An Explanation-Based, Visual Debugger for One-way Constraints

Bradley T. Vander Zanden, David Baker,
and Jing Jin
Computer Science Department
University of Tennessee
Knoxville, TN 37996
bvz@cs.utk.edu

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.

Categories and Subject Descriptors: D.2.2 [**Design Tools and Techniques**]: User interfaces; D.2.5 [**Testing and Debugging**]: Debugging aids; D.2.6 [**Programming Environments**]: Graphical environments, Interactive environments; D.3.2 [**Language Classifications**]: Constraint and logic languages, Data-flow languages; D.3.3 [**Language Constructs and Features**]: Constraints; H.5.2 [**User Interfaces**]: Graphical User Interfaces (GUI); I.2.5 [**Artificial Intelligence**]: Programming Languages and Software

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

INTRODUCTION

One-way, dataflow constraints are widely recognized as a potent programming methodology. They have found uses in a variety of applications including spreadsheets, graphical interfaces [13, 14, 4, 6, 7, 8], attribute grammars [10], programming environments [15], 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 [21]. 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 programmers 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 programmers can understand it after given only a brief explanation.
3. It is easy for the system 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 explanations. The programmer can select a reason and ask for a more detailed explanation in which case the debugger responds by highlighting certain portions of the dataflow graph.
3. It uses a technique called "constraint slicing" to limit the portion of a dataflow graph which a programmer views at any given time.

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 graph, called a *dataflow 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 a mark-sweep 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

The WhyLine debugger is perhaps most closely related to our work since it also allows the user to ask questions about what went wrong and tries to provide possible explanations [11]. It is also similar to our work in that it allows users to ask why things *did not* happen as well as why things happened, and in that it uses dynamic program slices to illustrate program behavior. It differs from our work in that its domain area is 3D scene creation rather than one-way constraints.

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 the constraint's outputs [12]. 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 [14]. 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 [18, 17]. 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. 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 where the errors are confined to a single page 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. This latter situation more closely conforms with constraints written for a graphical interface, where hundreds or thousands of constraints are often present. In effect spreadsheets suffer from the same drawback as C32, which is that only the dataflow graph in the immediate vicinity of the constraint can be easily visualized and many extraneous cells are also present.

Igarashi and his colleagues have tried a different visual technique that uses colors and animation to represent constraint relationships in a spreadsheet [9] 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 an-

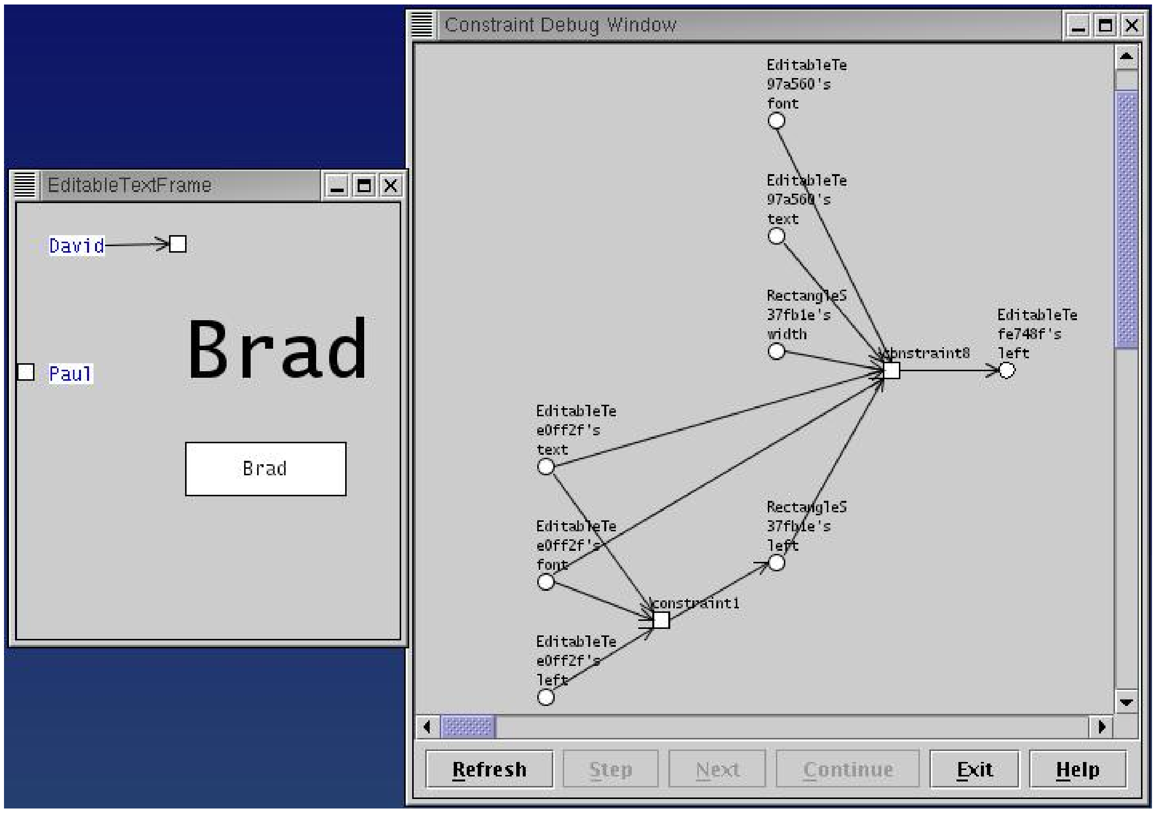


Figure 1: The dataflow graph shows the variables and constraints that determine the left property for the boldfaced "Brad" label. The circles denote properties and the rectangles denote constraints. `constraint8` directly determines "Brad's" left. It compares the values of two labels, "David" and "Paul", 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.

other 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 re-arrange cells plus the ability to elide irrelevant parts of the dataflow graph can alleviate this problem equally well.

Finally Burnett and her colleagues have studied non-visual techniques for spreadsheet debugging. The first of their techniques involves allowing a user to associate assertions with a cell that restrict its value [3]. These assertions are then checked by the spreadsheet solver each time a new value is computed or entered for a cell and it highlights cells that violate their assertions. Burnett and her colleagues have also examined whether it is better to interrupt a user to indicate that a potential error has occurred or to present a more subtle message that attracts the user's attention but does not prevent the user from continuing with the present task [16]. They found that it is better to present error information in a non-obtrusive rather than interrupt-driven manner. Burnett's non-visual techniques are complementary to our techniques it that her techniques are helpful in identifying that an error has occurred while our techniques are helpful in identifying the causes of an error and correcting that error.

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 [24, 23]. 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) [22]. It is true that not all networks are so well-behaved. In that case the programmer may have to scroll back-and-forth to observe the whole chain of constraints. However the explanation-based techniques described later typically narrow the problem to a specific part of the chain that can be easily observed.

Color Tagging

We have been observing programmers debugging constraints for 15 years and we have found that 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

The debugger can use this information to explain errors quite effectively. Again, drawing on our experience with observing programmers debugging one-way constraints, we have found that almost all logic-based constraint errors can be traced to the following causes:

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 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 but the user just noticed the error. 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. The former case 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 may 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 variable and the incorrect variable and highlights the cycle in yellow.

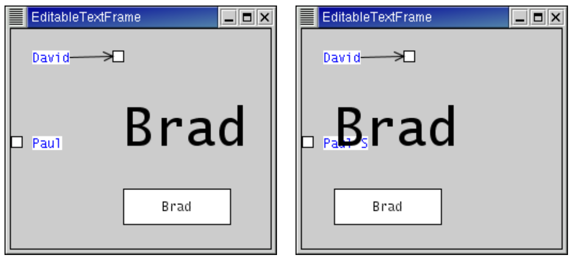
4. The changes to the edited variables quiesced before reaching the incorrect variable. The debugger does not currently check for this case. We plan to add this explanation in the future.

Based on these potential errors we have devised five messages from which the debugger can choose when suggesting an explanation for what might have gone wrong:

1. I used to depend on `object name/property name` but no longer do.
2. I just started depending on `object name/property name`, I didn't depend on it before.
3. Cycles were found.
4. I depend on `object name/property name`, which just changed.
5. I never depended on `object name/property name`, which just changed.

Typically the message "cycles were found" will accompany another message, such as "I just started depending on such and such, I didn't depend on it before." In other words, the two messages will jointly identify the problem.

Figures 2-5 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 since the white 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 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, right clicks to pop up a menu, and selects an option entitled "Something's Wrong. Please Suggest A Reason." The debugger pops up a dialog box that suggests two plausible reasons (Figure 4.a). 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 (Figure 4.b). Second, in the original dataflow graph window, it thickens the path from the edited variable to the selected variable (Figure 4.c). In the current example, the programmer can follow the thickened 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 5). 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.

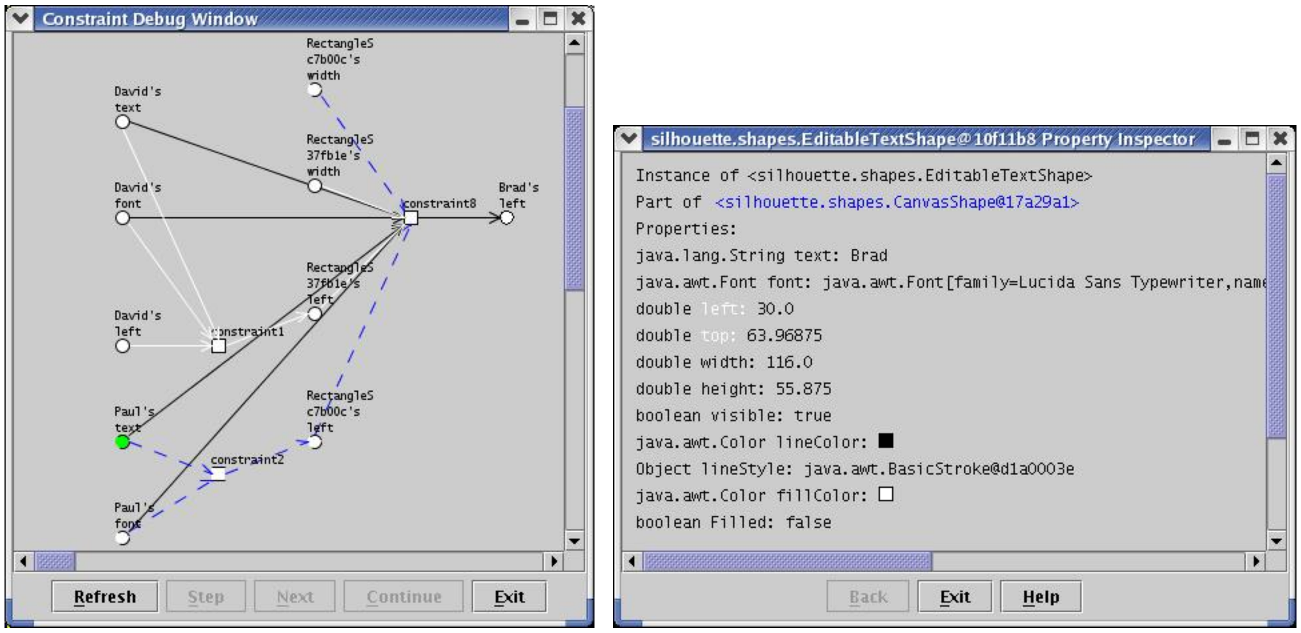
The source code shows a conditional statement that is the cause of the problem. The conditional aligns "Brad's" left with one of two rectangles, depending on the widths of two respective textboxes. Since Paul's text just changed one can reasonably infer that "Paul's" textbox is one of the two textboxes 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. The programmer must explicitly call a "start debugging" function in order for the debugger to be initialized and in order for it to start collecting information. Originally Silhouette automatically collected debugging information but we found that the collection of this information can significantly slow down the application when it is first starting up. Adding an explicit call gives the programmer the opportunity to control when the collection of debugging information should begin.

Separate Processes

The debugger was originally implemented in the same process as the 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 when the start debug-



```
Instance of <silhouette.shapes.EditableTextShape>
Part of <silhouette.shapes.CanvasShape@17a29a1>
Properties:
java.lang.String text: Brad
java.awt.Font font: java.awt.Font[family=Lucida Sans Typewriter,name
double left: 30.0
double top: 63.96875
double width: 116.0
double height: 55.875
boolean visible: true
java.awt.Color lineColor: ■
Object lineStyle: java.awt.BasicStroke@d1a0003e
java.awt.Color fillColor: □
boolean Filled: false
```

Figure 3: The constraint slice for "Brad's" left after "Paul's" text string has been edited. The dashed edges denote new inputs to constraints, the white edges denote deleted inputs to constraints, and the dark gray circle representing Paul's text denotes a recently edited property. The colors and line-styles of the edges have been altered for this paper so that the edges can be identified in a black-and-white figure. In practice blue edges denote new inputs, red edges denote deleted inputs, and green circles denote recently edited properties. The property sheet at right displays the values of the properties for "Brad's" text object. The properties highlighted in white are determined by constraints (in the actual property window they appear in purple).

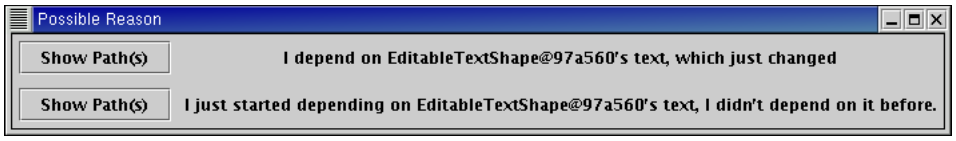
ging function is called and the two processes communicate via pipes.

The separation of the processes speeds up the application while it is running. When the debugger ran as a separate thread, the application had to pause while the debugger determined if it had to update its data structures. With the separate processes, the application posts the debugging information to a pipe but continues execution. The debugger eventually gains control during a pause in the application (e.g., the user finishes an interaction and starts thinking) and posts the debugging information to its own data structures. Hence the debugger can often update itself during the user's think time, rather than during the user's interaction time. The speed up in applications has been noticeable.

The separation has also simplified the debugger's code as we have started adding breakpoints to the debugger (discussed in future work). The simplification derives from the fact that the debugger is itself implemented in Silhouette and from the fact that the debugging windows must be handled differently than the application's windows. First, if the programmer is using breakpoints in the debugger, then the debugger's windows need to be updated when the breakpoints are reached but the application's windows should not be updated because the application's objects are in an inconsistent state. Second, the debugger wants its windows to respond to input events from the user whereas the application's windows should not respond to events from the user.

This dichotomy in the handling of events and updating of windows is simple to implement when the debugger and the application run in separate processes because each is running its own version of Silhouette. When they ran in the same process, we tried to run the debugger in a separate thread and we marked windows as being either an application window or a debugger window. Unfortunately, Java maintains only one event queue for an entire process rather than one event queue per thread. Further, window updates occur as a result of paint events and paint events cannot be executed until the previous event has been completely handled.

This event model caused considerable problems because a paint event initiates constraint solving. Until the constraint solving was completed and the window redrawn Java would not process any further events. However, the debugger windows were supposed to update themselves and handle input events as the constraint solver executed. In order to force Java to process the events the debugger thread had to commandeer the event queue (which is permitted by Java) and start dispatching events itself. Since the first few events on the event queue were typically paint events for the other Silhouette application windows, the debugger had to keep track of these events so they could be requeued once the constraint solving was completed. Naturally the commandeering of the event queue led to intricate code and occasionally applications would freeze up for undeterminable reasons. The separation of the debugger and the application solved these problems.

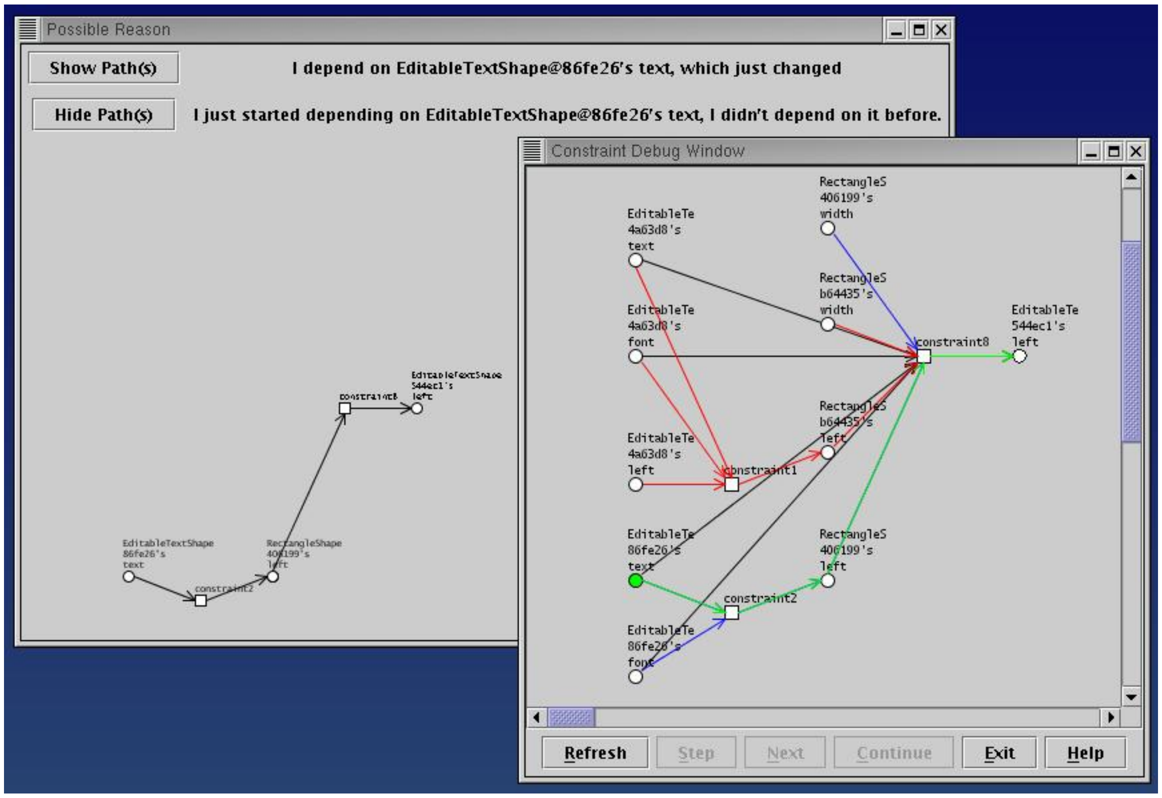


Possible Reason

Show Path(s) I depend on EditableTextShape@97a560's text, which just changed

Show Path(s) I just started depending on EditableTextShape@97a560's text, I didn't depend on it before.

(a)



Possible Reason

Show Path(s) I depend on EditableTextShape@86fe26's text, which just changed

Hide Path(s) I just started depending on EditableTextShape@86fe26's text, I didn't depend on it before.

(b) (c)

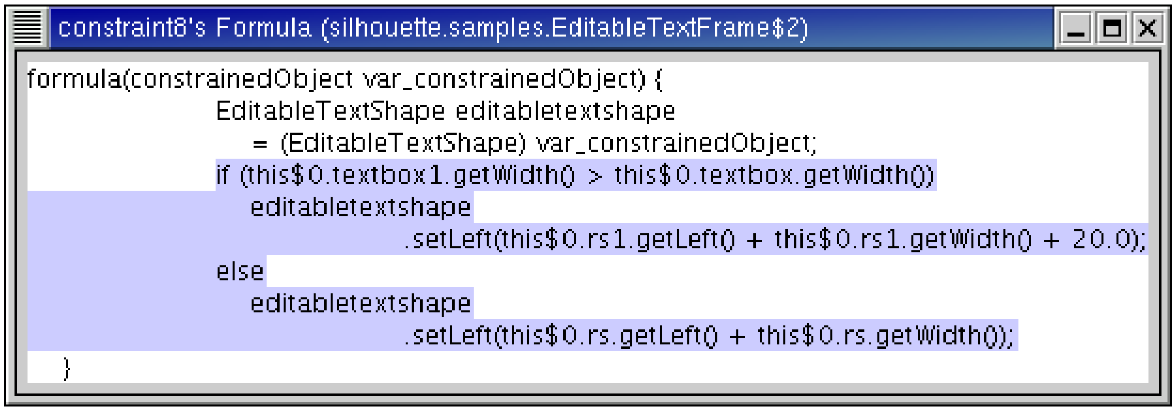
Figure 4: (a) A dialog box that provides two reasons for why "Brad's" label may have jumped to a new position. (b) The programmer has requested a more detailed explanation of the second reason by selecting the second item's "Show Path" button. The debugger isolates the path from the edited property to the incorrect property in the explanation window (b) and highlights the path as a thick black line in the dataflow window (c). In the actual debugger the thick black line would appear as a green line.

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 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 the variable might have an unexpected value, the debugger uses a reverse depth-first search of the dataflow graph to attempt 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 variable


constraint8's Formula (silhouette.samples.EditableTextFrame$2)

```
formula(constrainedObject var_constrainedObject) {
                EditableTextShape editabletextshape
                   = (EditableTextShape) var_constrainedObject;
                if (this$0.textbox1.getWidth() > this$0.textbox.getWidth())
                   editabletextshape
                                 .setLeft(this$0.rs1.getLeft() + this$0.rs1.getWidth() + 20.0);
                else
                   editabletextshape
                                 .setLeft(this$0.rs.getLeft() + this$0.rs.getWidth());
   }
```

Figure 5: 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.

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 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 [19]. 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 [5]. 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 taught at the University of Tennessee. 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

There are a number of things we would like to do to improve the current debugger:

1. Crash Management: 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.

2. Breakpoints: We would like to add a breakpointing facility to the visual debugger so that the user can stop execution at selected constraints.

3. Event Recording: 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.

4. Display Values: It would be helpful to display the current value of each variable in the constraint slice and perhaps even allow the user to look at previously computed values [20]. The Whyline provides a timeline of values that suggests one possible way to provide such a feature [11].

5. Object Identification: It would be helpful to have some way to show a correspondence between objects in the constraint slice and objects in the interface window. A simple technique would be to have the user select a property and then ask the debugger to blink the corresponding object in the interface window.

Another interesting direction for future work is to investigate whether the techniques described in this paper can be adapted to commercial spreadsheets. As described in the related work section it might be possible to augment spreadsheets with a window that displays a constraint slice for a cell. The question is whether the visualization disconnect from the spreadsheet grid would be too great. In other words, cells derive their meaning in many applications from their physical location rather than from their label. However, in other applications, such as Microsoft's Access, cells are labeled and are embedded in forms rather than in grids. In this situation the constraint slicing technique might be more helpful.

Conclusions

In this paper we have described the design and implementation of a visual, explanation-based debugger for one-way 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.

ACKNOWLEDGMENTS

This research was supported by NSF grant CCR-9970958. We would also like to thank the reviewers for their helpful comments.

REFERENCES

1. Bowen Alpern, Roger Hoover, Barry K. Rosen, Peter F. Sweeney, and F. Kenneth Zadeck. Incremental evaluation of computational circuits. In *ACM SIGACT-SIAM'89 Conference on Discrete Algorithms*, pages 32–42, Jan. 1990.

2. Polly S. Brown and John D. Gould. An experimental study of people creating spreadsheets. *ACM Transactions on Office Information Systems*, 5(3):258–272, 1987.

3. Margaret Burnett, Curtis Cook, Omkar Pendse, Gregg Rothermel, Jay Summet, and Chris Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *Proceedings of the 2003 International Conference on Software Engineering*, pages 93–103, Portland, OR, 2003.

4. Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F. Patterson, and Wayne Wilner. The rendezvous architecture and language for constructing multiuser applications. *ACM Transactions on Computer Human Interaction*, 1:81–125, June 1994.

5. Jochen Hoenicke, Bernard Farrell, Jonathan Nash, and Tobias Rademacher. Jode: Java optimize and decompile environment.

6. Scott E. Hudson. A system for efficient and flexible one-way constraint evaluation in C++. Technical Report 93-15, Graphics Visualizaton and Usability Center, College of Computing, Georgia Institute of Technology, April 1993.

7. Scott E. Hudson. User interface specification using an enhanced spreadsheet model. *ACM Transaction on Graphics*, 13(3):209–239, July 1994.

8. Scott E. Hudson and Ian Smith. Ultra-lightweight constraints. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 147–155, Seattle, WA, Nov 1996. Proceedings UIST'96.

9. Takeo Igarashi, Jock D. Mackinlay, Bay-Wei Chang, and Polle T. Zellweger. Fluid visualization of spreadsheet structures. In *1998 IEEE Symposium on Visual Languages*, pages 118–125, Halifax, Nova Scotia, Canada, Sept 1998. IEEE Computer Society.

10. D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, June 1968.

11. Andrew J. Ko and Brad A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. *CHI Letters*, 6(1):151–158, April 2004. Proceedings SIGCHI'2004.

12. Brad A. Myers. Graphical techniques in a spreadsheet for specifying user interfaces. In *Human Factors in Computing Systems*, pages 243–249, New Orleans, LA, Apr 1991. Proceedings SIGCHI'91.

13. Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

14. Brad A. Myers, Rich McDaniel, Robert Miller, Alan Ferrency, A. Faulring, Bruce Kyle, Andy Mickish, Alex Klimovitski, and P Doane. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6), June 1997.

15. T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM TOPLAS*, 5(3):449–477, July 1983.

16. T.J. Robertson, Shrinu Prabhakararao, Margaret Burnett, Curtis Cook, Joseph R. Ruthruff, Laura Beckwith, and Amit Phalgune. Impact of interruption style on end-user debugging. *CHI Letters*, 6(1):287–294, April 2004.

17. Michael Sannella. Analyzing and debugging hierarchies of multi-way local propagation constraints. In *Lecture Notes in Computer Science No. 874*, pages 63–77. Proceedings of the 1994 Workshop on Principles and Practice of Constraint Programming, 1994.

18. Michael Sannella. *Constraint Satisfaction and Debugging for Interactive User Interfaces*. PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, 1994. Also available as Technical Report 94-09-10.

19. Jos van Eijndhoven. Graphplace. 1994.

20. Brad Vander Zanden, Richard Halterman, Brad A. Myers, Rob Miller, Pedro Szekely, Dario Giuse, David Kosbie, and Rich McDaniel. Lessons learned from users' experiences with one-way constraints. *Software Practice and Experience*. To Appear.

21. Brad Vander Zanden, Brad A. Myers, Pedro Szekely, Dario Giuse, Rich McDaniel, Rob Miller, David Kosbie, and Richard Halterman. Lessons learned about one-way, dataflow constraints in the garnet and amulet graphical toolkits. *ACM Transactions on Programming Languages and Systems*, 23(6):776–796, Nov 2001.

22. Brad Vander Zanden and Scott Venckus. An empirical study of constraint usage in graphical applications. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 137–146, Seattle, WA, November 1996. Proceedings UIST'96.

23. G.A. Venkatesh. The semantic approach to program slicing. *sigplan*, 26(6):107–119, June 1991.

24. Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.