An Explanation-Based, Visual Debugger for Spreadsheet-like Constraints

ABSTRACT

This paper describes a domain-specific debugger for one-way constraint solvers. The debugger makes use of several new techniques. First, the debugger displays only a portion of the dataflow graph, called a *constraint slice*, that is directly related to an incorrect variable. This technique helps the debugger scale to a system containing thousands of constraints. Second, the debugger presents a visual representation of the solver's data structures and uses color encodings to highlight changes to the data structures. Finally, the debugger allows the user to point to a variable that has an unexpected value and ask the debugger to suggest reasons for the unexpected value. The debugger makes use of information gathered during the constraint satisfaction process to generate plausible suggestions. Informal testing has shown that the explanatory capability and the color coding of the constraint solver's data structures are particularly useful in locating bugs in constraint code. Although the debugger is implemented as part of a graphical user interface toolkit, we believe that the techniques can be applied equally well to the debugging of commercial spreadsheets.

Keywords: Visual debugging, one-way constraints, constraint satisfaction, software visualization, data structures

Introduction

One-way, dataflow constraints (also called spreadsheet-like constraints) are widely recognized as a potent programming methodology. They have found uses in a variety of applications including spreadsheets, graphical interfaces [12, 13, 3, 5, 6, 7], attribute grammars [9], programming environments [14], and circuits [1]. If one considers the number of spreadsheet users, one-way constraints are probably the most widely used programming technique in use today. However, there is considerable evidence that one-way constraints are difficult to debug and that this difficulty can pose both productivity and reliability concerns:

1. A survey of graphical interface programmers who have used constraints has shown that programmers find constraints difficult to debug [17]. In particular, they complain that constraints can seem like spaghetti code and that the manifestation of a bug often occurs far from the source of the problem.
2. A study in which experienced users created spreadsheets found that 44 percent of the spreadsheets contained bugs of one kind or another [2].
3. In our own research and instructional activities with students, debugging constraints has been a persistent, recurrent headache.

Despite such difficulties, one-way constraints have a number of qualities that should make them easier to debug than normal programs:

1. The constraint solving algorithm can be easily understood. We have found that the most commonly used constraint algorithm, a strategy called *mark-sweep*, can be explained to and understood by users in a matter of minutes.
2. The constraint solving algorithm can be easily visualized. The fundamental data structure used by one-way constraint solvers is a dataflow graph. This graph is easy to visualize and we have found that most users can understand it after given only a brief explanation.
3. It is easy to remember significant events during the constraint solving process. For example, it is possible to remember when constraints change their inputs or outputs. Recording such events should allow the debugger to provide helpful information to the programmer during the debugging process.

Unfortunately, existing debuggers do not exploit these capabilities of constraint solvers. Although they typically do provide a view of the dataflow graph, they do not take advantage of their knowledge of the constraint solver's actions nor do they scale up well to large constraint systems involving thousands of constraints.

In this paper we describe the design and implementation of a domain-specific debugger for one-way dataflow constraints that provides the following capabilities:

1. It uses color tagging to highlight important information in the dataflow graph.

2. It records significant events during the constraint solving process and then analyzes these events to help a programmer pinpoint the source of an error. The programmer utilizes this analysis feature by asking a general "What's wrong" type question and the constraint solver responds with a set of plausible reasons.

3. It uses a technique called "constraint slicing" to limit the portion of a dataflow graph which a programmer views at any given time.

Although we applied the techniques in this paper to a graphical user interface toolkit, they can be applied equally well to commercial spreadsheets. Additionally, the technique of exploiting domain-specific knowledge to aid the debugging process seems like a promising idea that can be applied to other applications as well.

Background

One-way constraints are often informally written as an equation. For example, the equation $rect2.top = rect1.bottom + 10$ constrains the top of `rect2` to be 10 pixels below the bottom of `rect1`. More formally, a one-way constraint is written as

$$v_0, v_1, \ldots, v_m = C(p_0, p_1, p_2, \ldots, p_n)$$

where C is an arbitrary function, each p_i is a parameter (also called an input) to C, and each v_i is an output variable. The constraint is said to be a one-way constraint because if the value of any of the $v_i's$ are modified by the user or the program, the changed variable does not influence any of the p_i's.

A one-way constraint solver typically maintains a directed, bi-partite graph to keep track of the relationships among variables and constraints. The vertices of the graph represent variables and constraints and the edges represent the flow of data among variables and constraints. A variable has a directed edge to a constraint if the variable is used as an input (i.e., parameter) to that constraint. A constraint has a directed edge to a variable if it assigns its result to that variable. An example of a dataflow graph is shown in Figure 1.

When an application or user changes the value of a variable, the constraint solver performs a depth-first search of the dataflow graph to find and mark all the variables and constraints that are potentially affected by the change. The constraint solver can then either re-evaluate all the affected constraints immediately or it can re-evaluate a constraint only when its value is needed. In the former case the constraint solver is called an eager evaluator and in the latter case it is called a lazy evaluator. Our experience with the visual debugger described in this paper is based on an eager evaluator.

Previous Work

As noted in the introduction, much of the previous research on debugging one-way constraints has focused on providing some type of visualization of the dataflow graph. The C32 spreadsheet tool associated with the Garnet user interface toolkit draws arrows from a constraint's inputs to to the constraint's outputs [11]. Amulet has an inspector tool that allows the user to textually view the inputs to a constraint and, if the inputs are themselves constrained, the inputs to those "second-level" constraints [13]. Both C32 and Amulet's inspector show only a limited portion of the dataflow graph in the immediate vicinity of the constraint.

The CNV debugger displays a dataflow graph for a multi-way constraint system as a user creates the graphical interface [15]. The CNV debugger is like our debugger in that it makes use of domain-specific information to help the user analyze the constraint system. However, it is more focused toward the planning stage of a constraint solver (helping the user to determine why the constraint solver chose to solve an equation for one variable rather than another variable) than toward the execution stage.

Spreadsheets also provide a number of debugging techniques. For example, Microsoft Excel uses arrows to display either the parameters or dependents of a cell [10]. Each press of a button reveals one more level of predecessors or dependents. Excel also provides a number of error types which are displayed if the user inputs an incorrect value or a constraint computes an incorrect value. By selecting a "trace error" tool, the user can use these error values to incrementally find the path in the dataflow graph that led to the error.

These techniques work well with small spreadsheets but they break down with large spreadsheets. For example, because the cells are in fixed positions, arrows can start to criss-cross like sphaghetti. As another example, dependent cells may be in widely separated areas of the spreadsheet, which forces users to skip around in the spreadsheet, potentially losing context as they do so. Our techniques address these shortcomings by 1) using a dataflow graph which naturally congregates related cells in the same vicinity, and 2) only showing the portion of the dataflow graph containing a cell and the variables and constraints that determine that cell. This elision of cells reduces the visual clutter and also allows the dataflow graph to be drawn more compactly.

Finally, Igarashi and his colleagues use colors and animation to represent constraint relationships in a spreadsheet [8] Like C32, only one relationship can be visualized at a time. The dependent cells are shown in one color and the destination cell is shown in another color. Animation may be used to move from one relationship to another to create the effect of flowing through the relationships. This research was based only on spreadsheets that can fit on one screen. It is not clear how these techniques will scale up to larger spreadsheets. The animation is aimed at avoiding problems caused by criss-crossing arrows but we believe that the ability to rearrange cells plus the ability to elide irrelevant parts of the dataflow graph can alleviate this problem equally well.

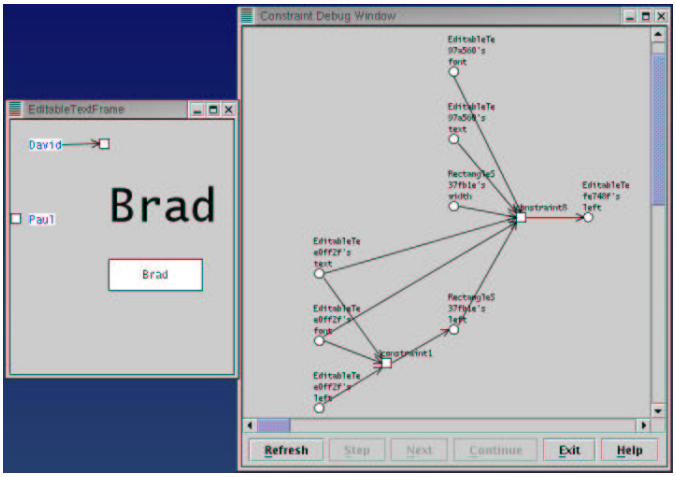

Figure 1: The dataflow graph shows the variables and constraints that determine the left property for the boldfaced "Brad" label. The circles denote properties and the rectangles denote constraints. `constraint8` directly determines "Brad's" left. It compares the values of two labels, "David" and "Paul", and aligns "Brad" to the right of the rectangle associated with the wider label, in this case, "David's" rectangle. "Brad's" left indirectly depends on `constraint1`, which places "David's" rectangle to the right of "David's" label. This dataflow graph is called a *constraint slice* because it shows only the slice of the dataflow graph that determines "Brad's" left rather than the dataflow graph for the entire graphical user interface.

Data Visualization Techniques

As noted in the introduction, our debugger uses constraint slicing to control the size of the dataflow graph, color tagging to aid the programmer in seeing how the inputs and outputs of constraints have changed, and an explanatory capability to assist programmers in determining the causes of bugs.

Slicing

In working with programmers, we have discovered that they typically start the debugging process by looking at a variable that has an incorrect value. They then work backwards from this variable, trying to pinpoint the original source of the problem. This type of debugging does not require that they view the entire dataflow graph. Instead, it only requires that they view the portion that extends backwards from the target variable. We call this portion of the dataflow graph a "constraint slice", since it resembles the slicing techniques used in the programming language literature, in which a program is "projected" on a certain set of statements so that only the parts of the program that have an effect on those statements is shown to the programmer [20, 19]. Figure 1 shows an example of a constraint slice. Constraint slices allow significant parts of the dataflow graph to be elided. They also allow the graph layout algorithm to rearrange vertices in the constraint slice to obtain a more compact representation that increases the amount of relevant information that can be displayed on the screen.

We have found that using constraint slices eliminates the need for panning and zooming because constraint slices tend to be narrow and short. This result is not surprising because a previous study has shown that dataflow networks tend to be modular (i.e., many disconnected networks), narrow (i.e., not too much branching out from constraints), and shallow (i.e., the longest path through a dataflow graph is typically less than 6 constraints or, equivalently, 12 vertices) [18].

Color Tagging

When debugging constraints, we have found programmers typically want an answer to one of the following three questions:

1. Why did the constraint not evaluate to the value I expect?
2. Why was this constraint not re-evaluated?
3. Why was this constraint evaluated (I did not expect it to be)?

Being able to view a generic dataflow graph is sometimes helpful but often it does not tell the programmer where to start looking for the bug. However, we have found that the answers programmers seek can frequently be found if the constraint solver stores information about its previous solving cycle (called *prior history*). There are two types of history information in particular that we have found to be useful to a programmer:

1. Changes to the dataflow graph: The addition or deletion of constraints, the execution of conditionals or a change in a value of a pointer variable can all cause edges to be added

to or deleted from the dataflow graph. By noting which edges are added to the graph and which edges are deleted, the constraint solver can later display these changes to the programmer. Such changes can be useful in helping a programmer to quickly identify how a cycle developed in the dataflow graph (because an edge was added) or why a constraint that the programmer expected to be evaluated was not evaluated (because a critical edge was deleted). In our debugger, newly added edges are shown in blue and newly deleted edges are shown in red. The edges remain colored until the next time the constraint's inputs or outputs change.

2. Edited values: Most dataflow constraint solvers allow an application to make multiple edits to variables before the constraint solver is invoked. The constraint solver can save these variables so that a programmer can later see what variables actually changed. We have found that sometimes a constraint is unexpectedly evaluated because a variable that the programmer did not expect to change was changed. Being able to see the set of changed variables can help to quickly identify this type of problem (a constraint slice quickly exposes the guilty variable by displaying a path back to that variable). The debugger highlights in green those variables that are changed by the application or the user, as opposed to changed by the re-evaluation of a constraint.

Explanation-Based Debugging

An initial version of the debugger allowed the programmer to utilize the above information by posing specific queries about changes to the dataflow graph, such as "show me all changed/deleted edges in the dataflow graph." However, we have found that the debugger can use this information to explain errors so effectively that specific queries are not needed (although they are still provided). The reason is that almost all logic-based constraint errors can be traced to the following factors:

1. There is not a path between the edited variables and the incorrect variable although a path used to exist. Typically this path disappeared in the last round of constraint solving because an edge was removed from the graph. However, sometimes the path never existed. In the former case the debugger tells the user that there used to be a path from an edited variable to the incorrect variable and highlights that path (the path includes a deleted edge). In the latter case the debugger the debugger tells the user that there is no relationship between the property in question and the edited property. This latter explanation is often useful because the programmer thought that a property depended on a variable that was being edited and was surprised when the property did not change at all.

2. There is an unexpected path between the edited variables and the incorrect variable. Typically this path appeared in the latest round of constraint solving because an edge was added to the graph. However, sometimes the path has been there for some time and the user just noticed it. In the former case the debugger tells the user that there is a new path from an edited variable to the incorrect variable and highlights that path (the path includes an added edge). In the latter case the debugger tells the programmer that the property in question depends on an edited property through a pre-existing path. This reason can arise when the value of an edited variable moves past a threshold, thus triggering a conditional that causes a constraint to depend on a different set of inputs. It can also arise when a pointer variable is changed, in which case a constraint will also often depend on a different set of inputs.

3. There is a cycle between the edited variables and the incorrect variable. Typically this cycle appeared in the latest round of constraint solving because an edge was added to the graph. In both cases the debugger simply indicates that there is a cycle between the edited varible and the incorrect variable and highlights the cycle in yellow.

4. The changes to the edited variables quiesced before reaching the incorrect variable. The debugger does not currently check for this case. We plan to add this reason in the future.

Figures 2-6 show a sample interaction between the programmer and the explanation-based portion of the debugger. The initial state of the interface is shown in Figure 2.a. In Figure 2.b, the string labeled "Brad" has inexplicably jumped to a new position when the user started editing the string labeled "Paul".

(a) (b)

Figure 2: The initial state of the graphical interface is shown in (a). In Figure (b) the boldfaced label "Brad" has inexplicably jumped to a new position during the editing of Paul's text string.

The programmer pops up a property inspector for "Brad" and notices that there is a constraint on the left property (the purple highlighting of "left" indicates that a constraint determines its value). The programmer now asks to see the dataflow graph that determines the left property and the debugger pops up a graph showing the appropriate constraint slice (Figure 3). The programmer can immediately see that the constraint that computes the left property has changed its

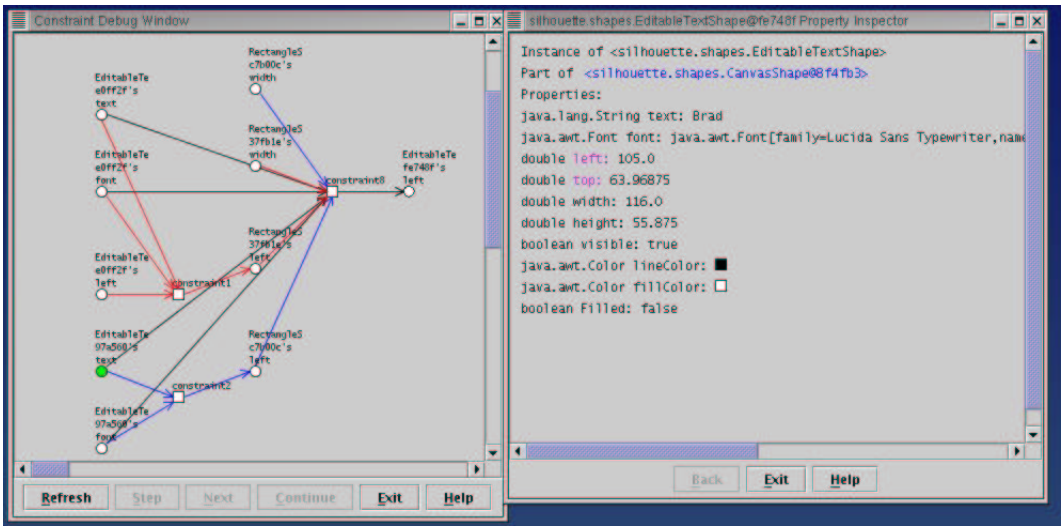

Figure 3: The constraint slice for "Brad's" label after "Paul's" text string has been edited. The blue edges denote new inputs to constraints, the red edges denote deleted inputs to constraints, and the green circle denotes a recently edited property. The property sheet at right displays the values of the properties for "Brad's" text object. The properties highlighted in purple are determined by constraints. The object highlighted in blue denotes a link that can be clicked on to bring up a property sheet for that object. In this case the highlighted object is the window containing "Brad".

set of inputs. However, there are a large number of added and deleted edges so the question now is what caused this large number of additions and deletions?

To explore further, the programmer selects the property and selects a menu option entitled "Something's Wrong. Please Suggest A Reason." The debugger pops up a dialog box that suggests two plausible reaons (Figure 4). The first reason is that the left property previously depended on the changed text property and the second reason is that the left property just started depending on the changed text property in a new way.

To get a more detailed explanation of the second reason, the programmer presses the "Show Path(s)" button. The debugger provides a more detailed explanation in two ways. First, in the explanation window it draws the specific path from the edited variable to the selected variable. Second, in the original dataflow graph window, it highlights in green the path from the edited variable to the selected variable (Figure 5). In the current example, the programmer can follow the green path and see that there are new edges between the edited text property and "Brad's" left property.

Finally, to see how a new path might have been established between "Brad's" left and "Paul's" text field, the user asks to see the source code for constraint 8, which determines "Brad's" left (Figure 6). We have found that the ability to view a constraint's source code interactively is very useful since the constraint name is frequently not descriptive. The programmer is allowed to assign names to constraints but typically does not do so. Therefore the constraint solver has to assign names to constraints, such as "constraint8", that are not meaningful.

The source code shows a conditional statement that is the cause of the problem. The conditional aligns "Brad's" left with one of two rectangles, depending on the widths of two respective textboxes. One can reasonably infer that "Paul's" textbox is one of the two textboxes being compared in the conditional and that this conditional is the source of the problem.

Implementation

The debugger is implemented in Java on top of the Silhouette interface development toolkit, which is also implemented in Java and is a toolkit we have been developing to help us explore constraint debugging. Originally, the debugger was implemented in the same process as a Silhouette application but we found that this bundling made the code awkward and also slowed down the application. As a result the Silhouette application now forks a separate process for the debugger and the two processes communicate via pipes. The debugger now does much of its work during a user's think time instead of during a user's interaction with the application.

Collecting Information

The debugger keeps track of added or deleted edges in the dataflow graph by taking a snapshot of a constraint's inputs and outputs each time the debugger is notified that the constraint is modifying its inputs or outputs list. Before the change is made, the constraint notifies the debugger and the

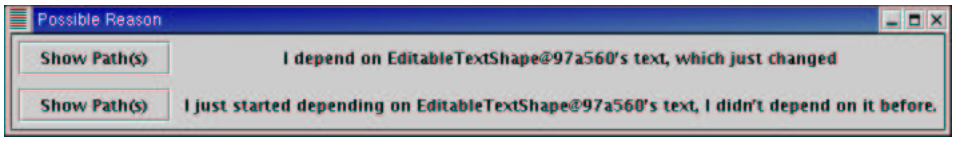

Figure 4: A dialog box that provides two reasons for why "Brad's" label may have jumped to a new position.

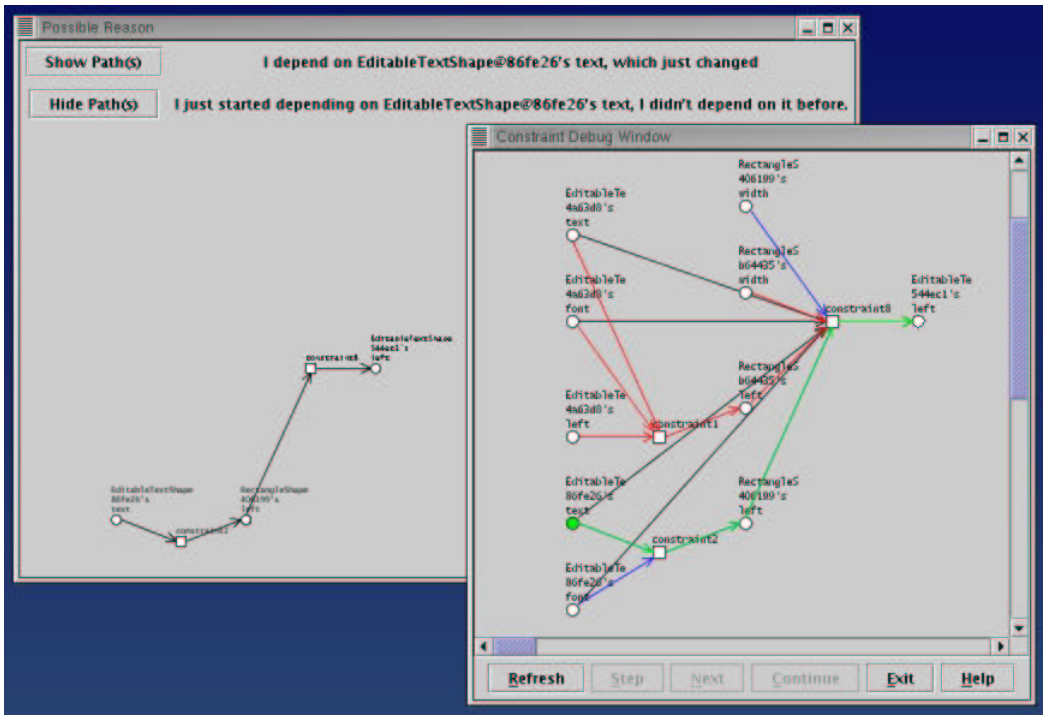

Figure 5: The programmer has requested a more detailed explanation of the second reason by selecting the second item's "Show Path" button. The debugger isolates the path from the edited property to the incorrect property in the explanation window and highlights the path in green in the dataflow window.

debugger saves the constraint's current set of inputs and outputs. When the user requests to see a portion of the dataflow graph that includes the changed constraint, the debugger compares the constraint's current input/output lists with the snapshotted input/output lists and computes the differences. The debugger does not throw away the snapshot until the next time the constraint changes its inputs or outputs. This "memory" feature is helpful because a bug frequently occurs as the user is moving or editing an object and hence several keystrokes or mouse events are processed before the user is able to stop and examine the problem. These events typically cause the affected constraints to be re-evaluated several times. However, their inputs and outputs typically do not change further and hence the relevant information is retained for the user.

Generating Explanations

When the user selects a variable and asks for reasons why something might be wrong, the debugger uses a reverse depth-first search of the dataflow graph to try to find paths from the selected variable back to each of the variables in the current set of edited variables. It tries finding paths through old edges, new edges, and deleted edges. If it finds a path, it generates a reason based on whether the path has a new edge, a deleted edge, or consists only of old edges. If it does not find a path, then it generates a reason stating that the selected property does not depend on this edited variable. The debugger also performs a strong connectivity test to determine whether there is a cycle between each edited varible and the selected property and, if so, generates a cycle explanation.

In searching for plausible reasons, the debugger does not list newly added or newly deleted paths between non-edited variables and the property in question. For example, in Figure 3 there are many paths involving new and deleted edges that do not go through the edited variable. Our rationale for not including such paths among the listed reasons is twofold. First, it would often lead to a large number of reasons that

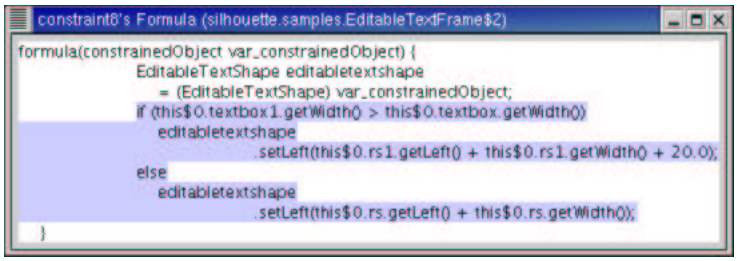

Figure 6: The source code for the constraint that determines "Brad's" left. The highlighted portion of the code shows that "Brad's" left depends on a conditional that compares the width of two textboxes. Based on the outcome of this test, "Brad's" label is aligned to the right of one of two rectangles.

a user would have to choose amongst. Second, the problem is caused by editing a variable so the explanation should be related to one of these edited variables.

Graph Drawing

For graph drawing we use a modified version of the Graphplace drawing package [16]. The modifications essentially involved translating the source code from C to Java and stripping out many of the unnecessary features related to postscript printing. The primary advantage of the Graphplace package is that it lays out a graph quickly and typically in a pleasing fashion. It does not guarantee that it will prevent edge crossings, even if edge crossings are preventable. In general, since dataflow graphs are narrow and shallow, edge crossings are not problematic.

Source Code Viewing

We use the JODE optimization and decompiler package to dynamically decompile a constraint object when the user asks to view the constraint's source code [4]. We modified the code in the package so that it only prints the source code for the function that computes the constraint, rather than all of the methods in the constraint object (the constraint object also has methods for evaluating constraints and marking constraints invalid).

Experience

An initial version of the visual debugger that displayed constraint slices and visually tagged the dataflow graph was used in a graphical interface class in Fall 2001. We observed that students frequently used the debugger to both examine the values of properties and to examine a dataflow graph. Students later reported that being able to view a dataflow graph decreased the amount of print statements that they felt they would have had to otherwise place in their code. Students also criticized the slowness of applications while the debugger was executing. Since that class, the debugger has been significantly speeded up by placing it into a separate process. Additionally the explanation-based facility and source code viewing of constraints has been added. While these new features have not been tested on programmers outside the Silhouette project, we have used the debugger to locate problems in the constraint solver. The color tagging and explanations have allowed us to identify problems much more efficiently than before the debugger was operational and we could rely only on print statements.

Future Work

Currently the visual debugger can be used to view the dataflow graph after a constraint crashes but it does not pinpoint the constraint that crashed. It would be quite helpful if a constraint crash would transfer control to the debugger which would then pop up a dataflow graph showing the constraint slice that originates at the crashed constraint. We also plan to add a breakpointing facility to the visual debugger so that the user can stop execution at selected constraints. Finally we plan to add an event recording feature so that a user can re-run an application up to a desired event and then start constraint debugging. Event recording would allow elusive bugs that appear only intermittently to be "captured" and then analyzed in detail.

Conclusions

In this paper we have described the design and implementation of a visual, explanation-based debugger for spreadsheet-like constraints. By restricting the domain of the debugger, we have been able to provide a pictorial visualization of the solver's state which is easy for a programmer to understand. We have also been able to make use of our knowledge about how constraint solvers typically manipulate their data structures in order to save state information that can be used to augment the visualizations with useful information about the recent behavior of the constraint solver. Further, by observing the types of bugs that frequently arise in constraints, we have successfully built an analysis component into the debugger that allows it to provide a range of plausible explanations for why a variable has an incorrect value. This feature has proven very helpful in quickly locating the source of a bug without having to resort to slow step-by-step, breakpoint execution of the constraint solver. Finally, by borrowing the technique of slicing from the programming languages literature, we have evolved a technique that allows our visualization techniques to scale up to very large scale constraint systems. The techniques described in this paper, while de-

signed for programming environments, can easily be adapted to commercial spreadsheets and hence can potentially provide a powerful new set of techniques for debugging spreadsheets.

Acknowledgements

This research was supported by NSF grant CCR-9970958.

REFERENCES

1. ALPERN, B., HOOVER, R., ROSEN, B. K., SWEENEY, P. F., AND ZADECK, F. K. Incremental evaluation of computational circuits. In *ACM SIGACT-SIAM'89 Conference on Discrete Algorithms* (Jan. 1990), pp. 32–42.

2. BROWN, P. S., AND GOULD, J. D. An experimental study of people creating spreadsheets. *ACM Transactions on Office Information Systems 5*, 3 (1987), 258–272.

3. HILL, R. D., BRINCK, T., ROHALL, S. L., PATTERSON, J. F., AND WILNER, W. The rendezvous architecture and language for constructing multiuser applications. *ACM Transactions on Computer Human Interaction 1* (June 1994), 81–125.

4. HOENICKE, J., FARRELL, B., NASH, J., AND RADEMACHER, T. Jode: Java optimize and decompile environment.

5. HUDSON, S. E. A system for efficient and flexible one-way constraint evaluation in C++. Tech. Rep. 93-15, Graphics Visualizaton and Usability Center, College of Computing, Georgia Institute of Technology, April 1993.

6. HUDSON, S. E. User interface specification using an enhanced spreadsheet model. *ACM Transaction on Graphics 13*, 3 (July 1994), 209–239.

7. HUDSON, S. E., AND SMITH, I. Ultra-lightweight constraints. In *ACM SIGGRAPH Symposium on User Interface Software and Technology* (Seattle, WA, Nov 1996), Proceedings UIST'96, pp. 147–155.

8. IGARASHI, T., MACKINLAY, J. D., CHANG, B.-W., AND ZELLWEGER, P. T. Fluid visualization of spreadsheet structures. In *1998 IEEE Symposium on Visual Languages* (Halifax, Nova Scotia, Canada, Sept 1998), IEEE Computer Society, pp. 118–125.

9. KNUTH, D. Semantics of context-free languages. *Mathematical Systems Theory 2* (June 1968), 127–145.

10. MICROSOFT CORPORATION. Microsoft Excel. 1998.

11. MYERS, B. A. Graphical techniques in a spreadsheet for specifying user interfaces. In *Human Factors in Computing Systems* (New Orleans, LA, Apr 1991), Proceedings SIGCHI'91, pp. 243–249.

12. MYERS, B. A., GIUSE, D. A., DANNENBERG, R. B., VANDER ZANDEN, B., KOSBIE, D. S., PERVIN, E., MICKISH, A., AND MARCHAL, P. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer 23*, 11 (Nov. 1990), 71–85.

13. MYERS, B. A., MCDANIEL, R., MILLER, R., FERRENCY, A., FAULRING, A., KYLE, B., MICKISH, A., KLIMOVITSKI, A., AND DOANE, P. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering 23*, 6 (June 1997).

14. REPS, T., TEITELBAUM, T., AND DEMERS, A. Incremental context-dependent analysis for language-based editors. *ACM TOPLAS 5*, 3 (July 1983), 449–477.

15. SANNELLA, M., AND BORNING, A. Multi-garnet: Integrating multi-way constraints with garnet. Tech. Rep. 92-07-01, Department of Computer Science and Engineering, University of Washington, Sept 1992.

16. VAN EIJNDHOVEN, J. Graphplace. 1994.

17. VANDER ZANDEN, B., MYERS, B. A., SZEKELY, P., GIUSE, D., MCDANIEL, R., MILLER, R., KOSBIE, D., AND HALTERMAN, R. Lessons learned about one-way, dataflow constraints in the garnet and amulet graphical toolkits. *ACM Transactions on Programming Languages and Systems 23*, 6 (Nov 2001), 776–796.

18. VANDER ZANDEN, B., AND VENCKUS, S. An empirical study of constraint usage in graphical applications. In *ACM SIGGRAPH Symposium on User Interface Software and Technology* (Seattle, WA, Nov. 1996), Proceedings UIST'96, pp. 137–146.

19. VENKATESH, G. The semantic approach to program slicing. *sigplan 26*, 6 (June 1991), 107–119.

20. WEISER, M. Program slicing. *IEEE Transactions on Software Engineering SE-10*, 4 (July 1984), 352–357.