Debugging Interactive Applications

Brad A. Myers *Alan Ferrency* *Rich McDaniel* *Roger Dannenberg*
Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{bam, af1x, richm, rbd}@cs.cmu.edu
http://www.cs.cmu.edu/~amulet

ABSTRACT

Although interactive, direct manipulation applications are known to be difficult to design and implement, the toolkits with which they are built generally do not contain any particular support for debugging. The Amulet toolkit contains a comprehensive collection of monitoring and debugging tools, including an interactive ''Inspector.'' These tools are provided in a machine-independent way in C++ without using hooks into the compiler, symbol tables or the run-time stack. Some of these capabilities are based on well-known techniques, but others are innovations that have never been provided before. Based on our experience with writing and debugging interactive applications, we have provided tools to address the most common and difficult programming bugs. The capabilities include: viewing values of objects as they change; breaking into the debugger when values change; viewing the inheritance and grouping hierarchies of objects; feedback for why objects are not visible or not interactive; tracing of constraint dependencies; and various techniques to search for objects. In addition, programmers can edit the values displayed in the Inspector, supporting rapid prototyping without requiring a C++ interpreter. These features make debugging interactive applications written using Amulet is substantially easier than with other toolkits.

KEYWORDS: Debugging, Software Development, Programming, Toolkits, Amulet, User Interface Development Environment, Software Engineering.

INTRODUCTION

The Amulet user interface development environment aims to make it significantly easier to design and implement highly-interactive, direct manipulation user interfaces for Windows or Unix in C++. This is accomplished primarily, by providing high-level models such as Interactors [2], command objects [5], constraints [10], and a structured graphics output model. Although programmers can write their code using Amulet's high-level abstractions for these models, they previously had to monitor and debug their code using the native, low-level C++ debuggers. Through long experience with our previous Garnet system and other toolkits, we learned the most common coding and debugging problems for interactive software. Based on this experience, we designed Amulet to support a number of high-level, run-time debugging and monitoring techniques. One is an interactive *Inspector* which provides access to a large number of novel debugging and monitoring techniques. Another is support for displaying the names of objects at run time.

The primary innovation in Amulet's debugging features is the support for debugging the *dynamic behavior* of objects. A number of previous systems have provided static object inspection techniques, but Amulet expands these to also show traces of the changes of values through time, the execution of Interactors and widgets, and the re-evaluations of constraints. Furthermore, Amulet provides these features, along with others conventionally found only in interpreted environments like Lisp and Smalltalk, in a fully-compiled, portable, efficient, C++ implementation designed for building large-scale applications. This paper discusses Amulet's novel debugging features, and how we were able to provide these in C++ without using an interpreter, preprocessor, or access to the compiler's symbol tables.

It is ironic that today's end-user applications generally follow good user interface principles, but the environments and toolkits used to program those applications generally do *not* follow those principles. As a result, the toolkits are difficult to learn and inefficient to use. We applied a number of user interface principles [1] when designing Amulet:

- **Help the user get started with the system**: The high-level abstractions in Amulet mean that simple programs are short. For example, the program that displays ''hello world'' in a window is 8 lines.
- **Be consistent**: The novel prototype-instance object model in Amulet, where objects are collections of slots

(instance variables), some of which are inherited, means that all properties of graphical objects, widgets, Interactors, command objects, etc. can be accessed and set the same way: by setting and reading the slots of objects. Higher-level abstractions like the command model, the Interactors and the debugging tools, treat all objects the same way, whether they are complex composite widgets or primitives like rectangles.

- **Use user-centered words in messages**: Unlike other systems that display pointer variables when debugging, Amulet strives to always display high-level names for methods, constraints, and even instances of C++ classes.
- **Prevent user errors**: Amulet tries to eliminate the need to deal with pointers and memory allocation, which plague programmers using other C++ environments. Amulet performs full compile-time or run-time type-checking, so errors are caught quickly.
- **Offer informative feedback**: Unlike other C++ environments where programmers are confronted with uninformative messages like ''Segmentation violation,'' ''Bus error,'' and ''Illegal instruction,'' Amulet provides helpful error messages that speak about the problem at the user's level, and often recommend how to fix the problem. This is unique among toolkits. Furthermore, we have added a number of checks and warning messages for common programmer errors that do not cause crashes.
- **No hidden state, and use direct manipulation**: The interactive Inspector makes the complete state of the objects visible. Many other views are provided so the programmer can more easily visualize the relationships of the objects and their dynamic behavior. There are views showing the current and old values of slots, the properties of slots, the constraints and their dependencies, methods, and the inheritance and aggregate hierarchies. There are also commands to ask where objects are or why they are not visible, what will happen when the mouse is pressed, and why objects are not interactive. The programmer can turn on various kinds of tracing and breakpoints for slot setting, constraint solving, method invocation, and Interactor execution. The Inspector allows values to be edited directly, which supports rapid prototyping, without requiring a C++ interpreter. This means that the Inspector could also be used by interactive builders, like an Interface Builder, to support the creation of objects.

Since Amulet is designed to be a portable toolkit (and currently runs on two platforms using at least 5 different compilers), we were careful to ensure that the techniques do not require compiler-specific hooks or access to symbol tables or the run-time stack. Although some of the techniques are specific to Amulet's models, most would be useful for programmers using any other user interface toolkit, such as Motif or Microsoft Windows.

RELATED WORK

Debugging tools in general have a very long history. The three basic forms of debugging, *trace*, *dump*, and *break* date back to the EDSAC computer of the 1940's [9]. More elaborate facilities have appeared with some systems, but today's popular C++ environments, including Visual C++, gdb, and ObjectCenter, still mainly provide only these techniques for accessing the dynamic state of the system.

An interactive Inspector was part of the early InterLisp-D environment [11], and could display and edit the values of Lisp structures, and Smalltalk has similar tools for its objects. Visual Basic and SUIT [7] provide property sheets for objects. However, Inspectors are rarely available for compiled languages like C++, since the required information is normally not available at run time. Furthermore, none of these tools support inspecting the dynamic behavior of objects.

Debugging tools for constraints have appeared in a few research systems, such as CNV [8], but these are mostly graphs of the dependencies and have not been applied to toolkits in widespread use or to debugging interactive applications.

Since most of the toolkits which provide high-level models are research systems which are never distributed for real use, it is perhaps not surprising that there are few corresponding high-level debugging tools. None of today's main-stream toolkits (e.g., Motif, MS Windows, Macintosh Toolbox), provide specialized debugging facilities. Our Garnet system [3], a predecessor of Amulet written in Lisp, explored some of the facilities presented here, but they were never reported, and Amulet provides many additional techniques, as well as exploring how to provide these in a compiled environment.

AMULET'S OBJECTS AND VALUES

Amulet uses a *prototype-instance* object system implemented in C++, unlike the class-instance system used by Smalltalk and C++. Instance variables of objects, called slots, can be dynamically added and removed from any object. Any slots that are not overridden by an instance are *inherited* from the prototype. Methods in Amulet are treated the same as data values in slots, and individual instances can have different methods. Amulet allows any slot of an object to contain a *constraint* which calculates the value. Methods and constraints can be composed of arbitrary C++ code [6].

In Amulet, the type of value in a slot can change at run-time. For example, the `LINE_STYLE` slot might contain a color object at one time, and later an integer, or a list. To support this, Amulet keeps a run-time type-field with each slot, which is checked by appropriate operations.

Often, programmers can use Amulet objects for all of their operations. If it is necessary to store C++ objects into Amulet slots, this can be accomplished with complete type-safety through the use of a ''wrapper'' mechanism. The default methods of wrappers provide a run-time type identifier, automatic memory allocation, and naming, and hide uses of pointers. For example, the `Am_Object` type is actually a C++ wrapper class that contains only a pointer to the internal Amulet object data structure. By wrapping the pointer in a C++ class, we can use the C++ constructors,

destructors and operator overloading to provide memory management and checking. Creating and referencing an Amulet object looks like:

```
Am_Object my_obj = Am_Rectangle.Create();
my_obj.Set(LEFT, 20);
Am_Object my_window = Am_Window.Create();
my_window.Add_Part(my_obj);
```

VIEWING OBJECTS

The main interface to debugging in Amulet is the Inspector window, shown in Figure 1, which is viewing all the slots of a rectangle called `colorrect`. The top few lines show that `colorrect` is an instance of `<Am_Rectangle>`, which is the top-level rectangle prototype in the Amulet library. `colorrect` is also a part of the window named `<window>`. The `LEFT` slot of `colorrect` contains 300, and the `LINE_STYLE` slot contains `Am_Line_1`. Inherited slots are shown in blue, so the programmer can easily see that `colorrect` is inheriting the value of `VISIBLE` from its prototype (or recursively, from its prototype's prototype, and so on. The slot properties pop-up window described below can be used to see where the slot is getting its value from).

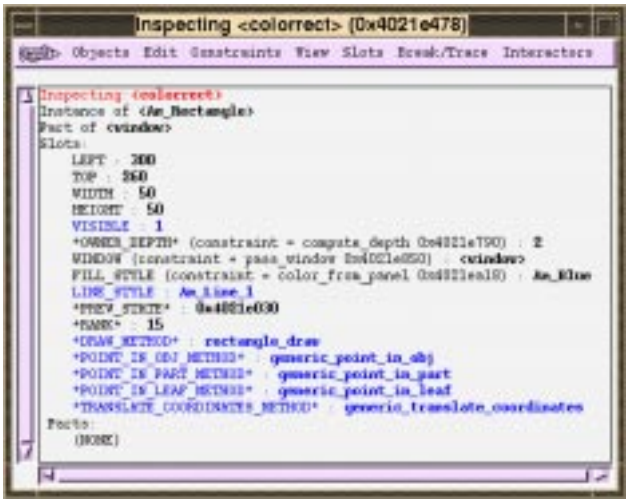

Figure 1:
The main Inspector window viewing the object `colorrect`.

Meaningful names are used for the display of most of the types and values. The `FILL_STYLE` slot contains a constraint called `color_from_panel` which currently has calculated the value `Am_Blue`. The `*DRAW METHOD*` slot contains the method `rectangle_draw` which is inherited. Amulet uses the convention that ''internal'' slots of objects, that are not generally supposed to be set by programmers, use a "*" in their print names.

INSPECTING OBJECTS

The easiest way to inspect an object is to put the mouse cursor over the object and hit the `F1` key.[1] If the user selects an object's name in the Inspector window, then that object can be inspected, either in the same window or in a new Inspector window. Another menu command allows the programmer to type the name of the object to be inspected.

A very common bug in interactive software is that a graphical object does not appear on the screen as expected. Amulet's Inspector provides a unique command that flashes an object or tries to determine why the object might be invisible. If the object is contained in some window, that window is brought to the front and a blinking rectangle is placed over the object. If the object is covered by another object, the flashing will show where it is hiding. However, if the object is outside of the window, the flash command will instead print a message explaining the problem. Other common reasons the object might not be seen are that the object was never added to any window, or that the object's `VISIBLE` slot, or the `VISIBLE` slot of its container, is set to `false`. In all, the flash command checks and reports on about 15 different reasons the object might not be visible.

A common debugging problem is finding the desired object if it is not on the screen. One way to find objects is using a hierarchy browser. All objects in Amulet participate in two hierarchies: the prototype-instance inheritance hierarchy, which corresponds to ''is-a,'' and a group-owner hierarchy, which corresponds to ''contains.'' Different views in the Inspector will display the complete inheritance and containment hierarchies, starting from a specified root object. For example, this can be used to see the hierarchy of all objects contained in a particular window, or all the instances of a particular prototype.

Since there may be thousands of objects, the it may be difficult to find objects by browsing. Therefore, a sophisticated search dialog box will allow the programmer to search for all objects that have particular values in their slots. The search can either start with all objects inside a particular window (or the screen) and thereby go through the containment (aggregate) hierarchy, or start with objects which are instances of a particular prototype. For example, the programmer could ask to see all objects in a window with a `LEFT` of 10, or all instances of `Am_Rectangle` which have a `FILL_STYLE` of `Am_Blue`.

NAMES

A problem with most toolkits and other high-level packages is that the debugging information is reported in terms of C++ low-level pointers. In Amulet we wanted instead to use high-level names that would be meaningful to the programmer. Therefore, we provide a number of mechanisms that associate the names with objects and values and make them available at run-time.

Slots

Amulet knows how to print the primitive types (ints, floats, strings, etc.) and most higher-level types use the wrapper mechanism, which saves meaningful names. In Figure 1, `Am_Line_1` and `Am_Blue` are wrapper values. The slot `*PREV_STATE*` is being used in a non-type safe manner,

[1]The particular keyboard key used can be easily changed.

and contains a `void*` pointer, which the lower levels of Amulet cast into a pointer of a specific type. Unlike other systems, Amulet's programming interface does not expose any of these kinds of pointers (and they are rarely used internally either). Instead, data is stored using wrappers so high-level names are available for inspection and operations are type-checked for robustness. An Inspector command will hide all of the internal slots so the programmer does not see these at debug time either.

To retain names for run-time, most object and wrapper creation functions take an optional parameter which is the name of the object. The wrapper or object stores the name in a global hash table for use when displaying the object. For editing, the user can also type the name, so, for example, `colorrect` can be changed to green by typing `Am_Green` into the `FILL_STYLE` slot.[2]

Wrappers can have custom print methods, which are used when no name is associated with them. For example, the `Am_Value_List` wrapper prints the elements of the list. Colors which do not have names will print their red, green and blue values. User-defined wrappers can also provide custom print methods.

Constraints

Providing names for methods and constraints is more tricky, especially since we wanted to provide a nice interface for the programmer without requiring a pre-processor. Since we do not want to use the compiler's symbol tables, we provide special macros that save the string names used for formulas and methods. Eventually, we plan to provide a sophisticated pre-processor, but for now, some C++ macros suffice.

To encapsulate a function as a formula or method, a macro is used which expands into a special header. For formulas, this looks like:

```
Am_Define_Style_Formula(color_from_panel)
{
  Am_Style new_color;
  //regular C++ code goes here to compute new_color
  return new_color;
}
```

The `Am_Define_Style_Formula` macro expands to code which defines an `Am_Constraint` global variable called `color_from_panel`. The constraint object is stored in this variable, and will contain the string `"color_from_panel"` used for debugging, along with a pointer to the function `color_from_panel_proc` containing the code. All formula functions must have the same standard parameters, and the return type depends on the type of the value to be calculated; here it is a Style (color). There are equivalent `Am_Define_`*xxx*`_Formula` macros for the other return types.

[2]Typing a new value into the Inspector will override the value computed by the constraint.

The `color_from_panel` constraint object can be stored into a slot using the standard `Set` method, just like other values:

```
colorrect.Set(LEFT, 300);
colorrect.Set(FILL_STYLE, color_from_panel);
```

Methods

Unlike formulas, which have a fixed signature for parameters and returns, methods in Amulet objects can have arbitrary parameters and return types. However, we still want to provide complete type-safety and flexibility, and avoid the ''Segmentation Faults'' common in other toolkits where the programmer coerces `(void*)` pointers into pointers to functions and then calls them, or where all methods have the same signature and the extra data is passed as a `(void*)` pointer. Instead, Amulet requires that all method *types* be pre-declared. Then, actual methods are defined using that type. Each type of method can have a different set of parameters and return values. Amulet checks at run-time to ensure that the correct type of method is being called. For example, the (internal) draw method type is defined to return nothing `(void)` and take an object to be drawn `(self)`, an internal `drawonable`, and two integers:

```
Am_Define_Method_Type(Am_Draw_Method,
 void, (Am_Object self,
        Am_Drawonable* drawonable,
        int x_offset, int y_offset));
```

Due to the limitations of the C++ macro facility, the return value of the method has to be passed as a parameter to the macro, and the parameters to the method function have to be in an extra set of parentheses.

The specific draw methods are then defined as:

```
//Define rectangle_draw as an Am_Draw_Method type method
Am_Define_Method(Am_Draw_Method, void,
 rectangle_draw, (Am_Object self,
    Am_Drawonable* drawonable,
    int x_offset, int y_offset))
{
  //regular C++ code goes here to draw a rectangle
}
```

Similarly to the constraint define macro, the `Am_Define_Method` macro allocates a global method object called `rectangle_draw` and fills it with the appropriate type information, the string `"rectangle_draw"`, and a pointer to the code. This method object can then be set into a slot in the standard way:

```
colorrect.Set(DRAW_METHOD, rectangle_draw);
```

We require the parameters and return value to be repeated in the `Am_Define_Method` call for a number of reasons. First, the method *type* declaration is usually far away (often in a `.h` header file), and the code is more readable when the names of the parameters are next to the code. Second, this is consistent with C++ function declarations in header files and definitions in code files. Third, C++ allows the actual names of the parameters to differ in the type declaration and the actual definition, so sometimes more meaningful parameter names can be used in the specific method definition. Finally, some C++ compilers give a warning if a parameter is not used, and the only way for the program-

mer to declare that a parameter is being ignored on purpose is to omit the name and leave the type, which requires repeating the parameters.

To call a method, the programmer uses the type defined with the `Am_Define_Method_Type` macro. `Call` is a C++ method defined by the type definition macro.[3]

```
Am_Draw_Method my_method =
            colorrect.Get(DRAW_METHOD);
my_method.Call(colorrect, draw1, 20, 30);
```

Amulet provides complete type safety on this call, by checking at run time that the `DRAW_METHOD` slot contains a method of type `Am_Draw_Method`. This is implemented using the type conversion mechanisms of C++.

Through the use of macros and clever C++ type conversion tricks, Amulet allows arbitrary C++ code in methods and formulas, along with arbitrary signatures for methods, while still providing names for debugging at run-time, full type-checking, and a reasonable syntax for the programmer. The same macros are used internally and in the programmer's code, so that meaningful names and error checking are available at every level. Furthermore, C++'s ''Illegal instruction'' and ''Bus Error'' messages are replaced with informative messages like ''Tried to assign a Am_Draw_Method with a method wrapper of type Am_Where_Method,'' or ''Invalid Method (with procedure ptr = 0) called.''

Many of these features would not be needed in a language like Smalltalk or Lisp. Amulet demonstrates how better type checking and meaningful error messages can be obtained regardless of the implementation language. Furthermore, most of the checking and data storage can be eliminated in Amulet using compile-time switches when greater efficiency is required, whereas in other systems data such as the names of atoms must always be retained at run time.

SLOT DISPLAY IN THE INSPECTOR

By default, the Inspector displays the slots in an order determined by when the slots were first set in the prototype, which usually puts the interesting slots at the top. Some objects have many slots, however, and it can be difficult at times to find a desired slot. Therefore, we provide various techniques to prune and sort the list of slots to be displayed. Commands will eliminate the display of the inherited slots and the internal slots, and sort the slots alphabetically. A novel feature is that a command will start a history of slot setting and subsequently display slots with the most recently set slots at the top. Finally, a command prompts for a particular slot name and displays that slot at the top.

In addition to their values, slots also have a number of properties which can be displayed in a pop-up window (see Figure 2). This window can be moved from slot to slot by clicking on the slot names. Among the information presented is the type of the current value in the slot, whether the slot value is currently inherited and if so, which prototype object holds the current slot value. Other lines show what types are declared to be legal for a slot (some slots may accept various types, like an integer *or* a float), and what the declared inheritance of the slot is (the programmer can control the inheritance of each slot by declaring that its value can be inherited dynamically, is copied to instances at object creation time, or must be declared locally). `Single Constraint Mode` determines whether the slot is allowed to contain multiple constraints at the same time, and the `Demon bits` show if any ''demon procedures'' will be executed when the slot is changed.

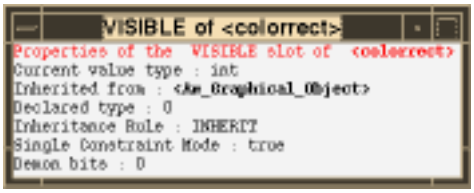

Figure 2:
A pop-up window that displays the properties of a slot.

EDITING

In addition to displaying the values of slots, the Amulet Inspector allows the values to be edited. The user can click on a value and edit it in place. The usual cut, copy, paste and undo will work for the editing of the values. This is extremely useful for rapid prototyping and testing of interactive software. For example, if an object is not quite in the right place, the programmer can simply edit the values in the `LEFT` and `TOP` slots in the Inspector, and the object will move. When the final desired values are achieved, the programmer can cut and paste them into the code. Named values can also be typed into the Inspector, including method names, constraint names, and wrapper names like `Am_Blue` and `my_bold_font`. (Because we are not using an interpreter, the names must have already been defined in C++ code.) The properties of slots can also be changed by editing in the slot properties window (Figure 2). New slots can easily be added to objects by typing the new slot name and value. Slots with local values can be deleted, but this may just cause the slot to change from local to inherited if a prototype has a value for the same slot.

TRACING AND BREAKPOINTS

One common debugging task is determining why a slot is set with the incorrect value. Therefore, we have provided high-level mechanisms to address this problem. The user can select a slot name in the Inspector window, and then choose a menu item to initiate tracing or breaking into the debugger when the slot's value is set. Breaks and traces can be set on multiple slots at the same time. There are various levels at which information can be provided. The simplest is that a history of all the values of a slot can be shown in the Inspector window, as in Figure 3. More complete information can be provided by another option which

[3]Note that this code is just for illustration; actual Amulet programs never call the draw method--it is only used internally.

prints the old and new values and the reason the value changed. The reasons include creation or deletion of the slot, direct setting of the slot, formula re-evaluation, or because the value changed in the prototype and this object is inheriting the value. If this is not sufficient information, a final option causes a break into the debugger whenever the slot's value changes. In the debugger, the programmer can then inspect the stack trace to see what procedure is causing the slot to be set.

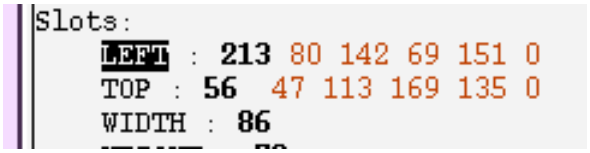

Figure 3:
The LEFT and TOP slots display the old values.

Sometimes code breaks because slots are set to a specific illegal value, such as NULL (0). To make this easier to debug, an option allows the break or trace to only happen if the slot is set to a specific value. Another option, which is useful for performance monitoring, will simply count the number of times that each slot is set and each formula is reevaluated, to check for unnecessary work.

CONSTRAINTS

As mentioned above, constraints in Amulet can be arbitrary C++ code, so the normal C++ debugging tools can often be used to help debug the constraint expressions. For example, a common technique is to put a breakpoint in the constraint formula. However, constraints present a number of new debugging challenges to the programmer, so special features have been added to the Inspector.

The slot tracing and breaking mechanism described above can be used to trace or break every time a constraint is re-evaluated. A history of the old values (Figure 3) can be displayed for slots which contain constraints in the same way as for all other slots. Whenever the constraint is evaluated, the display will be updated.

Constraints in Amulet can contain *indirect references* [10], which means that the object to which the constraint refers can be calculated by the constraint. Typically, the referent objects are retrieved from slots of other objects, but they may also be members of a list. For example, the constraint which computes the width of all of the components of a group looks like:

```
//define a formula named Am_Width_Of_Parts that returns an int.
Am_Define_Formula(int, Am_Width_Of_Parts) {
 int max_x = 0, comp_right;
 Am_Value_List components; //the list of components
 Am_Object comp;
 //Get the components out of my slot GRAPHICAL_PARTS.
 //This also sets up a dependency on the slot.
 components = self.GV(GRAPHICAL_PARTS);
 //Loop through all components.
 for(components.Start(); !components.Last();
          components.Next()) {
  //Get the component out of the list.
  comp = components.Get ();
  //  Compute how much of the component extends right of the
  //    origin.  This establishes dependencies on the left and
  //    width slots of the component.
  comp_right = comp.GV(LEFT)+comp.GV(WIDTH);
  max_x = MAX (max_x, comp_right);
 }
 return max_x;
}
```

This formula sets up dependencies on the GRAPHICAL_PARTS slot containing the list, and dependencies on the LEFT and WIDTH slots of *every* object in the list. Thus, this formula is re-evaluated whenever any of the objects move, or when the list of objects changes. Sometimes, bugs in constraint code arise from calculating the dependent objects incorrectly, rather than from the computation done *with* those objects. Therefore, the Inspector provides a facility to display the current dependencies of the constraint. Whereas CNV [8] used a node and arc graph to display the dependencies, our experience is that this kind of display breaks down for realistic applications because there are too many constraints and the names on the nodes are too long. Instead, we use a hierarchy display with elision, as shown in Figure 4. Clicking on the ellipsis will show more levels. Any of the objects can be selected for inspecting, flashing or any of the other Inspector operations. To get more explicit notification when the dependencies of a constraint change, the programmer can turn on tracing or breaking into the debugger whenever the dependencies of a constraint change.

Figure 4:
A pop-up window that displays the dependencies of the selected constraint.

Another command in the menus will tell which constraints *use* the value of a particular slot. The user selects a slot and can find out what constraints are currently using that value in their computation. This is helpful for ensuring that the value is used in appropriate places.

INTERACTORS AND COMMANDS

Amulet supports a novel, high-level model for handling input, based on *Interactor* objects [2]. Each type of Interactor implements a particular kind of behavior which is independent of the graphics to which the behavior is attached. For example, the `Choice_Interactor` can be used any time the user can choose among a set of objects using the mouse. An instance of the `Choice_Interactor` is used inside menu and button widgets, as well as to allow selection of graphical objects in a drawing editor. Programming with Interactors is quite different than input handling in conventional toolkits [4]. The programmer attaches Interactor objects to graphical objects and then the Interactors later manipulate the graphics in response to user input.

Our experience is that a typical bug with Interactors is that they do not operate when expected: the programmer sets up the Interactors and clicks the mouse and nothing happens. To make this kind of problem easy to debug, the Inspector provides a number of tracing mechanisms for Interactors. The programmer can specify certain Interactors to watch and then give the input that is expected to start the Interactors, such as clicking on a graphical object. The Inspector will then report why the Interactors are not running. If one or more of the Interactors *do* run, the Inspector will report exactly what graphical objects are modified.

Various modes of tracing are available. The programmer can trace just the setting of slots of objects by any Interactors, which is useful to see what the Interactors are doing without searching through long descriptions of other activities. Another option will just print a single line summarizing the operation of each Interactor that runs, which is useful for getting an overview of what is happening. Finally, full tracing can be turned on, which prints a lot of information about each Interactor.

For example, if the programmer is wondering what slots of which objects the Interactor named `grow_inter_in_handle_185` is setting, tracing on that Interactor can be started, and the following information will be displayed. The setting routines are highlighted with a ++:

```
<><><><><> LEFT_DOWN x=180 y=289 time=3114329169

Enter GO for <grow_inter_in_handle_185>,
  state=0...
Checking start event against wanted = LEFT_DOWN
  * SUCCESS
Checking start where..
  ~~SUCCESS=<Am_Rectangle_650>
Move_Grow starting over <Am_Rectangle_650>
  translated coordinates 169,268
Calculated attach point for non-line is
  Am_ATTACH_S
++Object <grow_inter_in_handle_185> setting
  Am_VISIBLE of <Sel_Rect_Feedback_197> to true
++Object <grow_inter_in_handle_185> setting
  obj=<Sel_Rect_Feedback_197> setting
  obj=<Sel_Rect_Feedback_197> LEFT=90 TOP=142
  WIDTH=182 HEIGHT=148
```

GENERAL DEBUGGING FEATURES

In addition to the specific features mentioned above, Amulet tries to help programmers in many smaller ways. For example, all of the error messages contain detailed explanations of the problem and often the probable cause. In some cases, we have been able to identify common mistakes that do not cause crashes, and there are specific checks and warnings in the system.

For example, the move-grow and create interactors allow the programmer to specify a feedback object which rubberbands as the mouse is dragged. Our experience shows that a common error is forgetting to add this object to a window so it does not show up on the display. Therefore, there is a special check in the code for this case which prints out a warning.

It was mentioned above that references to procedures and methods do not use pointers in Amulet. In fact, the typical Amulet programmer uses *no* pointers, which is quite unique for C or C++ code. The result is better error messages for uninitialized, null, or illegal pointers, dangling pointer errors are entirely eliminated (where a pointer refers to memory that was deallocated), and most memory allocation is handled automatically by reference counting. An early design for Amulet (that was never released) did not use this technique, and objects were referenced by pointers. We found that the code was difficult to debug, full of pointer errors and memory leaks, and when it crashed, it was difficult to tell why. The current design has significantly improved the robustness, readability and learnability of the system.

Another way pointers are eliminated is by ubiquitously using a general list mechanism instead of arrays or requiring the programmer to write their own lists. The `Am_Value_List` can hold an arbitrary number of dynamically typed objects. Iterating through a list uses a standard set of messages illustrated above in the constraint section. Note that again no pointers are exposed to the programmers (although they are obviously used internally). We find that most lists contain objects of various types, so arrays or even the C++ template mechanism are not adequate. For example, the list of labels for the items of a menu will contain strings and bitmaps.

STATUS AND FUTURE WORK

The Garnet toolkit incorporates an early version of the Inspector, and is available for general use.[4] A more sophisticated version of the Inspector is included in the version 1.0 release of Amulet [6].[5] The features described in this paper will be released with the next version of Amulet, which is expected in the late fall.

We are always looking for ways to make Amulet applications easier to debug. Eventually, we hope to incorporate an interpreter into the Inspector, so expressions can be

[4]Garnet is a Lisp toolkit available for free by anonymous FTP. See `http://www.cs.cmu.edu/~garnet` for more information.

[5]Amulet is available for free by anonymous FTP. See `http://www.cs.cmu.edu/~amulet` for more information.

typed for the values. This would also allow new constraints and methods to be entered at run-time. The next step is to use the Inspector as the property sheet display for interactive builder programs. For example, an Interface Builder would allow the user to place widgets and graphical objects into windows, and then to set other properties, and possibly enter the constraint and method code, using the Inspector windows. The author of the Interface Builder will not have to write the slot properties subsystem, but the users will have complete control over the many options for objects. Interface Builders will need a facility to save the designed objects to the disk, and this might be used separately by the Inspector so that the programmer will not have to cut and paste edited values to the code.

One goal of the Amulet project is to produce an easy-to-learn toolkit for use in college classes, in the style of SUIT [7]. For this version, easy debugging and error prevention will be critical. We plan to perform significant testing and iterative design of Amulet to ensure that the Inspector and the other features do, in fact, make it easy to learn how to create correct code.

Another area for future work is to investigate the speed and space tradeoffs for all the debugging and type checking information maintained at run time. A compile-time switch is already available to eliminate some of the run-time checking and all of the run-time storage of the names for slots, methods, constraints, etc. This can be used when the application is ready to be shipped. We have not yet investigated other optimizations because Amulet currently runs acceptably on our target platforms, such as a 486 or Sun SPARC 2. Even with the full debugging and checking facilities left on, Amulet runs significantly faster than Garnet, which is in Lisp, even though they use similar algorithms.

CONCLUSIONS

The Inspector was added quite late to the Garnet system, but it was so popular that we put it into the first version of Amulet. In fact, the Inspector was specifically cited by some of the early users of Amulet as a key unique feature. Kasem Abotel of the Univ. of Michigan said ‘‘I loved the Inspector’s power; it is very easy to change the slots of any object.’’ The new capabilities described here for monitoring the dynamic behavior of objects will make the debugging tools even more useful and thereby contribute to our goal of significantly improving programmer productivity when creating user interface software using Amulet.

ACKNOWLEDGEMENTS

Andy Mickish and Alex Klimovitski helped us with the design and implementation of Amulet.

This research is sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

1. Deborah Hix and H. Rex Hartson. *Developing User Interfaces; Ensuring Usability Through Product & Process.* John Wiley & Sons, Inc., New York, 1993.

2. Brad A. Myers. "A New Model for Handling Input". *ACM Transactions on Information Systems 8*, 3 (July 1990), 289-320.

3. Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces". *IEEE Computer 23*, 11 (Nov. 1990), 71-85.

4. Brad A. Myers, Dario Giuse, and Brad Vander Zanden. "Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods". *Sigplan Notices 27*, 10 (Oct. 1992), 184-200. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA’92.

5. Brad A. Myers and David Kosbie. Reusable Hierarchical Command Objects. Submitted for Publication.

6. Brad A. Myers, Rich McDaniel, Alan Ferrency, Andy Mickish, Alex Klimovitski, and Amy McGovern. The Amulet Reference Manuals. Tech. Rept. CMU-CS-95-166, Carnegie Mellon University Computer Science Department, June, 1995. also Human Computer Interaction Institute CMU-HCII-95-102. WWW = http://www.cs.cmu.edu/~amulet.

7. Randy Pausch, Matthew Conway, and Robert DeLine. "Lesson Learned from SUIT, the Simple User Interface Toolkit". *ACM Transactions on Information Systems 10*, 4 (Oct. 1992), 320-344.

8. Michael Sannella. *Constraint Satisfaction and Debugging for Interactive User Interfaces*. Ph.D. Th., Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, Sept. 1994. TR 94-09-10.

9. Edwin H. Satterthwaite. *Source Language Debugging Tools*. Ph.D. Th., Stanford University Computer Science Department, May 1975. Stan-CS-74-494.

10. Brad Vander Zanden, Brad A. Myers, Dario Giuse and Pedro Szekely. "Integrating Pointer Variables into One-Way Constraint Models". *ACM Transactions on Computer Human Interaction 1* (June 1994), 161-213.

11. *Interlisp Reference Manual.* Xerox Corporation, 1983. Pasadena, CA.