HotWire -- A Visual Debugger for C++

Chris Laffra and Ashok Malhotra

I.B.M. Thomas J. Watson Research Center,
P.O.Box 704, Yorktown Heights, NY 10598, USA
Email: {laffra,petsa}@watson.ibm.com

ABSTRACT

We argue that visualization is essential in a modern debugger. Instead of inserting debug statements throughout the code, it should be possible to easily define visualizations while running the program under control of the debugger, resulting in what might be called "visual printf's". A visualization of a C++ program can provide exciting insights. Bugs that cannot be found that easily with non-visual techniques are now found, just by watching the visualizations. However, the mechanisms to define the visualizations should be easy to understand, easy to apply and cause only minimal overhead to the programmer (who is the *end-user* of the visual debugger). HotWire is not only equipped with a couple of standard visualizations, but also with a small declarative script language (using constraints) that can be used to define new custom visualizations. This paper addresses user interface aspects of debugging tools. Specifically, the user interface of **HotWire**, a debugger for C++ and SmallTalk on AIX and OS/2 is described.

INTRODUCTION

Every programmer knows that debugging is very time consuming, yet it is universally considered to be a second-class activity. Programmers hate to admit that they create bugs. “Real” programmers don’t use a debugger. As a result, the development of debuggers is regarded as a third-class activity. Even though object-oriented (OO) techniques have been around for quite a while, debuggers for OO languages, and in particular C++, are still very immature. Most of them originate from debuggers for C. As a result, they use wrong metaphors, like structures and procedures. They do not recognize objects as objects. This is the typical C++ debugging scenario: Our C++ program crashes. We use a C++ debugger to determine where the illegal instruction occurred. After the location has been determined (the call stack is inspected), we may inspect one or two variables. But, the call stack is what we are after. We leave the debugger, edit the source code, insert some print statements, recompile the program, and run it. We repeat this (long) cycle, until we find the bug. Then we carefully have to remove the scaffolding to cleanup the source code again. To save time, C++ code is very often scrambled with #IFDEF DEBUG statements to use the debugging statements at a later stage. It is sad, but this is the case is most C++ program development today. However, we even know of C++ systems that take 5 hours for a complete compilation, and that cannot be loaded into a debugger within reasonable time. How do these people debug their C++ systems?

What is required, are filtering techniques that enable us to define and regulate our *focus* while we are trying to understand why a system behaves as it apparently does. Additionally, the enormous amount of information encountered while inspecting running applications should be represented in a fashion that allows programmers to actually interpret and use it. Breakpoint analysis, code stepping, or execution traces may indicate and solve bugs, but they typically do not work in real-world scenario’s. It is senseless to try and inspect a megabytes sized trace file.

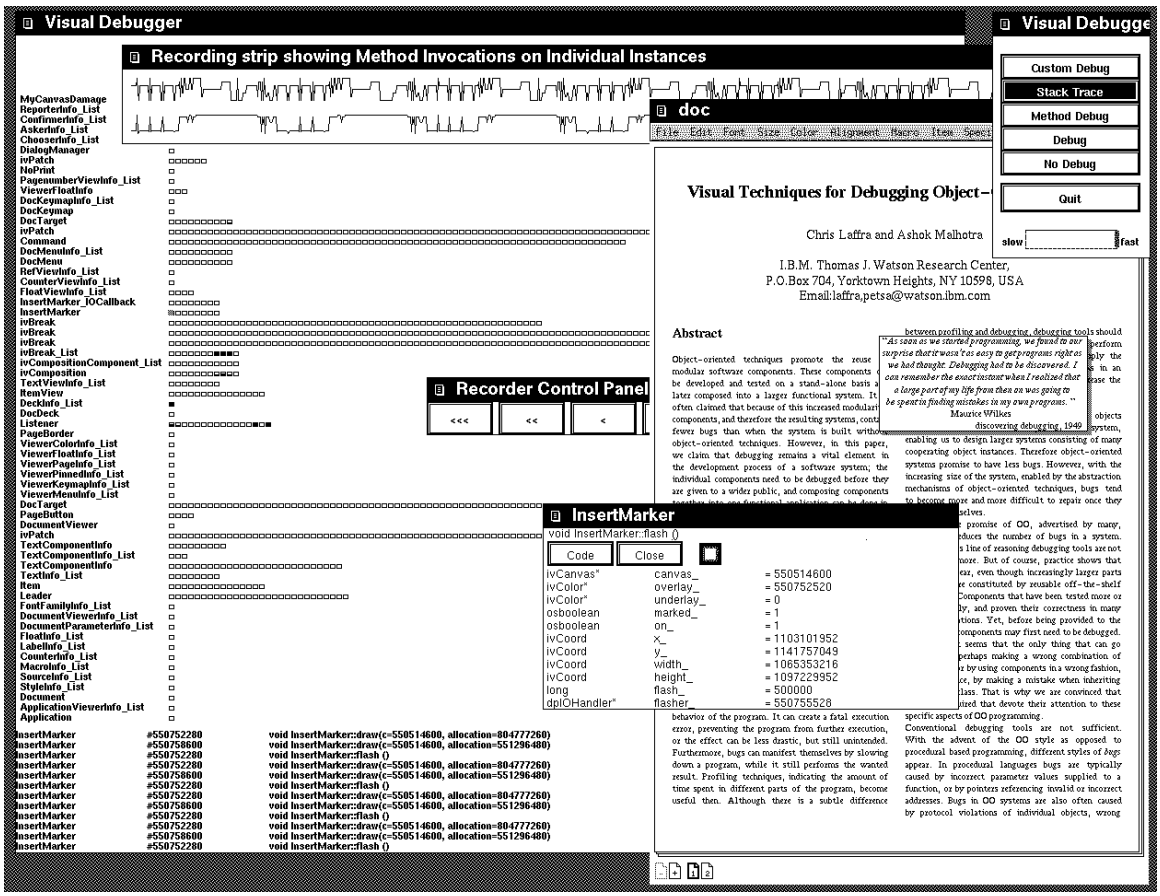


Figure 1: *A sample session with HotWire. The application shown in the window with the label "doc" (the large window on the right) is a wordprocessor. The other windows are part of the debugger. The large window on the left shows all instances in the application collected by type name. Each rectangle represents one instance. The rectangles are animated when a method is executed.*

We need abstraction techniques, showing the same information in another, more informative, way. This is where *visualization* comes into the picture. Visualization is the use of computerized images to enhance the insight of human beings into complex phenomena. Visualizations provide us with a way to improve the signal/noise ratio. Maps are one well-known abstraction mechanism with a very high information content [Miya 93]. Their signal/noise ratio is better than that of photographs or movies. The consistent characteristics of maps, such as easily measurable geometry, the use of symbols (always open to interpretation, though), and legends enable quick determination of source and destination. Aerial photographs are certainly less suitable for this purpose. The challenge is to find similar abstractions for displaying running OO programs. What techniques can be used to draw the map? What legends are required? What symbols are suitable? How can we change the resolution of the map and update the different abstractions and symbols on the map while zooming in (first look at the US, then at the Northeast, then at New York City, and finally at the streetmap of Manhattan). This hierarchical approach is one way to solve the problem of managing the information. But, there are many more.

Visualization techniques can help programmers by allowing them to detect anomalies in object activation and message patterns, simply by recognizing a "weird" aspect of a visualization. Visualizations offer a better representation of spatial relationships between objects, allows users to understand the picture "at-a-glance", can show trends and anomalies, and use cognitive associations (the viewer's pattern recognition skills). These points were recognized in earlier work, such as Balsa [Brown 88], Tango [Stasko 90], and the Object Visualizer [De Pauw 93]. Sound helps too [Smith 87], in that respect that incorrect behavior in the program

can be detected by recognizing that the program makes a "strange sound" (an important technique also used by car mechanics). Once the problem has been detected, it needs to be identified. Browsing techniques and more specific visualizations need to be applied to identify which particular part of the application is leading to the unexpected behavior.

Different applications and different situations require different approaches and visualizations to understand what goes on within the system, and how certain events are related to others. Debuggers should be equipped with built-in visualization techniques, but, more importantly, with facilities that allow for on-the-fly definition of visualizations, depending on the particular needs of the development team at that specific time. Furthermore, the mechanisms to define the visualizations should be easy to understand, easy to apply, and with minimal overhead.

As an aside, C++ compilers can make it more complicated by not generating enough information to allow C++ debuggers to do a decent job. People will have to become aware of the fact that debugging is an important aspect of program development, and that debuggers have a right to exist. A dialogue should be started between C++ compiler teams, debugger builders, and human factors experts.

In this paper we describe a visual debugger called **HotWire**. The debugger is equipped with fixed visualizations showing a selection of all object instances in the application and the method calls that are invoked on them (see Figure 1). In addition, the debugger is equipped with a special script language to define customized visualizations.

Before introducing the design issues that led to HotWire, we discuss specific type of bugs that could be introduced while using OO techniques. Then, some interesting visualization techniques in the context of OO debugging will be highlighted. It will be shown how they could enhance the understanding of the behavior of a program being debugged. Furthermore, different types of mappings that can be envisioned between the OO concepts and the visualizations will be discussed. Using visual techniques in the interface of a debugger enhances the understanding of a program. Understanding the program is the first step in detecting and identifying a malfunction.

WHAT CAN GO WRONG IN OO PROGRAMS?

Many problems in applying OO are related to the *syntax* of the language. Using the wrong type of brackets, omitting separators, etc. Most of these problems will be handled by the compiler, but some remain undetected until runtime. This is the fact with the list of classic bugs for Smalltalk [Johnson 93], which lists 12 problems with using Smalltalk. From that list, 6 bugs are directly related to the Smalltalk-80 environment. The rest are related to syntax, scope of local variables, object initialization, and different Smalltalk engines. In fact, only one addresses a problem that can be generalized to OO in general, and is not related to syntax or programming environment. It has to do with handling the Collection class.

When a given OO language is an extension of an existing language (like C), all problems that can occur in that underlying language, are automatically inherited in the OO language. Division by zero, loops that are traversed incorrectly, array bound violations, and invalid pointer arithmetic. These are some of the problems that debuggers may still need to take care of.

If we concentrate on OO concepts, the following things may go wrong, amongst others:

- Creating too many instances to do the job. There are a number of different forms of this problem:
  - Creating too many static objects. Normally, each block of code may contain the declaration of a local variable. With each execution of the block (it may be contained in a loop construct), the object is created, initialized, cleaned up, and removed. Typically, inefficiency is the result. Debuggers showing all instances dynamically, at runtime, address this problem (see Figure 2). Namely, the representations for the instances are created and deleted in a rapid rate. See also [De Pauw 93] for other interesting ways in which object histograms can be used.

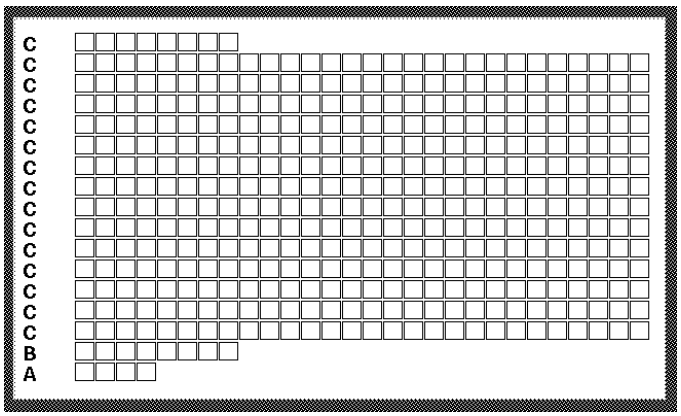


Figure 2: *This HotWire view shows all the instances in a given program. For each instance of the type* ***A*** *there exist two instances of the type* ***B****. There exist a large number of instances of the type* ***C****, which might indicate a memory leak.*

- Object groups. In many design patterns, objects are created in cooperating roles. An example could be that for each instance of the type A there should be two instances of the type B. Running the application, stopping the application at a particular moment in time, and counting the object instances shown in the debugger may reveal unexpected objects (see Figure 2).
- Unused objects instances (which cannot be collected as "garbage"). An application can create objects, and never actually use them at all during the execution. During development, some particular use was envisioned, and later changes in the program made these objects inappropriate. Debuggers that also show the activation on these objects (see Figure 3), might reveal this particular aspect, by highlighting at the end of the run all unused objects, and show where they were created. The debugger should remember where in the code instances are being created, so that selection of their representation will report that location.

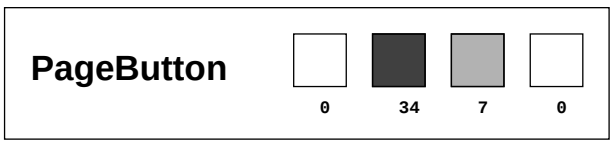


Figure 3: *This view shows that a given program contains 4 instances of the type* ***PageButton****. Instances 1 and 4 have never been used after creation, while instances 2 and 3 have until now executed 34 and 7 methods respectively. Instances 1 and 4 may very well be redundant objects. The shading of the instance also reflects the number of methods that have been invoked on the instance (shading poses a problem, in that there are only a limited amount of distinguishable shades).*

- Forgetting to delete instances. Objects may be explicitly created, manipulated, and passed around. After they serve their purpose, they should be deleted again. If they are not deleted, in languages without garbage collection, this results in memory leaks. A visual debugger indicates this problem. If you run your application long enough, it will simply fill up the debugger windows.
- Calling methods on the wrong target instance. For example, an instance can have references to a number of objects that perform some processing for it. Even though it may have been the intention of the programmer to equally divide the work among the different workers, a bug may result in having one single object doing all the work. Looking at the visualization of the method invocations shown in a visual debugger indicates this behavior (see Figure 4).
- Sending incorrect message contents to objects. Even when the target of a message is the correct object instance, and the actual method is the one that should be called, the contents of the method can still

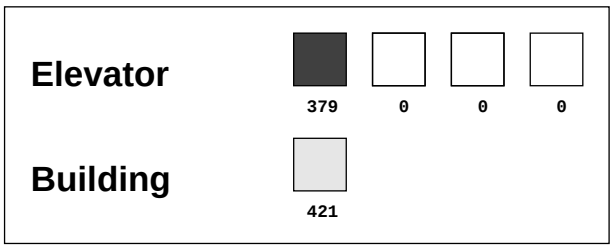


Figure 4: *This view shows a building with 4 elevators. It is clear that elevator 1 has done all the work, while elevator 2, 3, and 4 have done nothing so far.*

be invalid. Typical problems are passing references to wrong objects, wrong numerical values, null pointers, etc. Some problems directly lead to a program crash (rather simple to detect), but some may result in an incorrect program behavior (a lot harder to detect). A good OO debugger should remember the history of method calls, and should allow the programmer to browse through this recording to find incorrect calls. Figure 5 shows a snapshot where a call is made to a method with unexpected arguments.

- Sending messages to objects in the wrong sequence. Obeying the *protocol* of an object class or the combined protocol of a set of classes can be very difficult. In most OO languages, the programmer has no provisions to clearly specify the protocol of a class. When languages provide this facility (like the Protocol in Procol[van den Bos and Laffra 89]), it would ease the task of the programmer by providing a concrete specification of the access behavior of the object in question. Furthermore, if the specification is accessible at runtime, debuggers would be able to handle this very common access control problem much better, especially when the protocol could be visualized. As an alternative, keeping histories of method accesses may trigger the programmer to the occurrence of a protocol violation (see Figure 6). An important aspect of this problem is that it is especially apparent when using library classes. These toolkits typically consist of hundreds of classes, with thousands of methods. Using toolkits correctly is a major exercise and debuggers still have big difficulties in supporting this style of program development.

- Incorrect updates to the state inside a method. Often, the intended side-effect of a method is to update the state of the instance. Of course, this can be, and often is, done erroneously. Assertions over the state of the object, and pre-, and post-conditions (such as provided in Eiffel[Meyer 88]), can address this problem. But, simple visualizations of the state of the object are equally powerful, more versatile and dynamic, and more effective (see also Figure 10).

MAPPING OO CONCEPTS TO VISUALIZATIONS

The interesting question to be discussed next is how to connect the two very different universes of (1) OO concepts, such as classes, instances, and methods, and (2) custom visualizations, such as boxes, lines, texts, diagrams, and bitmaps, The form in which the mapping between these two worlds manifests itself is extremely important, as it will define the success of any visual debugger.

One way to define the mapping is to use the Model View Controller (MVC) paradigm [Goldberg 83]. The MVC concept is used in the Smalltalk environment to define user interfaces. The MVC triad separates the three components – functionality, input, and output. The underlying functionality is represented by a so-called *model*. The programmer can attach a *controller* and multiple *views* to each model. The view only contains the methods to make the model visible and to interact with it. The controller is used to control the communication between the views and the model. It ensures that each of the views always correctly represents the state of the model. In the context of this paper the model would reside in the program, and the views would be provided by the visualization library. The difficult part is defining the controller. If we are able to define a simple way to define controllers, it is relatively affordable to connect new views to models in our target program.

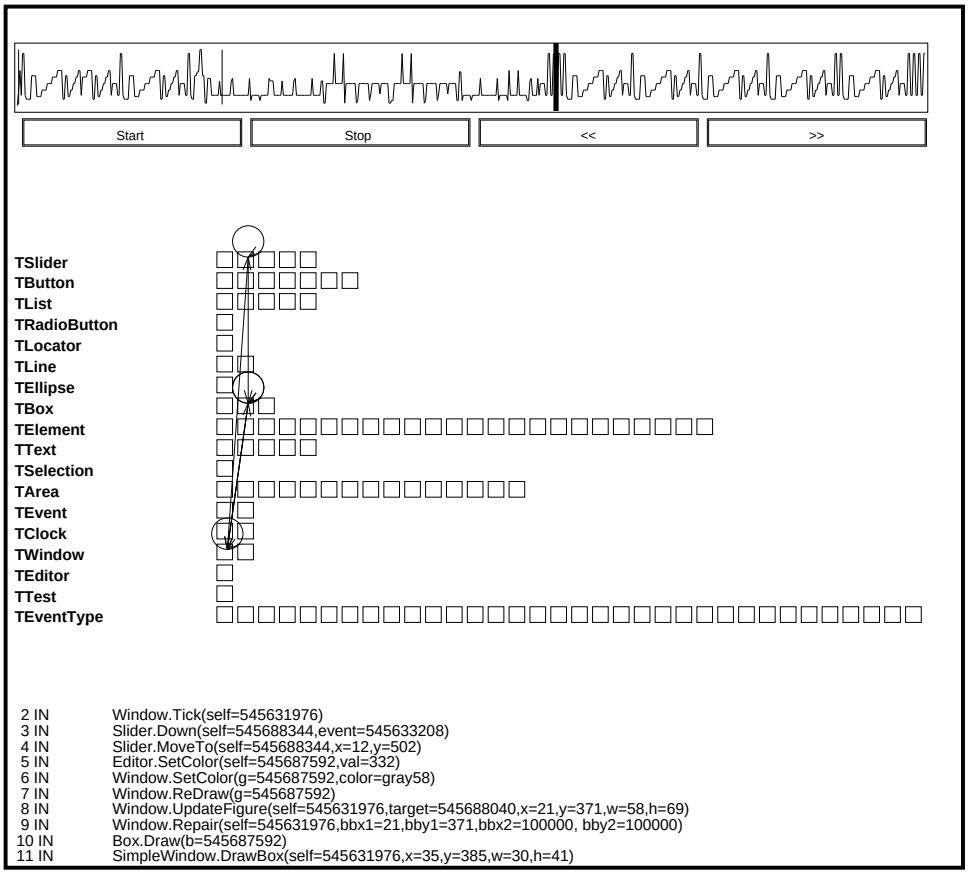


Figure 5: *This snapshot shows a debugging session of a drawing editor and debugger. It indicates a call to the method* ***Window.Repair*** *on line 9, with unexpected values for arguments* ***bbx2****, and* ***bby2****. The recording strip shows a history of methodcalls. Each pixel from left to right is one call. The height of the pattern is defined by the address of the instance. Each instance always appears at the same height. At the bottom, the callstack is shown at the selected time frame. The window in the middle shows all instances that were involved in the interaction pattern at that time.*

While the mapping between model and view in the MVC paradigm is defined in a *procedural* fashion, ideally the mapping should be declarative. A declarative mapping is easier to define, simpler to read, and understand. Furthermore, if the mapping is to be done in a visual fashion, a declarative "language" is also to be preferred. One could make the connections by simply clicking on objects or classes in the application and select an appropriate view from the library, again with a few mouse clicks. Once the mapping is given, the visualization system should maintain the relationship thus defined. Keeping the views up-to-date with their model is now a task for the visualization system, *not* for the programmer.

The development of a visualization or animation always requires a certain investment. Sometimes, the development of a specialized visualization is not justified, because other techniques allow for faster determination of a problem. However, when projects get larger, the initial effort of developing a visualization may prove to be very worthwhile at a later stage, either to test further developments of the system (effectively re-running the script from time to time), to report progress to third parties, or to educate new-comers to the project team.

Figure 7 shows a custom visualization of an elevator simulator. The emphasis here is on showing how simple it is to develop a visualization without the need of adding any extra statements to the code of the program. The visualization is entirely independent of the program.

Defining a visualization involves a number of steps. Step 1 is the identification of the elements of interest in the application (the models). Step 2 is the selection of appropriate visualizations. Step 3 is the mapping

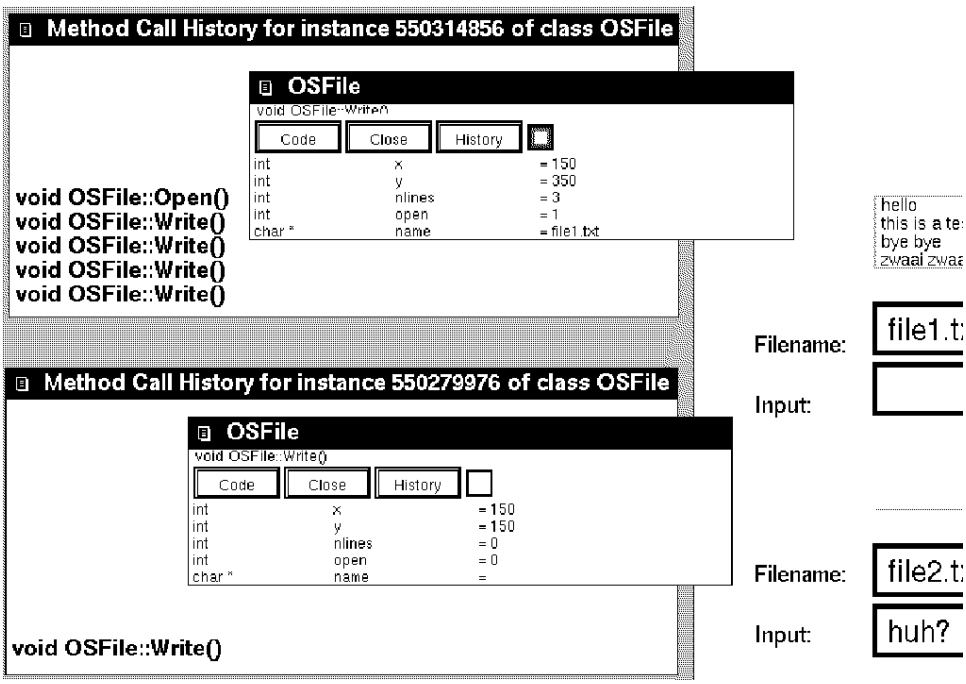


Figure 6: *This HotWire debugging session shows how keeping a history of method invocations can help identify object protocol violations. Two instances of the type* ***OSFile*** *are shown. Both are abstractions of the file concept. By typing in a name in the* ***Filename:*** *field, a particular file can be opened for writing. Then for each string that is given in the* ***Input*** *field, a line is written to the file. In the bottom interactor, the user forgot to hit the enter key, after typing in the filename to be opened. The instance browser in the left bottom reflects this fact, as no call to* ***Open*** *is recorded. Furthermore, the value of the* ***open*** *field is equal to 0, and not 1, as is the case in the top left window.*

between the models and their views. Step 4, the most difficult part yet, is to define spatial coordinates for the views, both absolute to the addressing space of the visualization environment, but also relative to each other. In most visualizations the relative layout of views is important, and their computation typically involves tedious calculations. The programmer has to come up with a translation from abstract relationships, such as *A must be left from B*, *C has to be inside D*, etc., into explicit coordinates to move individual views. Combinations of different relationships complicates matters even further. A different specification technique is required.

A powerful mechanism to define relationships is given by the *constraint* programming metaphor [Leler 88, Freeman-Benson 90]. Relationships are defined once, and the constraint solver will maintain the universe of constraints, ensure that all constraints are satisfied, and take measures when a certain condition invalidates a given constraint. Constraints can be one-way (working in one direction only), or two-way (updating both components, when one is changed) , see [Sanella 93]. In the current version of HotWire, one-way constraints are used to define relationship between graphical elements. Changes in the state are forward to dependents, which may again trigger other constraints. Circular dependencies are not dealt with perfectly. We are planning to incorporate a multi-way constraint system to handle the constraints in our specification language.

The following script shows how constraints are used to layout the bars in a bargraph animation for a sorting algorithm on a linked list. The linked list class is called `List` and has instance variables `value` and `next`:

```
class List {
public:
    int value;
    List *next;
    void SetValue(int value);
```

```
class Request {
public:
int floor;
...
};

class Elevator {
public:
int current_floor, target_floor;
int status;
...
};

class Building {
public:
Elevator *lifts[NLIFTS];
...
};
```

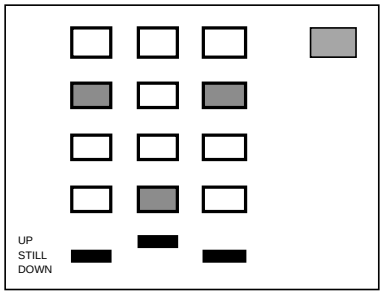
(a)

```
GLOBAL   window     Window 400 400 600 500
GLOBAL   x          Integer 100
GLOBAL   up         Text window 20 60 "Helvetica" 12 "black" "UP"
GLOBAL   still      Text window 20 40 "Helvetica" 12 "black" "STILL"
GLOBAL   down       Text window 20 20 "Helvetica" 12 "black" "DOWN"

TYPE     Elevator   Box window @x 100 60 50 "black" 2 "None"
TYPE     Elevator   Box window @x 200 60 50 "black" 2 "None"
TYPE     Elevator   Box window @x 300 60 50 "black" 2 "None"
TYPE     Elevator   Box window @x 400 60 50 "black" 2 "None"
TYPE     Elevator   Box window @x 60-status*20 60 12 "blue" 1 "blue"
TYPE     Elevator   Box window @x+3 3+current_floor*100 54 44 "red" 1 "red"
TYPE     Elevator   Assign x x+100

TYPE     Request    Box window 500 3+floor*100 54 44 "black" 1 "green"
```

(b)



(c)

Figure 7: *Usage of custom visualization techniques as a debugging tool. In (a) a partial listing of the elevator simulator is shown. A building consists of three elevators, and one request object to model pending requests. In (b) a visualization script is shown to visualize the running program. (c) shows one particular snapshot of the animation. A request is pending to have an elevator come to the fourth floor. The animation shows a clear flaw in the algorithm. Elevator two is coming, while elevator one and three are much closer by.*

```
    List(List *, int);
};
```

Observe the following script:

```
1.  GLOBAL Sort     Window 400 400 600 200
2.  LOCAL  List x   Integer 570
3.  TYPE   List     Box Sort x 50 9 value "black" 1 "gray80"
4.  ENSURE List     x < next.x-15
```

When this script is interpreted, Line 1 results in a window being created to show the animation. Line 2 adds an instance variable to the class definition of List. This is a symbolic addition, and the variable can be used later on in the visualization, as if it were an instance variable defined in the program. At line 3, a view is defined for `List` instances. This line defines that each instance of the type `List` should be represented by a `Box`. The position of the box is related to the actual value of the `x` field, which is defined at line 2. The width of the box is constant. Its height is a function of the `value` field, which is defined in the application (see the definition). At line 4, a constraint is defined, ensuring that each `Box` representing a `List` instance is always to the left of the instance referred to by the `next` field.

When this script is run in a custom visualization session, the result is as in Figure 8.

If the definition of the visualization is to be done visually, an approach should be found to define both the mappings and the constraints visually, using the mouse (as done in [Borning 86]). Doing the mapping visually has many advantages, such as ease of use. Also, all constraints are visible at all time, and can be understood by looking at the picture. However, in visual languages addressing components by *name* (a common mechanism in textual programming languages), is very difficult. In the design of HotWire, it was decided to use a textual format for the scripting language, for the time being. Also, exchanging textual scripts for visualizations of libraries, for instance, is much simpler.

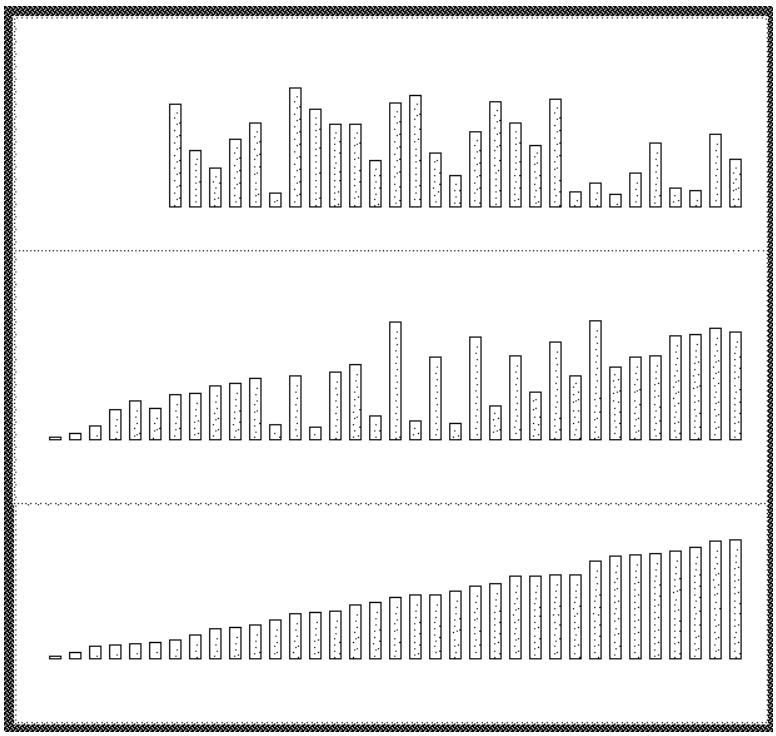


Figure 8: *A HotWire custom visualization of a sorting algorithm. The script given in the text of this paper will result in this animation of the algorithm. The top portion shows a snapshot during initialization of the datastructure. The middle part is taken during the execution of the algorithm, and the bottom view shows the state of the program, after it is finished.*

USING VISUAL TECHNIQUES FOR DEBUGGING

The foremost important result of using visual techniques for debugging is the enhanced understanding of the program. When one understands better how a program works, bugs or anomalies can be detected and identified much easier. Examples were given before how visualizations can address memory leaks, performance problems, incorrect execution threads, and flaws in the algorithm.

The visualization in the debugger can be static (created by the developers of the debugger), or dynamic (custom designed by the end-user of the debugger). In the first case, the debugger developer is striving to please a wide subset of programmers. Typically, a large number of different views of the executing program are supplied. The programmer will perhaps not use all of them, but specific problems may require different views highlighting particular aspects of the information that can be extracted from a running application.

An example is the approach taken it in the Procol runtime system [van den Bos and Laffra 89]. This started by showing the `MAIN` object (see Figure 9). Each Procol program contains one `MAIN` instance. Then, the mouse can be used to open up a browser for any instance shown on the screen, by clicking on it. When the browser is open, it can be dragged around the debugger window, and individual components defined in the instance, and shown by the browser, can be inspected into further detail, again by clicking on them. Each method that is activated on an instance, is animated by moving the cursor around (see the black rectangle in Figure 9). Also, at the end of a method, the values of the instance variables were checked, to see whether the display in the browser should be updated.

The inspection mechanism offered by this runtime system is useful for identifying a number of problems. However, the biggest drawback is that one first has to *find* the object that caused a specific problem. The instances shown in Figure 9 are the only ones that are shown, and most of the activity escapes our attention. In fact, we suffer from tunnel vision here.

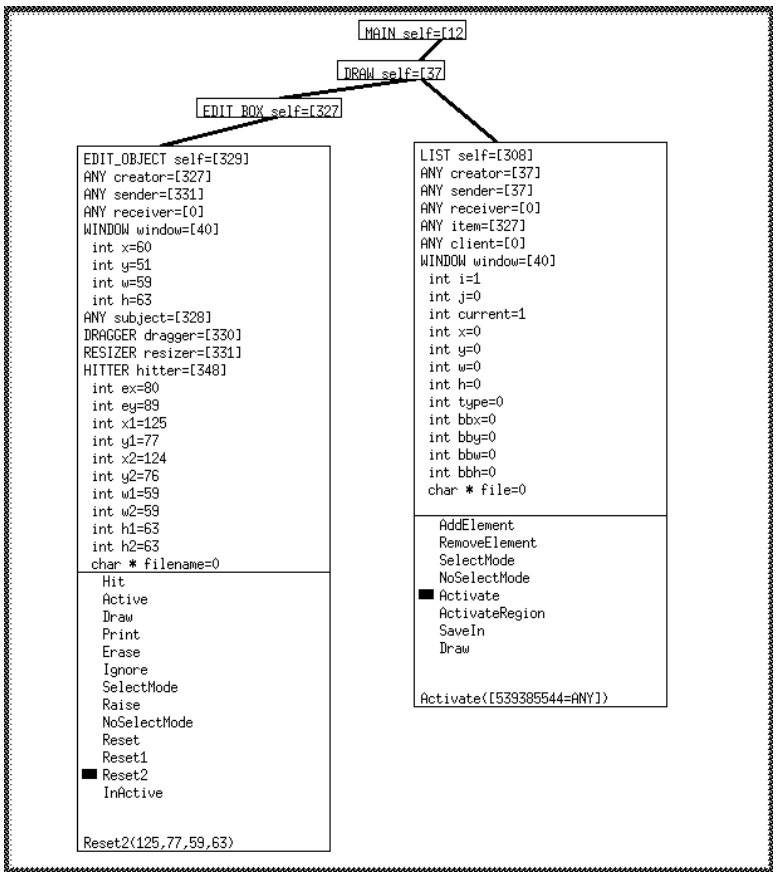


Figure 9: *This unprecedented, but still unsatisfactory view is produced by a previous attempt, the runtime debugger for the Procol language [van den Bos and Laffra 89]. It shows a couple of object browsers. Each of the browsers can be opened and closed. Shown are three closed browsers; for a* `MAIN` *object, a* `DRAW` *object, and an* `EDIT_BOX` *object. When a browser is open, individual elements can be clicked on, to create a browser for that specific instance too. Shown are the individual methods, and their activation is animated by moving the black rectangle to the currently active method. At the bottom, the name of the method last executed on this instance is listed, together with its arguments.*

Figure 10 shows a typical debugging session with HotWire. The program under study is *doc*, a word processor written in using the Interviews toolkit [Calder and Linton 91]. Figure 1 shows a snapshot of a HotWire debugging session of doc. First, a possible bug is observed. Namely, doc has a facility that one can create floating figures. After the figure has been created, it can be moved around on a given page. While doing that, our observation was that the figure moves fast in the bottom right corner, but moves slower and slower once we get closer to the top left corner. We experimented with different figures, and with different underlying documents. The effect remained the same. Our initial step towards the solution of this problem, was to look at the regular views that HotWire has to offer. We had HotWire show all objects and all method invocations. We tried the call stack view with the recording strip. But, none of the views really gave different results depending on the location of the figure. What we did find out was that a lot is happening already to draw a flashing cursor (the class InsertMarker in doc). Furthermore, even though there is always only one InsertMarker active in a doc session, a lot of others may also be invoked. Looking into more detail into the class InsertMarker (by clicking on the type name label in the instance window), we could see all methods that were invoked on all instances of type InsertMarker. Apparently, the sequence was that one InsertMarker is told to flash(), and as a result, it and a number of other InsertMarkers are told to draw() themselves. This observation leads to our first assumption: When the bug occurs, too much is drawn.

Reasoning that redrawing has all to go through one single object, the type Canvas, it led us to investigate the

damage() method. This method is the routine that is responsible for repairing damaged areas on the screen. Looking at the implementation of the method, we observed that an object of the type Damage is created for each repair operation. Sometimes, this object is saved from one operation to the next, to merge two areas that are overlapping into one new area. The Damage object will store the area that is to be repaired. It seems logical to test our assumption that too much is drawn, by inspecting the values of the instance variables of this Damage object. When we would be using an old-fashioned debugger, we would set a breakpoint in routine Damage::damage(), and inspect the values of the parameters. However, that really would not be practical, because we would stop too many times, as redrawing happens really often. Furthermore, the standard views of HotWire do not give any extra help here too. What we want to do next is to actually highlight the area that is redrawn inside doc. To do it in a general fashion directly in the doc canvas would be difficult to do, as it would make to many assumptions on the implementation of the application. Therefore, we will use a separate window of the same size of the canvas and visualize the area that is being redrawn in it.

In order to visualize the area that is damaged in the canvas, we define a small script. Basically, we tell HotWire to create a window, and to create a Box object (which is part of our scripting language) for each instance of the type Damage that is created during our debugging session. The size of the box is determined by the instance variables x, y, w, and h. These variables are C++ member fields, defined in the class Damage. Then, we run the application again, and observe what is happening. Now, we will see exactly which area is being redrawn. During initialization we ignore the flashing, but look more closely when we create a figure, and drag it around. When we start at the right bottom corner, the area that is repaired is roughly the same size as the figure itself. Occasionally, we move the mouse a little bit faster, and two areas are merged, so that the area that is repaired is somewhat larger. Which is ok. However, the more closely we move the mouse to the left top corner, the larger the damaged area gets! In fact, the are covers the entire window when we finally reach the top left corner. Experimentation learns us that the center of the canvas is the point of change. Looking at how a figure is implemented, we found out that it is using the class Patch, which is related to the Extension class. Visualizing that class in a similar way slowed down the visualization tremendously, as there happen to be a lot of extensions around, but in the end gave the same result. All this lead us to the conclusion that the patch allocation algorithm in the doc implementation has a serious problem.

As an aside, we learned a lot more from our visualization session of the damaged area. We found out that typing text in an empty window causes the entire line to be redrawn (until the right border of the page), even when there is no text after it. We did not consider that to be too serious, but still one might optimize that. What was far more serious, was the result of hitting the Enter key on the keyboard to go to the next line in our text. The result was a redraw of the entire page. Not one time, but exactly seven times. Visualizing the Insertmarker at the same time, learned us that the InsertMarker gets resized to be as high as the entire page, moves to the next line, and is then resized to the original line size again.

Summarizing, visualization can lead to interesting observations, and may highlight problems that one is not aware of, or that would be very hard to find using printf statements, tracing facilities, or existing debuggers.

IMPLEMENTATION

HotWire was initially developed on an RS/6000 running AIX with X Windows. Both the target and implementation languages were C++ and SmallTalk. The implementation is separated into two major parts. One part is the visualization code (approx. 4000 lines of C++). That includes all visualizations described in this paper, and also a parser and interpreter for the declarative scripting language. The other part is a graphical toolkit, with support for windows, structured graphics, display lists, double buffering, etc. (approx. 3000 lines of C++). The parser for C++ is developed using Lex and Yacc (approx. 1400 lines). An OS/2 port of the graphics toolkit is currently being worked on. No part of the functionality of HotWire needs to be changed to enable the porting. The reason for developing a new graphics toolkit, instead of using an available one, is the fact that efficiency was more important in our case than generality.

When a program is to be investigated, using HotWire, the C++ code is first annotated with a special pre-processor. This pre-processor reads in the code, instruments it at special locations, and feeds it into a standard

C++ compiler. The user does not have to change anything in the original source code. The instrumentation process is entirely automatic. The instrumented code makes calls to a runtime library, which is described before. It is essential that the visualization software runs in the same address space as the program, to reduce the overhead of the visualizations. HotWire behaves more like an intelligent runtime system, than as a debugger running in a separate process. The entire architecture is independent of the C++ compiler used. Furthermore, the interface to the graphics system is very small (one single class does all the graphics calls), all leading to a highly portable visualization tool.

CONCLUSIONS

C++ is being used for numerous program development projects for almost a decade already. Surprisingly, the state of the art in *debuggers* for C++ is still a few generations behind, and deserves a lot of attention. C++ debuggers use wrong metaphors, and especially the interface of most current debuggers is archaic, and needs to be replaced by one that uses more recent techniques. One promising approach is to use visualizations. They dramatically improve the quality of debuggers.

This paper describes **HotWire**, an experimental visual debugger for C++ and SmallTalk, developed at IBM T.J. Watson Research Center. It uses different types of visual techniques to enhance the understanding of the program under study. The main idea is to provide the user (the programmer debugging an OO system), with as many different views as required to help finding a bug. The visualizations are static (defined by the developers of HotWire), and custom built (defined by the end-user). The static visualizations emphasize on showing all classes and their instances, highlighting interactions on individual instances, showing method calls, recording all messages that happened in a particular time frame – and replaying and inspecting them afterwards, displaying the values of instance variables in a continuous manner, and showing relationships between instances. The custom visualizations are to be defined using a special visual scripting language. The script identifies the *models* – the objects of interest in the application –, the *views* – suitable shapes for displaying the objects –, and a *controller* – serving as a mapping between the two. The controller is declarative (an important aspect), and defines the views by referring to instance variables in the models. When the models get manipulated, the corresponding views will be updated by the visual debugger. In addition, triggers can be set on entering and leaving of methods, so that the visualization can be altered (animated), depending on activations of instances. Relationships between different views can be maintained using constraints.

There are problems that still need to be solved in the realm of visual debugging. One problem is how to display many classes and many instances. Appropriate abstraction mechanisms (as used in [Honda and Yonezawa 88]) and browsing techniques (like fisheye views [Sarkar and Brown 92]) need to be investigated to handle these types of complex (and realistic) systems. Furthermore, visualizations for libraries of source code could be developed. It is quite conceivable that toolkits of classes can be equipped with standard visualizations. For each part of the toolkit a special script could be defined, and be shipped with the toolkit or the debugger. The application of toolkits could be improved a lot, by educating programmers by showing visualizations. Finally, the structure of the script language is still being studied. The current version is the first generation. The aim is to keep the language declarative. Keeping the script declarative, reduces the complexity of the language and keeps the scripts small and concise, making the development of visualizations of C++ programs a simple and affordable task.

References

[Borning 86] Alan Borning, Defining Constraints Graphically, ACM–CHI'86 Conference Proceedings, April 1986, pp. 137-143.

[Brown 88] Marc Brown, Exploring algorithms using Balsa-II, IEEE Computer, 21, 5, May, 19988, pp. 14-36.

[Calder and Linton 91] Paul Calder and Mark Linton, Glyphs : Flyweight Objects for User Interfaces, UIST'91 Conference Proceedings, 1991.

[De Pauw 93] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides, Visualizing the Behavior of Object-Oriented Systems, OOPSLA '93 Proceedings, ACM SIGPLAN Notices, October 1993, pp. 326-337.

[Freeman-Benson 90] Bjorn Freeman-Benson, Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming, ECOOP–OOPSLA'90 Conference Proceedings, Ottawa 1990, pp. 77-88.

[Goldberg 83] Adele Goldberg, *Smalltalk-80, The Interactive Programming Environment*, Reading, Addison-Wesley, 1983.

[Honda and Yonezawa 88] Yasuaki Honda and Akinori Yonezawa, "Debugging Concurrent Systems Based on Object Groups", In Proceedings of *ECOOP '88*, pp. 267–286, August 1988.

[Johnson 93] Ralph Johnson (compiled the list), "Classic Smalltalk Bugs", stored at host `st.cs.uiuc.edu` in file `pub/st-docs/classic-bugs`, accessible through the internet, 1993.

[Leler 88] William Leler, Constraint Programming Languages – Their Specification and Generation, Addison Wesley, Reading Mass., 1988.

[Meyer 88] B. Meyer, *Eiffel – The Language*, Prentice Hall, London, 1992.

[Miya 93] Eugene N. Miya, *comp.graphics.visualization FAQ*, UseNet, 1993.

[Sanella 93] Michael Sanella, John Maloney, Bjorn Freeman-Benson, and Alan Borning, "Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm", Software – Practices and Experiences, 25, 5, pp. 529–566, May, 1993.

[Sarkar and Brown 92] Manojit Sarkar and Marc Brown, "Graphical Fisheye Views of Graphs", in Proceedings of *CHI '92*, pp. 83–91, May, 1992.

[Smith 87] Randall Smith, Experiences with the Alternate Reality Kit – An Example of the Tension Between Literalism and Magic. CHI+GI'87 Conference Proceedings, April 1987, pp. 61-68.

[Stasko 90] John Stasko, TANGO: A Framework and System for algorithm animation, IEEE Computer, 23, 9, September 1990, pp. 27-39.

[van den Bos and Laffra 89] J. van den Bos and C. Laffra, "PROCOL - A parallel object language with protocols", In *ACM–OOPSLA'89 Conference Proceedings*, New Orleans, Special Issue *SigPlan Notices*, 24, 10, pp. 95–102, October 1989.

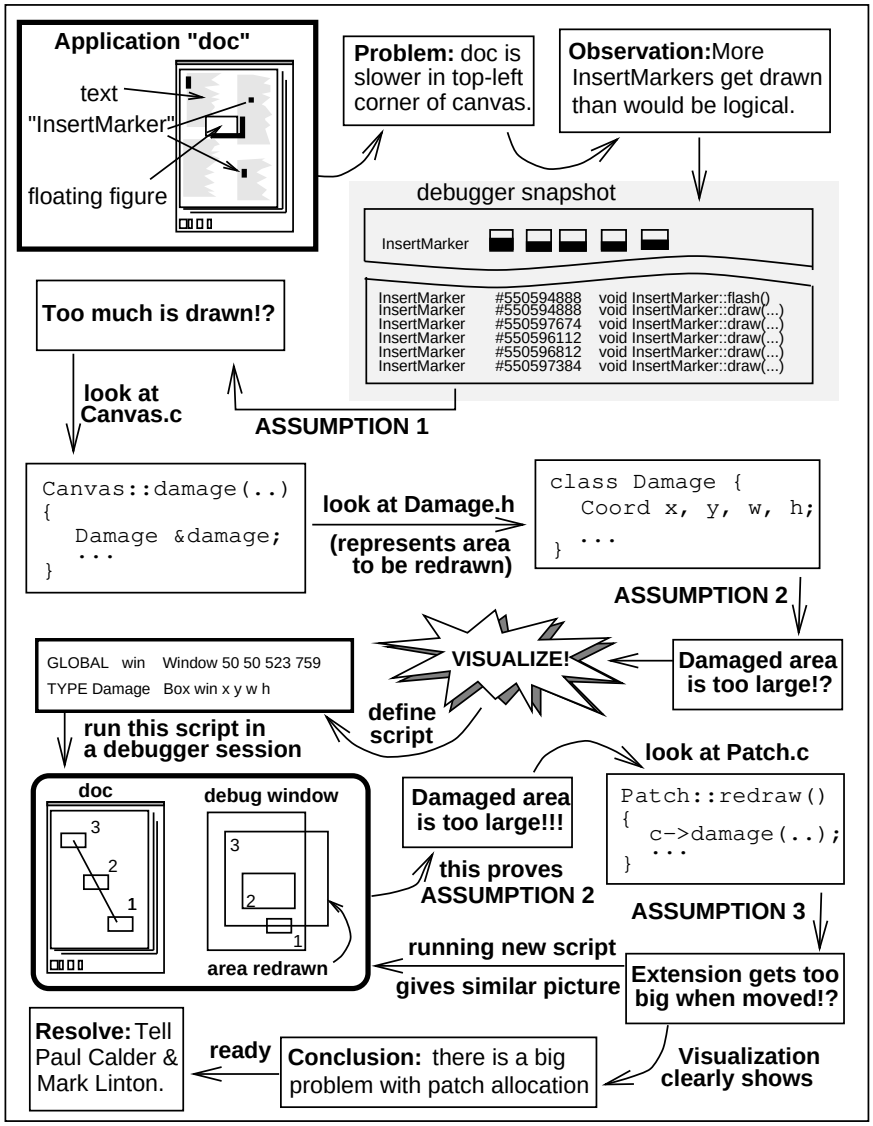


Figure 10: *A (busy) representation of the scenario for a typical debugging session with HotWire. Regular views are used to find anomalies, and customized visualizations are used to test and prove assumptions.*