The Amulet Environment: New Models for Effective User Interface Software Development

Brad A. Myers, Rich McDaniel, Rob Miller,
Alan Ferrency, Patrick Doane, Andrew Faulring,
Ellen Borison, Andy Mickish, and Alex Klimovitski

November, 1996
CMU-CS-96-189
CMU-HCII- 96-104

Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

bam@cs.cmu.edu
http://www.cs.cmu.edu/~amulet

Abstract

The Amulet user interface development environment makes it easier for programmers to create highly-interactive, graphical user interface software for Unix, Windows or Macintosh. Amulet uses new models for objects, constraints, animation, input, output, commands, and undo. The object system is a prototype-instance model in which there is no distinction between classes and instances or between methods and data. The constraint system allows any value of any object to be computed by arbitrary code and supports multiple constraint solvers. Animations can be attached to existing objects with a single line of code. Input from the user is handled by "Interactor" objects which support reuse of behavior objects. The output model provides a declarative definition of the graphics, and supports automatic refresh. Command objects encapsulate all of the information needed about operations, including support for various ways to undo them. An key feature of the Amulet design is that all graphical objects and behaviors of those objects are explicitly represented at run-time, so the system can provide a number of high-level built-in functions, including automatic display and editing of objects, and external analysis and control of interfaces. Amulet integrates these capabilities in a flexible and effective manner.

This research was 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 NCCOSC or the U.S. Government.

Keywords: Toolkits, User Interface Tools, User Interface Development Environments, User Interface Management Systems (UIMSs).

1. Introduction

Creating user interface software has proven to be very difficult and expensive. User interface software is often large and complex, and challenging to implement, debug, and modify. One study found that an average of 48% of the applications' code is devoted to the user interface, and that about 50% of the implementation time is devoted to the user interface portion [25]. Most of today's toolkits and interactive tools are still quite hard to use and lack flexibility. For example, to create new kinds of widgets, such as a scroll bar with two handles, or to add support for gesture recognition, is quite difficult with today's tools.

Amulet, a new user interface development environment for C++ that runs on X/11, Windows 95, Windows NT, and the Macintosh, facilitates user interface research and development. Amulet aims to make the design, prototyping, implementation and evaluation of user interfaces significantly easier, while supporting flexible experimentation with new styles. Amulet includes many design and implementation innovations including new models for objects, constraints, animation, input, output, commands, and undo.

In addition to incorporating innovations into its own design, Amulet has an open architecture to enable user interface researchers and developers to easily investigate their own innovations. For example, Amulet is the first system that supports multiple constraint *solvers* operating at the same time, so that researchers might easily investigate new kinds of constraint solvers. The undo model also supports new designs. The widgets are implemented in an open fashion using the Amulet intrinsics so that researchers can replace or modify the widgets. The goal is that researchers will only have to implement the parts that they are interested in, relying on the Amulet library for everything else. In addition, we aim for Amulet to be useful for students and general developers. Therefore, we have tried to make Amulet easy to learn, and to have sufficient robustness, performance and documentation to attract a wide audience.

Amulet, which stands for Automatic Manufacture of Usable and Learnable Editors and Toolkits, is implemented in C++. Amulet is based on our group's substantial experience from creating the Garnet user interface development environment [22], which was implemented in Common Lisp. Amulet brings to C++ the dynamic and rapid user interface design and implementation capabilities that Garnet provided in Lisp, while adding many new capabilities.

Since Amulet provides a structure for implementing the application-specific parts of a user interface, it can properly be called an *application framework* [19]. It is clearly much more than a "toolkit," which generally refers to a collection of widgets such as scroll bars and buttons. A key reason that Amulet provides a higher level of support than other systems is that all of the user interface objects are available at run-time for inspection and manipulation through a standard protocol, so that high-level, built-in utilities can be provided, which, in other toolkits, must be re-implemented for each application. For example, the graphical selection handles widget can get the list of graphical objects, and move and resize the selected object, through a standard protocol, even if the objects are custom-created and application-specific. Other facilities provided by Amulet include undo and operations like cut, copy, paste, save and load. This paper provides an overview of all the parts of the Amulet system.

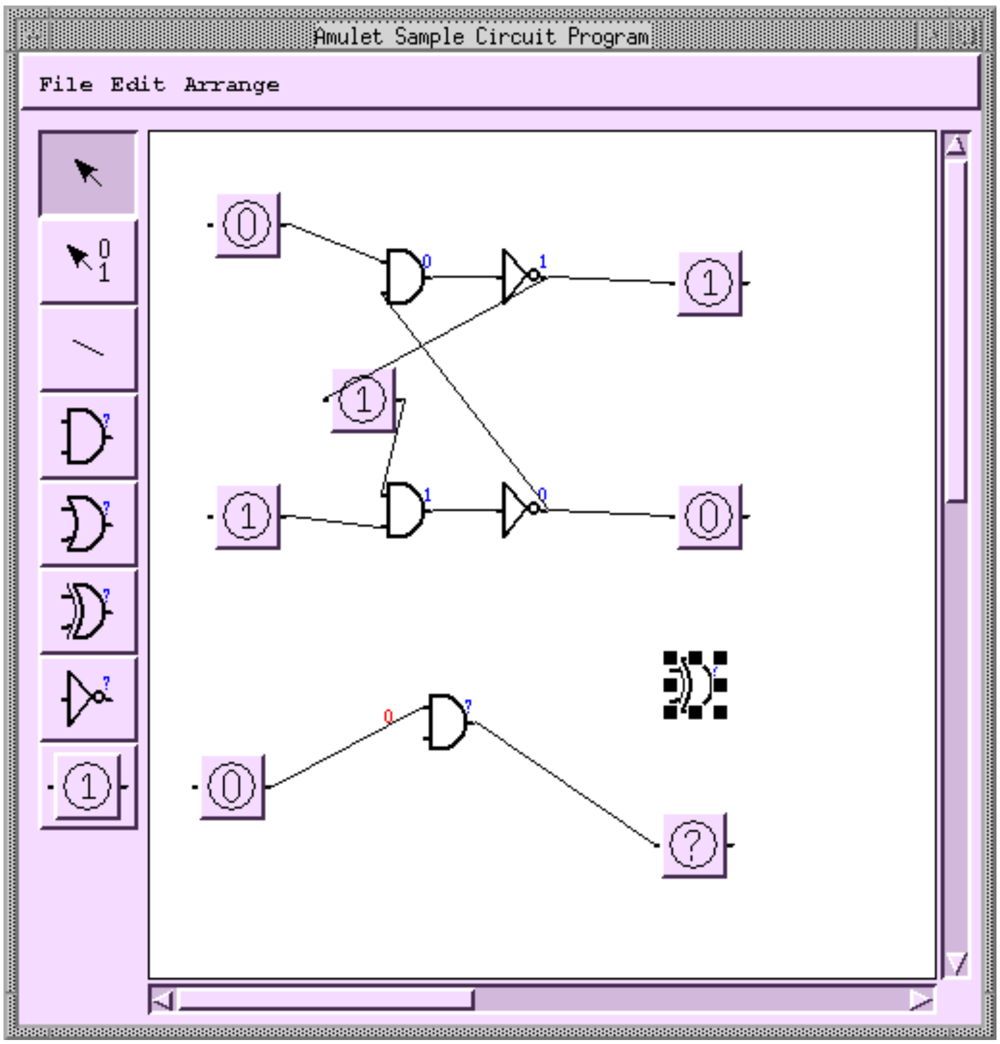


Figure 1: A simple circuit design program created with Amulet.

2. Goals

An important research objective of the Amulet project is to provide high-level support for the *insides* of application programs. Conventional toolkits like the Macintosh Toolbox and Motif provide a collection of widgets like menus, scroll-bars, buttons and text input fields. However, for graphical applications like drawing editors, CAD programs, visual language editors, visualizations, and charting programs, most of the programming for the user interface deals with the contents of the graphic windows, which do not contain any widgets. Consider the user interface shown in Figure 1. Other toolkits will provide the menubar at the top, and possibly the palette on the left, but they provide no help with the main area where the circuits are drawn. Instead, programmers must program directly at the window manager level without much support. In contrast, Amulet provides high-level support for these kinds of graphical applications, including

- Automatically redrawing the graphics,
- Constraints that automatically keep the wires attached to the circuit elements,
- Widgets such as selection handles which make interactive behaviors easy to implement, and,
- Built-in editing commands, such as cut, copy, paste, to top and bottom, and undo, that can often be used directly from the library without modification.

These facilities are made possible because Amulet's novel models make the components and structure of the user interface and application visible and manipulable by standard utilities.

We distribute Amulet for use by others because we feel this will help to demonstrate that the innovations in Amulet are sound and effective, and will hopefully facilitate technology transfer. Amulet is in the public domain and can be used for free. Version 1 of Amulet was released in July 1995 and Version 2 was released in May 1996. To get Amulet, including the complete source code, visit `http://www.cs.cmu.edu/~amulet` or send mail to `amulet@cs.cmu.edu`.

We want to make Amulet useful for:

- **Researchers**. Over 25 research projects all over the world are already using Amulet, including a number of thesis projects at all levels.
- **Students.** By aiming for Amulet to be useful to students, we are continually striving to make Amulet easier to learn. The success of SUIT [27] and Microsoft's Visual Basic show that it is possible to provide useful functionality in a way that is easy to learn, but unlike those other systems, Amulet provides a natural growth path to the complete fully-functional system. Amulet has been used in three courses at Carnegie-

Mellon University and at least two courses elsewhere, which have provided feedback used in refining the interface.

- **General Developers.** We also want Amulet to be useful for general user interface software construction. For this reason, Amulet runs on X/11, Windows 95, Windows NT, and the Macintosh. We also provide a high level of robustness, documentation and performance. A complete reference manual including a tutorial is available [20]. A few commercial products are even being built with Amulet.

3. Example: A Circuit Designer

Suppose you wanted to build an application like the circuit design program of Figure 1. This program should work in the standard way:

- Clicking on the palette on a circuit element, and then in the work window should create an object of that type. Gridding should be used to help lay out the objects neatly.
- The wires should be dragged from the source gate to the destination gate with the usual "rubber-band" feedback.
- Selection handles should appear on objects when they are selected, and objects can be dragged (moved) in the standard way. Multiple objects should be selected in the standard way, either by holding down the SHIFT key while clicking on objects, or dragging out a region. The wires must stay attached when elements are dragged. Growing of the gates should not be allowed.
- All the standard editing operations should be supported, such as cut, copy, paste, clear, clear-all, select-all, to-top, and to-bottom. When a gate is deleted, any attached wires must be removed.
- All operations must be undo-able.
- Circuit diagrams can be saved to a file and loaded back in.

In addition, this program should provide some advanced features:

- The program should simulate the operation of the circuit by showing the values calculated by the gates. The input nodes can be toggled and the outputs should display the correct values. Animations should be used to make the execution more understandable.
- Instead of repeatedly going back to the palette, the user should instead be able to use *gestures* to create all the kinds of objects using the right mouse button (on the Macintosh, using the Option keyboard key while pressing on the one mouse button). For a gesture, the path of the mouse is important, not just its start and end position. For example, the user can draw an "o" to make an OR gate, an "A" to make an AND gate, a ">" to make a NOT gate, a line to make wires, etc. The dot gesture should be undo, to make it easy to correct errors.

- The application should run on Unix, Windows NT, Windows 95, or the Macintosh and use widgets with an appropriate look-and-feel on each platform.

With most toolkits, such as Motif, Microsoft Foundation Classes, MetroWerks PowerPlant, Visual Basic, Borland's Delphi, Java AWT, etc., the code for this application would be tens of thousands of lines of code. However, using Amulet, this entire application requires only 850 lines of C++ code, due to Amulet's high-level features such as:

- **Graphical objects** with automatic refresh.
- **Interactors** that handle standard behaviors.
- **Command Objects** that handle editing operations.
- **Constraints** that maintain relationships among objects.
- **Gesture Recognition** as a built-in kind of Interactor.
- **Animations**, as a special form of constraint.

The rest of this paper will describe these and other features of Amulet that make creating interactive applications easier.

3.1. Graphical Objects

An important goal is to make Amulet easy to learn and use for developers, even though it has a large number of features. Therefore, we have concentrated on giving Amulet a uniform structure based on a few simple concepts. The main concept is that everything in Amulet is represented as an *object* which has a set of *slots* . A "slot" has a name and can hold a value of any type, for example the slot named Am_LEFT[1] might hold the value 10. Slots are similar to member variables or instance variables in other object systems. A new object is created by making an *instance* or *copy* of another object, which is called the *prototype.* An "instance" starts off inheriting all of its slot values from the prototype, and the slots can then be set with new values. A copy immediately gets a copy of all values in the prototype. For example, the following creates an And_Gate as an instance of the built-in Bitmap object, and then sets the IMAGE slot to the appropriate picture.

```
Am_Object And_Gate = Am_Bitmap.Create()
        .Set(Am_IMAGE, and_bitmap_image);
```

Any object can serve as a prototype to create other objects. There is nothing special about the objects in the Amulet library. For example, the following creates an instance of the And_Gate and then puts it in a particular place:

[1]Because C++ does not support separate name spaces, all exported names in Amulet start with "Am_".

```
Am_Object new_gate = And_Gate.Create()
        .Set(Am_LEFT, 10)
        .Set(Am_TOP, 43);
```

We allow the Sets to be chained together, but the previous code could instead be written:

```
Am_Object new_gate = And_Gate.Create();
new_gate.Set(Am_LEFT, 10);
new_gate.Set(Am_TOP, 43);
```

To make objects appear on the screen, they are simply added as a *part* to a window which is added to the screen. Because all the graphics on the screen are represented by objects in memory, changes to the screen are accomplished simply by setting the slots of objects with new values. For example, the new_gate could be made red by simply doing:

```
new_gate.Set(Am_LINE_STYLE, Am_RED);
```

When slots are set, Amulet automatically redraws the object, as well as any other objects that overlap it, so the screen is appropriately updated.

The following is the complete "hello world" program in Amulet, that displays a string, and redraws the string if the window becomes covered and then uncovered. This program would be about 2 pages long in Motif.

```
#include <amulet.h>
void main (void) {
  Am_Initialize (); //initialize Amulet
  Am_Screen
    //add a window to the screen using all of the default values
    .Add_Part (Am_Window.Create ()
     .Add_Part (Am_Text.Create () //create a text object and add it to the window
        .Set (Am_TEXT, "Hello World!"))); //set the string
  Am_Main_Event_Loop (); //display the window and then handle all input events
  Am_Cleanup (); //clean up Amulet
}
```

3.2 Interactors

To make objects respond to input, an instance of an *Interactor* object is attached to the graphics. Different types of Interactors handle different types of behaviors. For example, to make the new_gate object be movable, a move-grow Interactor can be attached to it:

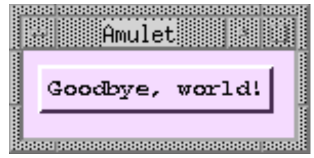
```
new_gate.Add_Part(Am_Move_Grow_Interactor.Create();
```

By default, the Interactor starts when the left mouse button is hit over the object, but this is easily changed by setting a slot of the Interactor. The following will allow new_gate to be selected with the right mouse button:

```
new_gate.Add_Part(Am_Choice_Interactor.Create()
        .Set(Am_START_WHEN, "right_down");
```

When an instance is made of an object that has parts, then Amulet makes instances of all the parts as well. Widgets, like buttons and scrollbars, contain graphical and Interactor objects as parts, but the designer can make an instance of the widget in the same way as instances of primitive objects are created:

```
Am_Object my_button = Am_Button.Create ()
     .Add_Part(Am_COMMAND, Am_Quit_Command.Create()
               .Set (Am_LABEL, "Goodbye, world!")));
```



All of widgets in the Amulet library are defined with multiple looks-and-feels, so a program can be written using Amulet, and it can then be compiled on Unix, Microsoft Windows or the Macintosh *without editing the source code*, and it will have the correct look and feel for the target platform.

3.4 Command Objects

When Interactors or widgets are operated by the user, instead of calling "call-back procedures" as in other toolkits, they allocate an instance of a *Command object,* and execute its "do" method. Command objects also support *Undoing*, *Redoing* and *Repeating* operations, as well as enabling and disabling (graying out), and help for operations. Many commands are available in the Amulet library and can often be used *without change*. The code above uses the built-in Quit command, so `my_button` can be added to a window to form the complete "goodbye world" program, which exits when the button is pressed.

3.5 Constraints

Another important feature of Amulet is the support for *constraints*, which are relationships that are declared once and then maintained by the system. Amulet supports multiple *kinds* of constraints, but the main kind is *formula* constraints, which act like spreadsheet formulas. When put into a slot of an object, formulas compute the value of the slot based on slots of other objects. Formula constraints can contain arbitrary C++ code, and use a special form of Get (called `GV`) that not only returns the value, but also registers the constraint to be re-evaluated if the dependent value changes. For example, the left of the scrolling region in Figure 1 stays to the right of the tool palette, even if the size of the tool palette changes, by using the following formula in its `Am_LEFT` slot:

```
// define a formula called right_of_tool_panel_formula which returns an int
Am_Define_Formula(int, right_of_tool_panel_formula) {
  // 5 pixels away from the right of the tool_panel
   return (int)tool_panel.GV(Am_LEFT) + (int)tool_panel.GV(Am_WIDTH) + 5;
}
...
scrolling_window.Set(Am_LEFT, right_of_tool_panel_formula);
```

3.6 Gesture Recognition

One of the built-in types of interactor objects supports *gesture recognition*. Using an interactive tool called "Agate", the designer gives about 10 examples of each gesture desired in the interface, and associates a string name with each gesture. Then, in the program, the designer can associate each name with an operation. Built-in commands provide a standard interface between the "normal" direct manipulation commands, for example to create and delete objects, and the gestures. For example, part of the code to handle gestures in the circuit program is:

```
Am_Object gesture_reader = Am_Gesture_Interactor.Create("gesture_reader")
    //gestures work when you hold down the right mouse button
    .Set (Am_START_WHEN, "any_right_down")
    //an object to show the gesture while in-progress (interim feedback)
    .Set (Am_FEEDBACK_OBJECT, gesture_feedback)
    //the gesture classifier read from a file that was created using the Agate tool
    .Set (Am_CLASSIFIER, gc)
    //first, a command to handle gestures that are not recognized
    .Add_Part(Am_COMMAND, Am_Gesture_Unrecognized_Command.Create())
    //now, a list of commands, one for each of the gestures defined by example
    .Set (Am_ITEMS, Am_Value_List ()
        //first, the gesture for the "And" gate
        .Add (Am_Gesture_Create_Command.Create()
            .Set (Am_LABEL, "and") //string defined in Agate for this gesture
            .Set (NEW_OBJECT_PROTO, and_proto) //object to create
                .Set (Am_CREATE_NEW_OBJECT_METHOD, gesture_creator))
        //next, the gesture for the "OR" gate
        .Add (Am_Gesture_Create_Command.Create()
            .Set (Am_LABEL, "or")
            .Set (NEW_OBJECT_PROTO, or_proto)
                .Set (Am_CREATE_NEW_OBJECT_METHOD, gesture_creator));
```

3.7 Animations

Amulet has a flexible constraint system, which allows new kinds of constraint *solvers* to be created. We used this facility to create an *animation constraint solver*. When an animation constraint is attached to a slot and the slot changes value, the animation constraint removes the new value, re-sets the slot with the old value, and smoothly sets the slot with values interpolated from the old value to the new value. For example, to move the little red numbers along with wires in the circuit program, the following code is used:

```
//first create the animator object
Am_Object number_animator = Am_Interpolator.Create ();

//now create the prototype number to be moved
animation_proto = Am_Text.Create("animation_proto")
     .Set(Am_TEXT, “0”)
    .Set(Am_FONT, small_font)
   .Set(Am_LINE_STYLE, Am_Red)
     .Set(Am_LEFT, Am_Animate_With (number_animator))
     .Set(Am_TOP, Am_Animate_With (number_animator));

//when ready to start an animation, first turn animations off and set the position
//to one end of the wire
   anim.Set(Am_LEFT, (int)line.Get(Am_X1), Am_NO_ANIMATION);
   anim.Set(Am_TOP, (int)line.Get(Am_Y1)-10, Am_NO_ANIMATION);
//then set the left and top again, to the other end of the wire, but this time letting
//the animations make the object move smoothly
   anim.Set(Am_LEFT, (int)line.Get(Am_X2));
   anim.Set(Am_TOP, (int)line.Get(Am_Y2)-10);
```

4. Details of the Design

The Amulet toolkit is divided into a number of layers (see Figure 2). These layers include an abstract interface to the window managers, novel models for objects, constraints, input, output, and commands, and a set of widgets. The following sections describe the overall design of each of these.

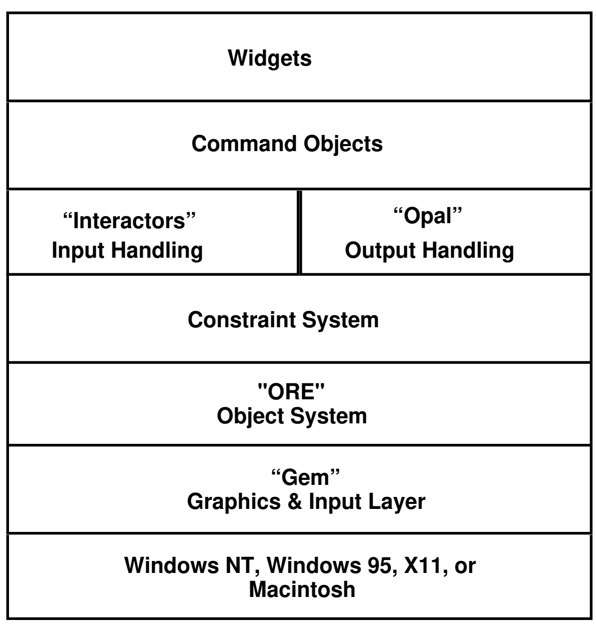


Figure 2: The overall structure of the Amulet system.

4.1. Gem: Abstract Interface to the Window Managers

Amulet provides a portable interface to various window managers called "Gem," which stands for the Graphics and Events Manager. Gem uses ordinary C++ objects and mechanisms to provide a simple graphics and input interface used by the rest of Amulet. Any code written using Gem will port to different windowing systems without change. Typical Amulet users never see the Gem interface, however, since the higher-level parts of the Amulet toolkit provide access to the same capabilities in an easier way. The circuit program of Figure 1 did not require any programming at the Gem layer. We export the Gem interface for advanced Amulet users. If the programmer wants to make something very efficient, calling Gem directly may be appropriate. For example, although widgets such as buttons and scrollbars can be implemented using the high-level Opal output model, as was done in Garnet [22], Amulet's widgets are implemented more efficiently using Gem-level drawing routines.

4.2. Object System

The "Ore" (Object Registering and Encoding) layer of Amulet implements a prototype-instance object system on top of C++. In a prototype-instance object system, there is no distinction between classes and instances: every object can be used as a prototype for other objects. Slots of the prototype can be inherited by instances, so that changes to the prototype's slot will be seen by any instances which do not override it.

4.2.1 Slots

Programming with a prototype-instance object system is a quite different style than conventional object-oriented languages. Much of the code is devoted to defining the slots and default values for prototype objects, and then creating instances, possibly overriding some slots. For example, to create an instance of the zero_one_proto object, which is the prototype for the zero-one buttons, the program just does:

```
Am_Object new_obj = zero_one_proto.Create ()
  .Set (Am_LEFT, new_obj_left)
  .Set (Am_TOP, new_obj_top);
```

Slots can be set with any type of value:

```
obj.Set(Am_LEFT, 40);
obj.Set(Am_TEXT, "Hello");
obj.Set(OTHER_OBJ, and_gate1);
```

The object system is *dynamic* in that slots in objects can be added and removed from objects at run time, and the types in slots can also change. Amulet performs run time type checking if a type is declared for the slot.

To allow the same Set and Get to work for all types in C++, we provide accessor and setting methods for the standard built-in types, void (untyped) pointers, Amulet objects, and a special class called a "Wrapper." Any new C++ type that the programmer wants to store into objects and have Amulet type-check can be made a subclass of Wrapper. Amulet will then also handle memory management for them.

C++'s overloading and type-conversion capabilities make the interface very convenient. For example, the Am_Object class defines a number of Set routines:

```
Am_Object Set (Am_Slot_Key key, Am_Wrapper* value);
Am_Object Set (Am_Slot_Key key, void* value);
Am_Object Set (Am_Slot_Key key, int value);
Am_Object Set (Am_Slot_Key key, float value);
Am_Object Set (Am_Slot_Key key, char value);
Am_Object Set (Am_Slot_Key key, const char* value);
```

The compiler will choose the correct one based on which type is actually used. Note that Set returns the original object, allowing Sets to be cascaded, so the code above could instead be:

```
obj.Set(Am_LEFT, 40).Set(Am_TEXT, "Hello").Set(OTHER_OBJ, and_gate1);
```

C++ does not allow overloaded functions to be chosen based on the return type, but we were able to get around this by returning a special Am_Value type, which then has type-conversion routines into the various primitive types. This allows code like:

```
int i = circuit_object_proto .Get(Am_VALUE);
bool b = this_command.Get(Am_GROW_INACTIVE);
//the next statement will work no matter what type is in the slot
Am_Value v = tool_panel.Get(Am_IMPLEMENTATION_PARENT);
if (v.type == Am_BOOL) ...
```

In the last lines we use the special Am_Value type which permits programmers to dynamically access and set the type and value.

4.2.2 Slot Inheritance

When an instance of an object is created, the slots that are not specified inherit their values from the prototype object's. If a slot of the prototype is changed, then the value also changes in all of the instances that do not override that property. For example, changing the Am_WIDGET_LOOK slot in the zero_one_proto will change the look in all instances, so it is easy to see what the circuit program will look like when running on a Macintosh or Windows. However, changing the Am_LEFT of the zero_one_proto will not affect new_obj since it has a local value for Am_LEFT. If the programmer does not want this behavior, then Amulet allows the inheritance of each slot to be specified as "copy" or "local". When an instance is made for a slot with copy inheritance, the value is copied into a new slot created in the instance, so later changes to the prototype do not affect the instance. A slot can be declared “local”, so the slot does not appear in the instance at all. This is useful for slots that hold information that is particular to the object. In the definition of the circuit_object_proto, the slots that will hold the connect input and output wires are declared to be local so that when copies or instances are made of an object, every instance of a circuit element will have unique wires. Therefore, if the user duplicates a circuit element, Amulet ensures that the copy will start off with its input and output slots empty.

The inheritance mechanism is an important distinction with other prototype-instance object systems, such as SELF [4], in which all the slots are always *copied* into instances so changes to prototypes never affect instances. Although Amulet’s model requires slightly more overhead, we think it is useful for prototyping to be able to change properties of prototypes and see the effect on all instances immediately.

4.2.3 Methods

An important feature of Amulet’s object system is that there is no distinction between *methods* and *data*: any instance can override an inherited method as easily as inherited data. In a conventional class-instance model such as SmallTalk or C++, instances can have different data, but only sub-classes can have different methods. Thus, in cases where each instance needs a unique method, conventional systems must use a mechanism other than the regular method invocation. For example, a button widget might use a regular C++ method for drawing, but would have to use a different mechanism for the call-back procedure used when the user clicks on the button, since each instance of the button needs a different call-back. In Amulet, the draw method and the callback use the same mechanism. In the circuit code, we set a method into the tool panel so that whenever the user changes modes, the selection will be cleared. This method is coded as:

```
Am_Define_Method(Am_Object_Method, void, clear_selection,
                        (Am_Object  /*cmd*/)){
  my_selection.Set(Am_VALUE, NULL);
}
```

and it can be put into a slot the same way as data is set into slots:

```
tool_panel.Get_Part(Am_COMMAND)
    .Set(Am_DO_METHOD, clear_selection);
```

4.2.4 Part-Owner Hierarchy

The object system also implements a part-owner hierarchy. The owner is usually a “group” (aggregate) object or a window, and the parts are either other groups or primitives like rectangles, lines and text. However, we have found many other uses for the part-owner hierarchy that are independent of graphical relationships. Supporting the part-owner hierarchy in the object system allows Amulet to provide “structural inheritance,” which means that when an instance is made of a group which contains parts, the new instance will have instances of all the prototype's parts, as shown in Figure 3. Thus, programmers can create instances of any type of object without knowing whether it is a primitive or a group, and the system will make sure that the instance has the same structure as the prototype. Changing the xor_gate from a simple bitmap to a group containing a bitmap and three lines as input and output ports only required changing the prototype—none of the *uses* of the prototype needed to change.

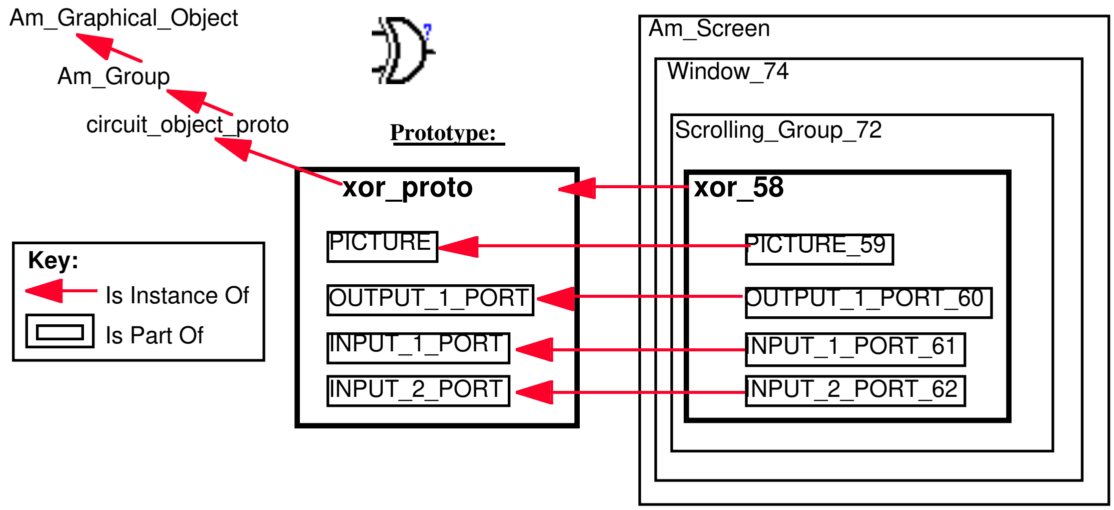


Figure 3: The xor_proto object contains 4 parts: a bitmap called PICTURE, and output and input ports which are instances of lines. The xor_proto is an instance of the circuit_object_proto, which in turn is an instance of an Am_Group, which is an Am_Graphical_Object. When an instance is made of the xor_prototype, Amulet automatically creates instances of each of the parts. If instances are not named, then Amulet makes up a name by appending a number. The instance, called xor_58, has been made a part of a scrolling group, which is part of a window, which is part of the screen, so it will be visible.

Some systems, such as FormsVBT [2] have hard-wired some slots to inherit values from their prototypes and others to inherit from their owners. Because the constraint mechanism is so easy to use and flexible in Amulet, it is sufficient to use constraints whenever slots should get their values from their owners rather than from their prototypes.

Amulet also provides a special form of group called an Am_Map that computes its parts dynamically. The Am_Map object uses a constraint to build a list of parts, usually based on a single prototype object, called the "item prototype." Typically, a list of strings, objects, commands, or something else, is provided, and the Am_Map creates an instance of the item prototype for each value in the list, setting a particular slot of the instance with the corresponding value from the list. The programmer will define a constraint somewhere in the item prototype that depends on the particular value copied from the list. Note that due to the flexibility of Amulet's constraint system (discussed below), *any* slot of the item can depend on the list of values supplied. For example, a list of strings might be supplied for a menu, a list of objects for a palette, or a list of locations for a scatter plot.

4.2.5 Other features

Amulet's object system also contains many other features that may be useful for programmers who need extra control. Automatic memory management using a reference counting scheme is available for objects and "wrappers" (used to "wrap" C++ types so they can be put into Amulet objects with full type-checking). A flexible "demon" mechanism allows procedures to be attached to objects or slots for invocation when the slots change. This mechanism enables Amulet to redraw objects when their graphical properties change. Programmers can also create their own demons. Type checking of slots is supported by Amulet, so that programmers can declare that a slot can only hold a specific kind of value. A complete set of querying functions allows determining objects' properties at run-time. These are used by the debugging facilities described below, and they can also be useful for application programs. All of these features are described in full in another paper [16].

4.2.6 Performance

The main disadvantage of the prototype-instance model over the conventional class-instance model has been performance. When slots are accessed, the system must perform a search through the object to see if the slot is there, and if not, it must search the prototypes up to the root. The same search is needed for both method and data slots. Dynamic type checking also adds some overhead. The forward and backward pointers and space for the types add space overhead. The SELF prototype-instance system [4] uses extensive compiler techniques to try to remove some of this search, but we have not found this necessary, and the performance of the Amulet system is quite good. There are no noticeable delays for normal size programs on a variety of modern hardware platforms. For example, the circuit program executes fine on Unix, Macintosh and PC platforms.

We optimized our previous Garnet prototype-instance object system for speed by copying all values to all instances, even if they were the same as the prototype's. However, this had a significant space penalty, so in Amulet, we only store the local slots, which in practice is only about half of the slots.

The times for various Get and Sets are shown in the following table for Amulet compiled fully optimized. The Unix timings were performed on a Sun SPARC 20, the Macintosh timings on a PowerMac 8500/150, and the PC timings are on a 133 MHz Pentium running Windows 95.

Times in Microseconds	Get Slot	Set Slot
Unix	2.5	2.9
Mac	1.7	1.9
PC	1.7	1.4

4.2.7 Discussion

There are many advantages of the prototype-instance model. Having no distinction between classes and instances, or between methods and data, means that there are fewer concepts for the programmer to learn and a consistent mechanism can be used everywhere. Another advantage of the prototype-instance object system is that it is very dynamic and flexible. All of the properties of objects can be set and queried at run time, and interactive tools can easily read and set these properties. In fact, most of today's toolkits implement some form of "attribute-value pairs" to hold the properties of the widgets, but Amulet's object system provides significantly more flexibility and capabilities.

Amulet's predecessor, Garnet, also used a prototype-instance object system [21], as have a few other systems such as SELF [4], Apple's NewtonScript and General Magic's MagicCap. Amulet's design is more complete and flexible, and we fixed a number of problems we experienced with Garnet, including adding control over the inheritance of slots, automatic management of a part-owner hierarchy along with the prototype instance hierarchy, support for multiple constraint solvers, and a flexible demon mechanism. Finally, it is worth pointing out that we are able to provide dynamic slot typing, a dynamic prototype-instance system, and constraints in C++ without using a preprocessor or a scripting language.

Garnet supported *multiple inheritance*, but we found it was not useful or necessary. In Amulet, we instead use the constraint mechanism to copy values among objects, which provides complete flexibility and control. Omitting multiple-inheritance has simplified much of Amulet's implementation leading to an easier-to-understand object creation procedure and better efficiency when searching for slots. It also eliminates the ambiguity and complexity for the programmer of resolving collisions of slot names from multiple prototypes.

Although designed to support the creation of graphical objects, many Amulet users have discovered that the prototype-instance object system is useful for representing their internal application data. The flexibility and dynamic nature of the objects make them ideal when varied and changing data types are necessary. Amulet objects are somewhat like "frames" used by artificial intelligence systems, so AI applications may find the model useful. The constraint system is also useful for maintaining data dependencies and consistency in application-specific data structures.

4.3. Constraints

Amulet integrates constraint solving with the object system. This means that instead of containing a constant value like a number or a string, any slot of any object can contain an expression which computes the value. If the expression references slots of other objects, then when those objects are changed, the expression is automatically re-evaluated. Thus, the constraints are primarily "one-way," like those of Artkit [9] and Rendezvous [7]. This kind of constraint resembles a spreadsheet formula, so it is called a "formula constraint" in Amulet. Constraint expressions can contain arbitrary C++ code, and the only restriction is that accesses to slots of objects must use a special function, called GV. In addition to returning the value of the slot, GV also sets up a dependency link so that the constraint will be re-evaluated when the other object changes.

Currently, Amulet does not use a preprocessor, so the syntax for specifying constraints is a little verbose. The macro Am_Define_Formula creates a constraint object of the specified name using the code that follows. The constraint object stores a pointer to the procedure to execute, the name of the constraint for debugging and tracing, and the list of slots used by and using this constraint. As an example, the following constraint from the circuit program determines the picture used for the zero-one buttons based on the value. Note that constraints can return any type. Here, the constraint is returning an image array, which is Amulet's machine-independent representation for a bitmap.

```
Am_Define_Image_Formula(zero_one_formula) {
  int value = self.GV(Am_VALUE);
   if (value ==1) return one_image;
   else if (value == 0) return zero_image;
  else return question_image;
}
...
obj.Set(Am_IMAGE, zero_one_formula);
```

The circuit program uses 24 custom constraints not counting all the constraints that are built into the objects themselves (for example, the built-in Am_Text object has constraints in its width and height slots that compute its dimensions based on the current string and font). Formulas are set into slots using the standard Set. In the future we hope to add a pre-processor to support a conventional "dot" notation for slot access (obj.slot) and to allow constraint expressions to appear inside the Set instead of only as a top-level procedure.

Slots are accessed the same way whether they contain constraints or constant values, and the code containing the Get normally does not know how the value was calculated. For example, the button widget does not care that the image was computed with a constraint. The object system is tied into the graphics system using demons so that whenever the value of a slot changes, either because the programmer set it or due to constraints, the object will be redrawn automatically.

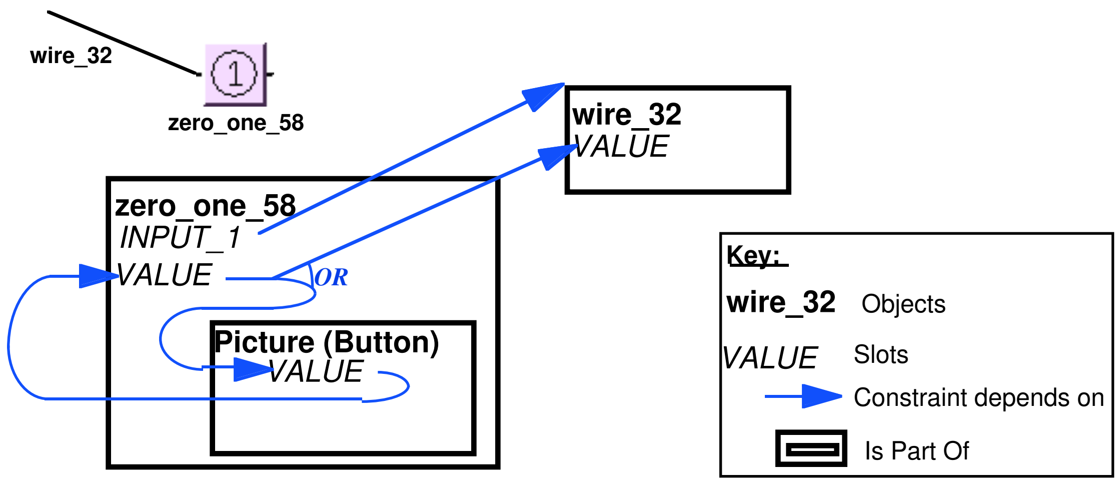


Figure 4: The VALUE slot of the zero_one objects have a constraint that depends on either the value of the input wire, if it exists, or else the value of the button if there is no wire (if the INPUT_1 slot is NULL). The button in the PICTURE slot has a constraint that depends on the value for the zero_one object so it will have the right value when there is a wire. This creates a cycle of constraints.

The Amulet constraint solver handles cycles in the constraints, so that a slot of object A can depend on a slot of object B and vice versa. This is used in the circuit program so that the zero-one buttons get their values from the input wires if present, and if not present then the value comes from the toggle button. This is implemented, as shown in Figure 4, by having a constraint from the zero-one-proto to the button, and another constraint from the button to the zero-one value. In evaluating circular constraints, Amulet simply goes around the cycle once, and uses the old value of any constraint that is already being evaluated. If the programmer uses constraints that are consistent, the values will be correct and this can be an effective way to set up mutual dependencies.

4.3.1 Indirect Constraints

The Amulet constraint system supports dynamic computation of the objects to which a constraint refers, so a constraint can not only compute the value to return, but also *which objects* and slots to reference. This allows such constraints as “the width is the maximum of all the components” which will be updated whenever components are added or removed as well as when one of the components’ position changes. An example from the circuit program is that the value of a gate is computed by first accessing the input wire objects from the INPUT slots of the gate, and then indirectly accessing the value of the line objects. For example, the value of the OR gate is calculated using the formula:

```
Am_Define_Formula(int, OR_value) {
  Am_Object in_line_1 = self.GV(INPUT_1);
  Am_Object in_line_2 = self.GV(INPUT_2);
   if (in_line_1.Valid() && in_line_2.Valid()) {
        int v1 = in_line_1.Get(Am_VALUE);
        int v2 = in_line_2.Get(Am_VALUE);
        return v1 | v2;
  }
  else return -1; //return for when have an illegal value
}
```

Most other constraint systems cannot handle these kinds of constraints. These "indirect constraints" [31] are also important for supporting object inheritance. When an instance is created of an object, Amulet also creates instances of any constraints in that object. These constraints refer to other objects indirectly using the structure of the groups. For example, in Figure 3, the constraint for the top of the INPUT_1_PORT is computed based on the center of the picture, which is its sibling in the part-owner hierarchy:

```
Am_Define_Formula(int, picture_center_y) {
  Am_Object picture = self.GV_Sibling(PICTURE);
   return (int)picture.GV(Am_TOP)+(int)picture.GV(Am_HEIGHT)/2;
}
```

Note that even though many input and output port objects share this same constraint, they will each calculate different values because the pictures will be at different places.

4.3.2 Multiple Constraints

Amulet also allows slots to contain multiple constraints at the same time. We find this very useful for situations where an inherited formula is necessary for the correct operation of an object, but the programmer wants an additional formula so values can flow in multiple directions. For example, we set a constraint into the Am_VALUE slot of the button widgets used in the zero-one objects to display the value of the input port, if any. Internally, however, the button widget uses a constraint in the Am_VALUE slot to make the slot change values when the user clicks on the widget. Both of these constraints can co-exist in the Amulet constraint system. When there are multiple constraints in a slot, normally only one will become invalid at a time, and so that one will be the one that is requested to recalculate the slot's value. When multiple constraints become invalid at the same time, Amulet evaluates the constraint that first becomes invalid.

4.3.3 Side Effects

Our experience with Garnet suggested that people wanted to put side effects into constraint expressions, and use them like "demon procedures" or "active values."

Therefore, Amulet's constraints are eagerly evaluated and can contain arbitrary side effects, even creating and destroying objects. For example, a constraint is used in the Am_Map object to create the instances of the item prototype based on the list in the Am_ITEMS slot. This constraint creates objects which themselves will contain constraints which need to be evaluated. Another use is that even though formula constraints must be put into a single slot, they can have the effect of multiple outputs by simply setting the other slots as side effects. For example, for efficiency, the constraint on the first end point of the wires are put into the X1 slot, but also sets the Y1 slot:

```
Am_Define_Formula(int, line_x1y1) {
 Am_Object source_obj = self.GV(INPUT_1);
 int x1 = (int)source_obj.GV(Am_WIDTH) + (int)source_obj.GV(Am_LEFT);
 int y1 = (int)source_obj.GV(Am_HEIGHT)/2 + (int)source_obj.GV(Am_TOP);
 self.Set(Am_Y1, y1); //set Y1 by side effects for efficiency
 return x1;
}
```

Unlike previous systems such as Rendezvous [7], Amulet does *not* require the programmer to use a special mechanism for side effects: the regular Set and Create calls are used. This works because we store any new constraints that need to be evaluated in a queue. When a constraint evaluation creates new constraints that need to be evaluated, they are simply added to the end of the queue. Amulet continues to evaluate constraints on the queue until the queue is empty, at which point Amulet redraws the objects that have changed.

When using side effects in constraints, programmers must be careful to avoid situations that will create an infinite loop. Constraints without side effects will always be evaluated exactly once each time the values change, since Amulet orders the constraint evaluation and checks for cycles of constraints, as discussed above. However, a programmer could set up a set of constraints that invalidated each other through side effects. If the constraints are consistent, so that slots are set to the same values no matter which constraints are used, then the evaluation will terminate even if the constraints contain cycles of dependencies and side effects. However, if the constraints calculate and set different values, an infinite loop can result.

4.3.4 Multiple Solvers

An important research area in user interface software is creating new kinds of constraint solvers (e.g. [6, 9, 30]). Therefore, Amulet contains an architecture that allows multiple *solvers* to co-exist. Currently, in addition to the one-way solver described above, Amulet supports a multi-output, multi-way solver called a "web," and an animation constraint solver.

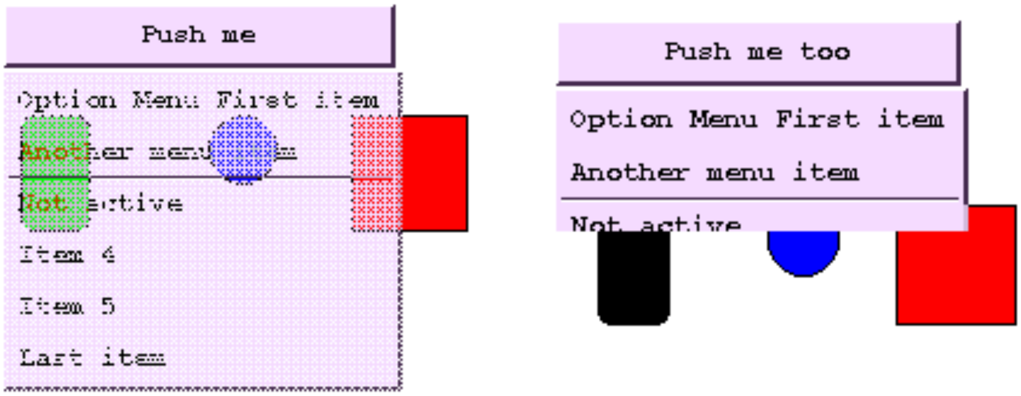
The web constraint can have an arbitrary number of input and output slots, and it can dynamically compute the dependencies like formula constraints. Webs also keep track of the order that dependencies change. We use this solver to keep the various slots of lines and polygons consistent. The line object has two sets of input slots. One set is point based and has slots called X1, Y1, X2, and Y2. The other set is rectangle-based and has slots called LEFT, TOP, WIDTH, and HEIGHT which are set when the line is moved without changing its orientation. However, if the slots X1, TOP, and WIDTH were set, the normal one-way formula mechanism would not necessarily evaluate the constraints in the correct order, but the web maintains the original order of slot changes, so the final result will be correct.

4.3.4.1 Animations

We have also created an novel *animation constraint solver* for animating objects [24]. Adding animation to interfaces is a very difficult task with today's toolkits, even though there are many situations in which it would be useful and effective. An animation constraint detects changes to the value of the slot, immediately restores the original value, and causes the slot to take on a series of values interpolated between the original and new values.

The advantage over previous approaches is that animation constraints provide significantly better modularity and reuse. The programmer has independent control over the graphics to be animated, the start and end values of the animation, the path through value space, and the timing of the animation. Animations can be attached to any object, even existing widgets from the toolkit, and any type of value can be animated: scalars, coordinates, fonts, colors, line-widths, vertex lists (for polygons), booleans (for visibility), etc.

A number of built-in animation constraints are used for special effects. For example, Figure 5 shows the effects of different constraints added to the Visible slot of a pop-up menu. A library of useful animation constraints is provided in the toolkit, including support for exaggerated, cartoon-style effects such as slow-in-and-slow-out, anticipation, and follow-through [5]. The programmer can also create custom animation types, if the built-in ones are not sufficient.



(a) (b)

Figure 5: Animation constraints can be added to the `VISIBLE` slot of a popup menu to make it (a) fade in using halftoning or (b) grow from the top.

Often, animation constraints can be added to an existing application with only a single extra line of code, which makes it easy to explore many new uses for animations. For example, undoing operations can animate the objects back to their original appearance, which makes it easier for users to see what has happened. Of course, animations can also be used to construct games (see Figure 6) and dynamic visualizations. The code to support this animation in the circuit program is only about 30 lines.

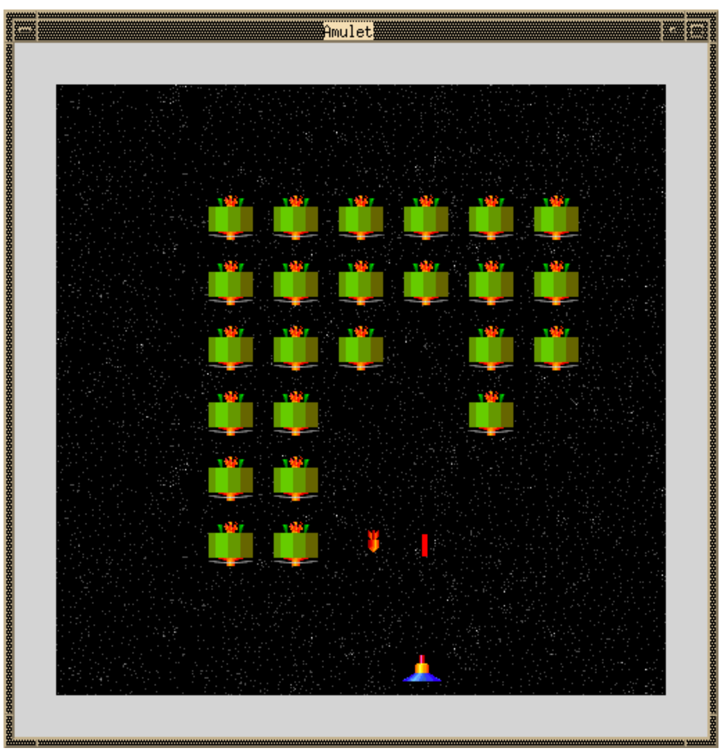


Figure 6: The ships, bullets and bombs are controlled by simple moving animators. For example, the bullets can be sent to the top of the screen by simply attaching an animation constraint to their TOP slot and then setting that slot to -10.

4.3.4.2 Design

We were able to add animation and Web constraints to Amulet without modifying the object system because there is a standard protocol that allows new solvers to be added. Every slot can contain two lists of constraints: the set of constraints that depend on the value of the slot, and the set of constraints on which the slot depends. Various messages to the slots themselves are available to the constraints, including:

- Set (to change the value of the slot),
- Invalidate (to notify the slot that its current value is not valid), and
- Get (to access the current value).

The messages that slots can send to constraints include:

- Change (for when the slot's value changes),
- Invalidated, which notifies all the constraints on a slot that some other constraint has caused this slot to be invalid (this causes the invalidation to be propagated), and
- Get, which requests the constraint to calculate a new value for the slot.

The slot sends the Get message whenever the value of the slot is requested, and the constraint is expected to generate a response. A constraint always has the option of not returning a value, in which case the slot sends the Get message to a different constraint. If no constraints return a value, then the slot will keep its original value and consider itself valid. The main research questions in this scheme are: in what order will constraints be sent the Get message, and how can multiple solvers coordinate setting the same slot? The policy implemented for Amulet is straight-forward, and just queries the constraints in the order they become invalid. Since practical constraints used for graphics do not usually compete, this policy has proven adequate. As we develop more constraint solvers, we will continue to investigate this issue.

4.3.5 Performance

For formula constraints which are valid (which already have the correct value), getting the value takes the same time as a regular Get, as reported above. The table below shows the times if the value of the slot needs to be recalculated. This includes the time to evaluate the constraint function and maintain the dependencies. We will be investigating why Unix seems so much slower than the other times.

Times in Microseconds	Formula re-evaluation
Unix	37.1
Mac	12.4
PC	19.7

4.4. Opal Output Model

The graphical object layer of Amulet is called Opal, the Object Programming Aggregate Layer. Opal hides the graphics part of Gem and provides a convenient interface to the programmer by using a retained object model, also called a structured graphics model or a display list. The programmer creates instances of the built-in graphical object prototypes, like rectangles, lines, text, circles, and polygons, and adds them to a group or window. Then Amulet automatically redraws the appropriate parts of the window if it becomes uncovered, or if any properties of the objects change. This frees the programmer from having to deal with refresh. Objects can simply be created and deleted and their properties can be set. Furthermore, Opal automatically handles object creation and layout when the data can be displayed as lists or tables.

Opal makes heavy use of the object and constraint models of Amulet. Of course, all graphical objects are Amulet objects. Adding parts to graphical groups simply uses the ORE-level Add_Part routine. Due to structural inheritance, the programmer can simply create instances or copies of groups in the same way as primitive objects, and the object system will automatically make instances or copies of the parts. The properties of objects that programmers do not care about can simply be ignored because they will inherit appropriate default values from prototypes. Due to this integration, simple programs are quite short.

The retained object model allows Amulet to provide many facilities that must be programmed by applications in other toolkits, including automatic refresh, as shown in Figure 7. Amulet uses a relatively efficient algorithm that calculates which objects will be affected when the object is erased, and redraws only the affected objects from back to front. Double buffering is typically used to minimize flicker. The following table shows the time it takes to redraw a small rectangle when its position is changed.

Times in Milliseconds	Double Buffered	Direct Drawing to the screen
Redraw on Unix	1.53	1.05
Redraw on Mac	1.91	1.46
Redraw on PC	2.1	0.53

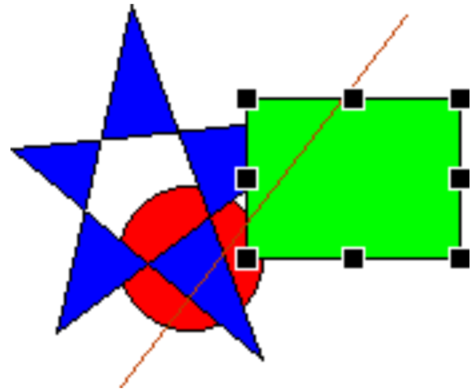


Figure 7: A collection of Opal objects. The color of the star can be changed by simply setting its FILL_STYLE slot, and Amulet will redraw the star and the other objects that overlap it. The Amulet toolkit includes selection handles shown around the rectangle which support selecting, moving and growing one or more objects.

4.5. Interactors

Programming interactive behaviors has always been the hardest part of creating user interface software, especially since most toolkits and window managers only provide a stream of raw input events for each window which the programmer must interpret and manage. Garnet introduced the "Interactor" model for handling input [18] which we refined in Amulet. Each Interactor object type implements a particular kind of interactive behavior, such as moving an object with the mouse, or selecting one of a set of objects. To make a graphical object respond to input, the programmer simply attaches an instance of the appropriate type of Interactor to the graphics. The graphical object itself does not handle input events.

Internally, each Interactor operates similarly. It waits for a particular starting event over a particular object or over any of a set of objects. For example, an Interactor to move the circuit elements of Figure 1 would wait for a left mouse button down over any of the circuit elements. When that event is seen, the Interactor starts running on the particular object clicked on, processing certain events. The moving Interactor processes mouse move events, while looking for a left button up event, or an abort event (usually Control-G, Command-dot, or ESC). While the Interactor is running, the user is supplied feedback, either as a separate object (such as a dotted rectangle following the mouse), or by having the original object itself move. If the Interactor is aborted because the user hits an appropriate key or by a program calls the abort method, the original object is restored to its original state, the feedback object is hidden, and the Interactor goes back to waiting for a start event. If the Interactor completes normally (because the mouse button was released), then the feedback is hidden, the graphical object is updated appropriately, and a "command object" is allocated (see Section 4.7). Interactors are highly parameterized so that the programmer can specify the start, end, and abort events, the objects the Interactor operates over and uses for feedback, along with other aspects such as gridding or how many objects can be selected. As a result, Amulet's six types of Interactors are sufficient to cover all the behaviors found in today's interfaces. Evidence for this claim is that in *none* of the 30 or so applications that have been created so far with Amulet, or the hundreds of applications that were created with Garnet, did programmers ever need to go around the Interactors to get to the underlying Window Manager events.

The six types of Interactors currently in Amulet are:

- **Choice Interactor,** which is used to choose one or more object from a set. The user can move the mouse among the objects (getting interim feedback) until the correct item is found, and then there will often be final feedback to show the final selection. The Choice Interactor can be used for selecting among a set of buttons or menu items, and for choosing among the objects that have been dynamically created in a graphics editor. Parameters to the Choice Interactor include whether a single or multiple items can be selected.
- **One Shot Interactor,** which is used to cause something to happen immediately when an event occurs, for example when a mouse button is pressed over an object, or when a particular keyboard key is hit.
- **Move Grow Interactor**, which is used to have a graphical object move or change size with the mouse. It can be used for dragging the indicator of a scroll bar widget, or for moving and growing objects in a graphics editor. Parameters support gridding and minimum sizes.
- **New Points Interactor**, which is used to enter new points, such as when creating new objects. For example, you might use this to allow the user to drag out a rubber-band rectangle for defining where a new object should go. Parameters include how many points are needed for the object, gridding, and minimum sizes.
- **Text Edit Interactor**, which supports editing the text string of a text object. Parameters include a flexible key translation table so that the programmer can easily modify and add editing functions. The built-in functions support the standard text editing behaviors.
- **Gesture Interactor**, which supports free-hand gestures, such as drawing an "X" over an object to delete it, or encircling the set of objects to be selected. Gestures can be defined by example using the "Agate" gesture trainer (see Figure 8), and can be easily added to conventional direct manipulation interfaces without writing much additional code. For example, adding gestures to the circuit program took about 50

lines of code, most of it to associate the seven gestures with the appropriate existing command. Gesture recognition in Amulet uses an algorithm created by Dean Rubine [28].

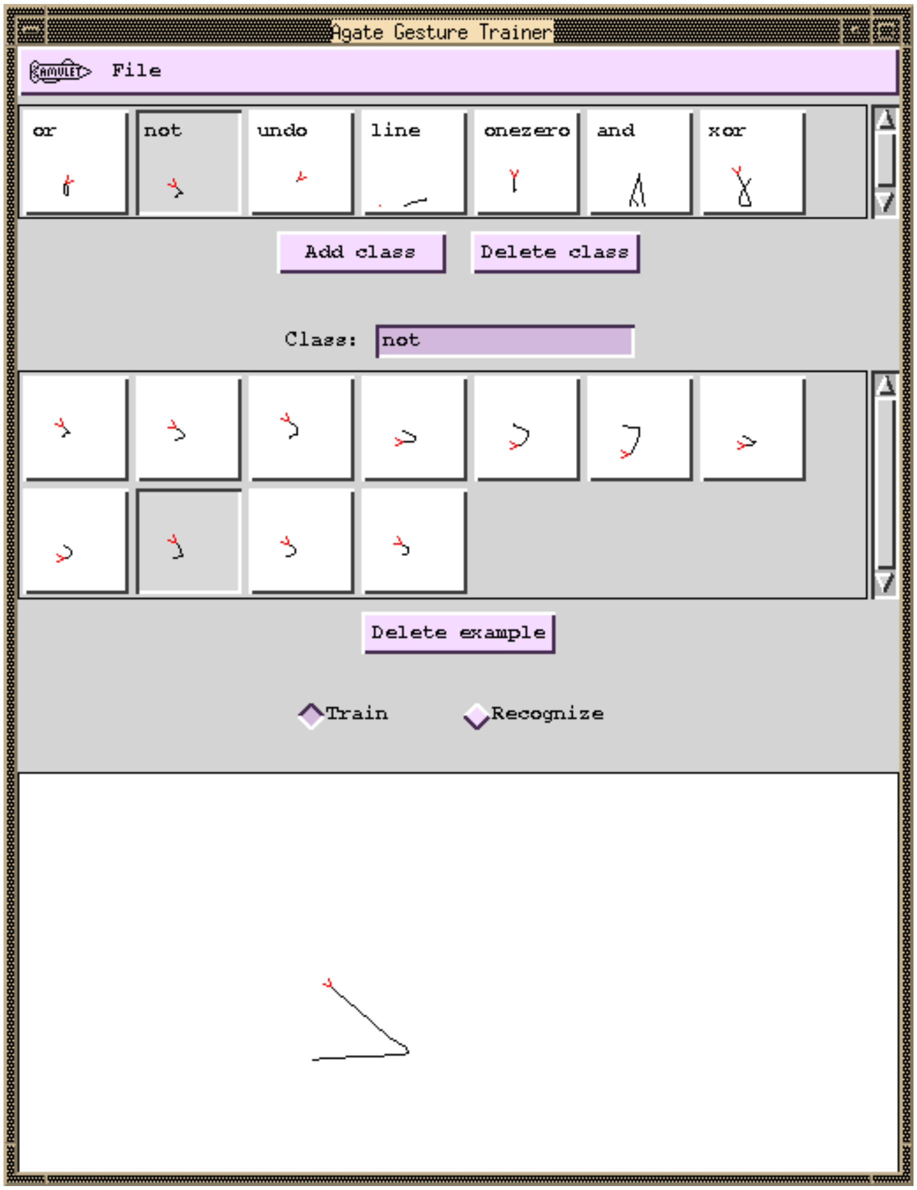


Figure 8: The Agate interactive tool allows gestures to be created by example. Here, the gesture set defined for the circuit program is shown. A file is written describing the gestures which can then be read by a gesture Interactor and used in applications. "Agate" stands for A Gesture-recognizer And Trainer by Example and is based on a similar tool in Garnet [14].

The Interactors are implemented using Amulet objects, so parameters are simply slots the programmer can set, or leave at their default values. Constraints can also be used to compute the parameters. For example, the Am_ACTIVE slot of an Interactor often contains a constraint depending on the global mode, and a constraint in a single move-grow

Interactor might determine whether objects are moved or grown based on which mouse button was held down.

Normally, the Interactor operates on the object it is attached to. An important feature of Amulet's Interactors is that they can also operate on a *set* of objects. For example, the choice Interactor can select among any elements of a group, and the move-grow Interactor can be attached to a window to manipulate any object added as a part of the window. By default, Interactors make this choice based on the type of object they are attached to (group vs. non-group), but the programmer can explicitly specify which is desired.

As an example of the use of Interactors, in the circuit program, the following code is used to create new lines. Note that the constraint line_tool_is_selected is used to make this behavior be available only when the correct tool in the palette is selected.

```
created_objs
    .Add_Part  (Am_New_Points_Interactor.Create("create_line")
          .Set(Am_AS_LINE, true)  //want to create a new line
          .Set(Am_FEEDBACK_OBJECT, lfeedback)//feedback while dragging
          .Set(Am_CREATE_NEW_OBJECT_METHOD, create_new_line)
            .Set(Am_ACTIVE, line_tool_is_selected))
```

The circuit program uses the selection handles widget provided by the Amulet library to select and move objects, so no new code needed to be written for these functions. The following are some examples from other applications that show how Interactors can be used for moving and selecting objects. Note that in the simplest cases, an object can be made interactive with a single line of code:

```
//allow my_object to be moved while the left mouse button is held down
my_object.Add_Part(Am_Move_Grow_Interactor.Create());

//allow any part added to my_group to be grown using the right button
my_group.Add_Part(Am_Move_Grow_Interactor.Create()
                          .Set(Am_GROWING, true)
                          .Set(Am_START_EVENT, "RIGHT_DOWN"));

//allow one or more parts of my_group to be selected with the left button
my_group.Add_Part(Am_Choice_Interactor.Create()
                          .Set(Am_HOW_SET, Am_CHOICE_LIST_TOGGLE));
```

4.5.1 Discussion

The Interactor model is a successful implementation of the "Model-View-Controller" idea from Smalltalk [12]. The model contains the data, the view presents the data, and the controller manipulates the view. Most previous systems, including the original Smalltalk implementation, had the View and Controller tightly linked, in that the controller would have to be reimplemented whenever the view was changed, and vice versa. Indeed, many later systems such as Andrew [26] and InterViews [15] combined the view and controller and called both the "View." In contrast, Amulet's Interactors are independent of graphics, and can be reused in many different contexts.

Another common design in other systems is to just have each graphical object have a standard set of methods or events that it handles, for example for becoming selected and moving. Visual Basic is an example of this design the programmer can code methods that are activated when the user clicks on or drags an object. There are a number of advantages to Amulet's design of having *explicit* objects (the Interactors) representing the behaviors of the graphics. First, it provides significantly greater reuse for such common features as gridding, undo, and enabling and disabling operations, since these are provided in a single place, instead of being re-implemented with each graphical object. Second, being able to analyze, inspect, and manipulate the behavior objects makes debugging and tracing easier, and enables external agents, tutors and alternative interfaces like speech and gestures to control the interface without modifications to the graphical objects or the existing behavior logic.

4.6. Widgets

Amulet supplies a complete set of widgets, including pull-down menus, buttons, check boxes, radio buttons, text-input fields, scroll bars, etc. Each widget has a different drawing routine for the Motif, Microsoft Windows, and the Macintosh look and feel (see Figure 9). Amulet re-implements all the widgets rather than using the built-in widgets from the various toolkits so that we can provide flexibility and control to programmers who want to investigate new behaviors. This is necessary, for example, to create a scroll bar with two handles [1] or to support multiple people operating with a widget at the same time for a multi-user application. Widgets are completely integrated with the object, constraint and command models, so properties of widgets can be computed by constraints, and the actions of widgets are represented by command objects (see section 4.7), so they are easily undone. The various kinds of button and menu widgets can accept strings, bitmaps or arbitrary Amulet objects to display as the labels (most other toolkits only allow strings or bitmaps). This is easy in Amulet since there is a standard way for the buttons to query objects for their size and tell them where to draw. In the circuit program, the *actual* prototypes of the gates are displayed in the button panel, so it was not necessary to construct bitmap pictures of the gates. Since widgets are objects, it is easy to use constraints to compute their parameters. This design is used in many places in the circuit program, for example to enable and disable the widgets based on the appropriate global state, and to lay them out appropriately. Amulet's widgets also have an extra parameter to disable them without greying out, which was added so the actual widgets can be used by interface-builder programs which need the widgets to be selected and moved when clicked on, instead of performing their normal functions. This feature proved useful for the circuit program as well, since the zero-one button prototype is disabled when displayed in the tool panel, but we do not want the number to be grayed out.

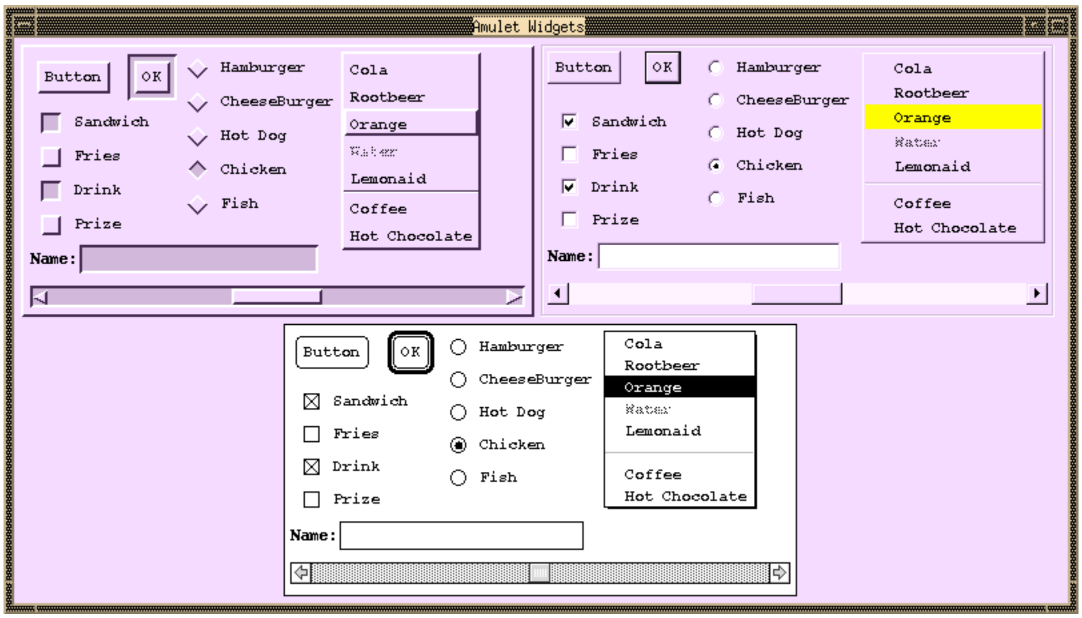


Figure 9: Some of the widgets in Amulet with the Motif, Windows 95, and Macintosh look-and-feel in a window running under Motif. All of the widgets are implemented using Amulet intrinsics, and the choice of the look-and-feel is controlled by a slot which defaults to an appropriate value for the particular machine, but can also be set explicitly.

In addition, Amulet supplies other widgets for the *insides* of application programs. For example, the selections-handles widget implements the familiar squares around the edges of graphical objects that show what is selected and allows the selected objects to be moved and resized (see Figure 7). The circuit program uses this widget as well, but disables the growing of objects, since they must stay the same size. All other toolkits require programmers to re-implement selection handles and all their standard behaviors in every application, but in Amulet, programmers only need to add an instance of this widget to their window .

4.7. Command Objects

Often, the Interactors and widgets operate simply by setting the appropriate slots of objects and having the values computed by constraints. For example, toggling the zero-one button changes its value, and all other values are computed with constraints. In other cases, extra actions are required. Rather than using a "call-back procedure" as in other toolkits, Amulet allocates a *command object* and calls its “Do” method [23]. Amulet’s commands also provide slots and methods to handle undo, selective undo and repeat, enabling and disabling the command (graying it out), help, and “balloon help” messages. Thus, unlike MacApp [33], the command objects provide a single place for describing a behavior.

Furthermore, commands promote re-use because commands for such high-level behaviors such as move-object, create-object, change-property, become-selected, cut, copy, paste, duplicate, quit, to-top and bottom, group and ungroup, undo and redo, and drag-and-drop are supplied in a library and can often be used by applications *without change.* This is possible because the retained object model means that there is a standard way to access and manipulate even application-specific objects. The circuit program uses the standard operations from the library.

The commands in Amulet are *hierarchical*, so that a behavior may be composed of high-level and low-level commands [23]. For example, a scroll bar command might internally use a move-object command. This improves modularity and re-use because each command is limited to its own local actions. This feature was necessary for the circuit program because the built-in Delete commands know how to delete the selected objects, but would not have deleted the attached wires. Therefore, the circuit program needed another command to delete the attached wires, and this was linked to the built-in command. This command contained an undo method to put the wires back.

4.7.1 Undo

All of the built-in operations in Amulet support undo. Thus, if programmers use the standard Interactors and command objects, all operations are automatically undoable without writing any extra code. If the programmer creates custom commands that perform application-specific actions, like deleting the wires when the gates are deleted, then a custom undo method will have to be written as well. However, we have found that the Amulet object and constraint models make writing undo methods very easy since any needed data can be stored as slots in the command objects, and due to constraints, undoing operations is usually only a matter of resetting some slots.

Amulet’s commands also support investigation of various undo mechanisms. Currently, Amulet supplies three different undo mechanisms that the developer can choose from: single undo like the Macintosh, multiple undo like Microsoft Word Version 6 and Emacs, and a novel form of undo and repeat, where any previous command, including scrolling and selections, can be selectively undone, repeated on the *same* object, or repeated on a new selection [23]. Figure 10 shows the experimental dialog box which is supplied to support this new style and allow users to select a previously executed command. Researchers can also create their own undo mechanism and integrate it into the Amulet system.

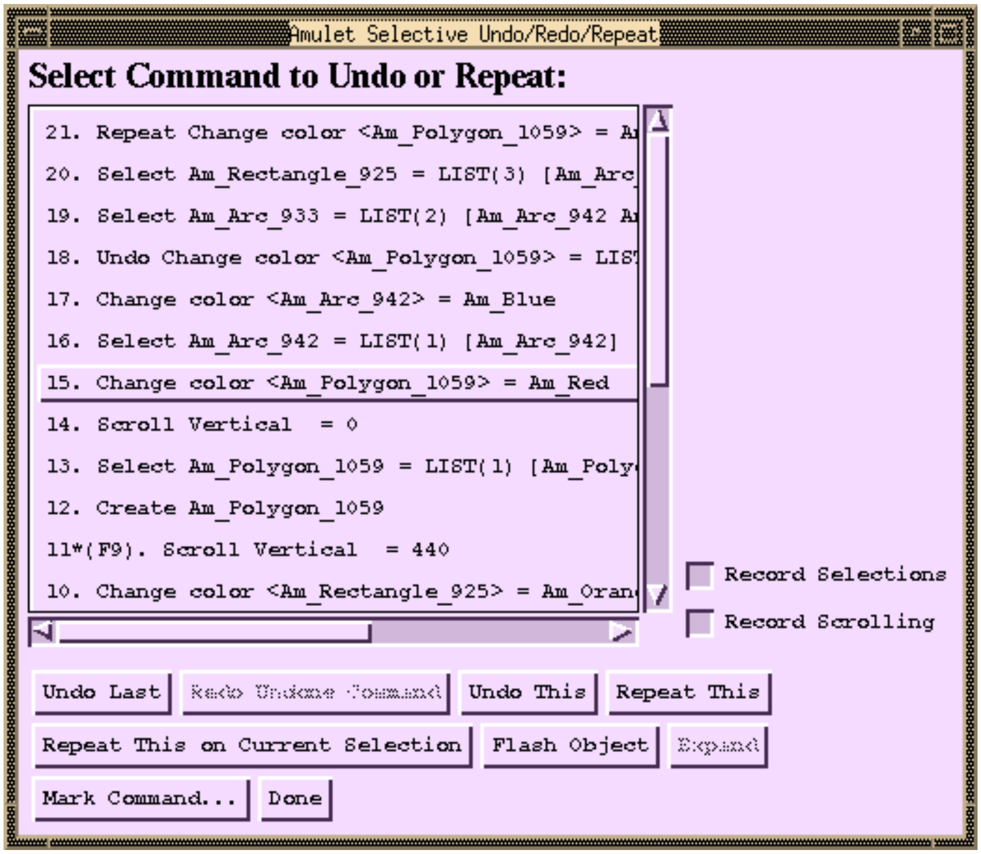


Figure 10: The experimental dialog box that allows users to access the regular undo and redo operations (the first two buttons below the scrolling list). Other buttons operate on the command selected in the list (here, number 15), and will undo it, repeat it, or repeat it on new objects. "Flash Object" shows the object associated with the command. The "Expand" button will allow a command which operates on multiple objects to be separated into separate commands. Commands can be "Marked" to allow them to be repeated with a single keystroke. Command 11 has been marked with the F9 keyboard key as an accelerator. The display of each command shows the action, the name of the objects affected, and the new value. The radio buttons on the right cause scrolling and selection commands to be queued, so that, for example, an accidental deselection of a set of objects can be undone.

5. Debugging Tools

Debugging interactive applications requires additional mechanisms than supplied with conventional development environments. Amulet provides an interactive *Inspector* that displays the object's properties, traces the execution of Interactors, pauses, single-steps and traces animations, and displays the dependencies of constraints (see Figure 11). From the Inspector, programmers can also set breakpoints or have messages printed whenever the value of a slot changes. Furthermore, extensive error checking (when debugging is enabled) and helpful messages make Amulet applications easy to develop and debug. We try to make sure that programmers using Amulet never see “Segmentation fault” or other common but unhelpful C++ error messages.

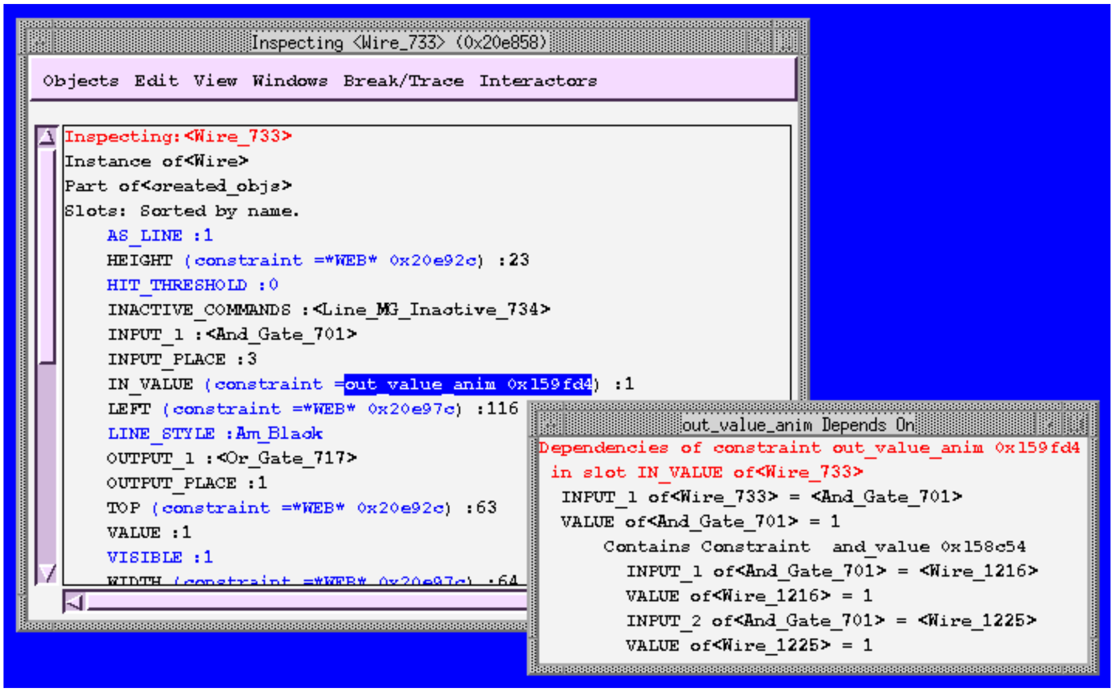


Figure 11: Inspecting a wire in the circuit program, and the constraint in its Am_VALUE slot. The slots which are inherited are shown in blue. Notice that the names of methods, constraints and objects are shown.

6. Status and Future Work

The current version of Amulet (V2.0) has been released for Unix, Windows NT, Windows 95, and the Macintosh (to get Amulet, see http://www.cs.cmu.edu/~amulet). There are over 30 projects all over the world using this version, and Figure 12 shows a few examples of the applications built with Amulet. Version 3, which includes support for animations and the Macintosh and Windows 95 look-and-feel, will be released soon.

In the future, we will be investigating techniques to support speech recognition, 3-D, visualizations, World-Wide Web access and editing, and multiple people operating at the same time (also called Computer-Supported Cooperative Work — CSCW). An important focus will be on *interactive tools* that allow most of the user interface to be specified without conventional programming. Our ultimate goal is to allow the user interface designer to simply *draw* examples of the graphics of the interface, and then *demonstrate* the interactive behaviors to show how the interface should react to the user. The motivation for this is that whereas today’s programming frameworks such as MacApp [33], have demonstrated productivity gains of factors of 2 to 5, interactive tools like HyperCard, the NeXT Interface Builder, and Visual Basic have demonstrated productivity gains of factors of 10 to 50. We want to create an interface builder for Amulet to lay out widgets, which will probably incorporate ideas from SILK, where gestures can be used to sketch interface ideas [13]. Another tool in progress is GAMUT, which is an interactive tool for creating games and educational software by demonstration [17].

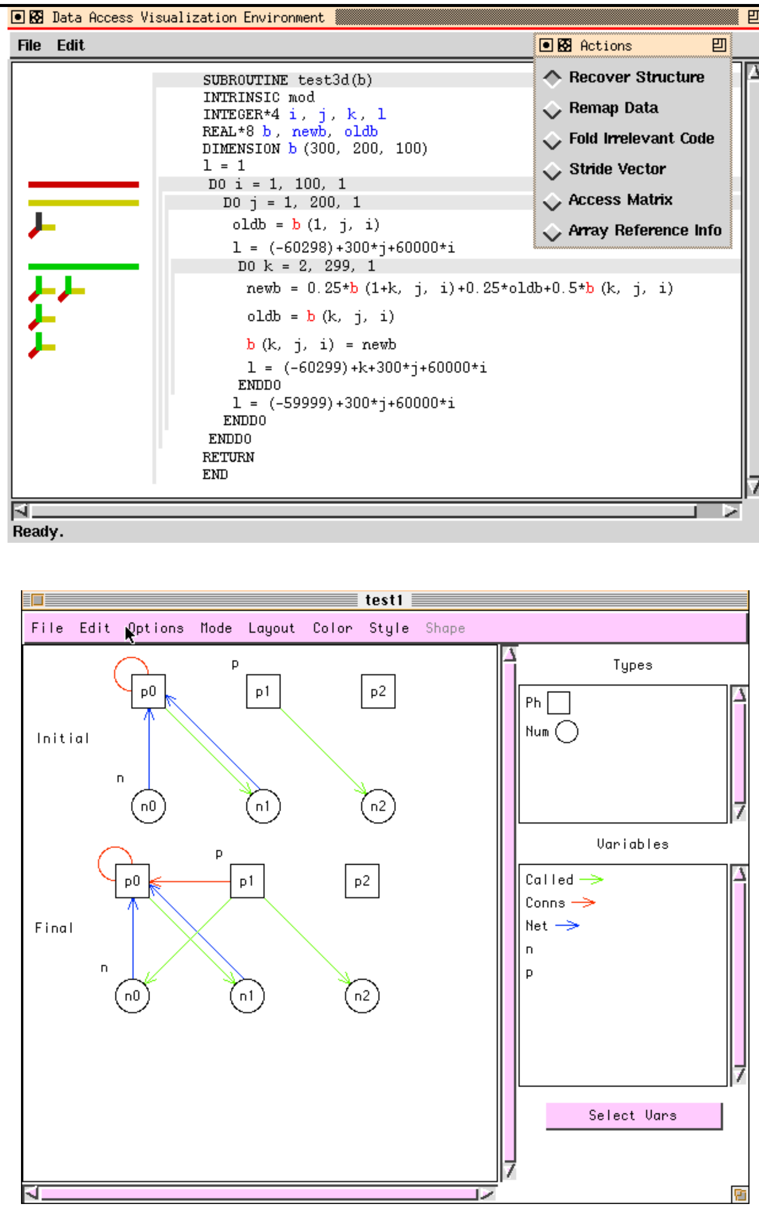


Figure 12: Examples of two applications built using Amulet. The top one is the Data Access Visualization Environment, a program designed to aid programmers in optimizing and parallelizing Fortran programs. (Courtesy of Galen C. Hunt, Computer Science Department, University of Rochester.) The bottom picture is of a system to visualize the counter-examples discovered by the Nitpick analyzer for software specifications in Z, created as a class project by Craig Damon and Geoff Langdale at CMU.

7. Related Work

Amulet builds on many years of work on user interface toolkits (see [19] for a survey). It is primarily influenced by our previous Garnet toolkit [22]. The main other research project investigating the prototype-instance model is SELF [4]. There are many differences between the SELF and Amulet models, however. SELF is its own language, so it does not have to integrate with an existing language. SELF uses a pure copy-down semantics, so after an instance is created, changes to the prototype are not reflected in the instances. Finally, SELF does not support constraints.

There are many research systems which support constraints. The idea for indirect constraints and integrating constraints with a prototype-instance object system originated in Garnet [30]. EVAL/vite [8] integrates constraints with C++ by using a preprocessor and a special sub-language for the constraints. EVAL/vite is a one-way solver like Amulet's formula constraints. MultiGarnet [29] integrated a multi-way solver with Garnet's one-way solver, and inspired Amulet's goal for providing an architecture to make this kind of investigation easier. Rendezvous [7] was designed to help create multi-user applications in Lisp. Like Amulet, Rendezvous allows multiple one-way constraints to be attached to a variable. However, Rendezvous requires that variables be explicitly declared and uses a different implementation algorithm. Also, Rendezvous requires that all side-effects from constraints be deferred. The Artkit toolkit [10] provided a mechanism to support animations, but it did not use the constraint system and it required writing new methods for each object which was to be animated.

Amulet's Interactors model is based on Garnet's [18]. Using command objects to support undo was introduced in MacApp [33] and has been used in many systems including InterViews [32] and Gina [3]. Katie [11] introduced the idea of hierarchical events and explored some implementation issues. There is a long history of research into various new undo mechanisms, and Amulet is specifically designed to allow new mechanisms to be explored. The selective undo mechanism in Amulet is closest to the Gina mechanism [3], but adds the ability to repeat previous commands, and to undo selections and scrolling.

8. Conclusions

We are very excited about the potential for Amulet to be a useful and efficient platform on which to perform user interface research. We hope it will also be popular for user interface education and for the implementation of real systems. The innovations in Amulet and the integration of novel object, constraint, input, output, command, and undo models, make it effective for supporting both today's and tomorrow's user interfaces.

Acknowledgments

We want to thank the many users of Amulet who have helped us find bugs and improve the system. For help with this paper, we would like to thank Bernita Myers.

References

1. C. Ahlberg, C. Williamson, and B. Shneiderman. "Dynamic Queries for Information Exploration: An Implementation and Evaluation," in *Proceedings ACM CHI'92 Conference.* 1992. pp. 619-626.

2. G. Avrahami, K.P. Brooks, and M.H. Brown. "A Two-View Approach To Constructing User Interfaces," in *Proceedings SIGGRAPH'89: Computer Graphics.* 1989. Boston, MA: **23**. pp. 137-146.

3. T. Berlage, "A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects." *ACM Transactions on Computer Human Interaction*, 1994. vol. 1, no. 3, pp. 269-294,

4. C. Chambers, D. Ungar, and E. Lee, "An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes." *Sigplan Notices*, 1989. vol. 24, no. 10, pp. 49-70, ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'89.

5. B.-W. Chang and D. Ungar. "Animation: From Cartoons to the User Interface," in *Proceedings UIST'93: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 1993. Atlanta, GA: pp. 45-55.

6. M. Gleicher. "A Graphics Toolkit Based on Differential Constraints," in *Proceedings UIST'93: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 1993. Atlanta, GA: pp. 109-120.

7. R.D. Hill, "The Rendezvous Architecture and Language for Constructing Multiuser Applications." *ACM Transactions on Computer-Human Interaction*, 1994. vol. 1, no. 2, pp. 81-125,

8. S.E. Hudson, "A System for Efficient and Flexible One-Way Constraint Evaluation in C++," 1993, Graphics Visualizaton and Usability Center, College of Computing, Georgia Institute of Technology: 10.

9. S.E. Hudson and I. Smith. "Ultra-Lightweight Constraints," in *Proceedings UIST'96: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 1996. Seattle, WA: pp. To appear.

10. S.E. Hudson and J.T. Stasko. "Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions," in *Proceedings UIST'93: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 1993. Atlanta, GA: pp. 57-67.

11. D.S. Kosbie and B.A. Myers. "Extending Programming By Demonstration With Hierarchical Event Histories," in *Human-Computer Interaction: 4th International*

Conference EWHCI'94, Lecture Notes in Computer Science, Vol. 876,. 1994. Berlin: Springer-Verlag. pp. 128-139.

12. G.E. Krasner and S.T. Pope, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system." *Journal of Object Oriented Programming*, 1988. vol. 1, no. 3, pp. 26-49,

13. J. Landay and B.A. Myers. "Interactive Sketching for the Early Stages of User Interface Design," in *Proceedings SIGCHI'95: Human Factors in Computing Systems.* 1995. Denver, CO: pp. 43-50.

14. J.A. Landay and B.A. Myers. "Extending an Existing User Interface Toolkit to Support Gesture Recognition," in *Adjunct Proceedings INTERCHI'93: Human Factors in Computing Systems.* 1993. Amsterdam, The Netherlands: pp. 91-92.

15. M.A. Linton, J.M. Vlissides, and P.R. Calder, "Composing user interfaces with InterViews." *IEEE Computer*, 1989. vol. 22, no. 2, pp. 8-22,

16. R. McDaniel and B.A. Myers, "A Dynamic And Flexible Prototype-Instance Object And Constraint System In C++," 1995, Carnegie Mellon University Computer Science Department: also Human Computer Interaction Institute CMU-HCII-95-104.

17. R.G. McDaniel. "Improving Communication in Programming-by-Demonstration," in *Conference Companion for CHI'96: Human Factors in Computing Systems.* 1996. Vancouver, BC, Canada: pp. 55-56.

18. B.A. Myers, "A New Model for Handling Input." *ACM Transactions on Information Systems*, 1990. vol. 8, no. 3, pp. 289-320,

19. B.A. Myers, "User Interface Software Tools." *ACM Transactions on Computer Human Interaction*, 1995. vol. 2, no. 1, pp. 64-103,

20. B.A. Myers, "The Amulet V2.0 Reference Manual," 1996, Carnegie Mellon University Computer Science Department:

21. B.A. Myers, D. Giuse, and B. Vander Zanden, "Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods." *Sigplan Notices*, 1992. vol. 27, no. 10, pp. 184-200, ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'92.

22. B.A. Myers, "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces." *IEEE Computer*, 1990. vol. 23, no. 11, pp. 71-85,

23. B.A. Myers and D. Kosbie. "Reusable Hierarchical Command Objects," in *Proceedings CHI'96: Human Factors in Computing Systems.* 1996. Vancouver, BC, Canada: pp. 260-267.

24. B.A. Myers. "Easily Adding Animations to Interfaces Using Constraints," in *Proceedings UIST'96: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 1996. Seattle, WA: pp. To appear.

25. B.A. Myers and M.B. Rosson. "Survey on User Interface Programming," in *Proceedings SIGCHI'92: Human Factors in Computing Systems.* 1992. Monterey, CA: pp. 195-202.

26. A.J. Palay. "The Andrew Toolkit - An Overview," in *Proceedings Winter Usenix Technical Conference.* 1988. Dallas, Tex: pp. 9-21.

27. R. Pausch, N.R. II Young, and R. DeLine. “SUIT: The Pascal of User Interface Toolkits,” in *Proceedings UIST’91: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 1991. Hilton Head, SC: pp. 117-125.

28. D. Rubine. “Specifying Gestures by Example,” in *Proceedings SIGGRAPH’91: Computer Graphics.* 1991. Las Vegas, NV: **25**. pp. 329-337.

29. M. Sannella. “SkyBlue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction,” in *Proceedings UIST’94: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 1994. Marina del Rey, CA: pp. 137-146.

30. B. Vander Zanden, “An Incremental Algorithm for Satisfying Hierarchies of Multi-way, Dataflow Constraints,” 1995, Computer Science Department, University of Tennessee: 55.

31. B. Vander Zanden, “Integrating Pointer Variables into One-Way Constraint Models.” *ACM Transactions on Computer Human Interaction*, 1994. vol. 1, no. 2, pp. 161-213,

32. J.M. Vlissides and M.A. Linton, “Unidraw: A Framework for Building Domain-Specific Graphical Editors.” *ACM Transactions on Information Systems*, 1990. vol. 8, no. 3, pp. 204-236,

33. D. Wilson, *Programming with MacApp.* 1990, Reading, MA: Addison-Wesley Publishing Company.