixture is an environment that is established before a test run is started. A test fixture can be reused for many test cases. TeeJay does not support test fixtures, although they can be emulated by recording additional preparation and cleanup tests which can then be used with different test cases inside of test suites.

Another advantage of the testing features of BlueJ is due to the integration of JUnit into BlueJ. The tests created by BlueJ are Java classes suited for JUnit. It is thus possible to run them also in the traditional JUnit environment - the BlueJ environment in which the tests were created is not needed for the execution. On the contrary, TeeJay is always needed to run the tests created with it. Furthermore, the created tests in BlueJ can be manually edited at a later time.

In TeeJay, there are no source files needed for the classes being tested, whereas in BlueJ, it is only possible to define test classes for those classes whose source code is available (unless the test code is written manually). In general, methods or constructors of classes cannot be interactively called in BlueJ unless the source files are available. On the contrary, in TeeJay classes can be displayed and used without the need of knowing the corresponding source code.

Another difference between the two interactive testing environments is the graphical representation of classes and interfaces. In BlueJ, UML class diagrams are used, whereas classes, interfaces, and objects are represented by trees in TeeJay (see subsection 2.1).

Furthermore, TeeJay was developed with a special focus on testing remote RMI objects. As far as we know, this is not possible in BlueJ. In addition, after having found a remote object in an RMI registry, it is possible in TeeJay to load the class file for this object dynamically over the network.

Finally, some features of TeeJay regarding the definition of tests are not possible in BlueJ (unless the test code is written manually). E.g., in BlueJ it is not possible to save the return value of a called method in the test environment if this value has a primitive type. This is mainly because the BlueJ environment only manages objects, not primitives. Therefore, it is not possible to create a test which checks whether the (primitive) return value of two methods, called one after the other, is the same.

3.4 JavaCHIME

Like BlueJ, JavaCHIME (Tadepalli and Cunningham, 2004) is more oriented towards teaching than testing purposes. It is another tool which allows direct interaction with objects and classes like BlueJ and TeeJay do. It is, however, currently under development and can not be tested by the authors yet. Thus, we are not able to provide further information.

4 Possible Usage of TeeJay in Practical Education

TeeJay can be used in practical education as a valuable tool for programmer novices. It provides a simple visual overview of the classes and objects currently in use, and enables the students to directly see the effect of a performed action such as a method call. TeeJay could thus help to understand important principles of object oriented programming in a similar way as BlueJ does.

More important is TeeJay as a tool that fosters the integration of testing into the software development process right from the beginning of computer science education. We believe that this is as important as the integration of software modelling techniques from early on. The students should be encouraged to create tests already for their small exercise programs that they have to write in their first year. After changing the underlying code, the same tests can be used again to perform a regression test without any more work to do. Students will discover that some of the test cases will not work any more. If they did not expect that the code changes have such an impact to their test cases, they will gain some important experiences and insight into the area of software development. Especially, they will get used to the testing process and they will achieve a better understanding of the eminent and increasing importance of testing software thoroughly before actually using it in a running system.

The achieved effect of using TeeJay in practical education could be evaluated i.e. by separating the students into two groups: one that is teached to work and test with TeeJay and another one that does not use TeeJay. We assume that the programs created by the first group will be better (less bugs, more robustness), since these programs will be better tested than those of the other group.

In order to get significant results, the experiment has to be carried out with larger student groups working on a number of different exercises. To realize the comparison of the student programs, a tool for the automatic assessment of the student programs (such as those described e.g. by Reek (1989) or Jackson and Usher (1997)) would be very helpful.

5 Summary and Outlook

TeeJay is a testing tool for programmer novices that offers the possibility of defining and executing tests in an interactive manner. A test case consists of method calls and checks. TeeJay is a capture/replay tool comparable to many test tools used for testing graphical user interfaces. A test is defined by recording all executed method calls and checks. The method calls can be selected for execution in a way similar to the way method calls are executed in BlueJ. TeeJay can be used as a tool for programmer novices that fosters the integration of testing into the software development process right from the beginning of computer science education.

Up to now, TeeJay supports only the definition of tests by explicit specification of each step. These steps are basically at the level of source code statements. This way of test definition is well suited for simple tests, but is too time-consuming for larger tests. JUnit is much more efficient and flexible in this respect.

The usability of TeeJay can be improved significantly if test cases are derived automatically from a given higher-level specification. State charts are an example of such higher-level specifications. By applying appropriate traversing algorithms for state charts, TeeJay could automatically find the needed test cases to cover all possible transitions of a state chart.

Such an approach is described in the diploma thesis of Sokenou (1999) which deals with the definition of state charts for class testing. The work focusses on a technique for the inheritance of state charts in class hierarchies.

Apart from such fundamental extensions, some smaller improvements are also planned such as modifications of the user interface. An extended view of the different contents would be useful to give an even better overview of the test environment to the user. An example for this would be a deeper display of the attribute values of objects as can be seen in many advanced debuggers. Some currently not used abilities of the Reflection API could be used to access members with limited access modifiers.

A possibility to edit defined (recorded) tests would be another useful feature which is not available at the moment. A little fault made at definition time would no longer force the whole recording process to be repeated.

The addition of further remote resource types is another direction for future work. Cur-

rently, only RMI registries are available as remote resources. Other possible remote resource types that can be made available for TeeJay are *JINI Lookup Services*, *Java Spaces*, *CORBA Naming Services*, and *Web Services*.

References

David J. Barnes and Michael Kölling. *Objects First with Java: A Practical Introduction using BlueJ*. Prentice Hall / Pearson Education, 2nd edition, 2004. ISBN 0-13-124933-9.

Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3 (7):37–50, July 1998.

David Jackson and Michelle Usher. Grading Student Programs using ASSYST. In *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, pages 335–339, 1997.

Andrew Patterson, Michael Kölling, and John Rosenberg. Introducing Unit Testing with BlueJ. In *Proceedings of the 8th conference on Information Technology in Computer Science Education (ITiCSE)*, 2003.

Kenneth A. Reek. The TRY System - or - How to Avoid Testing Student Programs. In *Proceedings of the 20th SIGCSE Technical Symposium on Computer Science Education*, pages 112–116, 1989.

Dehla Sokenou. Ein Werkzeug zur Unterstützung zustandsbasierter Testverfahren für JAVA-Klassen. *Softwaretechnik-Trends (in German)*, 19(1):37–50, Februar 1999.

Pallavi Tadepalli and H. Conrad Cunningham. JavaCHIME: Java Class Hierarchy Inspector and Method Executer. In *Proceedings of the ACM-Southeast Conference*, pages 152–157, April 2004.

Ralph Weires. Entwurf und Implementierung eines Werkzeugs zur interaktiven Definition und Ausführung funktionsorientierter Tests für lokale und über RMI nutzbare Java-Objekte. Diplomarbeit (in German), Fachhochschule Trier, 2003.

JavaMod: An Integrated Java Model for Java Software Visualization

Micael Gallego-Carrillo, Francisco Gortázar-Bellas, J. Ángel Velázquez-Iturbide
ViDo Group, Universidad Rey Juan Carlos, Madrid, Spain

{mgallego, pgortaza, a.velazquez}@escet.urjc.es

1 Introduction

Given the practical importance and complexity of object-oriented programming, there are many software visualization systems (VSs) for these languages. These systems use different forms of visualization to assist in understanding object-oriented applications. In particular, some VSs are designed to visualize programs written in the Java programming language (and they are often implemented in such a language, too). Java is an attractive language for visualization developers, because it is a "comfortable" language and it is simple to build visualizations in Java. In the particular case of Java VSs implemented in Java itself, there is an additional advantage: the Java Virtual Machine provides an interface to debug programs written in Java, namely JPDA (JPDA). This interface avoids the need of using external debuggers or of generating program traces. The former often involves obscure interfaces; the latter requires to introduce additional code within the target program in order to extract information at run-time.

Our ultimate goal is to build a infrastructure adequate to the comprehensive, flexible and systematic design of Java visualizations. Many Java VSs have been developed using different Java program representations, for instance, Evolve (Wang, 2002) based on Step (Brown, 2003), Jeliot (Myller, 2004) based on a Java interpreter, and JIVE (Gestwicki and Jayaraman, 2002) based on JPDA.

Our proposal provides an architecture to support three models of Java programs: source code, execution and trace. The final result is a set of APIs that allows working with a comprehensive model of any Java application in a uniform and homogeneous way. Note that we use the term "model" as synonymous for representation; it is inherited from the software engineering community, where representing entities is named modeling.

In the following sections, we briefly describe such an architecture. The second section outlines our three models: source code, execution and tracing. The third section sketches the construction of a debugger based on these models. The fourth section gives a comparison with VSs and tools. Finally, we summarize our future work.

2 Java Models

A program model is a representation of the program in a given programming language. Notice that such a model involves two programming languages: the language in which the target program is written and the modeling language. We are interested in building Java models of Java programs, so we use one only programming language for both roles.

A program can be modelled in different ways, depending on the point of view. We can distinguish the following three models:

- The *code model* provides a static representation of a program. It contains information that can be determined at compile time, such as subclass relationships. The most important code models are based on structural object information, abstract syntax trees (AST) like those generated by compilers constructed with SableCC (Gagno, 1998), or decorated ASTs (i.e. also including semantic information).

- The *execution model* provides a dynamic representation of a program. It contains information that can only be determined at run time, such as the value of variables. It

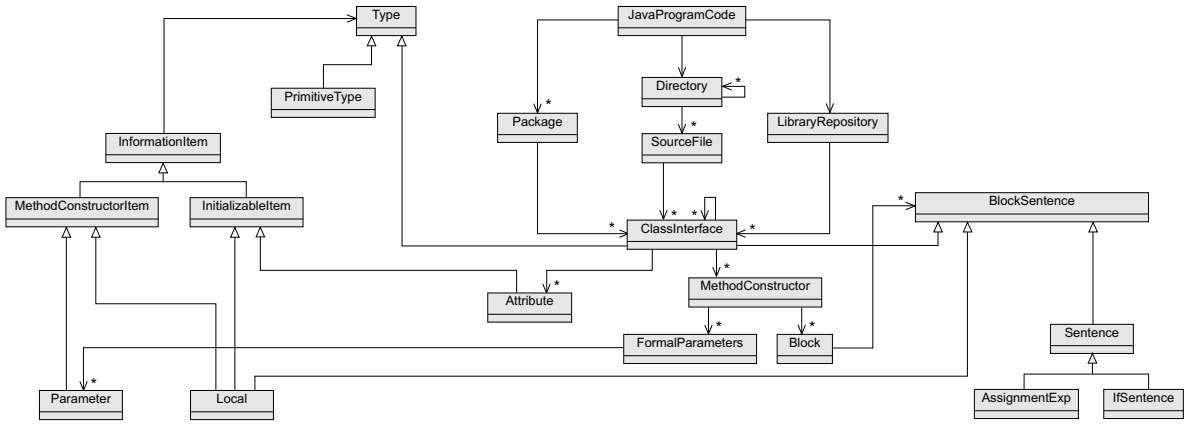


Figure 1: An overview of javaMod code model

is also common to include some code information, such as method names. The most important execution models are based on interpretation, debugging, or instrumentation (i.e. automatically modifying source code to introduce communication with the model at relevant points).

- The *trace model* also provides a dynamic representation of a program. The difference between the execution and the trace models of a given program is that the former only gives information about its current state of execution, while the latter gives information about its execution history. There are no agreed representations for this model. Existing models are based on their intended use, such as profiling or visualization. For instance, Omniscient Debugger (Lewis) is a trace system of Java programs for debugger purposes.

Our proposal for modeling Java programs in Java is called javaMod (`http://vido.escet.urjc.es/javamod`). The model allows representing a Java program with the three models. It is an integrated model, so that semantically related concepts that appear in the three models are explicitly related.

2.1 The javaMod Code Model

The code model of a program represents the contents of its source files and the libraries it uses. It is represented by an instance of the class `JavaProgramCode`. Each program element is modelled with lexical, syntactic and semantic information, as well as information about its location in the source file. Typically, some relevant physical information is also modelled, for example, paths of the `CLASSPATH` variable or path of the virtual machine.

All the elements of any source are modelled as decorated ASTs. The code model is based on classes such as `MethodConstructor`, `ClassInterface`, `Local`, `Attribute` and `Sentence` (examples of sentences are a control flow statement, an object instantiation, a method invocation, an assignment or a "break" sentence). This model is simple enough to be easily understood, because the elements present in source files and the modeling language are represented as objects of those classes (see Figure 1).

2.2 The javaMod Execution Model

The execution of a program is represented as an instance of the class `JavaProgramExec`. Entities present at any instant during the execution of a program are modelled. For instance, the value of local variables can be extracted from the execution stack represented in the model.

This model is based on classes such as `MethodConstructorExec` (that represents the execution of a method or constructor), `ClassInterfaceExec` (that represents a class or interface after being loaded in memory) or `LocalExec` (that represents the memory space of a local).

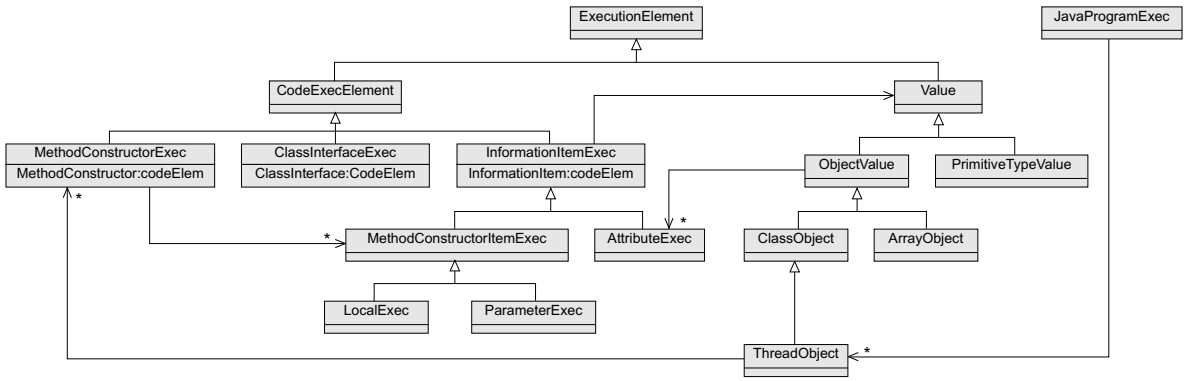


Figure 2: An overview of javaMod execution model

This model represents each thread of a program as an instance of `ThreadObject`. For each thread, its execution stack is represented as a list of `MethodConstructorExec` objects. For each method or constructor under execution, we can know the sentence that is being executed, the object that is serving the message, and the list of variables (see Figure 2).

Each object at the execution model is related to its corresponding object at the code model. For instance, each object of the class `MethodConstructorExec` is related to the object of class `MethodConstructor` it represents.

2.3 The javaMod Trace Model

The trace model records all the information that is generated during the execution of a program. Some pieces of information that can be determined by this model are the number of times a method has been executed, the values a variable has stored or the number of objects instantiated of a given class. Our trace model is in an advanced phase of elaboration, but it is not yet complete. However, we outline the services it will provide.

In this model, there is a class named `MethodConstructorTrace` that represents a method invocation along time, and therefore it stores the instant its execution started, the instant it finishes and the trace of each method invocation made from its body. The class `LocalTrace` represents the trace of the memory space allocated to a local variable, including the allocation time of such a memory space, the set of values stored and the instants they were assigned.

A program trace is represented as an instance of the class `JavaProgramTrace`, which contains a list of `ThreadObjectTrace` objects. Each `ThreadObjectTrace` hosts a tree with all the `MethodConstructorTrace` objects that represent each of the method invocations performed.

Recall that the trace model represents an execution record along the execution time of a program. Consequently, the information provided by this record at a given instant is equivalent to that provided by the execution model.

3 A View of the Debugger Process Using javaMod

In this section we show the use of javaMod to model and build an example application, namely a debugger. First, the overall debugger architecture and its design are presented and then its educational application is introduced.

3.1 Construction of a Java Debugger Using javaMod

To illustrate the versatility of javaMod, we have built a debugger (see Figure 3). It is divided into three blocks: components of the user interface, interest models, and javaMod. In Figure 3, we show component composition as circle ended lines. The user interface is a window that contains a tool bar to interact with the program being debugged (`JDebuggerToolbar`). Four dif-

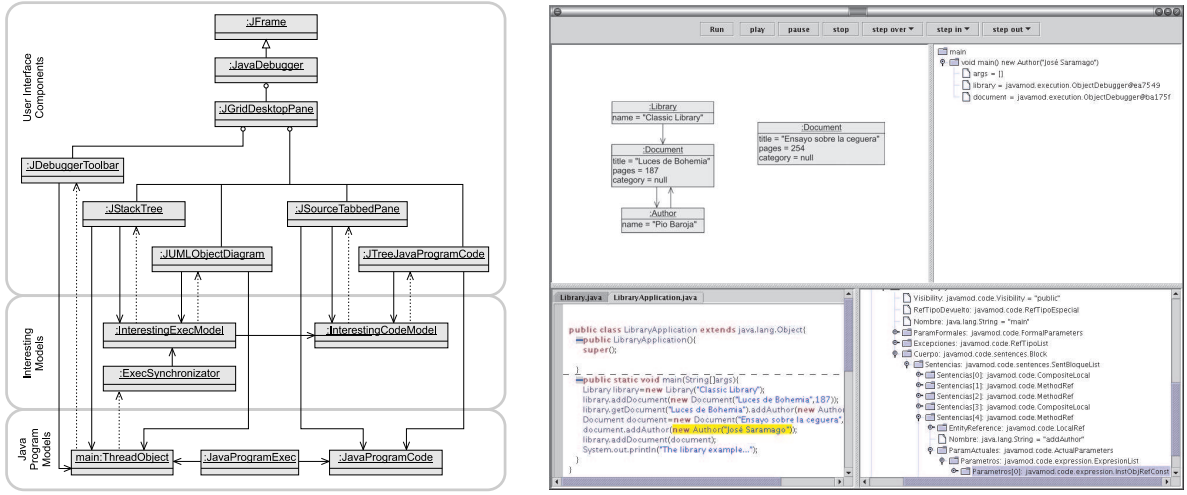


Figure 3: Java debugger architecture and snapshot

ferent, synchonized views of the program are shown: an object diagram (`JUMLObjectDiagram`), a tree showing the execution stack of the thread that executes the method main (`JStackTree`) and two views of the code model. `JTreeJavaProgram` shows a directory structure and each source file with its AST. `JSourceTabbedPane` shows source files with syntax highlighting. The development of this user interface would be the greatest burden in the work of the visualization designer.

We have used the Observer pattern (Gamma et al., 1997), in which the MVC architecture is based, to build javaMod. This pattern defines the generation of events to notify changes of state. In Figure 3 we show the association between subject and observer as dotted lines. We use this event model (in `JavaProgramExec`) to report changes of values in information items (locals, parameters and attributes). We also use it to report the beginning of method execution, the end of thread execution, etc. Our event model is based on the standard event model of Java defined in JavaBeans.

The `JavaProgramExec` instance allows initiating and finalizing the execution of the program. This instance models all the threads under execution in a program. For the sake of simplicity we only consider the main thread in this example, represented by an instance of `ThreadObject`. This class offers operations to pause execution, to resume it and to run step by step. The `ThreadObject`s trigger events when they change of state (paused to resumed or vice versa).

In this way, there is a `JavaProgramCode` that represents a Java program, a `JavaProgramExec` that represents an execution of that program, and a `ThreadObject` that represents the execution of the main thread. The user of the tool can control such a thread execution with a `JDebuggerToolbar`, which contains controls to resume, pause and manage a thread execution. The bar also associates event listeners to the thread to show its state (paused or active).

Typically, the debugger of integrated development environments (IDEs) shows, in the component that visualizes code, the next sentence to execute. In addition, the methods in the stack are shown in the components that visualize execution. The values of local variables of the method at the top of the stack are also commonly shown. This functionality is achieved in our tool by allowing components to access information related of the models.

The components in charge of visualizing the code model do not know or refer to any element of the execution model. As a consequence, the components of the graphical user interface of a tool are more modular and reusable. For instance, some visualizations in tools that do not require program execution can be built such as pretty-printers or metric gatherers.

The views of the code model and the execution model must also be synchronized. For

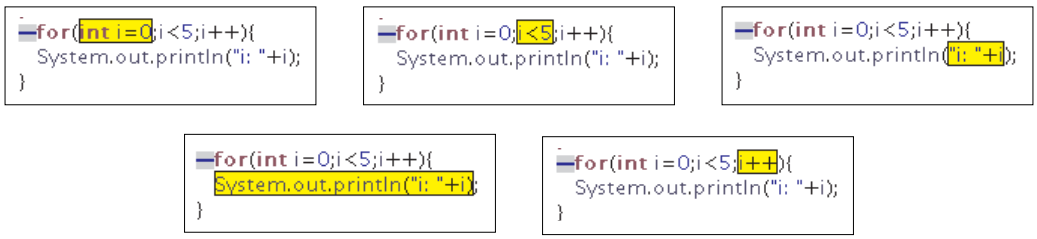


Figure 4: Structural debugger snapshot

instance, if the element of interest for the components that visualize the execution model is the execution of a method or constructor (`MethodConstructorExec`), the element of interest for the components that visualize the code model will be the definition of the method (`MethodConstructor`) or the next sentence (`Sentence`) to be executed.

The point of "interest" at any moment must also be identified, and it either is determined by the current state of execution of the program or is selected by the user interacting with one of the views. In the same way as the selected elements are managed in Java user interface (JFC) through `SelectionModel`s, we have created `InterestingModel`s.

Basically, the `InterestingModel`s keep the element of interest and the components to which changes must be notified. The element of interest will show the current point of execution if it is indicated to the `InterestingExecModel`. An instance of the class `ExecSynchronizator` is used for this purpose. This instance will map the events of the main thread and the `InterestingExecModel`.

Provided there is an element of interest in the code model and an element of interest in the execution model, there is an `InterestingModel` for each of them. In addition, the `InterestingExecModel` is associated to the `InterestingCodeModel` to guarantee synchronization of the views. Establishing an interest element in the `InterestingExecModel` also implies that the corresponding element is established as the element of interest in the `InterestingCodeModel`.

Finally, the components that visualize each of these models must be associated to `InterestingModel`s. This association is established directly by setting the interest element through the user interface or the other way around by means of events, so that the views are notified of a change in the interest element.

3.2 Using the Debugger in Education

From an educational point of view, the debugger we have built using javaMod has several advantages over traditional debuggers. It is a "structural debugger" (Gallego-Carrillo et al., 2004), in contrast to traditional line-based debuggers. A structural debugger allows inspecting the state of the program based on its syntactic structure. Consequently, the gap between the execution dynamics of a program and its static declaration is shorter. From an educational point of view, it allows instrumenting programs according to the operational semantics explained to students in the classroom. The comprehension of such static-dynamic relation is one of the problems most students have on learning programming. The facilities of our debugger could be applied to imperative languages in general, although it is currently applied to the Java language.

The construction of a Java debugger that implements the ideas of structural debugging has been possible by the way javaMod integrates the code model (with the syntactic information) and the execution one (with the capabilities of a traditional debugger). The use of the code model as an aid for the execution model allows for the inclusion of the operational semantics of the language in the debugger process.

For instance, suppose the *for* loop is to be explained in a classroom. In a *for* loop the initialization part is executed first and once, then the condition is evaluated and if it succeeds

Table 1: Some software visualization and educational tools

Tool Name	**Code**	**Exec**	**Trace**	**Technologies**
BlueJ (BlueJ)	√	√	×	JPDA
DrJava (DrJava)	√	√	×	DynamicJava (DynamicJava)
JGrasp (JGrasp)	√	√	×	DynamicJava & JPDA
ProfessorJ (ProfessorJ)	√	√	×	DrScheme plugin (DrScheme)
Jacot (Leroux et al., 2003)	×	√	×	JPDA
Omniscient Debugging	×	×	√	Instrumentation
JRat (JRat)	×	√	×	Instrumentation & others
Evolve	×	×	√	Step trace protocol
Fujaba (Fujaba)	√	√	×	JPDA

the body of the loop is executed. Afterwards, the loop variable is updated and the condition is evaluated again, and so on. In this context, a debugger showing what is the next part to be executed or evaluated could be really valuable for the students if used in conjunction with the theoretic explanations (see Figure 4).

Structural debugging allows performing operations such as showing what are the next suitable sentences to be executed (by taking into account where the program is stopped at the moment). As an example, if the program is stopped in the condition part of an if-statement at least two things can happen. If the condition evaluates to true, the then-part is executed, so the first sentence in that part could be highlighted. If the condition evaluates to false, then either the first sentence of the else-part (if exists) is executed or the first sentence after the if-sentence is executed, so they could also be highlighted. Even if the normal execution flow can be broken via an exception this can be detected with javaMod, and the corresponding catch block can be visualized too as a possible point to step to.

4 Related Work

The available Java VSs are based on their own specific architectures. Those that are capable of step-by-step debugging are usually based on JPDA. However, the management of source code and libraries and the execution trace is done without any standard. Moreover, the elements visualized have to be synchronized and this synchronization process has to be done by the tool itself. Table 1 shows some tools, identifies the program models each one manages and cites the technologies used to obtain such information from the Java program.

Some tools such as Fujaba, Evolve or BlueJ have a plugin architecture to include new functionality. Even so, they have not been constructed allowing other to build VSs just using its internal representation for Java programs (without making use of their user interface or installation procedure). JavaMod, however, is an API which has been designed to obtain all the relevant information from a Java program and to make this information available to build any kind of tool.

The models available for building tools are not integrated. Each one offers an specific functionality and it is not easy to integrate them. In Table 2 several APIs are shown focusing on the main aspect they are intended for.

5 Conclusions and Future Work

We have described a new approach to modelling Java programs in Java, called JavaMod. JavaMod allows defining three models: source, execution and trace. It has also been compared with advantage to other models. Currently, it provides a framework to build ambitious programming tools and visualizations. We have also illustrated it by applying it to build a structural debugger.

Table 2: Some Java program management APIs

API Name	Definition	Execution	Trace
Java Reflection (JavaReflection)	√	×	×
BCEL (BCEL)	√	×	×
Javassist (Javassist)	√	×	×
PMD (PMD)	√	×	×
RECODER (RECODER)	√	×	×
BARAT (BARAT)	√	×	×
OpenJava (OpenJava)	√	×	×
Eclipse JDT Core (JDT)	√	×	×
DynamicJava	×	√	×
BeanShell (BeanShell)	×	√	×
JPDA	×	√	×
Eclipse JDT Debug (JDT)	×	√	×
STEP	×	×	√

A useful feature that could be integrated into the code model consists in making it modifiable so that modifications are notified as events. This would permit that the code→debugging→execution→profiling cycle were integrated in one single model. We also plan to design educational tools that will make use of javaMod to generate graphical explanations of programs.

From a practitioner's point of view, it would be very useful to facilitate the use of the model in standard environments. To this aim, we are developing a tool to transform Java models into UML 2.0 (UML), as defined in the UML2 project by Eclipse (UML2), that provides serialization in the standard format XMI. Another interesting feature to make our model more useful consists in being able to incorporate it as a plug-in in the most relevant IDEs such as Eclipse (Eclipse) and NetBeans (NetBeans).

Finally, it would be useful to build a model generator, so that given a specification for a language, models were generated that included lexical, syntactic, semantic, and execution elements of the language.

6 Acknowledgements

This work is supported by the research projects TIC2000-1413 of the Spanish Research Agency CICYT and TIN2004-07568 of the Ministerio de Educación y Ciencia.

References

BARAT. URL `http://sourceforge.net/projects/barat`. (seen September 2004).

BCEL. URL `http://jakarta.apache.org/bcel/`. (seen September 2004).

BeanShell. URL `http://www.beanshell.org/`. (seen September 2004).

BlueJ. URL `http://www.bluej.org/`. (seen September 2004).

Rhodes H. F. Brown. STEP: A framework for the efficient encoding of general trace data. Master's thesis, McGill University, 2003. URL `http://www.sable.mcgill.ca/step/`.

DrJava. URL `http://www.drjava.org/`. (seen September 2004).

DrScheme. URL `http://www.drscheme.org/`. (seen September 2004).

DynamicJava. URL http://koala.ilog.fr/djava/. (seen September 2004).

Eclipse. URL http://www.eclipse.org/. (seen September 2004).

Fujaba. URL http://www.fujaba.de/. (seen September 2004).

Étienne Gagno. SableCC, an object-oriented compiler framework. Master's thesis, McGill University, 1998.

Micael Gallego-Carrillo, Francisco Gortázar-Bellas, and J. Ángel Velázquez-Iturbide. Depuración estructural: Acercando la práctica a la teoría de la programación. 6th International Symposium on Computers in Education. To appear, 2004.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.

Paul Gestwicki and Bharat Jayaraman. Interactive visualization of Java programs. In *Symposia on Human Centric Computing Languages and Environments*, pages 226–235, 2002.

JavaBeans. URL http://java.sun.com/products/javabeans. (seen September 2004).

JavaReflection. URL http://java.sun.com/j2se/1.4.2/docs/guide/reflection/. (seen September 2004).

Javassist. URL http://www.csg.is.titech.ac.jp/~chiba/javassist/. (seen September 2004).

JDT. URL http://www.eclipse.org/jdt/. (seen September 2004).

JFC. URL http://java.sun.com/products/jfc. (seen September 2004).

JGrasp. URL http://www.jgrasp.org/. (seen September 2004).

JPDA. URL http://java.sun.com/products/jpda/. (seen September 2004).

JRat. URL http://jrat.sourceforge.net/. (seen September 2004).

Hugo Leroux, Annya Réquilé-Romanczuk, and Christine Mingins. Jacot: a tool to dynamically visualise the execution of concurrent Java programs. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, 2003. ISBN 0-9544145-1-9.

Bil Lewis. Debugging backwards in time. URL http://www.lambdacs.com/debugger/debugger.html. (seen September 2004).

Niko Myller. The fundamental design issues of Jeliot 3. Master's thesis, University of Joensuu, 2004. URL http://cs.joensuu.fi/jeliot/.

NetBeans. URL http://www.netbeans.org/. (seen September 2004).

OpenJava. URL http://openjava.sourceforge.net/. (seen September 2004).

PMD. URL http://pmd.sourceforge.net/. (seen September 2004).

ProfessorJ. URL http://www.professorj.org/. (seen September 2004).

RECODER. URL http://recoder.sourceforge.net/. (seen September 2004).

UML. URL http://www.uml.org. (seen September 2004).

UML2. URL http://www.eclipse.org/uml2. (seen September 2004).

Qin Wang. Evolve: An extensible software visualization framework. Master's thesis, McGill University, 2002. URL http://www.sable.mcgill.ca/evolve/.

Towards Tool-Independent Interaction Support

Guido Rößling, Gina Häussge
Department of Computer Science
Darmstadt University of Technology, Darmstadt, Germany

{guido, huge}@rbg.informatik.tu-darmstadt.de

Abstract

Interaction may be one key aspect for achieving improved learning outcomes using algorithm visualization software. Only a limited set of tools actually supports interaction, and the types of interaction supported are usually incompatible. In this paper, we present a tool-independent interaction support component that is easy to incorporate into existing systems.

1 Introduction

A group of eleven experts met in the course of ITiCSE 2002 to determine the role of visualization and engagement in CS education (Naps et al., 2003). They managed to isolate several key reasons why algorithm or program visualizations (abbreviated "AV" for the rest of this paper) have not been adopted as often as researchers in the field would hope. Chief among the reasons is *time* - the time needed to search for good examples, learning new tools, developing visualization and adapting materials to the use in the classroom.

In those cases where the use of AV software was evaluated, research has shown an important trend: learners who are actively engaged with the visualization technology have consistently outperformed learners who passively view visualizations (Hundhausen et al., 2002). One key part of the working group report is therefore the *engagement taxonomy*. The taxonomy defines six different levels of engagement in combination with AV materials:

1. *No viewing* – no use of AV technology,
2. *Viewing* – from passive to controlling the direction and pace of the animation, possibly including the use of different views or accompanying textual or aural explanation,
3. *Responding* – answering questions about the AV content presented by the system. This can for example include questions about the coding, efficiency, debugging or a prediction of the next step(s) in the algorithm,
4. *Changing* – modifying the AV content, for example by providing specific input data,
5. *Constructing* – building a new visualization of a given algorithm or data structure, usually based on educator-defined limitations,
6. and *Presenting* – presenting AV content to an audience for feedback and discussion. The AV content may or may not have been created by the learners themselves.

Figure 1 illustrates the dependency of the different levels of engagement, with *Viewing* acting as the base for the higher forms of engagement. *No Viewing* is outside the figure's window, as it does not entail any use of AV software.

The basic hypotheses of the working group can be summarized as follows:

1. There is no significant difference between *No Viewing* and *Viewing* - that is, learners do not benefit from purely passive use of AV tools.
2. Apart from the first two hierarchy steps *No Viewing* and *Viewing*, each higher step can result in statistically significant (and thus measurable) better learning outcomes than the previous steps.

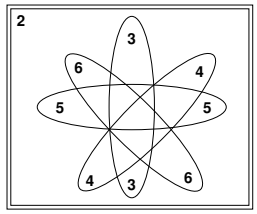


Figure 1: Possible Overlaps in the Engagement Taxonomy. The basic regions are 2 (*Viewing*), 3 (*Responding*), 4 (*Changing*), 5 (*Constructing*), and 6 (*Presenting*).

The working group report also includes a set of example experiments to verify these hypotheses. However, there is (at least) one obstacle for interested users: evaluating the different levels of engagement requires a tool which can handle the appropriate functionality. For example, to verify if *Responding* is more beneficial than *Viewing*, the AV tool used for the experiment must be able to handle question pop-ups during the visualization.

Currently, this type of interaction support is mostly restricted to highly context-specific tools such as *PILOT* (Bridgeman et al., 2000) or *Interactive Data Structure Visualization (IDSV)* (Jarc et al., 2000). Both systems are essentially restricted to graph problems and let the user select the next graph node or execution step for the given algorithm and graph.

The *MatrixPro* system (Karavirta et al., 2004) offers far-reaching interaction. Users can directly interact with given data structures and simulate underlying algorithm. The animation essentially illustrates the reaction of the underlying data structure to the user's graphical insert or delete operations. Users are not asked any questions during the animation, apart from the lead-in question outlining the task they have to accomplish. *MatrixPro* is difficult to place in the engagement taxonomy – it certainly addresses level 2 (viewing) and 4 (changing the values). It also supports parts of level 5 (constructing). The support for level 3 (responding) is still too limited to use *MatrixPro* for experiments that require this level of engagement.

2 Current Interaction Support in *JHAVÉ* Using Animal

JHAVÉ (Naps et al., 2000) is one of the few general-purpose AV systems that offer full interaction support. This includes free-text answers, multiple choice, true/false questions, and showing a documentation page in HTML format.

The interaction implementation for *JHAVÉ* contains an *infoFrame* class which is used to display questions and HTML documentation. The supported question types cover the standard elements *true / false*, *select m out of n*, and *fill in the blanks* (free-text). With a small effort, the AV system Animal could be adapted to use the same interaction front-end (Rößling and Naps, 2002).

The implementation works fine for the current AV systems incorporated in *JHAVÉ* - namely, *GAIGS*, *JSamba* and Animal (Rößling and Naps, 2002). However, the structure of the implementation prevents easy adoption into other systems. This is mostly due to the following three factors: notation, embedding, and portability.

First, the interaction components are defined in a scripting notation that has to be parsed. This effectively means that any AV system author interested in incorporating the interaction support has to modify the parsing component to address a set of other commands. How easily this can be accomplished depends on the structure of the AV system in question.

We would ultimately have n nearly identical parsing components in n different AV systems - where each author has to reinvent the wheel for his or her system. Furthermore, the notation used for the interaction components may not fit the notation used so far. For example, some systems use integer values as object IDs, whereas JHAVÉ's interaction support uses strings placed in double quotes.

Second, the interaction commands are directly embedded into the animation code. Both *JSamba* and ANIMAL use a scripting language to generate the visualization content. To ensure that the interaction elements occur at the right point in time, a specific place-holder is used. The animation ends with the actual definition of the interaction elements.

Finally, parts of the *infoFrame* code are system-specific and cannot easily be adapted to other systems. For example, *JHAVÉ* offers a *quiz for real* mode. In this mode, each new question freezes the animation and stores the answer (once it is given) over the WWW in the answer database on a central server. Other systems, such as *Interactive Data Structure Visualizations* (Jarc et al., 2000), use a different – and probably incompatible – approach for storing learner answers.

3 Designing a Tool-Independent Interaction Component

To address the problems listed in the previous section, we have to take three steps:

- separating (as much as possible) the definition of interaction elements from the animation itself,
- providing a general parser and graphical interface for interaction events,
- and offering a flexible evaluation back-end for handling answered questions.

Our new component requires only the invocation of the following two methods in the *avinteraction.InteractionModule* Java class:

interactionDefinition("location") defines the location of a file describing the interaction components. The structure of the file and indeed the type of location should be flexible. For example, *location* could be a scripting file on the local hard disk, an XML file on a Web server, or a dynamically generated resource anywhere on the Web.

Optionally, either the MIME type of the definition file or a concrete parser can be passed in as a second parameter. In this way, it is easy to ensure that nearly any implementation can work without touching the core implementation of the *avinteraction* package.

interaction("interactionID") invokes the interaction element with ID *interactionID* at this point of execution. The ID can be any arbitrary string, and thus may also encode contextual knowledge such as the exhibited "skill" of the learner (Rößling and Naps, 2002), provided that the input parser or interaction generator support this.

Both methods are easy to use and should therefore be easy to incorporate in most AV systems. Depending on the type of underlying AV system, the appropriate places in which the methods are invoked can be determined by embedding commands in the scripting notation, adding (non-visual) elements, adding appropriate declarations for declarative systems, or adding API invocations for API-based systems. This also holds for the determination of the command parameters. The only restriction is that the *interactionDefinition* command must appear before the *interaction* commands. Typically, the definition is placed in the animation header or among the first animation commands.

AV system developers may also need a flexible evaluation back-end that can store the learner's answers and possibly submit them to other evaluation engines. For example, answers for the *JHAVÉ* / ANIMAL cooperation have to be submitted to the web server for *JHAVÉ* in the *quiz for real* mode. For this end, we need a flexible interface that allows us to easily switch evaluation back-ends. This is relatively easy to do in Java if coded carefully.

Figure 2 shows the schematic structure of the proposed interaction component. The upper part shows how a single *interactionDefinition* and several *interaction* commands are sent to the interaction parser. After parsing the associated interaction definition file, the interaction

component has an internal representation of the visualization's interaction elements. The interaction parser is designed in modules to support customized definition file formats for defining the interaction components.

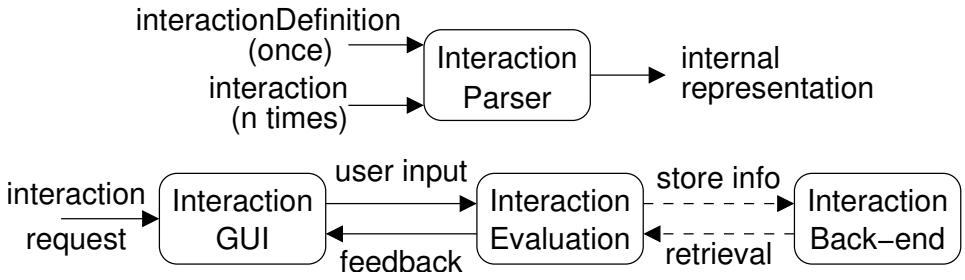


Figure 2: Schematic Overview of the proposed interaction component

The lower part of Figure 2 illustrates the typical data flow during interaction. The interaction request (with a concrete request ID) is resolved in the internal representation of interactions and then pops up the GUI, if not already visible. After a while, the user provides his or her input, which is sent to the built-in evaluation component. Feedback is then sent back to the user over the GUI.

Optionally, the learner's answer may also be sent over the interaction back-end to the appropriate "listener", typically a data base or learning management system. User data may also be incorporated into the evaluation of the answers, for example by dropping certain questions after the user has shown sufficient proficiency in the topic (Rößling and Naps, 2002).

4 Supported Features

The currently supported interaction types are identical to those offered by JHAVÉ / ANIMAL:

- *true / false* questions, containing one radio button for *true* and *false*, respectively,
- *free-text* questions, where the user can enter an arbitrary text as the answer. Note that this type of input is notoriously difficult to evaluate, due to the large number of possible answer formulations. For example, to indicate that the answer to a given question is "the first array element", the user could write "a[0]", "first element", "the first" or "7".

  For practical reasons, the evaluation of the free-text input is restricted to a case-insensitive string matching. Note that sub-string matching ("any expression that includes ...") may be more hindrance than help: clever learners will simply incorporate several possible answers into their text and hope that at least one of them matches. Even in graded tests, the educator may rely on the automatic assessment rather than checking the exact input for each submission. To alleviate this situation, the educator can define an arbitrary number of correct or acceptable answers.
- *multiple choice* questions, where the user selects a subset of the defined answers,
- *HTML documentation* links to further documentation or other pages.

Each question is set in a text area including scroll bars if necessary. The AV interaction designer can provide a comment for each possible answer to tell the users why their answer is correct or wrong. In the latter case, the answer may also give hints to the correct answer.

The package contains two default back-ends, both of which simply echo any user input on the standard output. The default parser is called *AnimalscriptParser* and parses interaction elements based on the notation used in JHAVÉ and ANIMAL (Rößling and Naps, 2002).

```
%Animal 2
interactionDefinition "http://www.algoanim.net/iav/interactionDemo"
array "a" (100, 100) length 5 int {3, 7, 8, 2, 12}
interaction "sortComplexity"
# further animation commands...
```

Listing 1: Example animation code including interaction components

Listing 1 shows a very brief example interaction script. In this listing, the command notation is nearly identical to the API calls, except for the missing parentheses and class names. Mapping the script input to API invocations is therefore trivial. The implementation also prevents "curious" learners from gleaning the correct answer from the animation code itself. The definition of the interaction elements is defined in a separate file which has to be parsed by the interaction manager – *not* by the AV system.

Figure 3 shows an example screen shot of the interaction element using the multiple choice answer defined in Listing 2. The user has selected two correct answers. The feedback for each selected answer is displayed at the bottom of the window. The system also comments if correct answers are missing or incorrect answers were chosen. For layout reasons, this feedback is not shown - but note the scroll bars of the text area at the bottom of the window.

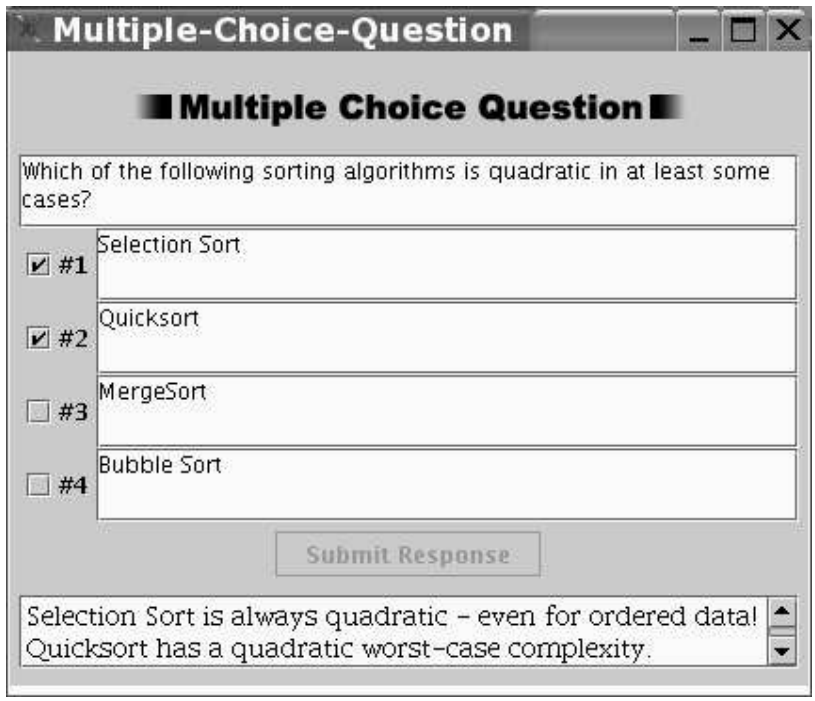


Figure 3: Example Interaction Screen Shot

Especially in long visualizations, users may become bored or even frustrated by repeatedly being forced to answer easy questions. On the other hand, the personal perception of the depth of understanding and the actual level of knowledge tend to differ. Additionally, not all questions are equally difficult. We should therefore enable good learners to skip questions that they have answered correctly sufficiently often.

Listing 2 shows the example code for the example definition file for the animation. Line 2 declares the actual points to be gained in this question. Line 3 declares the question as belonging to a group of questions named "general complexity", which may be repeated until the learner has correctly answered questions from this group two times (`nrRepeats 2`). After

this, further questions of this question group are skipped.

The actual question is defined in lines 5 to 6. The rest of the definition contains one block for each answer. Each block defines an answer text (lines 8, 14, 20, and 26), ended by *endchoice*. An answer can be given a specific number of points. Here, the understanding that *Quicksort* is quadratic in *at least some cases* gives two points, while the same answer for *Bubble Sort* and *Selection Sort* gives only one point. Each answer also has its own comment to help the learner's understanding (lines 12, 18, 24, and 30). The question ends with the definition of the set of correct answers in lines 33-35. Usually, there should be at least one correct answer and one incorrect answer.

```
mcQuestion "sortComplexity"
points 4
questiongroup "general complexity"
nrRepeats 2
"Which of the following sorting algorithms is quadratic in"
" at least some cases?"
endtext
"Quicksort"
endchoice
points 2
comment
"Quicksort has a quadratic worst-case complexity."
endcomment
"Bubble Sort"
endchoice
points 1
comment
"Bubble Sort is indeed quadratic."
endcomment
"Selection Sort"
endchoice
points 1
comment
"Selection Sort is always quadratic - even for ordered data!"
endcomment
"MergeSort"
endchoice
points 1
comment
"Mergesort has a worst-case complexity of O(n log(n))!"
endcomment
answer
1
2
3
endanswer
```

Listing 2: Example interaction script

There is also a default feedback for correct answers ("Yes, you are right!"), in case the author has not provided a more concrete answer comment. The number of possible free-text input values is effectively indefinite. The *comment* entry for this question type therefore acts as a "catch-all" for all possible incorrect answers. The system currently does not support the specification of expected "incorrect answers" with predefined reactions. In general, free-text

answers require some leniency to prevent learner frustration caused by giving a correct but unrecognized answer.

Multiple choice questions always randomize the order of the answers before they are displayed. This way, the same question asked four times may still require clicking in different places each time. The evaluation of the question ends with the assignment of points. The *points* command in line 2 defines the number of points given if the question has been correctly answered. For most question types, assigning points is a straightforward process. The interaction back-end, as shown in Figure 2, has to decide if and how the points for a given question or all questions are shown to the learner. The default back-end simply displays the number of points reached on the default Java output stream.

For multiple choice questions, the actual point number is determined by several parameters. A number of points can be defined for the whole question (see line 2 in Listing 2). Additionally, every answer option can have its own number of (positive) points (lines 10, 16, 22, and 30 in Listing 2). The number of points assigned to each selected correct answer is added to a separate sum of points. Similarly, the number of points assigned to a selected incorrect answer is deducted from this sum. In this way, partly incorrect answers can be translated into a decrease of already gained points. If no point value is given, no points are deducted from the point sum - this is especially useful for "radio button" (1:n) questions.

If the sum calculated in this way is less than zero, it is reset to zero. If a question is answered correctly by selecting exactly all correct answers, and the sum of points is still less than the total number of points, the total number of points possible is awarded. This can happen if the author does not ensure that the number of subpoints add up to the desired total points.

Developers can easily implement their own parser based on a simple interface. The new parser class should belong to the package *avinteraction.parser*. Making the parser visible for the system is achieved by modifying the ASCII-based configuration file *parser.config* that accompanies the package. Here, the MIME type of the input format has to be mapped to the parser class name in the notation shown in this example:

```
"text/animalscript" = "avinteraction.parser.AnimalscriptParser"
```

5 Summary and Further Work

We have presented the design for a tool-independent interaction component. Principally, this component can easily be incorporated into most existing AV systems based on Java or capable of invoking Java methods. For example, we finished a *proof of concept* embedding of our package into the *JAWAA* system (Akingbade et al., 2003) within just five hours. The largest part of this time was actually spent in determining the concrete classes to modify inside JAWAA and to figure out how the interactions could be integrated as "events" in JAWAA.

Due to the common GUI and evaluation component, AV system authors are not required to implement anything apart from the two simple interaction API invocations. More advanced or specific systems can also provide a special back-end to support answer feedback and advanced evaluation, for example for testing or scientific evaluation. They may also use special interaction parsers for other definition file formats, such as formats based on XML or interaction generation depending on database content. By using this modular structure in the component, we hope to ensure a high flexibility that enhances the component's usability.

Using the *avinteraction* component makes it far easier for AV system developers and users to design experiments using the *Responding* level of engagement. Such experiments could then be used to test the Working Group's hypothesis that responding to questions during a visualization may increase learning outcomes. Here, our package plays a key part in enabling not only AV system developers, but virtually *any* user to define interactions for a given system, and evaluate the way users interact and potentially benefit from this.

As further work, we plan to evaluate the ease of use of the AV interaction package. For this end, we need to convince authors of others tools such as *Jeliot* (Moreno et al., 2004) or *Matrix* (Korhonen, 2003) / *MatrixPro* (Karavirta et al., 2004). Any interested developer is encouraged to contact the first author of this paper in order to explore possible cooperations. Additionally, the system is freely available on `http://www.animal.ahrgr.de` under the link *Downloads*.

References

Ayonike Akingbade, Thomas Finley, Diana Jackson, Pretesh Patel, and Susan H. Rodger. JAWAA: Easy Web-Based Animation from CS 0 to Advanced CS Courses. In *Proceedings of the* 34^{th} *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2003), Reno, Nevada*, pages 162–166. ACM Press, New York, 2003.

Stina Bridgeman, Michael T. Goodrich, Stephen G. Kobourov, and Roberto Tamassia. PILOT: An Interactive Tool for Learning and Grading. *Proceedings of the* 31^{st} *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 139–143, March 2000.

Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing*, 13(3): 259–290, 2002.

Duane Jarc, Michael B. Feldman, and Rachelle S. Heller. Assessing the Benefits of Interactive Prediction Using Web-based Algorithm Animation Courseware. *Proceedings of the* 31^{st} *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 377–381, March 2000.

Ville Karavirta, Ari Korhonen, Lauri Malmi, and Kimmo Stålnacke. MatrixPro – A Tool for On-The-Fly Demonstration of Data Structures and Algorithms. In *Proceedings of the Third Program Visualization Workshop, University of Warwick, UK*, pages 26–33, July 2004.

Ari Korhonen. *Visual Algorithm Simulation*. PhD thesis, Department of Computer Science and Engineering, Helsinki University of Technology, 2003.

Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing Programs with Jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI 2004), Gallipoli (Lecce), Italy*, pages 373–380. ACM Press, New York, May 2004.

Thomas Naps, James Eagan, and Laura Norton. JHAVÉ: An Environment to Actively Engage Students in Web-based Algorithm Visualizations. *Proceedings of the* 31^{st} *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, pages 109–113, March 2000.

Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the Role of Visualization and Engagement in Computer Science Education. *ACM SIGCSE Bulletin*, 35(2):131–152, June 2003.

Guido Rößling and Thomas L. Naps. Towards Improved Individual Support in Algorithm Visualization. *Second International Program Visualization Workshop, Århus, Denmark*, pages 125–130, June 2002.

Taxonomy of Visual Algorithm Simulation Exercises

Ari Korhonen and Lauri Malmi
Helsinki University of Technology
Department of Computer Science and Engineering
Finland

archie@cs.hut.fi, lma@cs.hut.fi

Abstract

This paper presents a taxonomy for algorithm simulation exercises that allow to build learning environments that not only portray a variety of algorithms and data structures, but also distribute tracing exercises to the student and then automatically evaluates his/her answer to the exercises. The taxonomy systematically classifies the exercises into 8 separate categories that have 8 subcategories each. Each category is characterized and demonstrated by describing an example exercise that falls in the category.

The taxonomy provides a thinking tool to systematically diversify the set of possible simulation exercises. Thus, the taxonomy promotes new perspectives to come up with novel exercises of completely new genre. Moreover, we demonstrate a fully working web based learning environment that already includes implementations for such exercises.

1 Introduction

Today, algorithm animation is primarily utilized for both supporting teaching in the lectures and for studying in open and closed labs. A variety of systems are available for us on the web. However, from the pedagogical point of view, many of them lack the potential to give automatic formative or summative feedback on a student's performance, which is an essential factor in the learning process. Fortunately, the unambiguity of algorithms and data structures allows one to set up automatically assessed exercises and compare the student's solution to the correct model solution. This gives an opportunity to produce systems that not only portray a variety of algorithms and data structures, but also distribute tracing exercises to the student and then evaluate his/her answer to the exercises. One possible method for building such systems is *visual algorithm simulation* (Korhonen, 2003), which allows to practise, for example, such core CS topics as sorting algorithms, search trees, priority queues, and graph algorithms on a conceptual level without writing any code. When automatic evaluation of students' submitted work is included into the system, we refer to this as *automatic assessment and feedback of algorithm simulation exercises*.

Very few systems fully support algorithm simulation exercises with the automatic assessment and feedback capability. However, TRAKLA (Hyvönen and Malmi, 1993; Korhonen and Malmi, 2000), and its follower TRAKLA2 (Korhonen et al., 2003) both do, and they form the basis of our discussion. Some other systems support algorithm simulation exercises, as well, but only within a limited scope. PILOT (Bridgeman et al., 2000) is targeted to tracing exercises, but covers only graph algorithms and provides only formative feedback. On the other hand, "stop-and-think" questions requiring an immediate response from the learner, such as introduced in JHAVÉ (Naps et al., 2000), can be interpreted to be algorithm simulation exercises, too. The learner is supposed to understand the algorithm either by mentally executing the algorithm or by doing such a simulation with paper and pen. Thus, the system illustrates exercises where the learner is asked to manually trace an algorithm on a small data set. However, the system does not give feedback on the correctness of such a trace, but merely asks questions that should confirm that the learner has understood the concept. Finally, if we generalize the concept algorithm to be any well-defined procedure to solve computational problems, we can also bring in tools such as the problets introduced by Krishna and Kumar (2001). They illustrate problem generators on the topic of precedence and associativity of operators in which the learner is to evaluate expressions by solving sub-expressions in correct

order. The task is to solve exercises by following the predefined rules of operator precedence and associativity (that could also be expressed as an algorithm).

In this paper, we introduce a systematical classification for algorithm simulation exercises. Our aim is to illustrate the full range of exercises this method supports, and thus better discover their potential in supporting learning. The taxonomical evaluation divides the exercises into separate categories that are described by characterizing each category first. In addition, each category is demonstrated by describing an example exercise that falls in the category. We have a fully working web based learning environment that includes implementations for many of the exercises. Thus, you can try out the exercises in live action[1].

In Section 2, we briefly describe how to apply algorithm animation and simulation together to construct a *learning environment* with exercises for data structures and algorithms. Section 3 introduces the taxonomy of algorithm simulation exercises, and in Section 4 we present sample exercises in our learning environments TRAKLA and TRAKLA2. Finally, in Section 5, we discuss some aspects how this taxonomical approach could be applied in practice.

2 Algorithm simulation and animation in a learning environment

We define a learning environment as a system that is capable of meaningful interaction with the user with respect to the topic of the class to which this environment is attached. We refer to the users of such an environment as learners. In the context of data structures and algorithms, our vision is that a good learning environment should provide the learner a selection of interactive learning objects to view algorithms working (animation), interact with the animations, *i.e.*, control or simulate the algorithms, get feedback on the simulation in order to test one's knowledge on its working, and explore the general behavior of the algorithm through simulation with smaller or larger data sets. Moreover, the learner should be able to operate on the algorithm both on a conceptual level and on the implementation level to fully grasp its working. Implementing such a vision is a major task, and requires typically several different visualization tools. In this paper, we consider only working on the conceptual level, and the visualization of algorithm code execution is out of scope of this paper. First, we briefly define the concepts *algorithm simulation* and *automatic algorithm animation*, since they form the basis for the rest of the paper.

In visual algorithm simulation, the user manipulates graphical objects on the screen that are visual representations of actually implemented underlying data structures. Typically a simulation sequence consists of a number of context sensitive drag & drop operations, each simulating, *e.g.*, basic variable assignments, reference manipulations, or operation invocations such as insertions and deletions. The system interprets the operations and modifies the corresponding underlying data structures, such as arrays, lists or trees according to the operations, and automatically updates the visual representation on the screen. Thus, visual algorithm simulation applies automatic algorithm animation to update the screen. The conceptual difference between algorithm animation and algorithm simulation is that in the animation all changes in data structure representations are based on the execution of a predefined algorithm whereas in the simulation, the user is the active part initiating the changes. As a whole, a seamless combination of these methods, such as introduced in the Matrix application framework (Korhonen and Malmi, 2002), allows the user to explore the working of different algorithms and interact with the system in many ways.

If compared with plain algorithm animation tools, the simulation facility enables us to define new interesting types of algorithmic exercises, because the user-generated simulation sequence can be compared with a sequence generated by a true implemented algorithm. Thus, we can build exercises that train and test different aspects of the working of algorithms. We

[1]The research pages for TRAKLA2 learning environment at http://www.cs.hut.fi/Research/TRAKLA2/ includes fully working applets that demonstrate the algorithm simulation exercises.

can request the user to imitate a real algorithm with a given initial data, as depicted in Figure 1. Or, we can ask the user to solve a counter example: for a given algorithm, generate the initial data that produce the given output. There are also other possibilities as we can see in the next section.

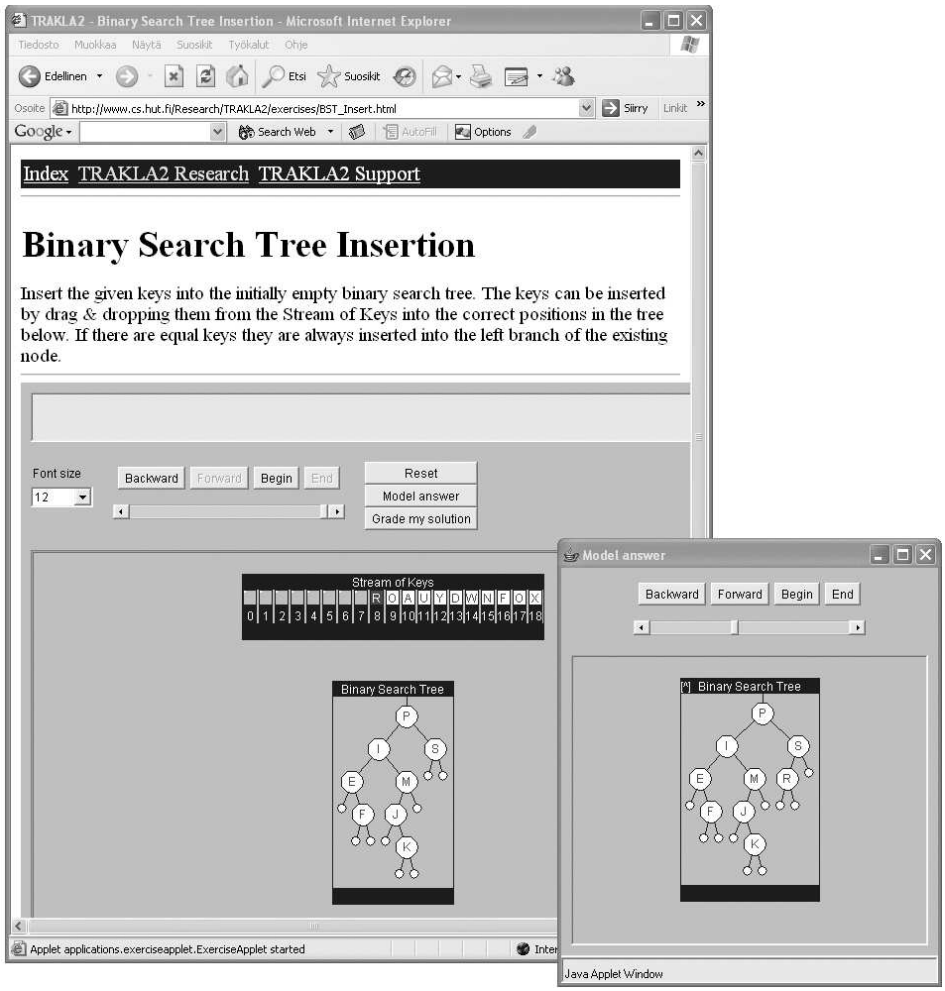


Figure 1: TRAKLA2 applet. The exercise window includes the data structures and push buttons. The model solution window is open in the front.

In the figure, we show a sample exercise in which the learner is to insert (drag and drop) the keys from the given array (initial data) one by one into the binary search tree. After completing the exercise, the learner can ask the system to grade his or her performance. The feedback received contains the number of correct steps out of the maximum number of steps. The learner can also view the model solution for the exercise as an algorithm animation sequence. The model solution is shown as a sequence of discrete states of the data structures, which can be browsed backwards and forwards in the same way as the learner's own solution.

3 Taxonomy

As the dynamics of a simulation is targeted to algorithms, we can derive the taxonomy of algorithm simulation exercises by examining the function $P : I \rightarrow O$ that an algorithm A is supposed to compute. An *algorithm simulation exercise* can employ any of the following three components, *i.e.*, the algorithm A (instructions to compute P), the input I or the output O, or any combination of them, while the other components are fixed (predefined in the assignment). Thus, an *algorithm simulation exercise* is a tuple $E_P = (A, I, O)$, where P

is the fixed problem to be solved by an algorithm A, I is the legitimate input set (a problem instance), and the solution is the obtained output set $O = A(I)$.

An *algorithm simulation exercise type* is a tuple $E_T = (X_1X_2X_3)$, where X_i is substituted by F if the corresponding E-component, *i.e.*, algorithm, input or output is fixed, respectively, and Q if it is in question. Most of the TRAKLA exercises ask the student to simulate a particular algorithm A with individually tailored input sequence $f_1, f_2, \ldots, f_k \in I$, and show how the corresponding structures change by determining the output sequence $q_1, q_2, \ldots, q_k \in O$. Thus, the exercise is of type $E_T = (FFQ)$ and we denote the input and output sequences as F_k and Q_k, respectively. The question is, in general, what is the output for the given algorithm with the given input. For example, F_k can be an ordered set of keys to be inserted into an initially empty binary search tree or it can be the parameter(s) an algorithm receives in each recursive call. Here, for each f_j the corresponding $q_j = A_{q_{j-1}}(f_j)$, *i.e.*, the binary search tree after each insertion or the computed result after each recursive call to A.

Two different subtypes for E-components can be identified: implicit and explicit. An *explicit question* (denoted by capital case Q) requires the learner to produce an answer for the question, for example, the output in the previous example. *Implicit questions* (denoted by lower case q) only require that the learner is familiar with the topic in question but no explicit answer is required and the learner may choose from among a set of alternatives. For example, the learner may be asked to apply any algorithm that produces topological sorting and not any particular one. Usually, this happens when there is more than one Q-component in an exercise. On the other hand, also fixed E-components can be implicit. For example, it is obvious what is the output structure while sorting an array of keys. Thus, we denote the implicit output as lower case f.

Eight different basic types of exercises can be named and characterized and 256 in total if the subtypes are taken into account. In the following, we summarize the 8 basic types briefly and give an example of each that includes also the subtype definitions.

1. $E_1 = (FF_if)$ - *Determining characteristics*: Which items in the array F_i are compared with the given search key $k \notin F_i$ in binary search?

2. $E_2 = (F_aF_iQ)$ - *Tracing exercise*: i) Insert the set of input keys F_i into an initially empty binary search tree. ii) Insert the set of keys F_i into an initially empty hash table using linear probing with the given hash function F_a.

3. $E_3 = (FQF_o)$ - *Reverse engineering exercise*: Determine a valid insertion order for the keys resulting the AVL tree F_o in question.

4. $E_4 = (FQq)$ - *Exploration*: Determine such an input string for Boyer-Moore-Horspool algorithm that satisfies the statement coverage (every statement is executed at least once with the test set). Describe the output of such an execution.

5. $E_5 = (QF_iF_o)$ - *Determining algorithm*: The following binary tree F_i was traversed in different orders. The resulting traversing order of nodes are F_o. Name the algorithms.

6. $E_6 = (qFQ)$ - *Open tracing exercise*: i) Trace topological sort on a given graph. ii) Compare several recursive sorting algorithms with each other. Show the state of the input array F after each recursive call.

7. $E_7 = (qQf)$ - *Open reverse engineering exercise*: Compare several sorting algorithms to each other. Determine an example input for each of them that leads to the worst case behavior.

8. $E_8 = (QQQ)$ - *Completely open question.* Let us consider the following broken binary search tree. All duplicate keys are inserted into the left branch of the tree, but the deletion of a node having two children replaces the key in the node with the next largest

item (from the right branch). Give a sequence of insert and delete operations with keys of your choice that results in a tree that is no longer a valid binary search tree (Hint: the search routine can only find duplicates from the left branch).

Obviously the taxonomy presented here is not exhaustive in the sense that there exist exercises that are not algorithm simulation exercises at all. However, the taxonomy gives us a systematic way to cover one particularly interesting area of exercises throughout, and consider how such exercises could be supported by visualization tools.

4 TRAKLA and TRAKLA2 exercises

TRAKLA (Hyvönen and Malmi, 1993; Korhonen and Malmi, 2000) was initially implemented in 1991 to assess manual algorithm simulation exercises, *i.e.*, the learners solved the exercises on pen and paper and submitted the solution to the TRAKLA server by email. Later on, a visual front end for editing the answer was added, but there was no change in the system capabilities, because the front end was a dummy drawing tool with no understanding on the underlying data structures. TRAKLA2 (Korhonen et al., 2003), on the other hand, is a web-based learning enviroment built on the Matrix framework that provides full support for visual algorithm simulation and automatic algorithm animation.

We summarize the types of exercises that TRAKLA and TRAKLA2 systems support in Table 1. We use the systems as a proof of concept to address the many possibilities that automatically assessed algorithm simulation exercises have in the broad context of data structures and algorithms.

Table 1: Examples of automatically assessed exercises supported by TRAKLA and TRAKLA2.

Exercise type	TRAKLA2	TRAKLA
$E_1 = (FFF)$	FF_if	FF_if
$E_2 = (FFQ)$	F_aF_iQ	F_aF_iQ, FF_iQ, fF_iQ
$E_3 = (FQF)$		
$E_4 = (FQQ)$	(FQq, Q implies q)	
$E_5 = (QFF)$		
$E_6 = (QFQ)$		qF_iQ, q implies Q
$E_7 = (QQF)$	(qQ_if)	
$E_8 = (QQQ)$	$QQ'q, Q'$ implies q	

All the exercise types marked for TRAKLA and exercise types FF_if, FF_iQ,and $QQ'q$ in TRAKLA2 have been in production use with hundreds of students. For the FQq exercise, we have a reference implementation. For the tree traversing exercises ($E_5 = (QF_iF_o)$) mentioned on page 121 and Huffman code exercise ($E_6 = (qF_iQ)$) introduced with TRAKLA, however, we have a little bit different scheme. They will be implemented as a tracing exercise ($E_2 = (FF_iQ)$), and an open reverse engineering exercise ($E_7 = (qQ_if)$), respectively. It should be noted, however, that also exercise types not marked for TRAKLA2 can and will be incorporated into the system in the future, but for brevity these are not discussed here any further. In the following, we map each example exercise type described above to an actual exercise implemented or designed.

1. FF_if; binary and interpolation search; the algorithm is explicitly determined as well as the parametrized input. The result is obvious as the key is not found from the structure. The user must determine which items in the array F_i are compared with the key to be searched during the search.

2. F_aF_iQ, FF_iQ, fF_iQ; linear probing, deque, radix sort; the algorithm can be parametrized as it is in case of linear probing, it can also be expressed implicitly as it is done on radix sort (the learner must determine which radix sort is simulated by examining the intermediate states of the input structure).

3. FQq (TRAKLA2 only); BMH-algorithm; determine such an input Q for Boyer-Moore-Horspool algorithm that satisfies the statement coverage (every statement is executed at least once with the test set). Describe the output of such an execution. The selected input Q implies the output q.

4. qF_iQ, q implies Q; topological sort, Huffman code; in both of these exercises several correct solutions are possible for a given input. The algorithm applied by the learner implies the resulting output.

5. qQ_if; Huffman code; the learner is asked to simulate Huffman's algorithm q to form the Huffman code f in order to decode the original text string Q_i. The actual visible input for the algorithm is the frequencies of the characters in Q_i.

6. $QQ'q, Q'$ implies q; broken binary search tree; many procedures can lead into a correct solution. Several correct answers are possible due to the nature of the exercise. The input keys chosen by the learner imply the resulting output.

As mentioned in the introduction, there are few other systems that support algorithm simulation exercises. PILOT (Bridgeman et al., 2000) is targeted to tracing exercises that employ graph algorithms. There is also an option to allow parametrized input graphs. Thus, the type of the exercises is $E_2 = (FF_iQ)$. On the other hand, "stop-and-think" questions requiring an immediate response from the learner introduced in JHAVÉ (Naps et al., 2000) can be interpreted to be $E_1 = (FF_if)$ questions where the learner determines characteristics of the given algorithm. Moreover, the same system illustrates exercises in which the learner is asked to manually trace an algorithm on a small data set. However, in this case, there is no automatic assessment involved.

5 Discussion

The taxonomy of algorithm simulation exercises was first presented by Korhonen (2003). However, an interesting analogy exists independently within a completely different area of teaching. In their paper, Sutinen and Tarhio (2001) present a similar example of problem classes that are applied for characterizing problem management in thinking tools. They also have three components (Start (input), Technique (algorithm), and Goal (output)) that they call dimensions. Again, these expand to eight classes, if restricted to binary values "open" and "closed" (in question and fixed above). This construction spans the creative problem management space.

Such a problem management space is useful for teachers in various ways. Regardless of whether we are designing a single course or a larger program, we can use this space as a reference to better identify what are the learning goals of the exercises. Typically the exercises change from closed exercises to more open ones, when students progress in their studies. In closed exercises students generally train specific skills and techniques whereas in open exercises they have to apply their knowledge to solve new and varying problems. However, the presented problem space provides for a broader view of this issue than such a one-dimensional closed – open axis.

Let us now consider more closely, how this applies to a data structures and algorithms course. When learning this topic, students first have to learn how various well-known algorithms, such as quicksort and binary search algorithms work. These can be trained with exercises of types FFF and FFQ (and possibly qFQ). Next, they should understand the behaviour

of the algorithms. Typically this is strongly related to mathematical analysis of presented algorithms, especially considering their worst-case behaviour. Attached to this theme, common assignments deal with generating worst case or best case input data for a given algorithm (type FQf), or more generally compare worst cases / best cases of a set of algorithms (type QQf).

In traditional education of the topic there are, however, seldom exercises in which students could explore the behaviour of the algorithm using different kinds of input sets. The obvious reason is that this may be clumsy on pen and paper, or laborious, if it requires coding. On the other hand, using visual algorithm simulation such exercises are easy to organize. Moreover, providing automatic feedback on the answers is often simple, as well. As an example, a student could be given an FQq type of exercise: "Insert the following keys into a red-black tree in such order that the height of the resulting tree is at least 6", or an FQF type exercise "Insert the following keys into a binary search tree in such order that the resulting tree is a complete binary tree". Another exercise of type FQq is "Consider the following weighted graph. You may modify the weight of at most 3 edges to create a graph in which both Prim's and Djikstra's algorithms starting at node A create the same spanning tree. Is this possible?". Note that in such exercises it is not enough just to apply the algorithm. The student has to recognize how the result is affected by changes in the input data, which is something else than understanding the worst-case or best-case behavior of the algorithm. Still, automatic assessment of the solution is straightforward.

Constructing new algorithms to solve new problems is a standard type of exercises in an algorithms course. Such exercises are of type QFF or QFq. If we allow the student freely demonstrate his/her skills on the course theme, we end up in QQQ type of totally open exercises, even though we can limit the exercise topic as we did in the broken binary search tree exercise.

In the previous discussion, we have provided classifications for existing exercises. Conversely, the teacher can try to devise exercises of each type in the taxonomy, to find out new ways of handling the topic. Especially exercises which are related to exploring the behaviour of the algorithm (FQQ, FQF) can enrich the commonly used exercise sets. The taxonomy thus promotes creating new course material.

Next, we continue to discuss some other aspects of the taxonomy. Two main characteristics should be taken into account while designing new algorithm simulation exercises. First, we should make clear how these exercises are delivered to the students. We identify two main classes here, namely *closed labs* and *automatically assessed exercises*. The main difference between these two is the available support from instructors. In closed lab sessions, we assume that an instructor is present and takes actions to guide the learner by asking specifying questions, giving additional feedback, and so on. The exercises can be solved by "exploring" the state space and the correctness of such an exploration can be assessed by the instructor. Thus, the exercises are more open in their nature. On the other hand, automatically assessed exercises should be self-explanatory so that the learner can cope with the assignments by himself. In addition, the feedback should be explicitly targeted to the assignment in question and aid the learner to find the correct solution. Roughly speaking, the most open questions (types QFQ, QQF, QQQ) are more suitable for closed labs while tracing exercises (FFF, FFQ, FQF) suit well to automatic assessment. There are, however, counter examples that contradict this discrete view.

Finally, we mention an important characteristics (especially with automatically assessed exercises) in which the fixed E-components can be parametrized (denoted by subindex) so that each learner has his own individualized exercises. For example, in the tracing exercise $E_2 = (FF_iQ)$, where keys are inserted into an initially empty binary search tree, the keys F_i to be inserted can be randomly drawn from the set of all possible keys. Thus, each learner has an individually tailored set of keys to be inserted. This is the approach used both in TRAKLA and TRAKLA2 although they have slightly different policies concerning

resubmission of solutions. In TRAKLA exercises, the learner is allowed to resubmit his/her solution to a particular exercise to the server only a few times, and each time the initial data remains the same. In TRAKLA2, on the other hand, the initial data is randomized each time the learner wants to continue after requesting grading or the model solution, but then there is no limit for the number of grading requests.

In the case of input and output components, the randomization is trivially achieved by randomly picking a suitable number of keys into the corresponding structure. Of course, some constraints must be taken into account in order to prevent the exercise turning out to be trivial (for example, the AVL tree insertions should include both single and double rotations). However, parametrized algorithm exercises $E_{1...4} = (F_a XX)$ are slightly trickier. Usually, the implementation of such a parametrization depends on the algorithm. For example, in linear probing, the hash function $h(k) = (k+p) \bmod q$ can be parametrized by individually tailoring p and q.

6 Summary

We have presented a novel method for classifying algorithm simulation exercises. The presented taxonomy can be used not only as a classification tool, but also as a design tool for teachers when they create new exercises for their data structures and algorithms courses. The taxonomy can give more insight into this design process, and as such aid teachers to refine the learning goals they wish to set up for their students. It seems that in most cases the exercises can be supported with automatic feedback on students' answers. Our TRAKLA2 has demonstrated this widely with tracing exercises, but we have a lot interesting work to do to implement new types of exercises in different fields of algorithms.

Finally, we hope that our taxonomy will promote other systems to implement algorithm simulation, and apply these ideas in other environments, as well.

References

S. Bridgeman, M. T. Goodrich, S. G. Kobourov, and R. Tamassia. PILOT: An interactive tool for learning and grading. In *The proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, pages 139–143. ACM, 2000. URL `citeseer.nj.nec.com/bridgeman00pilot.html`.

Juha Hyvönen and Lauri Malmi. TRAKLA – a system for teaching algorithms using email and a graphical editor. In *Proceedings of HYPERMEDIA in Vaasa*, pages 141–147, 1993.

Ari Korhonen. *Visual Algorithm Simulation*. Doctoral thesis, Helsinki University of Technology, 2003.

Ari Korhonen and Lauri Malmi. Algorithm simulation with automatic assessment. In *Proceedings of The 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, pages 160–163, Helsinki, Finland, 2000. ACM.

Ari Korhonen and Lauri Malmi. Matrix — Concept animation and algorithm simulation system. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 109–114, Trento, Italy, May 2002. ACM.

Ari Korhonen, Lauri Malmi, and Panu Silvasti. TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. In *Proceedings of Kolin Kolistelut / Koli Calling – Third Annual Baltic Conference on Computer Science Education*, pages 48–56, Joensuu, Finland, 2003.

Aravind K. Krishna and Amruth N. Kumar. A problem generator to learn expression: evaluation in CSI, and its effectiveness. In *Proceedings of the sixth annual CCSC northeastern conference on The journal of computing in small colleges*, pages 34–43. The Consortium for Computing in Small Colleges, 2001.

Thomas L. Naps, James R. Eagan, and Laura L. Norton. JHAVÉ: An environment to actively engage students in web-based algorithm visualizations. In *Proceedings of the SIGCSE Session*, pages 109–113, Austin, Texas, March 2000. ACM.

Erkki Sutinen and Jorma Tarhio. Teaching to identify problems in a creative way. In *Proceedings of the 31st ASEE/IEEE Frontiers in Education Conference*, page p. T1D813. IEEE, 2001.

What a Novice Wants: Students Using Program Visualization in Distance Programming Course

Osku Kannusmäki, Andrés Moreno, Niko Myller, and Erkki Sutinen
Department of Computer Science, University of Joensuu
Joensuu, Finland

{okannus, amoreno, nmyller, sutinen}@cs.joensuu.fi

1 Introduction

Evaluation is an important part of the development cycle of a new application. Especially, software tools for learning and teaching should be evaluated early on and the evaluation should guide the application's implementation and development.

The Jeliot family is a collection of program and algorithm visualization tools designed for novices learning programming, algorithms, and data structures (Ben-Ari et al., 2002). The latest member of the Jeliot family (see `http://cs.joensuu.fi/jeliot/`) is Jeliot 3 (Moreno et al., 2004). It is a program visualization tool that is based on the automatic animation of Java programs.

During the development of the different versions of Jeliot, evaluation of the actual use of the software has been an important part of the development process (Sutinen et al., 1997; Markkanen et al., 1998; Lattu et al., 2000, 2003; Ben-Bassat Levy et al., 2003). Qualitative descriptions of the use of the tool and its limitations have been fruitful for further development of Jeliot (Ben-Ari et al., 2002).

In this paper, we present some results from a qualitative analysis of the use of Jeliot 3, as well as students' requests and proposals for the development of Jeliot 3. The students were taking the second programming course in the ViSCoS (Sutinen and Torvinen, 2003) program at the University of Joensuu. Qualitative information was gathered from different sources such as forums and emails with comments and exercises. All these documents were later compiled to form a knowledge base from which we extracted our preliminary conclusions.

2 Related Work

Most software visualization tools have been empirically evaluated in order to test their effectiveness in teaching new concepts (Hundhausen et al., 2002). Most of the results summarized in Hundhausen et al. (2002) were obtained from tests done after, and maybe before, using the visualization tools (pre- and post-tests). However, Kehoe et al. (1999) proposed not to depend on the results of tests, but rather on the results of performing certain tasks in a less constrained environment—a homework learning scenario. Empirical evaluations, while providing important information, are difficult to performed in a distant learning course.

Bassil and Keller (2001) use a method similar to ours when evaluating seven software visualization tools. They use a qualitative and quantitative approach to compare those tools. First they try to fit the tools into the taxonomy described by Price et al. (1993); then they collected the users' opinions of those tools. The participants of their study mentioned the benefits of SV tool they were using. They also proposed further development of the tool of their choice, by integrating it with third-party tools. However, their evaluation was of tools used in industry, not those used in education.

3 Jeliot 3

Jeliot 3 visualizes the execution flow of a Java program by showing the current state of the program (e.g., methods, variables, and objects) and animations of expression evaluations and loops. Jeliot 3 evolved from a previous version called Jeliot 2000. The new version was

developed in order to achieve two new goals: to provide support for object-oriented programs, and to improve software modularity. Jeliot 3 still retains the user interface and the animation style from Jeliot 2000.

Currently, Jeliot 3 animates a larger subset of the Java language than Jeliot 2000, with features like values and variables of primitive data types (e.g. int, boolean, char), strings, primitive type 1-dimensional arrays, expressions including operations and assignments, control structures, error reporting and I/O operations. Furthermore, it also animates object-oriented programming (e.g. inheritance and method calls). The implementation of Jeliot 3 now makes it easier to develop new extensions due to its modularity and through the use of program trace code (M-code) that provides the means for communication between the visualization engine and a Java interpreter, DynamicJava (Hillion, 2002).

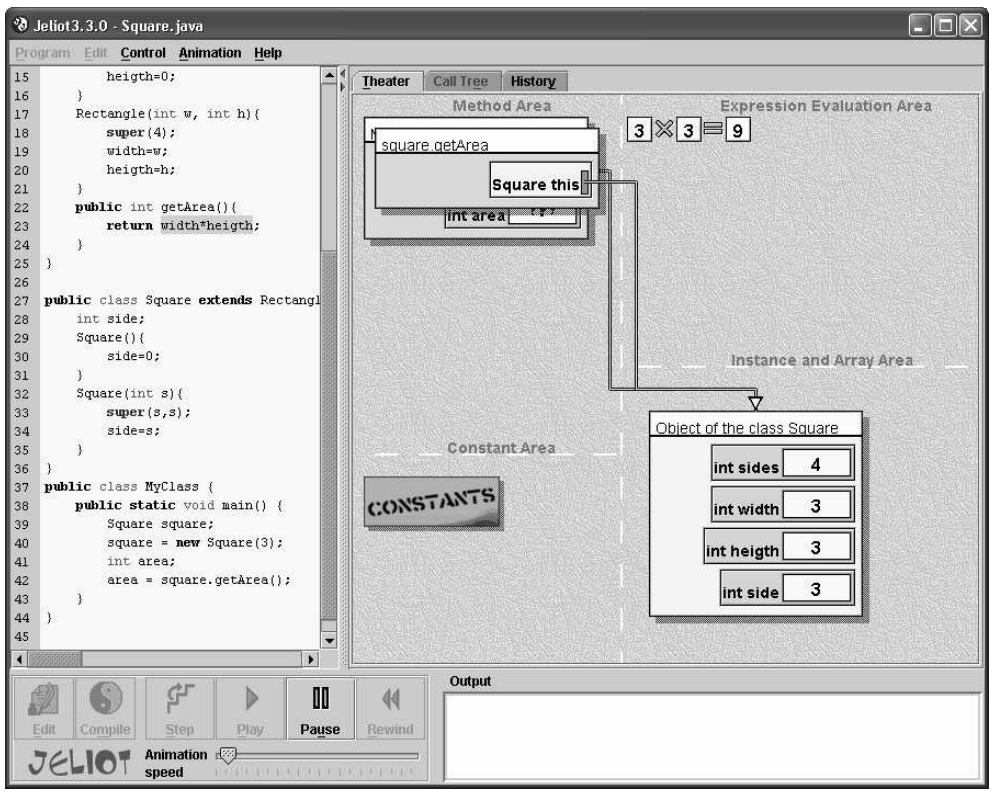


Figure 1: The user interface of Jeliot 3.

The user interface of Jeliot 3 is shown in Figure 1. Interaction takes place in the editor pane which is situated on the left side. There, users modify the source code. By pressing the Compile button, animation begins in the main pane, called the theater, and the editor pane follows the execution of the source code by highlighting the portion of source code being animated. The theater is divided into four sections:

- The *Method area* displays method frames. The frames contain the local variables of the methods.
- The *Expression Evaluation Area* animates the evaluation of the expression. Values are animated from their origin (e.g. a variable value in a method frame) to their place in the evaluation expression area. Complex evaluations are shown as a stack. Input boxes asking for input are laid out here, integrated within the normal flow of the animation.
- Constants appear from the *Constant Area* and are animated like variable values directly to a variable field, or to fill an expression in the expression evaluation area. Furthermore, the static variables are shown in this area.

- Finally, the *Instance and Array Area* displays instances of objects and arrays. They are connected to variables with arrows, indicating the reference semantics of Java. Output data are animated from the theater to the output console at the bottom of the window, where they are shown and stored.

Jeliot 3 can be used in several ways for teaching and learning to program. Here are some examples:

- Lecturers can use Jeliot 3 as a part of the lecture material. They can explain the different concepts of programming through Jeliot animations. This will facilitate the construction by the students of the correct relationship between the animation and the concept, and enable them to apply it later with a reduced possibility of error (Ben-Bassat Levy et al., 2003).
- The students may use Jeliot 3 by themselves after lectures to do assignments.
- Jeliot 3 can be used in an interactive laboratory session, where students may utilize their recently acquired knowledge by writing programs and debugging them through Jeliot 3.
- Finally, Jeliot 3 provides a tool that can aid in courses where external help is not available (e.g. in distance education). Its visualization paradigm creates a reference model that can be used to explain problems by creating a common vocabulary between students and teacher (Ben-Bassat Levy et al., 2003).

4 Empirical Study

As a part of the evaluation study of Jeliot 3, students used Jeliot 3 during their second course of programming—object-oriented programming (3 ECTS points)—in the Virtual Studies of Computer Science (ViSCoS) (Sutinen and Torvinen, 2003) distance learning program at the University of Joensuu's Department of Computer Science. In this study, our aim was to find out how students used the tool and what features they would like to have included in Jeliot in the future.

4.1 Method

In this evaluative case study, we used qualitative methods. With these methods, we tried to determine how students used the tool and what kind of features the novice users of Jeliot 3 would like to have in a visualization and program development tool. The answers to exercises and texts from discussion forum messages from the course were collected and analyzed. Since it was an exploratory study, our aim was not to give precise recommendations about what should be done. Rather the aim of the study was to describe the different ways of using the tool and to analyze the suggestions for improvement given by our sample of programming students.

The texts were analyzed by the third author of this paper who has also been involved in designing and implementing the system. However, this is an exploratory study, the researcher's involvement should not strongly affect the results. During the first reading, the material was coded and the focus of the study was decided. The methodological approach taken was similar to the method Lattu et al. (2003) used in their qualitative study concerning the use of visualization tools as demonstration aids. After the first reading, the coding was revised and the codes were grouped into three main categories. The *usage patterns* category describes the different uses of the tool. The *usage problems* category describes the problems that students encountered while using the tool. Those problems are divided into general, visualization, language, and code-editing related problems categories. The students also described their feelings and opinions, both positive and negative, about using Jeliot 3 as a visualization tool; these descriptions were put in the category *opinions and suggestions*.

4.1.1 Participants

The participants were secondary school students who were taking part in the ViSCoS program. During the course of study, they were taking the second programming course in the ViSCoS program, which dealt with object-oriented programming.

There were 57 students who took part in the second programming course and who agreed to be included in the study. However, only 35 of the students actually returned any assignments that could be analyzed.

Since some of the students were more experienced with computers, especially in programming, than others, and this could have affected the requirements of the program visualization tool for the students. A few students had previous knowledge about Jeliot 2000 and were familiar with its basic features. This is important because the user interface of Jeliot 3 is very similar to Jeliot 2000.

4.1.2 Procedure and Materials

The students used Jeliot 3 in four assignments. They had about one and one half weeks to complete each of the assignments. This means that Jeliot 3 was used, in total, for about six weeks. Each assignment consisted of two questions where students were not required to use Jeliot 3, and two questions where they were required to use Jeliot 3. The assignments were related to the learning material that was divided into chapters, so that each chapter could be completed in about a week. After each assignment, the students were asked to reflect on their usage of Jeliot 3. In addition to the students' reflections, we used the messages on the discussion board, and the feedback emails from the course in the analysis.

The grades of the previous course and this course were used to divide the students into strong ($n = 9$), mediocre ($n = 12$) and weak ($n = 14$). These categories were then used to indicate different levels of requirements as the students had different knowledge levels.

4.2 Results and Discussion

Four different patterns of doing the exercises were found. In two of the patterns, Jeliot 3 was used. In the other two patterns, students only used Jeliot 3 to be sure that their program ran on Jeliot 3, or they did not use it at all.

- Jeliot was used for coding and visually debugging and testing the program.
- A text or code editor was used to code the program and Jeliot was used to visually debug and test the program.
- A text or code editor was used to code the program and a standard Java compiler and JVM were used for debugging and testing.
- An advanced IDE (e.g. Eclipse) was used to code, debug and test the program.

Figure 2 shows how the different patterns appeared in the different user groups. As there were several assignments during the experiment, the same student could report different usage of the tool during different assignments, so the bars can add up to more than 100 per cent; some students did not report anything so some groups do not add up to 100 per cent.

In the first course of programming, students could choose the environment and working style that suited them. During that time they may have got used to a certain environment and working style. When they needed to switch to Jeliot 3, the switch may have indicated changes to their working style, and this could have increased the discomfort with Jeliot 3 and caused the rejection of the tool.

On the other hand, some students found Jeliot to be a nice environment to work with and used it to solve the problems as suggested. Some of the more advanced students used

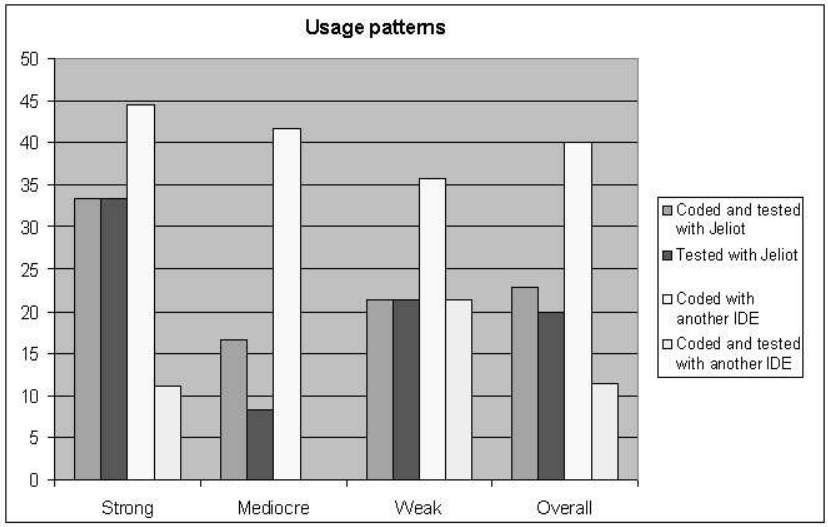


Figure 2: Patterns of use found from the students answers (percent).

advanced IDEs or just a simple text editor, Java compiler, and JVM for development. This indicates that already in the second programming course students' requirements for a learning tool have changed according to their knowledge level.

Different kinds of *usability problems* were identified from the students' answers and comments. They were divided into four different subcategories: *general*, *language*, *editor and visualization related problems*, and *propositions*.

Most of the general usage difficulties had one common feature: they were somehow related to the fact that Jeliot 3 is still under development and there are bugs and unimplemented features that can cause problems. The fact that students needed to install a second version of the tool during the course produced some extra problems. The installation problems of Java SDK also contributed to some of the adoption issues. These problems should be solved as the tool matures.

In several cases, students said that the *error messages were unclear and hard to interpret*. Error messages are an important part of the learning process and they need to be modified according to the user's knowledge (Hristova et al., 2003; Lang, 2002). This indicates that novices would need different kinds of error messages compared to more advanced users. In a similar way, example programs and the user guide of Jeliot were said to be insufficient.

Interestingly, several students seemed to have a misconception about the Java language used by Jeliot 3 since they claimed that *Jeliot 3 was not using standard Java*. However, the subset of Java language that can be visualized with Jeliot 3 is in accordance with Java Language Specification. The subset, however, was different from the one they had been used to, so this may have explained the complaints. The main differences between the Java specification and the subset of Java implemented in Jeliot 3 were: different I/O classes, the Java language support was not full, and the possibility to write `main()` instead of `main(String[] args)` at the beginning of the program (even though `main(String[] args)` is also accepted in Jeliot). The reason for accepting the simpler `main()` method was that it simplifies programs for the novices, so that during the first weeks they do not need to understand parameters and methods.

The students had gotten used to the code editor that they had been using during the first programming course. This seems to have affected the students' requirements for the editor since find & replace and other common editor commands, syntax highlighting, auto-indentation, and bracket indication were requested. The error messages were also requested to be shown during coding as in Eclipse.

Currently, Jeliot 3 requires that all the classes be in a single file, because we want to reduce complexity, and furthermore, code visualization is easier to grasp when the whole program is in a single file. Several students proposed that classes should be in separate files and that it should be possible to run several files at the same time. This is reasonable demand when the programs grow larger as in the second programming course, but before that several files can just confuse the novices.

The problems related to visualization came mostly from those students that used Jeliot 3 only to test the programs, while using an IDE or a code editor and a Java compiler. They complained that the animation was too slow, and they even wanted to have it disabled, or at least to have the possibility to only visualize certain lines of code. Some of the students also said that they can simulate the program in their mind and do not need a visualization tool to do it.

Students also proposed that Jeliot should be used in the first programming course, but not in the second, because it was not needed anymore by that time. The authors of this paper agree with the statement that it should be used in the first course, and that possibly in its current form the tool is not needed in the second programming course by the average student. Nevertheless, it can still help the weak students in the second programming course (see Figure 3). Furthermore, if students would have used Jeliot 3 in the first programming course, it would not have led to a situation where students were not used to the visualization tool and found it hard to change their working style to accommodate a new tool.

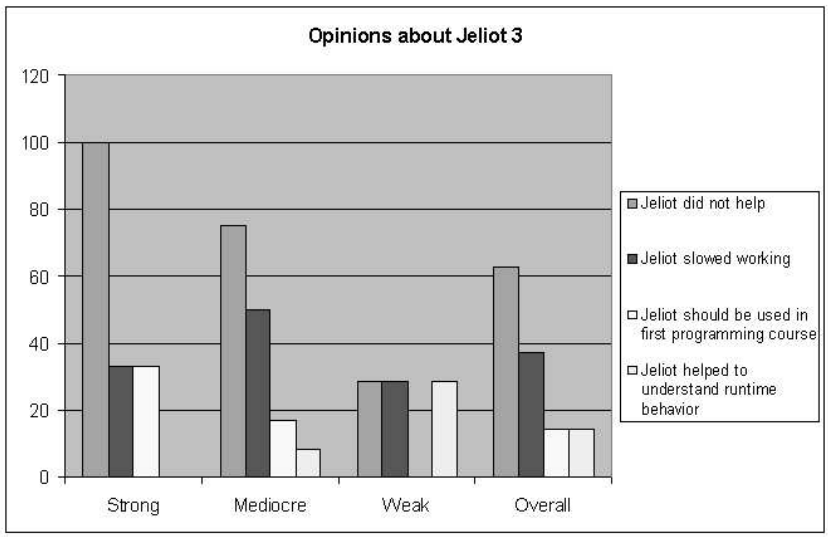


Figure 3: The opinions about Jeliot divided into different knowledge levels (in percentages).

Although students were encouraged to use Jeliot, some students thought that they did not need the tool and regarded it as a waste of time. This is similar to the findings of Ben-Bassat Levy et al. (2003) who found that the strongest students refused to use the previous version of Jeliot, Jeliot 2000. However, in our case, all but the weakest students refused to use Jeliot (see the Figure 3).

One of the students complained that she has a different model of computation that is incompatible with the model that Jeliot uses. The student, however, did not report anything more about her model of computation. If there really are different, but effictive models, developers can program different views or animation paradigms to Jeliot 3 by interpreting the intermediate code generated by Jeliot 3.

In one message, it was mentioned that Jeliot 3 contains too many graphics and that a plain debugger view should also be available. A view that would show classes and their methods and fields was also proposed.

Finally, students also found some positive points concerning when to use Jeliot 3:

- The if-statements and loops were made understandable to the students by Jeliot 3.
- Jeliot 3 helped students grasp objects and their use.
- Jeliot 3 showed in a step-by-step way what happens inside the program, what is wrong with it, and where the errors are.
- The compilation and execution of a Java programming with Jeliot is actually faster although the running time is longer.

Mostly of these comments came from the weak students. The three first comments were anticipated as that is what Jeliot is developed for. The fourth one is also understandable, but the reason for it is that students do not need to compile the programs, as they do when they are using Java compiler from command line. The students need to only push the compile button and the source code is then send to the interpreter which creates the animation.

5 Conclusions

We have described a program animation system called Jeliot 3 and presented some preliminary results from an evaluative study of the use of Jeliot in a distance education context. Students' answers to assignments were qualitatively analyzed and some future research areas were found. The analysis is still preliminary and a deeper analysis is needed. However, it is already clear that in the early stages of learning to program, the needs for programming environments change and that students with different levels of knowledge need different tools.

Advanced students, and even students with just some experience in programming, are very sensitive about changing the tool they have used, unless it provides a significant improvement over the previous tool. This means that the individual characteristics of learners, including levels of knowledge, should be taken into deeper consideration as the students requirements change rapidly.

Jeliot 3 could be improved in several ways. It can become a tool that can help the novices during the first programming course, and then an additional plug-in tool would be used together with an advanced IDE to provide some help for more advanced students. Jeliot 3 is fully automatic and supports novices learning to program, but Jeliot 3 probably could support more advanced students with a semi-automatic view where students could easily modify the properties of the program visualization as is possible in Jeliot I.

In this study, we have identified different kinds of problems that are typically of the problems found by the user of Jeliot 3. The issues can be divided into several categories such as issues related to Jeliot 3's philosophy, its implementation, and programming language considerations. The next step in the research will be to focus on one of these problem areas more carefully. The subsequent analysis could be supported with evaluation that is tightly connected to tool usage, for instance, by logging the users' actions in order to get feedback during the usage of the tool and not just after it.

References

Sarita Bassil and Rudolf K. Keller. A Qualitative and Quantitative Evaluation of Software Visualization Tools. In *Proceedings of the Workshop on Software Visualization*, pages 33–37, Toronto, Canada, May 2001. Held in conjunction with the 23rd Intl. Conf. on Software Engineering (ICSE'2001).

Mordechai Ben-Ari, Niko Myller, Erkki Sutinen, and Jorma Tarhio. Perspectives on Program Animation with Jeliot. In Stephan Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 31–45. Springer-Verlag, 2002.

Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. The Jeliot 2000 rogram Animation System. *Computers & Education*, 40(1):15–21, 2003.

Stéphane Hillion. DynamicJava. WWW-page, 2002. (Koala project) `http://koala.ilog.fr/djava/` (accessed 10.6.2004).

Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, pages 153–156. ACM Press, 2003.

Chris D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing*, 13(3):259–290, 2002.

Colleen Kehoe, John T. Stasko, and Ashley Taylor. Rethinking the Evaluation of Algorithm Animations as Learning Aids: An Observational Study. Technical Report GIT-GVU-99-10, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, March 1999.

Bob Lang. Teaching new Programmers: a Java Tool Set as a Student Teaching Aid. In *Proceedings of the Inaugural Conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate Representation Engineering for Virtual Machines 2002*, pages 95–100, National University of Ireland, 2002.

Matti Lattu, Veijo Meisalo, and Jorma Tarhio. A Visualization Tool as a Demonstration Aid. *Computers & Education*, 41(2):133–148, 2003.

Matti Lattu, Jorma Tarhio, and Veijo Meisalo. How a Visualization Tool Can Be Used — Evaluating a Tool in a Research & Revelopment Project. In *12th Workshop of the Psychology of Programming Interest Group*, pages 19–32, Corenza, Italy, 2000. `http://www.ppig.org/papers/12th-lattu.pdf` accessed 10.6.2004).

Janne Markkanen, Pertti Saariluoma, Erkki Sutinen, and Jorma Tarhio. Visualization and Imagery in Teaching Programming. In John Domingue and Paul Mulholland, editors, *10th Annual Meeting of the Psychology of Programming Interest Group*, pages 70–73, Knowledge Media Institute, Open University, Milton Keynes, UK, 1998.

Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing Program with Jeliot 3. In *Proceedings of the International Working Conference on Advanced Visual Interfaces, AVI 2004*, pages 373–380, Gallipoli (Lecce), Italy, 2004.

Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

Erkki Sutinen, Jorma Tarhio, Simo-Pekka Lahtinen, Antti-Pekka Tuovinen, Erkki Rautama, and Veijo Meisalo. Eliot – an Algorithm Animation Environment. Report A-1997-4, Department of Computer Science, University of Helsinki, Helsinki, Finland, 1997. `http://www.cs.helsinki.fi/TR/A-1997/4/A-1997-4.ps.gz` (accessed 10.6.2004).

Erkki Sutinen and Sirpa Torvinen. The Candle Scheme for Creating an on-line Computer Science Program — Experiences and Vision. *Informatics in Education*, 2(1):93–102, January 2003. `http://www.vtex.lt/informatics_in_education/htm/INFE009.htm` (accessed 10.6.2004).

Selecting a Visualization System

Sarah Pollack, Mordechai Ben-Ari
Department of Science Teaching, Weizmann Institute of Science, Israel

moti.ben-ari@weizmann.ac.il

1 Introduction

This paper describes the selection of an algorithm visualization system for use in a high-school course on data structures. While the selection criteria and the evaluations of the systems are highly specific to the particular educational context, we believe that it will be helpful to share our experience with others. Even more important, we wish to point out areas in which *none* of the systems fulfilled our requirements, so that developers of existing or future visualization systems can take them into account.

1.1 The educational context

During the 1990s, a new curriculum was developed for Israeli high schools (Gal-Ezer et al., 1995). The intent was to enable students to study computer science as a full-fledged scientific subject for their matriculation examinations. These subjects can be studied in three or five 90-hour *units* spread over three years of high school. The five-unit curriculum includes three required units: a two-unit course on the foundations of computer science and a unit called *Software Design*, as well as two elective units, one theoretical and one on a second programming paradigm. The Software Design unit is normally studied in the last year of high school; its syllabus includes the study of (a) designing non-trivial programs and implementing them in modules, and (b) data structures (lists, stacks, queues, trees) and elementary algorithms on these structures.

We have had experience in the design and development of the program animation system Jeliot (Ben-Ari et al., 2002). Our research has shown that visualization can improve the learning of programming by novices by providing them with a concrete vocabulary for describing the execution of programming constructs (Ben-Bassat Levy et al., 2003). However, Jeliot visualizes the lowest level of program execution (individual variables, expression evaluation and instruction execution), and thus is not suitable for use in the Software Design course, where we need to visualize more abstract entities like lists and trees. This paper describes our evaluation of visualization systems to support learning and teaching in this course.

1.2 The visualization systems evaluated

Based upon our previous acquaintance with visualization systems, we chose Animal (Rößling and Freisleben, 2002) and Matrix (Korhonen and Malmi, 2002) as the candidate systems. Later, we also decided to evaluate Interactive Data Structure Visualizations (IDSV) (Jarc, 1999).

Animal is an interactive system for building visualizations and animations. It is an open tool, meaning that there are no data structures and algorithms predefined by the system; instead, the tool supplies graphics primitives that are useful in creating such visualization. (A library of visualizations of data structures and algorithms is available.) In addition to its interactive mode, a scripting language (Rößling and Freisleben, 2001) is available; this provides an alternate method of generating visualizations (one that may be faster in the hands of an experienced user) and it enables the construction of meta-tools to generate visualizations. Such a tool is available for generating visualizations of sorting algorithms. Animal can be downloaded from `http://www.animal.ahrgr.de/`.

Matrix is a system for visualizing data structures and algorithms, primarily, advanced tree and graph algorithms. It is closed, meaning that there is a fixed set of structures and

algorithms supported by the system. We evaluated the most recent version of the system called MatrixPro that can be downloaded from `http://www.cs.hut.fi/Research/MatrixPro/`.

IDSV is a web-based client-server tool that integrates HTML documents with animations of data structures and algorithms. IDSV can be run interactively from `http://nova.umuc.edu/~jarc/idsv/`. IDSV is also a closed tool, offering a limit number of visualizations, but the documents can be easily changed from outside the tool. In our environment, we can assume that schools (and students' homes) are reasonably well-equipped with computers, but we cannot take for granted the existence of good Internet access. By courtesy of the developer, we received the source code of IDSV and created a self-contained Java application suitable for our environment.

2 Selection criteria

If there were only one set of criteria for choosing a visualization, there would be only one visualization system. Clearly, the multiplicity of systems means that different researchers and educators have different sets of criteria. In this section, we set out the criteria that we used in our selection.

The first and most important criterion is the appropriateness of the visualization for the intended users—high-school teachers and students—and the probable contribution to learning from use of the visualization system. The users can be characterized as having very good computer literacy skills, and beginning programming skills in the language used in the course (Pascal or C). Therefore, ease of installation and use is a primary consideration, and the effective use of a visualization tool *cannot* require proficiency in Java programming. (All three systems are written in Java and we had no problems installing or using them.) There are plans to use Java and C# in Software Design course in the future, but we can never expect teachers and students to acquire the skills to effectively modify large Java programs.

Here is a list of aspects of the tools that were assessed under this criterion:

- Ease of installation and operation.
- Support for the teacher to easily create demonstrations.
- The ability to integrate the tool within the type of teaching activities that the teachers routinely use. (Teachers tend to be conservative in adopting new educational technology, especially if it affects their existing corpus of lesson plans and teaching materials.)
- Beyond the basic visualization of the data structures and algorithms, the tool should enable the students and teachers to analyze data types, for example, to analyze the efficiency of a tree search in relation to the height of the tree.
- The visualization tool should support communications between the teacher and student: Can the student use the tool to ask questions and can the teacher use the tool to answer them?

The second criterion is that a visualization tool must support visualizations of the data structures and algorithms actually used in the course, preferably through built-in visualizations that will not require extra effort on our part. The algorithms to be studied include creation of the data structure, adding and removing elements, and searches and traversals. In order to support the creation of exercises for students, it is essential that the tool have the ability to easily input data sets.

The third criterion is the quality of the visualization. We looked for the ability to control the speed of animation, to choose step-by-step or continuous execution, and to step backwards and forwards. Control of the step granularity is very helpful and it is important that the visualization be coordinated with a display of the source code (whether in a programming language

or pseudocode). A save/load facility is essential to help teachers prepare visualizations for demonstrations.

In terms of the criteria given by Anderson and Naps (2000), we want Level I (understanding of the algorithm as a recipe) and Level 2 (understanding the relationship of the algorithm to its implementation) of their Algorithm Understanding Scale. For their Instructional Design Scale, the important levels are Level 1 (the visualization ... demands no interaction from the viewer), and Level III (allow the student to design input data for the algorithm). Flexibility of the visualization is less important, because the students are young novices, and the presentation of textual material is not too important, because we intend to supply our own material tailored to contents of the course.

3 Evaluation

The evaluation was carried out by the first author, who is a very experienced teacher of computer science in high schools; in particular, she has taught the Software Design course for many years. Four typical exercises using binary search trees (BST) were written in the style that such exercises would be given to students or used by a teacher for demonstration: constructing a BST, adding an element to a BST, searching for a value in both balanced and unbalanced BSTs, and using a BST for sorting. The tools were then analyzed to determine their potential for improving the learning experience when used with these exercises. A table was created for each of the criteria and sub-criteria, and scores and weights were assigned.

3.1 Animal

As an interactive tool ANIMAL is very easy to use, because its intuitive drag-and-drop interface makes it possible to produce new visualizations with little training. However, the amount of time required to create a visualization is quite large. While investing this amount of time can be justified when preparing a large formal lecture, it would not be practical in a high-school classroom. The classroom dynamics is such that a teacher should be able to use the visualization to quickly respond to questions such as: "What would happen if the last value to be inserted in the tree were 15 and not 30?" Therefore, the tool received a low score for the criterion of aiding communication between the student and the teacher.

The problem is that the ANIMAL tool does not contain algorithm-specific knowledge. While it does supply primitives that facilitate the creation of algorithm animations, we could not see that teachers and students would find these sufficiently more powerful than those available in a generic tool like PowerPoint, with which they are very familiar.

The attractive mode of the use of ANIMAL is to use meta-tools to generate ANIMALSCRIPT for algorithm visualizations. Meta-tools enable the data set to be easily changed for exercises. The ANIMAL web site includes such a tool for sorting algorithms, but these algorithms are covered in the second year course, not in the Software Design course. Building meta-tools is not difficult, but it is not feasible given the low programming skills of the teachers and students. The option of manually writing ANIMALSCRIPT commands is not an attractive option for a target audience who are used to working with interactive tools.

The ANIMAL library includes many contributed visualizations, which would be invaluable in preparing lectures, but the visualizations are static in the sense that to change them to use different data requires that they be entirely recreated. Therefore, it would be difficult for the teacher to integrate the visualizations within the existing activities in the course.

The quality of the visualizations and the control that the user has is good. The visualization can be run step-by-step or it can be displayed as a smooth animation. A unique advantage of ANIMAL is that its visualization primitives support the display of source code, and you can (manually) coordinate between the code and the visualization. Finally, ANIMAL contains no support for assessment.

3.2 Matrix

Matrix comes with a large built-in selection of algorithm visualizations, mostly for advanced algorithms. The developers kindly agreed to add visualizations of simple data structures like stacks and queues, so that we have a set of visualizations covering the data structures taught in the course.

Matrix uses step-by-step animation, though it can export the animation in the Scalable Vector Graphics (SVG) format and these animations are smooth. The control of the visualization is good and animations can be saved and reloaded. Of particular importance is the ability of Matrix to read external files of data so that the teacher can easily prepare different demonstrations and exercises for a particular algorithm, for example to show the dependency of the efficiency of a BST search on the balance of the tree. Since the files are text files, they could even be modified during a class using an editor. Matrix also includes a self-assessment facility.

The control of the animation is excellent: not only can you run the animation forwards or backwards, and step-by-step or continuously, but you can also define breakpoints and specify the granularity of the animation step.

Matrix has the novel ability to *simulate* algorithms (Korhonen, 2003). In algorithm simulation, the user can manually modify the visual representation of the data structure, and these modifications are applied to the internal representation of the structure. We have not yet decided if this feature is important in our context.

The most serious problem with Matrix is that there is no display of source code nor is there any coordination with explanatory material. It is also difficult to extend it to visualize new algorithms. Such an extension was demonstrated to us by the developers, but it requires significant Java programming experience and familiarity with the structure of the source code, both of which are way beyond the capability of high school teachers.

3.3 IDSV

We modified the original IDSV tool to be a Java application using the Swing user interface; source code and explanations are displayed from text files distributed in the tool archive. However, the source code is just displayed, not coordinated with the animation of the algorithm.

IDSV uses smooth animation, where you can see an element slowly travel throughout the data structure, rather than just appear in its correct position. Based upon our experience with Jeliot, we believe that smooth animation is better for beginning students learning elementary algorithms. Furthermore, like Jeliot, IDSV displays helpful comments at each step. Since these features proved to be extremely useful in teaching introductory students, this made IDSV extremely attractive for our high schools students.

The control of the animation is less complete than in the other tools, and you cannot run animations backwards.

There are two main problems with IDSV. First, there is a limited selection of built-in data structures and algorithms, and like with Matrix, adding new visualizations requires significant effort and Java programming experience. Second (and more important), while IDSV enables the user to see animations run with random data or data entered interactively, there is no way to supply a file with a data set.

4 Our selection

The three tools were evaluated according to a set of criteria and weighted numerical scores were given and justified.[1] Data adapted from that report are shown in Table 1. More important

[1] The full report—in Hebrew—is available from the authors.

Criterion	Subcriterion	Weight	Subweight	Matrix	IDSV	Animal
Usability		20%		5	3	3.8
	Installation		20%	5	5	5
	Existing animations		40%	5	3	5
	Creating animations		40%	5	2	2
Visualization		30%		3.7	2.8	3.1
	Visualization of ADT		10%	5	2	3
	Control of animations		35%	5	3	5
	Coordinated with algorithm		30%	1	4	5
	Reinitialize ADT		30%	5	3	1
Pedagogy		50%		5	2.75	1.75
	ADTs and their operations		25%	5	3	3
	Open to new ADTs		25%	5	3	2
	Building exercises		25%	5	2	1
	Learning activities		25%	5	3	3
Total		100%		4.6	2.8	2.6

Table 1: Weighted scores of the visualization tools

than the actual scores, however, the match between the capabilities of the tools and the needs of our students and teachers.

We found that ANIMAL supports the visualization of the data structures that are taught in the course, but it is difficult to integrate into the type of activities that are currently used. Furthermore, neither of the modes of the use of ANIMAL—interactive and scripts—is appropriate for our environment.

In deciding between IDSV and Matrix, IDSV has the advantages of smooth animation and display of code and explanations, while Matrix has a larger set of built-in visualization and better control of the visualization. We decided to adopt Matrix, primarily because of its excellent flexibility: the ability to save and load animations, and to create data sets for exercises and examinations. We hope that future versions of Matrix will improve in the two areas in which it is deficient: display of source code and ease of extension.

5 The problem with all of them

The emphasis in the Software Design course for high-school students is on understanding the concept of abstract data type (ADT), building modules to implement ADTs and using them to solve problems. Here is a list of typical problems that would be given as exercises on homework or examinations:

- Let `L1` and `L2` be ordered lists of integers. Write an algorithm that returns the list resulting from the removal of all elements of `L1` from `L2`.
- Write an algorithm for the boolean-valued operation `immediately-after(L,x,y)`, where `L` is a list of integers and `x` and `y` are integers; the operation returns `true` iff `x` appears in `L` immediately after `y` or `y` appears in `L` immediately after `x`.
- Write an algorithm to compute the number of nodes in a binary tree.
- Write an algorithm to check if two binary trees are "mirror images" of each other.
- Define a *sum-tree* as a binary tree such that the value at every node is larger than the sum of all the values of the nodes in the left subtree and smaller than the sum of the

values of the nodes in the right subtree. Write algorithms to create a sum-tree and to check if a tree is a sum-tree.

With closed systems like Matrix and IDSV, we cannot create animations for these problems. With an open system like ANIMAL, any animation can be created, but this is a task that is separate from the task of solving the problem. To build an animation, a student would first have to solve the problem, but once she solves the problem, there doesn't seem to be any point in putting in the extra effort to create the animation.

We believe that if *student-written* algorithms could be easily animated, it would improve the students' ability to successfully solve problems. Ideally, we would want a visualization system to be able to work directly from student-written source code, as is done in Jeliot (Ben-Ari et al., 2002).

The objection that is made to self-animation is that it cannot be as good as animations that are hand-crafted for particular algorithms. While this is true to some extent, courses in data structures and algorithms study a very limited number of generic structures: arrays, lists, trees, graphs. (These are called *fundamental data structures* in Matrix (Korhonen, 2003).) It ought to be possible to have a visualization system interpret source code manipulating these structures. The feasibility of this approach was demonstrated in the first version of Jeliot, which animated array, stack and queue algorithms by modifying the source code with no further user intervention.

A related issue that we would like to see addressed is that of visualization of self-assessment facilities. As implemented in Matrix and IDSV, the student simply receives a score and the correct answer. What we would like to see is a visualization that would point to a node and display "the key 5 is less than the value 10 in *this* node, but you entered the right subtree instead of the left subtree."

6 Other systems

There are other visualization systems that are worth evaluating, but that were not appropriate for our study because they are intended for teaching that uses the Java programming language. Two of them are worth mentioning, because they address issues discussed in this paper.

Dot.java (Hamer, 2004) is a Java class that can be included in a student's program. It provides a method `drawGraph` that creates a drawing of any Java object. The drawing is visualized with the GraphViz utility. Although it requires intervention in the source program, the intervention is very simple. The primary advantage of Dot.java is that it enables visualization of *any* student-written program, which we believe to be of great importance.

jGrasp (Hendrix et al., 2004) is a well-known pedagogical development environment. The latest version contains the beginnings of a dynamic visualization feature. The advantage of here is that the visualization is integrated into an IDE so a separate tool is not required.

7 Conclusions

The choice of Matrix was primarily dictated by the flexibility it offered for building exercises that would enable the analysis of algorithms. We were not aware of Karavirta et al. (2002) when performing our comparison, so it is gratifying to see that our analysis is consistent with the concepts and analysis presented in that paper. Given the level of the students and teachers it is essential that the visualizations be as *effortless* as possible. Figure 1 of Karavirta et al. (2002) shows that ANIMAL is generic and high effort, whereas Matrix is specific and low effort; similarly, Figure 2 ranks ANIMAL as having a primitive graphical vocabulary and high effort, whereas Matrix has a complex graphical vocabulary and is low effort. (Note that the scripting language of ANIMAL was not evaluated in this study.)

We would like to see tool developers explicitly discuss the intended pedagogical use of their tools; an example of such an explicit discussion is given in Chapter 9 of Korhonen (2003). We

believe that a priority for future development of algorithm visualization systems is to enable the visualization of student-written algorithms.

Acknowledgements

We would like to thank the developers of the visualization systems for their willingness to answer our questions. This research was partially funded by the Israeli Ministry of Education.

References

Jay Martin Anderson and Thomas L. Naps. A context for the assessment of algorithm visualization systems as pedagogical tools. In *Proceedings of the First Program Visualization Workshop*, pages 121–130, Porvoo, Finland, 2000.

Mordechai Ben-Ari, Niko Myller, Erkki Sutinen, and Jorma Tarhio. Perspectives on program animation with Jeliot. In *Software Visualization: International Seminar*, Lecture Notes in Computer Science 2269, pages 31–45, Dagstuhl Castle, Germany, 2002.

Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, 2003.

Judith Gal-Ezer, Catriel Beeri, David Harel, and Amiram Yehudai. A high school program in computer science. *IEEE Computer*, 28(10):73–80, 1995.

John Hamer. A lightweight visualiser for Java. In *Proceedings of the Third Program Visualization Workshop*, pages 54–61, Warwick, UK, 2004.

T. Dean Hendrix, James H. Cross II, and Larry A. Barowski. An extensible framework for providing dynamic data structure visualizations in an lightweight IDE. *SIGCSE Bulletin*, 36(1):387–391, 2004.

Duane J. Jarc. *Assessing the Benefits of Interactivity and the Influence of Learning Styles on the Effectiveness of Algorithm Animation Using Web-based Data Structures Courseware.* PhD thesis, George Washington University, 1999. `http://www.student.seas.gwu.edu/~idsv/djj-dissertation.pdf`.

Ville Karavirta, Ari Korhonen, Jussi Nikander, and Petri Tenhunen. Effortless creation of algorithm visualization. In *Proceedings of the Second Finnish / Baltic Sea Conference of Computer Science Education*, pages 52–56, 2002.

Ari Korhonen. *Visual Algorithm Simulation.* PhD thesis, Helsinki University of Technology, 2003.
`http://lib.hut.fi/Diss/2003/isbn9512267950/isbn9512267950.pdf`.

Ari Korhonen and Lauri Malmi. Matrix—Concept animation and algorithm simulation system. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 109–114, Trento, Italy, 2002.

Guido Rößling and Bernd Freisleben. AnimalScript: An extensible scripting language for algorithm animation. *SIGCSE Bulletin*, 33(1):70–74, 2001.

Guido Rößling and Bernd Freisleben. Animal: A system for supporting multiple roles in algorithm animation. *Journal of Visual Languages and Computing*, 13(2):341–354, 2002.

Survey of Effortlessness in Algorithm Visualization Systems

Ville Karavirta, Ari Korhonen, and Petri Tenhunen
Helsinki University of Technology
Department of Computer Science and Engineering
Finland
{vkaravir, archie, ptenhune}@cs.hut.fi

Abstract

This paper reports the results of an on-line survey conducted among computer science educators to examine effortless creation of algorithm visualizations. Based on the results, we give a proposal for measuring effortlessness in this sense. The aim is to enhance the understanding of the visualization tools adequate in computer science education.

1 Introduction

The idea of using visualization to promote the understanding of abstract concepts, like data structures and algorithms, has become widely accepted. Despite its benefits, algorithm visualization (AV) has failed to become popular in mainstream computer science (Stasko, 1997; Baecker, 1998). One of the main obstacles for fully taking advantage of AV systems seems to be the time and effort required to design, integrate and maintain the visualizations (Naps et al., 2003b). According to Hundhausen et al. (2002) the process of creating AVs is thought to be too laborious to be worthwhile. Thus, "a future challenge is to create tools and methodologies which will result in the use of SVs by the majority of computer science educators" (Domingue, 2002).

Several attempts have been made to introduce a system that levels out the burden of creating new visualizations (e.g. Haajanen et al., 1997; Korhonen and Malmi, 2002; LaFollette et al., 2000; Naharro-Berrocal et al., 2002). However, none of these systems has gained wide recognition. In our previous study (Karavirta et al., 2002), we examined four systems, in order to identify why the creation of software visualization is such a laborious process. We wanted to emphasize the instructor perspective because "the visualization research has focused on the developer and designer while research in CS education has focused on [...] student learning. In contrast, virtually no research has focused on the needs of the instructor" (Naps et al., 2003b). However, it is the instructor that plays the key role in taking the AV system in use.

Unfortunately, effortlessness is a highly subjective measure including many factors. For example, the different systems can be put on the same line only by first determining the context where the systems are utilized. With one system, it might be easy to create static visualization from an already existing code whereas another system might be effortless in the sense that the animations can be created on-the-fly in a lecture situation. Creating visualizations with an inadequate system for a certain task might require a lot of extra effort compared with more feasible system. Thus, algorithm visualization systems are often seen laborious because people are drilling with a hammer. More research is needed to identify the essence of effortless creation of algorithm visualizations.

In this study, we have created a questionnaire for computer science educators to sort out the elements of effortless creation of algorithm visualization. In our survey, we concentrate on the instructor's point of view to identify the typical use cases when visualizations are created, and typical expectations when starting to use an algorithm visualization system. In the rest of the paper, we describe the survey, results, give a proposal for how to measure effortlessness in AV systems, and finally make some conclusions.

2 Survey

The starting point for this research was the work done by the working group on "Exploring the Role of Visualization and Engagement in CS Education" at the ITiCSE 2002 conference (Naps

et al., 2003a). The results from the three surveys evaluated in that paper (Grissom's survey from ITiCSE'00, ITiCSE'02 WG pre-conference on-line survey, and ITiCSE'02 WG Index Card survey) motivated us to develop a new, more detailed on-line survey from the *effortlessness* point of view. Using some items from the three surveys, we refined a survey of our own that was first tested with the attendees of the Third Program Visualization Workshop (PVW'04). The preliminary results were also presented there. However, our survey continued after the conference and we advertised the research in various mailing lists (*e.g.*, for PPIG (Psychology of Programming Interest Group), SIGCSE (ACM Special Interest Group on Computer Science Education), and PVW members) to get a more versatile sample. The final results are reported in this paper. Respondents were from USA (7), UK (5), Finland (2), Germany (2), Israel (2), New Zealand (2), Island (1), and Spain (1), which makes total of 22 responses. This is less than in the previous syrveys. Moreover, as the terminology in the field varies it causes some problems to collect reliable data. For example, some of the respondents did not answer the survey as they felt that the survey covered only very narrow set of courses.

In the ITiCSE WG survey they asked much more detailed questions on the background of the respondent as well as their attitude towards AV. We expected to reach a similar population, thus we concentrated more on how they have applied AV in practice. The only background questions, in addition to the above, were the ones on subjects taught and in which course levels. The subjects varied from basic programming courses to computer graphics, and from computer architecture to discrete mathematics. Moreover, all course levels were almost evenly distributed as many of the respondents were teaching several courses. After the few background questions, previous usage of AV systems and usage in teaching were asked.

Five respondents denied the use of algorithm visualizations in their teaching, three of whom have not used any tools to create algorithm visualizations. However, there is a total of six respondents that have not used AV tools to create algorithm visualization, thus three respondents must have used them, but not in teaching.

The respondents who have used AV system(s) were asked about the origins of the system(s) they have used. We have 25 responses from 18 respondents. Ten have developed a tool or system by themselves. One of these, however, has never used it to create algorithm visualizations (but possibly used it in teaching, see the previous questions). Six have developed system(s) in a team within a single institution. Four respondents have developed a new tool by reusing existing tools or libraries. Again, four of the respondents uses software developed by researchers in other institutions. Only one respondent used software developed by some other team within his or her own institution. None of the respondents selected "Commercial software" even though some examples of these are mentioned later in free text responses.

The rest of the survey was divided to two phases. First, we wanted to collect free text descriptions on the actual use cases and situations in which the respondents were using AV. After the first phase, the respondent was directed to another page where we provided ready-made scenarios and asked the respondents to state how well these apply in their context. The whole survey, with some additional data not presented in this survey, can be located on the web at `http://www.cs.hut.fi/Research/SVG/survey/`.

3 Results

Most respondents described several use cases where they, as teachers, could create and use AV – 16 respondents and 28 use cases. Even though in most cases the task was clearly identifiable, the ultimate goal within the use case was often blurred. Some typical tasks, however, can be identified from the data (i.e., *tracing and debugging*, *problem solving and exercises*, *data abstraction and representation issues*, and *teaching data structures and algorithms in general*).

By lowering the abstraction level, we can also identify the context in which AV is used. The following items were mentioned with more than one use case (number of use cases where the context occurred is marked in parenthesis): Programming – data structures, data flow, and

control flow (6); Sorting algorithms (4); Graph algorithms (4); Mathematics / theoretical CS (3); and Geometric algorithms (2). Thus, tracing and debugging of an implemented algorithm (i.e., real execution) is the most commonly mentioned use case. Albeit important one, the *actual implementation* of an algorithm, however, was not an issue in most of the use cases.

The tools that respondents have used could be divided into three groups: commercial products, visualization and graphics packages, and non-commercial AV systems. Commercial systems mentioned in the responses were PowerPoint, Visual Basic, and QuickTime. Visualization packages were the Visualization ToolKit (VTK), GraphViz, and OpenGL. Non-commercial products, however, was the most common category with products like Jeliot, JAWAA, Animal, JHave, TRAKLA2, MatrixPro, Poplog, SimAgent, JFLAP, and several Java applets.

According to respondents, the most common pros of the visualization systems were the features allowing to create animations easily, and sufficient navigation possibilities in the final animation. The lack of these very same properties were mentioned as cons of some other systems. Not so commonly mentioned pros were: easy installation, freely available, modular structure, and the fact that use of these tools might activate thinking. Respective cons were: limited domains where the system can be used and the observation that the interaction with such tools may be mechanical and therefore decrease thinking. A benefit mentioned especially with programming tools was that "since it's a programming language, I can get it to do just about anything I want".

The last question in the first part of the survey asked the respondent to describe an ideal tool that could be used to complete the use case. The responses were more of a wish-list about the system features or descriptions of current systems than any new visions about possible future tools. The most-often mentioned features are represented in Table 1.

Table 1: The ideal tools.

Category name	**Examples**	**Count**
Ease-of-use	Easy generation, own input data, customization of visualizations, example generators	7
Programming	Java, JavaScript, drag&drop coding	5
Special features	For beginners and experts, flexible notation and timing, interaction, visual effects	5
Abstraction level	General tools, small tools combined, many representations	4
Professionalism	Help functionality, internationalization, better user interfaces	3
Formats	*e.g.*, Flash, common file formats between systems	2

The respondents were also asked how much time they can afford to use and how much time they have used for the whole course and for different teaching related tasks. The tasks were developing the course contents (lectures, exercises, demos, text, ...), and several AV related tasks (*i.e.* searching, installing, and learning to use for good AV examples and tools, developing visualizations, adapting visualizations to the teaching approach and/or course content, solving problems in using the tool, and setting up visualizations in the classroom). The number of responses for this question was 19.

In most cases there was no difference between the time used and time available. However, when looking at the responses on the item "searching for AV tools", a clear difference was found. People have more time to search AV systems, than they are actually using for that.

The respondents were asked how much time they have available for predefined use cases and how much it would actually take to complete those use cases. The tasks were divided into four categories based on the situation: lecture, practice session, producing teaching material and examination/summative evaluation. The categories and the corresponding use cases are shown in Table 2. The number of respondents who answered this question was 20.

For each use case the respondents were asked how relevant it is according to the course he or she is teaching. The scale was from 1 (not relevant at all) to 5 (very relevant). The number of respondents that thought a use case is relevant (answers 3-5) is shown in Table 2

(column Rel). In addition, there is the column (Most Rel) that represents the number of respondents that consider the use case to be within the three most relevant tasks in their teaching. Moreover, the respondent was asked to state whether or not AV could be used to reduce the required work to complete the use case (column AV). The results show that AV is considered most useful to produce lecture examples and on-line illustrations. These two tasks are also considered to be the most relevant ones.

Table 2: Predefined use cases related to teaching. The columns Rel, Most rel, and AV show the numbers of responses in which the use case is considered to be relevant to the respondent, within one of the three most relevant use cases, and considered to reduce the work load of the use case, respectively.

Use case	**Rel**	**Most Rel**	**AV**
Lectures			
single lecture example	14	6	9
answering students' questions during a lecture	14	2	5
preparing questions/problems for a lecture	14	1	4
Producing teaching material			
on-line illustrations (static or dynamic)	12	5	8
text book/lecturer's notes illustrations (static)	12	1	3
Examination/summative evaluation			
creating exercises for examination (that students solve)	12	1	3
Practice session			
creating exercises for practice session (that students solve)	12	2	4
preparing a demo for a closed lab such that a tutor shows	9	3	6
preparing a demo students interact with in closed labs	7	3	5
preparing a demo students interact with in open labs	6	4	3

The section included also a question about the best tool the respondent has used for the use case, and the three most important features of the system. The features mentioned more than once were "saves time", "must be reliable", "one has control over the final visualization" (for example, possibility to move backwards and forwards, pause, play the animation), "allows student input data", "helps explaining/understanding", and "easy/automatic creation". There was also one response that stated that "all [systems] have increased my workload".

Most of the use cases are estimated to be completable just within the time available. Creation of text book illustrations and demos for closed labs was estimated to take a litle more time than what is available. A clear difference was observed in on-line illustrations. The average time to complete such a use case was 0.69 hours, whereas the time available was 1.23 hours.

The next question was about the content generation approaches. Table 3 lists the different approaches and illustrates that the approaches were almost equally preferred.

Table 3: Preferred content generation approaches.

Content generation approach	**Count**
Interactive simulation (user can experiment with methods; easy for implemented concepts, new concepts require understanding of underlying API)	11
Fully prepared examples (easy to use, no work; not adaptable)	9
"Generators" for some AV system (need only specify parameters, easy to use; restricted to certain topics, limited adaptability)	9
Automatic generation based on source code (easy to use, no work; requires source in appropriate language, display usually not adaptable)	7
Generation using a script (adaptable and designable; extra work)	7
Visual generation by drag and drop (highly adaptable, extremely flexible; takes much time, changing a value may not adapt the algorithm)	7

The last question of the survey was what are the tasks where a Graphical User Interface (GUI) is essential and when it is not needed at all. Tasks in which a GUI is needed according to the respondents are code/algorithm simulation, and customizing the visualization. Tasks that GUI is not essential in are automatic production from code, and textual tracing (for example, visualizing procedure calls).

4 Proposal for Measuring Effortlessness

In our previous study (Karavirta et al., 2002), we defined the effortless creation of algorithm visualization based on the following two criteria: the possibility to use the system on-the-fly basis, and available WYSIWYG user-interface. In addition, some of the categories presented in Taxonomy of Price et al. (1993) were applied. Moreover, we compared several systems with each other in order to discuss the relevance of such criteria. However, as we have already stated in our previous paper, our original criteria set are quite subjective in their nature. One of the main obstacles here is that the systems evaluated are usually targeted to different use cases, and the comparison among them is unreasonable. Thus, the idea of this proposal is to come up with a more objective set of requirements that measure effortless use of AV, and gather further evidence to support our vision of effortlessness.

We sum the proposal up with the following three main categories that are the *scope*, *integrability*, and *interaction techniques*. The idea is to apply all the categories to one single system to come up with an estimate of system effortlessness.

Before applying the proposed categories, our proposition is that the task should be defined *problem based*, thus leaving enough space for interpretation how to solve the problem with the current system. Different systems might provide different perspectives on how to tackle a problem. This is our starting point, and examples of such problem based tasks are, for example, anything between "demonstrating quicksort algorithm" and "teaching sorting algorithms", but not "how to animate quicksort partition algorithm" (as the target system might have some other alternative than animation to visualize the partition algorithm). Only after we have agreed the context, it should be possible to evaluate and compare different systems *that are targeted to such activity*.

Category Scope is basically defined in how wide context one can apply the system. The results show clearly that the lack of time is the most common reason for not using AV as widely as expected. Thus, a system providing a more broader context to apply is definitely more attractive choice than many single purpose systems because it saves the time to install and learn various tools that all can be applied only for a very limited scope. This can also be seen from the responses for the question about *ideal tool*. The respondents had a vision of "general tool" that combines several small tools together or comes true in several tools that have similar usage (see, e.g., Table 1). Moreover, in Section 3, we can see that in the free responses the scope of the creative use is not just a single specific topic, but a broader context.

The scope can have the following subcategories defined, for example, in terms of the subdivisions of a course: lesson-specific, class-specific, domain-specific, and generic (not any domain specific). A generic tool, however, can still be, for example, lesson-specific: the tool provides tutorials and ready-made examples for some specific lesson (e.g., sorting algorithms). The deeper the system covers the subcategories the better. The respondents of the survey seem to be keen on downloading not only tools but also something they can use for their class (e.g., data structures and algorithms), thus there is also need for good tutorials as well as ready-made examples. Even if the examples of a generic tool do not contain the desired learning objects, they should help one to figure out how to apply the tool to produce more content. Similarly, a generic tool can have very small scope if there is no evidence that it will be used for other domains (e.g., computer science) as well. A good indication that a system has large scope is that a third party can produce content with it.

Category Integrability covers topics such as easy installation, but it is a much more broader category than that. We also include features such as customization of visualizations, platform independence, internationalization, good tutorials, etc. Thus, this category lists the features that makes the system attractive to use because there is a way to integrate the system into a course. No single feature makes the system effortless in this sense, but together they have to provide a meaningful way to make the system applicable. It is even better if there are several ways to reach the goal as there are many content generation approaches that meet with support (See Table 3).

Subcategories for integrability can be defined in terms of desired features such as described, for example, in Rößling and Naps (2002). They rank the systems (among others) according to the following requirements: interactive prediction support (e.g., stop-and-think questions or algorithm simulation exercises), database support (for course management), and integration of hypertext. The list is not exhaustive, but gives a hint what kinds of features are expected. If there are many feasible features available, it means more ways to complete a task. Moreover, the more ways there are to achieve the goal, the more effortless the use of the system is. Ultimately, integrability is measured in terms of how well a third party can adopt the system to their course.

Category interaction techniques. Even though a system has a very large scope, and it is easy to integrate it to a course, it may lack this very important aspect. The trend is towards activity where more interaction is required. Table 2 shows that there are many tasks that all seem to be relevant at least to some extent. However, in order to cover as many of these as possible, a tool must support more interaction than what comes with a simple VCR type of animator panel that has backward, forward, and play buttons. This is true with production of material for teaching (towards dynamic on-line illustrations instead of static lecture illustrations) as well as production of material for practice sessions (towards student interaction instead of tutor-centered demonstrations). It is not only important that the interaction between the system and visualizer (content creator) be adequate, but the system has to support interaction between the visualization and the end user as well. Here the end user might be, for example, a learner that solves exercises.

Interaction techniques have two subcategories, which are the two point of views described above: *producer* and *consumer*. Both point of views share some common interests, and we could measure the systems in terms of usability. We do, however, not discuss this or other human computer interaction (HCI) issues here any further. The taxonomy of Price et al. (1993), for example, suggest categories to measure interaction methods, but as we have argued in our previous study (Karavirta et al., 2002) they usually do not cover the various later interaction methods very well. For example, it is very hard to categorize Matrix (Korhonen and Malmi, 2002) that promotes visual algorithm simulation, a technique that allows the user to interact with the underlying data structures in terms of direct manipulation, in the taxonomy. Thus, instead of trying to rule out all the possible interaction techniques at the moment and in the future, we look at the subcategories by determining how well they correspond to the required uses cases (see, *e.g.*, Table 2).

Table 3 implies that there is a number of content generation approaches a single system can support, all of which were preferred by many of the respondents. Interestingly, the most cited technique *interactive simulation* is the one that requires the most interaction. Moreover, each approach requires possibly different interaction techniques. Thus, in general, the more approaches a system supports the better. Or other way around, the more interaction techniques (what ever they are) the system supports, the more content generation approaches can be achieved with it. The question is, can we cover the task with the tool or not, *i.e.*, does the system support the required interaction techniques involved? However, the question how well does systems support an approach is left to be measured in the other categories.

Even though the point of view in our survey was the producer's perspective, we cannot neglect the consumer who eventually uses the produced visualizations. Thus, we have to look

into the interaction methods also from this point of view. We are not, however, interested in the "can we cover" question anymore, but instead in how well does the interaction techniques involved support the various learning strategies. There exist several indicators (see, *e.g.*, Kolb (1984); Felder (1996); Bloom (1956); Naps et al. (2003a)) that can be used to measure such things depending on the purpose of the evaluation. For example, one can apply the *engagement taxonomy* introduced in Naps et al. (2003a) to evaluate the level of activity in the learner–system communication. The levels introduced are viewing, responding, changing, constructing, and representing. A single system can support any combination of these, and the more levels it supports the better. The effortlessness comes from the fact that as our knowledge on how to support learning increases, we might want to change the level of activity, but not the tool. If the tool supports many levels, there is a better chance that we can continue to use it even though the requirements change. Moreover, several activity levels might be attractive in a single course in order to be able to apply the same tool for several levels (e.g., viewing in lectures, and changing in practice session).

5 Conclusions

In this paper, we have presented the results of the on-line survey conducted prior to the PVW'04 conference. In addition, based on the results, we have proposed a taxonomical approach to measure the effortlessness of algorithm visualization systems. The taxonomy contains three main categories that are the *scope*, *integrability*, and *interaction techniques*. The categories try to characterize the evaluated system by determining the extent of its scope by answering whether a third party can produce content with it; integrability by measuring how well a third party can adopt the system to their course; and interaction techniques by looking at the system from producer's and consumer's perspective, and by determining how well the system corresponds to several common use cases.

The survey had a smaller sample set compared to other similar surveys carried out in the past five years. In addition, too few non-developers answered the survey and that might have biased the results. However, we still believe that we have managed to gather data that reliably supports our view of effortless AV creation.

One concern that raises up from the survey data was the observation that respondents are keen to use AV quite *passively*, *i.e.*, the instructor is active with the tool and the learner is only passively viewing the visualization. In our opinion, however, the use of AV has much more potential than that. In this sense, we must pay attention to the *interaction techniques* supported by the AV tools since as long as there are no suitable systems available, the use of AV retains its passive form.

Finally, it seems that the instructors have more time available to search AV tools than they actually use. Thus, developers should pay attention to ease the system integration as well as to promote the system use for different contexts.

In the future, our aim is to define a full taxonomy for effortlessness in algorithm visualization by applying it to different use cases (producer's perspective) based on this work. Some of the existing algorithm visualization systems will be classified based on the use cases they are suited for. We strongly believe that this kind of classification would help us design better algorithm visualizations systems for different needs and increase users ability to select correct tool for a specific problem. Also, when compared to e.g. Price's Taxonomy (Price et al., 1993) and applied on several systems this would give us knowledge about the consequences of different design selections.

Acknowledgments

We thank Guido Rößling and other participants of PVW 2004 who commented the questionnaire and earlier versions of this paper.

References

Ronald M. Baecker. *Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science*, chapter 24, pages 369–381. The MIT Press, Cambridge, MA, 1998.

Benjamin S. Bloom. *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain*. Addison Wesley, 1956.

John Domingue. Software visualization and education. In Stephan Diehl, editor, *Software Visualization: International Seminar*, pages 205–212, Dagstuhl, Germany, 2002. Springer.

Richard M. Felder. Matters of style. *ASEE Prism*, 6(4):18–23, 1996.

Jyrki Haajanen, Mikael Pesonius, Erkki Sutinen, Jorma Tarhio, Tommi Teräsvirta, and Pekka Vanninen. Animation of user algorithms on the Web. In *Proceedings of Symposium on Visual Languages*, pages 360–367, Isle of Capri, Italy, 1997. IEEE.

Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290, June 2002.

Ville Karavirta, Ari Korhonen, Jussi Nikander, and Petri Tenhunen. Effortless creation of algorithm visualization. In *Proceedings of the Second Annual Finnish / Baltic Sea Conference on Computer Science Education*, pages 52–56, October 2002.

David A. Kolb, editor. *Experiential Learning: Experience as the Source of Learning and Development*. Prentice-Hall Inc, New Jersey, USA, 1984.

Ari Korhonen and Lauri Malmi. Matrix — Concept animation and algorithm simulation system. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 109–114, Trento, Italy, May 2002. ACM.

Paul LaFollette, James Korsh, and Raghvinder Sangwan. A visual interface for effortless animation of C/C++ programs. *Journal of Visual Languages and Computing*, 11(1):27–48, 2000.

Fernando Naharro-Berrocal, Cristobal Pareja-Flores, Jaime Urquiza-Fuentes, J. Ángel Velázquez-Iturbide, and Francisco Gortázar-Bellas. Redesigning the animation capabilities of a functional programming environment under an educational framework. In *Second Program Visualization Workshop*, pages 59–68, HornstrupCentret, Denmark, 2002.

Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodgers, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bulletin*, 35(2):131–152, June 2003a.

Thomas L. Naps, Guido Rößling, Jay Anderson, Stephen Cooper, Wanda Dann, Rudolf Fleischer, Boris Koldehofe, Ari Korhonen, Marja Kuittinen, Charles Leska, Lauri Malmi, Myles McNally, Jarmo Rantakokko, and Rockford J. Ross. Evaluating the educational impact of visualization. *SIGCSE Bulletin*, 35(4):124–136, December 2003b.

Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

Guido Rößling and Thomas L. Naps. A testbed for pedagogical requirements in algorithm visualizations. In *Proceedings of the 7th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'02*, pages 96–100, Aarhus, Denmark, 2002. ACM.

John T. Stasko. Using student-built algorithm animations as learning aids. In *The Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, pages 25–29, San Jose, CA, USA, 1997. ACM.

Author Index