

The Prototype-Instance Object Systems in Amulet and Garnet

Brad A. Myers, Rich McDaniel, Rob Miller,
Brad Vander Zanden, Dario Giuse, David Kosbie and Andrew Mickish

Final Draft

May 6, 1998

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>

To appear in:

James Noble, Antero Taivalsaari and Ivan Moore, eds.,
Prototype Based Programming, Springer-Verlag, 1998.

Abstract

Over the last 10 years, the CMU User Interface Software Project has been investigating prototype-based programming in two large-scale systems: Garnet in Lisp and Amulet in C++. The goal of these systems is to provide an effective way to prototype and implement user interface software. In addition to using a prototype-instance object model, these systems also use *constraints* to tie objects' values together, and new models for input and output. The result is a significantly different style of programming than conventional class-based object systems, and even than other prototype-based systems.

Copyright © 1998 — Carnegie Mellon University

This research was sponsored by Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597, and 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.

1. Introduction

We have built two user interface development environments that use prototype-instance object models. Garnet [Myers 1990b], is in Lisp and was started in 1988. Amulet [Myers 1997], is in C++ and was started in 1994. The goal of these systems is to investigate ways to make the creation of user interface software significantly easier, especially for highly-interactive, graphical, direct manipulation user interfaces. These systems also use *constraints*, which are relationships that are declared once and maintained by the system, to tie objects' values together. Amulet and Garnet also include many other innovations, which are described in various papers [Myers 1989][Myers 1990a][Vander Zanden 1990][Myers 1991b][Myers 1991a][Hashimoto 1992][Myers 1994][Vander Zanden 1994][Vander Zanden 1995b][Myers 1996a][Myers 1996b][Myers 1998]. This chapter concentrates on the prototype-instance object systems in Garnet and Amulet.

In Garnet's and Amulet's prototype-instance object systems, there is no concept of a "class" since every object can serve as a prototype for other objects. Values of objects are stored in "slots" (sometimes called "fields" or "instance variables") each of which has a name and can hold a value of any type. Any slots that are not declared locally in an object are inherited from the prototype. Another important feature of these object systems is that there is no distinction between methods and data: individual objects can override an inherited method in the same manner as inherited data (unlike class-instance systems where a different method requires a sub-class, whereas each instance can have different data). The object systems are *dynamic* in that slots in objects can be created, set, and destroyed at run time, and the types of values in slots can also change (for example, the value in a slot can change from a string to a float). The object systems also support a part-owner hierarchy, by which objects can be grouped together. For instance, the graphics in a window are added as *parts* of the window.

Amulet and Garnet also provide *constraints* which can be arbitrary code. Any slot of any object can contain a constraint, rather than a regular value. We have found that the style of writing code in these systems is quite different than in other systems. By writing constraints, the programmer can specify relationships that should hold in the interface in a *declarative style*, and leave it up to the system to maintain them.

In the terms of the "Treaty of Orlando" [Stein 1989], our object systems can be classified as a prototype-instance model with dynamic, implicit, per-object sharing. It is dynamic because the inheritance can be changed at any time, implicit because objects inherit from their prototypes and you cannot explicitly declare how slots are inherited

(except by using constraints), and per-object because there is no such thing as classes. The “templates” (prototypes) are entirely “non-strict,” which means that an instance can gain or lose slots at any time.

For Amulet, we took the opportunity to fix a number of problems we experienced with Garnet, and Amulet also contains a number of important innovations, including incorporating the part-owner hierarchy into the object system, control over the inheritance of slots, support for multiple constraint solvers, and a flexible demon mechanism. In addition, Amulet’s default constraint solver is more flexible than other one-way systems. Finally, it is interesting to note that we were able to provide dynamic slot typing, a dynamic prototype-instance system, and constraints in C++ without using a pre-processor or a scripting language.

2. Summary of the Garnet and Amulet Systems

The Garnet and Amulet user interface development environments aim to make the design, prototyping, implementation, and evaluation of user interfaces significantly easier, while supporting flexible experimentation with new styles of interaction. The term “user interface development environment” is used to signify that these systems are more than just a collection of widgets (also called “controls”) like menus, buttons and scroll bars, like the Macintosh or Motif widget sets. Instead, Garnet and Amulet are higher-level tools that provide support for creating the complete application, and therefore might be classified as “application frameworks” comparable to MacApp [Wilson 1990] or the Microsoft Foundation Classes. For a complete discussion of user interface tools, see [Myers 1995].

Amulet and Garnet include a number of design and implementation innovations including new models for constraints, objects, input, output, commands, undo, and animation. An important goal is to support new kinds of *interactive tools* where the designer can create much of the user interface for a system by direct manipulation or by demonstration [Myers 1992] rather than by writing code. For example, we created a variety of “interface builders” which allow widgets to be laid out interactively using the mouse, and their parameters can be set, in the style of Visual Basic. Other tools allow constraints and behaviors to be demonstrated without writing code.

Garnet, which stands for Generating an Amalgam of Real-time, Novel Editors and Toolkits, is in Common Lisp and runs on Unix X/11 and the Macintosh. Amulet, which stands for Automatic Manufacture of Usable and Learnable Editors and Toolkits, is in

C++ and runs on Unix X/11, Microsoft Windows 95 and Window NT, and the Macintosh.

We have three main target audiences for these systems. First, for user interface researchers (like ourselves), they should be very flexible and effective tools so parts can be replaced and new technologies and widgets can be easily created and evaluated. For example, we have made it easy to investigate new kinds of widgets and new kinds of interactive tools. The second goal is to be useful for students, which means that the systems should be easy to learn. Finally, the systems should provide sufficient performance, robustness and documentation so they will be useful for general user interface developers.

We distribute the systems for use by others because we feel this will help to demonstrate that the innovations we have incorporated are sound and effective, and will hopefully facilitate technology transfer. Both systems are in the public domain and can be used for free. The distributions include the complete source code, manuals and tutorials. To get Garnet, see <http://www.cs.cmu.edu/~garnet>. Note that Garnet is no longer being supported. To get Amulet, see <http://www.cs.cmu.edu/~amulet> or send mail to amulet@cs.cmu.edu. Amulet is being downloaded about 1000 times a month.

3. Why a New Object System?

We did not set out to investigate different object models, but we decided that existing object systems were not sufficiently flexible to support our requirements for user interface tools. When we started Garnet, the standard Common Lisp Object System (CLOS) was not yet available, and after it appeared, we decided that it did not fully meet our needs anyway. For Amulet, we found the C++ object system was much too restrictive.

Our particular requirements for an object system are:

- **Class creation and reflection at run-time.** Since an important goal of these systems is to support interactive tools, it is required that the tools be able to generate and modify all properties of the user interface elements easily. The ability to query objects' properties is sometimes called "reflection" [JavaSoft 1996]. C++ now has a reflection capability (still not implemented by many compilers) called "real-time type information" (RTTI) but there is no ability to modify or create classes at run time.
- **Querying the names of slots.** The tools often do not know *a priori* which slots of an object are important. This information might be stored with the object itself.

For example, the “slots-to-save” slot of an object tells the system which slots of that object should be written to the disk when the object is saved. The system iterates through that list of slots, and then gets the value of each slot in the list and writes the value to a file. Another example is the “Inspector” for debugging programs which displays all the slots of any kind of object. In C++, there is no portable way to access an instance variable of an object without knowing all the names at compile time.

- **Dynamic typing.** Sometimes the label for a button or menu item is a string, and other times it is a bitmap or other picture, or even a combination of a picture and a string. Most toolkits find this difficult to support since there must be a specific type for the label parameter. We wanted instead to be flexible and allow any type to be supplied. Similarly, the value returned when the user selects an item in widgets such as menus should be able to be a string, a number (index) or an arbitrary other value. Lisp provides this dynamic typing, but C++ does not.
- **Different methods in instances.** Both CLOS and C++ require that every instance of a class have the same methods, although the data in each instance can be different. A new sub-class must be created to have a different method. We wanted to treat methods similarly to data, so that an instance could have a different method than its prototype.
- **Support for constraints.** Another important motivation was to be able to easily integrate a constraint system with the object system. Although it would be possible to add constraints to C++ or CLOS objects, most systems that do this (e.g., [Hudson 1993a][Hill 1994]) have required the use of pre-processors or special-purpose constraint languages. In contrast, we wanted to support arbitrary code in constraints without using a pre-processor. Additionally, C++’s lack of dynamic typing makes it cumbersome to implement a constraint system in C++.

4. Object System Design

Both Garnet and Amulet are layered systems with many different parts. The object system part of Garnet is called “KR,” because it is based on a Knowledge Representation language [Giuse 1989]. The object system part of Amulet is called “Ore,” which stands for Object Registering and Encoding.

An important goal for Garnet and Amulet is to be easy to learn and use for developers, even though they have a large number of features. Therefore, we have tried to provide a uniform structure based on a few simple concepts. The main concept is that everything is represented as an object which has a set of slots. The objects support completely dynamic

redefinition of prototypes with automatic change propagation to the prototype's instances. An instance can add any number of new slots, and slots that are not overridden in an instance inherit the values from the prototype. In fact, the inheritance can change dynamically, as an object can add or remove slots at any time. There is no distinction between data and method slots. Any slot can hold any type of value, and a method is just a type of value. This allows the methods that implement messages to change dynamically, which is not possible in conventional object systems like C++. The ability to dynamically add, delete, and modify methods has proven important in graphical interface builders since they often will temporarily insert their own methods during “build” mode, and then remove them during “test” mode.

A new object is created by making an *instance* of another object, which is called the *prototype*. An instance starts off inheriting everything from the prototype, and the slots can then be set with new values. Alternatively, an object can be *copied* (or “cloned”), in which case it immediately gets a copy of all values in the original.

In Garnet, slot names are Common Lisp *keywords*, so they start with colons, and can contain any number of printable characters (e.g., `:left`, `:interim-selected`, `:obj-over`). In Amulet, slots are variables or constants which evaluate to short integers. For example, `Am_LEFT`¹ is a pre-defined slot name which evaluates to the index 100. Garnet uses a hash table internally to map the slot names into a reference to the actual slot structure, and Amulet just uses a linear search through the slot list associated with each object.

For example, the following Amulet code creates an `And_Gate` as an instance of the built-in `Am_Bitmap` object, and then sets the `Am_IMAGE` slot to the appropriate picture.

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

The equivalent code in Garnet would be²:

```
(create-instance 'and-gate opal:bitmap
 (:image and-bitmap-image))
```

There is nothing special about the objects in the library (like the bitmap object used above). Any object can serve as a prototype from which to create other objects. For example, the following Amulet code creates an instance of the `And_Gate` and then puts it in a particular place:

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

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

²“Opal” is the Garnet package that exports the graphical objects.

We allow the `Set` operations to be chained together in Amulet, 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);
```

In Garnet, the `create-instance` call is actually a complex macro, so it also allows multiple slots to be set at the same time:

```
(create-instance 'new-gate and-gate
 (:left 10)
 (:top 43))
```

Slots can be set with any type of value, as the following Amulet code illustrates:

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

The following sections elaborate on these features.

4.1 Slot Inheritance

When an instance of an object is created, the slots that are not set locally inherit their values from the prototype object. 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_IMAGE` slot in the `And_Gate` created above will change the look in all instances. However, changing the `Am_LEFT` slot of the `And_Gate` will not affect `new_gate` since it has a local value for `Am_LEFT`.

If the programmer does not want standard slot inheritance, then that slot can 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. (*Copy* inheritance was not supported by Garnet.) Alternatively, a slot can be declared to be *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 one object. For example, the `drawonable` slot of a `Window` object holds the machine-specific pointer to the underlying window-manager window, and should not be inherited by instances of the window, so it is declared to be local. Each instance of a `Window` has to create and assign its own value for the `drawonable` slot.

The inheritance mechanism is an important distinction from other prototype-instance object systems, such as `Self` [Ungar 1987][Chambers 1989], in which all the slots are always *copied* into instances so changes to prototypes never affect instances. Although Amulet's model requires 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. Many Garnet and Amulet applications have taken advantage of this capability.

4.2 Methods

An important feature of Garnet's and Amulet's object systems is that there is no distinction between the inheritance for *methods* and *data*: any instance can override an inherited method in the same manner as inherited data. In a conventional class-instance model such as Smalltalk, C++, or CLOS, instances can have different data, but only subclasses 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, or create a new subclass and a single instance of that subclass each time. 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 and Garnet, the draw method and the callback use the same mechanism. Methods can be set into a slot in the same way that data is set into a slot.

In Garnet, methods are simply function pointers that are set into slots. We did this in early versions of Amulet, but it resulted in too many "Segmentation Faults" because there was no type checking. This is not a problem in Garnet because Common Lisp does more run-time checking. Originally, if the programmer passed the wrong number or wrong type of parameters to a function, there was no way for Amulet to find out. Also, Amulet could not display meaningful names for the methods. Therefore, we developed a macro for defining methods, for example:

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

The `Am_Define_Method` macro creates a new method object of the type of the first argument (here `Am_Object_Method`). This method object invokes a function which returns the type of the second argument (here `void`). The method object is assigned to a variable with the name of the third argument (here `clear_selection`). The function takes as parameters the list in the fourth argument (which must be specified in an extra set of parentheses). The `Am_Define_Method` macro then defines a function for the method to invoke, and the body of the function follows the macro. The name of the method (here "clear_selection") is stored into a slot of the method object as a string to support debugging, and the compiler can perform full type-checking on the parameters. The compiler will make sure that a method of type `Am_Object_Method`

is being called and that the correct types of parameters are being passed. The syntax to call a method in Amulet is:

```
Am_Object_Method amethod = obj.Get(Am_DRAW_METHOD);
amethod.Call(obj);
```

4.3 Part-Owner Hierarchy

An innovation in Garnet and Amulet is the support for “structural inheritance,” where the *parts* of an object can also be inherited, not just the values. In Garnet, this is supported by special aggregate objects, and all of the parts of an aggregate had to be graphical objects. In Amulet, the part-owner hierarchy is available for *any* type of object, and we have found many uses for the part-owner hierarchy that are independent of graphical relationships. When an instance is made of an object which contains parts, the new instance will have instances of all the prototype’s parts, as shown in Figure 1. This can be distinguished from “normal” slots, where just the value of the slot is copied, so if a normal slot contains a reference to an object *obj1* which is *not* a part, then the instance will contain another pointer to the *same* *obj1* object, rather than having a new object created as an instance of *obj1*.

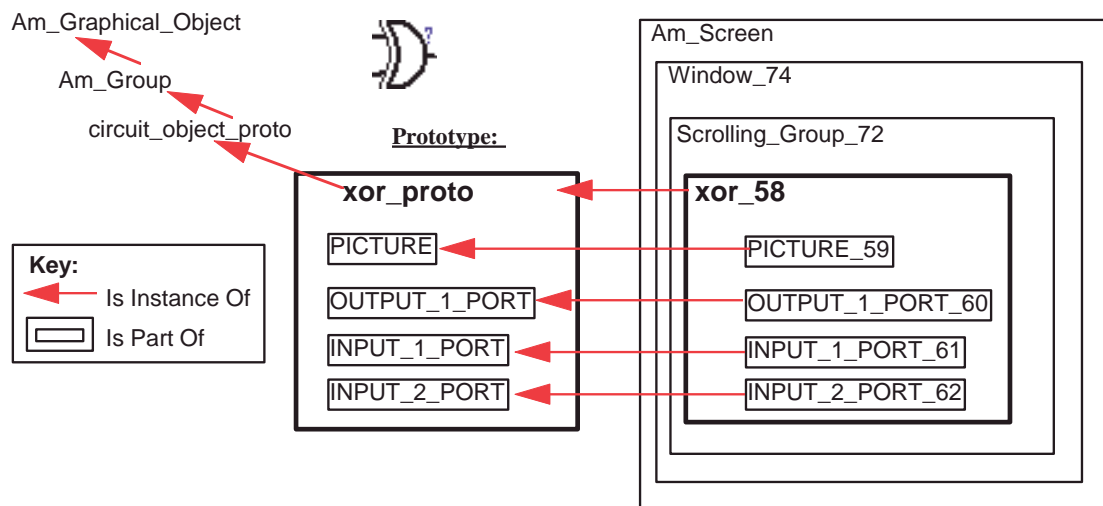


Figure 1: The `xor_proto` object contains 4 parts: a bitmap called `PICTURE`, and output and input ports which are instances of lines. The `xor_proto` object is an instance of the `circuit_object_proto` object, which in turn is an instance of an `Am_Group`, which is an `Am_Graphical_Object`. When an instance is made of the `xor_proto`, 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.

Structural inheritance is an extremely powerful abstraction, because it provides *encapsulation* for the implementation of objects. The user does not need to know whether an object being instanced is a primitive object like `rectangle` or a composite like `button`. In both cases, the create call is the same, and the system will make sure that the instance has the same structure as the prototype. Changing the `And_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 need to change. Thus, the implementation of an object can be changed without changing the code that uses the object. Programmers also do not need to write clone methods or copy constructors.

There is no inheritance of values from the owner to the parts. For example, the default color of a part is inherited from the part's prototype, *not* its owner. Some systems, such as FormsVBT [Avrahami 1989] have hard-wired some slots to inherit values from their prototypes and others to inherit from their owners. Because the constraint mechanism (see below) 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.

The syntax for creating parts is different in Garnet and Amulet. The `create-instance` macro in Garnet was enhanced to know about parts, and appropriate objects are created. For example, a button might be composed of three rectangles and a text object, as shown in Figure 2. The programmer can declaratively list these as part of a button. Then, when the user creates `my-button1` using `button` as the prototype, instances are automatically created of the three rectangles and the text. Of course, any of the parts could themselves be groups, and the instancing would be applied recursively. Constraints (described below) are used to declare how the properties of the components are connected. An outline of the Garnet code is:

```
(create-instance 'button aggregate
  (:parts
    `((:top-edge ,rectangle ....) ;white left & top edges
      (:bottom-edge ,rectangle ....) ;black right & bottom
      (:fill-inside ,rectangle ...) ;gray interior
      (:label ,text ... ;string inside button
        (:string "Label")))))

(create-instance 'my-button1 button
  (:left 100)(:top 5)(:string "First"))
(create-instance 'my-button2 button
  (:left 100)(:top 35)(:string "Second"))
(create-instance 'my-button3 button
  (:left 100)(:top 65)(:string "Third"))
```

A problem with this macro is that users are always getting messed up with quotes, backquotes and commas, and the wrong number of parentheses. In Amulet, `Add_Part`

calls can be chained just the same way as regular Set calls, which makes it much easier to use and learn. The following is an outline of the Amulet code:

```
Am_Object Am_Button = Am_Group.Create("Button")
    .Add_Part(Am_TOP_EDGE, Am_Rectangle.Create() ...)
    .Add_Part(Am_BOTTOM_EDGE, Am_Rectangle.Create() ...)
    .Add_Part(Am_FILL_INSIDE, Am_Rectangle.Create() ...)
    .Add_Part(Am_LABEL, Am_Text.Create() ...
        .Set(Am_STRING, "Label"))

Am_Object my_button1 = Am_Button.Create()
    .Set(Am_LEFT, 100) .Set(Am_TOP, 5) .Set(Am_STRING, "First");
Am_Object my_button2 = Am_Button.Create()
    .Set(Am_LEFT, 100) .Set(Am_TOP, 35) .Set(Am_STRING, "Second");
Am_Object my_button3 = Am_Button.Create()
    .Set(Am_LEFT, 100) .Set(Am_TOP, 65) .Set(Am_STRING, "Third");
```

A feature of the Garnet and Amulet object systems is that edits made to the prototype are automatically reflected in all instances. For example, if the color of `fill-inside` were changed in the `button`, it would automatically also change in `my-button1` and all the other instances (see Figure 3). More significantly, if a part is added or removed from the prototype, then the corresponding object will be added or removed from all instances³. For example if `top-edge` was removed from `button`, then the appropriate rectangle would also be removed from `my-button1` and the other instances. Garnet and Amulet store pointers in each prototype to all instances to support these operations.

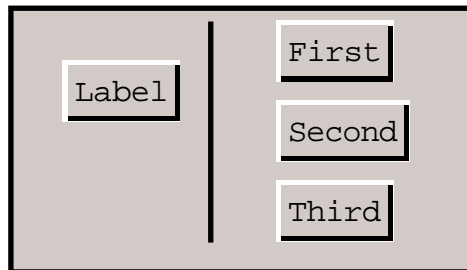


Figure 2 : A prototype button (shown on the left) and some instances created from it.

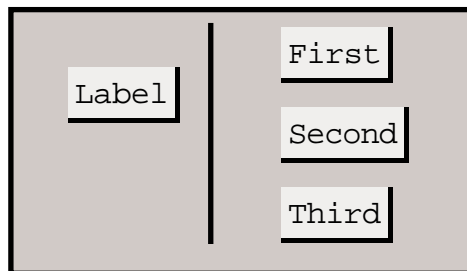


Figure 3 : When the color of the `fill-inside` rectangle is changed to light-gray and the `top-edge` rectangle is removed from the `button` prototype in Figure 2, these changes propagate automatically to the instances.

³Creating and removing parts from the instances when prototypes are edited is not fully implemented in Amulet.

Similarly, if the programmer wants to create an object which is a slight modification of an existing object, it is only necessary to override the divergent parts. In a conventional object system, the programmer would instead be required to rewrite the entire `draw` method (and probably the `erase` and many other methods as well). Here, only the specific parts to be changed need to be mentioned, and only in the object definition.

A very significant advantage of this technique is that it is possible to provide graphical, interactive tools that will create the graphical objects. For example, Lapidary in Garnet [Vander Zanden 1995b] allows programmers to draw pictures of new widgets (like the buttons above) and of new application-specific prototypes. The interface of Lapidary is much like a conventional drawing program like MacDraw. The programmer can specify which slots will be parameters (for the button, they might be the position and string). Interactive behaviors and relationships among the components can also be defined. Note that unlike other interactive interface builders, such as the NeXT Interface Builder, Lapidary allows the designer to define *entirely new* objects, not just to choose pre-defined objects from a palette. The various features of the object system make this much easier to implement. In particular, prototypes can be constructed interactively using Lapidary, and then instances can be immediately created of the prototypes, without having to compile the prototype description to create a class, as would be necessary in most other object systems. Therefore, in the same Lapidary session, the user could create a prototype and then create a scene using instances of the prototype.

Since edits to a prototype are reflected in its instances, it is even possible to interactively change the appearance of objects *while* they are being used in an application. When the prototype is changed, all of the instances are updated immediately, even if they appear inside an application that is currently running. This helps achieve the goal of supporting rapid prototyping of interfaces, since the designer can see the results of the edits in context. In conventional models, it would usually be necessary to stop and recompile to see the results of edits.

Garnet and Amulet also provide a special form of group that computes its parts dynamically. Called an `aggrelist` in Garnet and an `Am_Map` in Amulet, these groups build the list of parts based on a single prototype object, called the “item prototype.” Typically, a list of strings, objects, commands, or something else is provided, and the `aggrelist` or `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 defines a constraint somewhere in the item prototype that depends on the particular value copied from the list. Note that due to the flexibility of the 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.

4.4 Creating New Slots

One problem in Garnet and early versions of Amulet is that users would accidentally set and get the wrong slots. In Garnet, setting a slot that was not there just created it, and getting a slot that was not there returned NIL. Often, this resulted in hard-to-find bugs when slot names were spelled wrong, or the right slot name was used, but in the wrong object. We addressed this problem in the current version of Amulet by requiring all slot names to be declared first, so the compiler insures that all slot names are spelled correctly, and by checking on Get and Set that the slot already exists. In Amulet, special forms of Get and Set must be used when the slot might not exist already, to make sure that the programmer is not just accidentally accessing the wrong slot. It is also possible in Amulet to declare slots as read-only, so their values will not be set accidentally.

4.5 Other features

The object systems also contain many other features that may be useful for programmers. In Amulet, we added automatic memory management using a reference counting scheme for objects and “wrappers” (used to “wrap” C++ types so they can be put into Amulet objects with full type-checking). Of course, this is not necessary in Lisp for Garnet.

A flexible “demon” mechanism allows procedures to be attached to objects or slots for invocation when the slots change. This mechanism enables Garnet and Amulet to redraw objects when their graphical properties change. Programmers can also create their own demons in Amulet (in Garnet, the demon mechanism was not extensible). Type checking of slots is supported, so that programmers can declare that a slot can only hold a specific kind of value. A complete set of querying functions allows objects’ properties to be examined at run-time. These are used by the debugging facilities, and they can also be useful for application programs.

4.6 Performance

The main disadvantage of the prototype-instance model over the conventional class-instance model has been performance. When a slot is 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. We have investigated various indexing and hashing schemes to help reduce this overhead,

including hardwiring some common slots, but this increases complexity and sometimes does not improve performance. 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 [Ungar 1987][Chambers 1989] uses extensive compiler techniques to try to remove some of this search, but we have not found this necessary. An important difference between Self and Garnet/Amulet is that Self uses the prototype-instance system for *everything*, right down to integer arithmetic, whereas we use the efficient underlying C and C++ mechanisms for basic computation and only use the prototype-instance model for the user interface portion. The performance of our object systems is quite good in the typical user interfaces applications created with them. There are no noticeable delays for normal size programs on a variety of modern hardware 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.

4.7 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 our object system provides significantly more flexibility and capabilities.

Another advantage of our prototype-instance object systems over C++ is the ability to treat "classes" as first-class objects. In C++, one cannot store a class object in a variable so that different kinds of objects can be created at run-time. For example, C++ does not allow code like:

```
obj_to_create = Rectangle;           //NOT ALLOWED IN C++
new_object = new obj_to_create;     //NOT ALLOWED IN C++
```

Instead, the operand of the new operator must be a fixed class, leading to large case statements and other inflexible and error-prone constructs. In contrast, since one can create an instance of any object in a prototype-instance object system, one can store a reference to an object in a variable and later use the variable to create a new object:

```
Am_Object obj_to_create = Am_Rectangle; //Typical Amulet code
Am_Object new_object = obj_to_create.Create();
```

Although designed to support the creation of graphical objects, many 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. Our objects are somewhat like “frames” used by artificial intelligence systems, so AI applications may find the model familiar. The constraint system also helps to maintain data dependencies and consistency in application-specific data structures.

5. Constraints

A very significant difference between the Garnet and Amulet object systems and others is the support for *constraints*, which are relationships that are declared once and then maintained by the system. Constraints result in a quite different programming style, and they influence other aspects of the object system design as well.

Garnet and Amulet both support *formula* constraints, which act like spreadsheet formulas. 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. If the other objects’ values have *not* changed, then the formula is *not* re-evaluated and a cached value is returned instead. Thus, the constraints are primarily “one-way.” Formula expressions can contain arbitrary code. Amulet also supports multiple *kinds* of constraints, including multi-way constraints and animation constraints, which are discussed below.

Although many other research systems have provided constraints, Garnet was the first to truly integrate them with the object system and to make them general purpose. They are also well-integrated in Amulet. An important result of this is that constraints are used throughout the systems in many different ways. For example, the built-in `text` object has constraints in its width and height slots that compute its dimensions based on the current string and font. Garnet’s implementation of a Motif radio button widget uses 58 constraints internally, and the Lapidary graphical editor, which is a large and complex application in Garnet, contains 16,700 constraints. Many of these are only evaluated once when the system starts up, however.

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.

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.

Formula constraints can contain arbitrary source code. In Garnet, a different version of `Get` is used (called `gv`) inside of a constraint. In Amulet, the standard `Get` routine can tell whether it is being invoked from inside of a formula or not. If called from inside a formula, in addition to returning the value, `gv` and `Get` also set up a dependency. Then, whenever a slot's value changes, the system knows which constraints depend on that value, and can cause the constraints to recalculate.

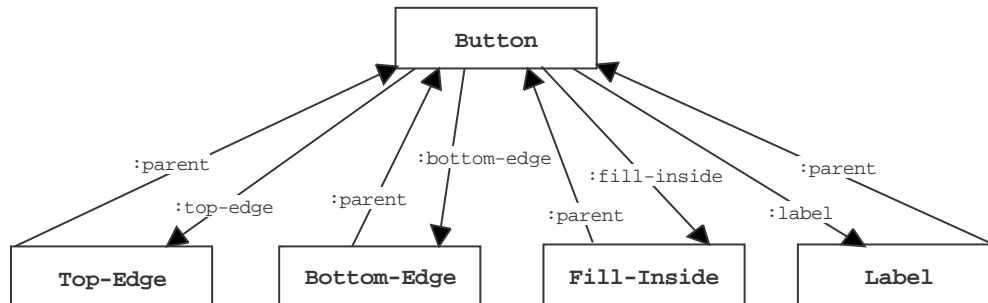
Since they can contain arbitrary code, constraints might be considered to be like methods, and, in fact, they serve a similar purpose: to define the operation of objects. The important point is that programming with constraints is a different style than programming with methods, in the same way that programming with methods is a different style than conventional procedural programming. For one thing, constraints are automatically evaluated when necessary, rather than requiring the programmer to invoke them at appropriate times. Secondly, constraints are declarative, in that they compute the values of variables (slots) based on values of other variables. Constraints can provide more information hiding than conventional methods, as discussed in Section 9. Finally, by focusing on data values, constraints make programming more data oriented, rather than procedure oriented.

One obvious use of constraints is to tie parts of composite objects together. When the programmer collects together a set of objects to make a composite, it is necessary to specify how the parts relate. An innovation of the Garnet constraint system, which is also provided by Amulet, is that the objects can be referenced through pointer variables (see section 5.1). This allows the code of the constraint to be independent of the specific objects used for the parts. Instead, the constraint will reference the object using a "path" through the aggregate hierarchy. For example, in the button of Figure 2, the `bottom-edge` rectangle can refer to the width of the label object using code like:

```
(gv :Self :parent :label :width) ; Garnet code
self.Get_Owner().Get_Object(Am_LABEL).Get(Am_WIDTH) //Amulet code
```

As shown in Figure 4-a, this starts from the `bottom-edge` rectangle, goes up to the parent aggregate (called the `Owner` group in Amulet), down to the `label` part, and gets the width from there. A short-cut in Garnet is that `gv1` stands for `gv :self`. In Amulet, `Get_Sibling(xx)` stands for `Get_Owner().Get_Object(xx)`. These are used in the code in Figure 4-b and 4-c. Thus, the width of the `bottom-edge` will be the same as the width of the `label`. This will work in the prototype, as well as in all

instances, since Garnet and Amulet set pointers to the appropriate objects into the slots. This makes it easy to create instances of the entire aggregate (including the constraints), since the constraint code does not need to be edited. Figure 4-b and 4-c shows the constraints used to tie together the parts of the button of Figure 2.



(a)

```

(create-instance 'button aggregate
 (:left 20) ; These are
 (:top 20) ; the parameters
 (:string "label"). ; to the button
 `(:parts (
 (:top-edge ,rectangle
 (:left ,(formula (gvl :parent :left)))
 (:top ,(formula (gvl :parent :top)))
 (:width ,(formula
 (+ (gvl :parent :label :width) 8)))
 (:height ,(formula
 (+ (gvl :parent :label :height) 8)))
 (:color white)
 (:bottom-edge ,rectangle
 (:left ,(formula (+ 2 (gvl :parent
 :left))))
 (:top ,(formula (+ 2 (gvl :parent :top))))
 (:width ,(formula
 (+ 6 (gvl :parent :label :width))))
 (:height ,(formula
 (+ 6 (gvl :parent :label :height))))
 (:color black)
 (:fill-inside ,rectangle
 (:left ,(formula
 (gvl :parent :bottom-edge :left)))
 (:top ,(formula
 (gvl :parent :bottom-edge :top)))
 (:width ,(formula
 (- (gvl :parent :bottom-edge :width) 2)))
 (:height ,(formula
 (- (gvl :parent :bottom-edge :height) 2)))
 (:color ,gray)
 (:label ,text
 (:left ,(formula
 (center-x (gvl :parent :fill-inside))))
 (:top ,(formula
 (center-y (gvl :parent :fill-inside))))
 (:string ,(formula
 (gvl :parent :string))))))
 Am_Object Am_Button = Am_Group.Create("Button")
 .Set(Am_LEFT, 20) //These are the parameters
 .Set(Am_TOP 20) // to the button
 .Set(Am_STRING, "label")
 .Add_Part(Am_TOP_EDGE, Am_Rectangle.Create()
 .Set(Am_LEFT, 0)
 .Set(Am_TOP, 0)
 .Set(Am_WIDTH, Am_From_Sibling(Am_LABEL, Am_WIDTH, 8))
 .Set(Am_HEIGHT, Am_From_Sibling(Am_LABEL, Am_HEIGHT,
 8))
 .Set(Am_FILL_STYLE, Am_White))
 .Add_Part(Am_BOTTOM_EDGE, Am_Rectangle.Create()
 .Set(Am_LEFT, 2)
 .Set(Am_TOP, 2)
 .Set(Am_WIDTH Am_From_Sibling(Am_LABEL, Am_WIDTH, 6))
 .Set(Am_HEIGHT, Am_From_Sibling(Am_LABEL, Am_HEIGHT,
 6))
 .Set(Am_FILL_STYLE, Am_Black))
 .Add_Part(Am_FILL_INSIDE, Am_Rectangle.Create()
 .Set(Am_LEFT, 2)
 .Set(Am_TOP, 2)
 .Set(Am_WIDTH, Am_From_Sibling(
 Am_BOTTOM_EDGE, Am_WIDTH, -2))
 .Set(Am_HEIGHT, Am_From_Sibling(
 Am_BOTTOM_EDGE, Am_HEIGHT, -2))
 .Set(Am_FILL_STYLE, Am_Gray))
 .Add_Part(Am_LABEL, Am_Text.Create()
 .Set(Am_LEFT, Am_Center_X_Is_Center_Of_Owner)
 .Set(Am_TOP, Am_Center_Y_Is_Center_Of_Owner)
 .Set(Am_TEXT, Am_From_Owner(Am_STRING));

```

(c)

(b)

Figure 4 : (a) The structure of the objects in the button of Figure 2 showing the references. (b) The complete code used to produce the button in Garnet, and in (c) Amulet. This shows the constraints which put the graphics in the correct places and copy the parameter values to the parts. Two of the differences in the Amulet version are: the coordinates of the parts of a Garnet group are with respect to the window, so the parts in (b) need formulas for their left and top (but not in Amulet); and we cannot have in-line constraints in Amulet, but all of the needed constraints in this code are available as built-in constraints.

An issue with this path-based referencing scheme is that the code of the constraints needs to be edited if the objects are restructured. Also, we find a quite common bug is to accidentally reference the wrong object by coding a path incorrectly. An alternative would be to directly name the particular object desired. This is supported by the constraint system (e.g., a constraint could reference `my_obj_3.Get(Am_LEFT)`), but this is not practical since all the names have to be unique and have to be global variables, which means that the constraints can never be re-used in more than one object. In particular, a prototype and its instances could not share formulas. Another possibility we considered was providing some sort of name scope for object references, so that an object's name could be available in more than just the immediate owner (as in Amulet and Garnet) but still not be a global variable. We were never able to work out a good design for this, however, so the path mechanisms seems to be the best alternative.

The constraints in Figure 4 are fairly simple, and in the Amulet version, only the built-in constraints are needed. However, some objects have quite long and complex constraints. For example, the `aggregraph` object in Garnet is a special type of aggregate that displays its components as a tree or graph, and it has a very large constraint that computes the graph layout information. Figure 4-b shows the syntax for defining constraints in Garnet using the `formula` macro which creates a lambda expression out of its parameter.

Amulet does not use a preprocessor, so the syntax for specifying constraints is a little verbose. In C++, it is impossible to create new functions inside of other functions, so all formulas must be defined at the top level before they are used. For example:

```
// 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.Get(Am_LEFT) +
           (int)tool_panel.Get(Am_WIDTH) + 5;
}
...
// now use the formula to compute the left of the scrolling_window
scrolling_window.Set(Am_LEFT, right_of_tool_panel_formula);
```

The macro `Am_Define_Formula` defines a formula object which returns the type of its first argument (here `int`) where the object is named with the second argument (here `right_of_tool_panel_formula`). Note that constraints can return any type, and a single constraint can even return different types at different times by using the special `Am_Value` type (discussed below). The `Am_Define_Formula` macro then defines a procedure to be executed by the formula, and the code following the macro is used as the procedure's body. The formula 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 this constraint.

In addition to layout, another important use of constraints is to copy values and parameters around. Although the slots which serve as parameters are in the top-level button aggregate in Figure 4, for these values to actually take effect they must be copied down to the appropriate places in the components. For example, the string value is specified at the top level in Figures 2 and 4, but it is needed by the text object. So there is a constraint in the text object that copies the value of the parameter. Of course, since constraints can be arbitrary code, the values can be transformed arbitrarily as needed. Since constraints are used to propagate the values, the objects do not have to do anything special to allow changes at run-time: if one of the parameter slots is changed, the constraints automatically propagate the change appropriately, and the update algorithm will make sure the object is then redrawn.

As another example, the Motif button widget prototype takes the string label, the color, and the position as parameters (among others). These parameters are supplied as values in the slots of the top-level widget. When the object is created, the programmer can specify whichever slots need different values and the rest are inherited. Of course, any value can be changed later while the widget is displayed, if desired. Note that this is quite different from a conventional system that requires the widget creation method to take a large parameter list with all possible values to be set, and therefore requires a custom creation method for each object. Here, the standard create routines are used for all objects, and then any desired slots can be set.

An interesting observation about this use of constraints is that it allows *arbitrary* delegation of values, not just from prototypes. Any slot can get its value from any slot of any other object through constraints. Therefore, the constraints can be used as a form of inheritance. Of course, constraints are more powerful than conventional inheritance since they can perform arbitrary transformations on the values.

As with the graphical objects themselves, constraints can be defined interactively using various editors. Lapidary [Vander Zanden 1995b] provides some iconic menus for defining the most popular constraints. We have found that these are sufficient for most graphical applications. For more complex constraints, a spreadsheet-like tool in Garnet, which is called C32, provides a number of features to help programmers who do not know the exact syntax [Myers 1991a]. For example, C32 has menus that will insert commonly used functions. Also, the user can point to objects with the mouse and C32

will insert a reference into the constraint using the correct path expression. We are working on similar tools for Amulet.

The use of constraints provides the programmer with a number of important benefits. The most obvious is that the system maintains the relationships among objects that otherwise would be the responsibility of the programmer. Constraints also allow objects to provide an abstract interface through top-level variables, and the programmer can declaratively specify how to transform the values for all components. In fact, if you need to use methods, constraints can even be used to dynamically determine which method to use for a message based on the current state. This works because the value of any slot can be computed using a constraint, and the value returned can be a function. However, we do not know of anyone using this feature.

Our constraint solvers handle cycles in the constraints, so that a slot of object A can depend on a slot of object B and vice versa. In evaluating circular constraints, the solver simply goes around the cycle once and uses the old value of any constraint that is currently 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.

5.1 Indirect Constraints

The Garnet constraint system was the first to allow the 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. This capability is also provided by Amulet.

Most other constraint systems cannot handle these kinds of constraints. These “indirect constraints” [Vander Zanden 1994] 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, most of the constraints in Figure 4-c use indirection based on parts of the button. Note that even though each instance of the button will share the same constraints, they might each calculate different values if the label string is a different size.

The indirect constraints are a form of “procedural abstraction” since the constraints can be thought of as relationships that can be reused in multiple places, with different values for their parameters. In fact, there is a library of predefined constraints, which can be used for many of the basic relationships frequently found in user interfaces as used in the

code in Figure 4-c. It is worth reiterating that unlike other systems such as SubArctic [Hudson 1996] that *only* supply these predefined constraints, Amulet allows arbitrary code in constraints.

5.2 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.” However, since Garnet uses a lazy evaluation scheme, which, by design, only evaluates the minimal set of constraints, it is difficult to get constraints containing side effects to be evaluated at the right times. Therefore, Amulet’s constraints are eagerly evaluated and can contain arbitrary side effects, even creating and destroying objects. For example, a constraint in the `Am_Map` object creates 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 in a circuit program is put into the `Am_X1` slot but also sets the `Am_Y1` slot:

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

Unlike previous systems such as Rendezvous [Hill 1994], 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. If the constraints calculate and set different values, an infinite loop can result.

There was some fear that switching to an eager evaluation scheme might hurt performance, so we did a small experiment where we switched all the constraints that did not contain side effects to use lazy evaluation. Our initial analysis suggests that in practice, the resulting performance was the same, or possibly even worse using the lazy version. The only advantage of the lazy version is that some constraints are evaluated later than they are in the eager version, so the constraint code can often do less checking that their parameters are valid.

5.3 Multiple constraints in the same slot

Unlike Garnet, Amulet allows multiple formula constraints in a single slot at the same time. We find this useful for situations where the value of the slot might be computed two different ways. For example, the value slot of a scroll bar might contain a constraint that computes the value based on where the user drags the indicator. However, in some application, the programmer might want a scroll bar to take its value from an application variable, and so add another constraint to the value slot. Keeping both constraints in the slot allows the value to be updated appropriately when either the application's data changes or the user manipulates the scroll bar.

5.4 Multiple Solvers

An ongoing research area in user interface software is creating new kinds of constraint solvers (e.g. [Gleicher 1993][Vander Zanden 1995a][Hudson 1996]). One Ph.D. research project was to try to add a multi-way constraint solver into Garnet that would cooperate with the existing formula solver [Sannella 1994a]. The difficulty of this task in Garnet inspired us to create an architecture in Amulet that allows multiple constraint *solvers* to coexist. Currently, in addition to the one-way solver described above, Amulet supports a multi-output, multi-way solver called a “web”⁴ and an animation constraint solver.

A 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

⁴Web constraints are not related to the “world-wide-web.”

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.

5.4.1 Animations

We have also created a novel *animation constraint solver* for animating objects [Myers 1996b]. 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. Amulet's animation constraints detect changes to the value of the slot, immediately restore the original value, and cause the slot to take on a series of values interpolated between the original and new values.

Animation constraints provide significantly better modularity and reuse than previous approaches. 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)⁵, and so on.

5.4.2 Design

We were able to add animation and web constraints to Amulet without modifying the object system because there is a standard protocol in the object system 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, usually because some slot that the constraint depends on has changed), 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 that depend on a slot that some other constraint has caused this slot to be invalid (this causes the invalidation to be propagated), and

⁵Special animation constraints are used for the Visible slot which cause the object to fade out or fly off the screen. The value of the Visible slot itself therefore only changes when the animation is complete.

- 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 value is invalid, 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. As we develop more constraint solvers, we will continue to investigate this issue.

5.5 Performance

For formula constraints which are valid (which already have the correct value), getting the value takes the same time as a regular Get. When the formula is invalid, the procedure for the formula must be called and executed, and the dependencies of the formula might need to be updated. With typical formulas, we have found we can re-evaluate about 50,000 constraints per second.

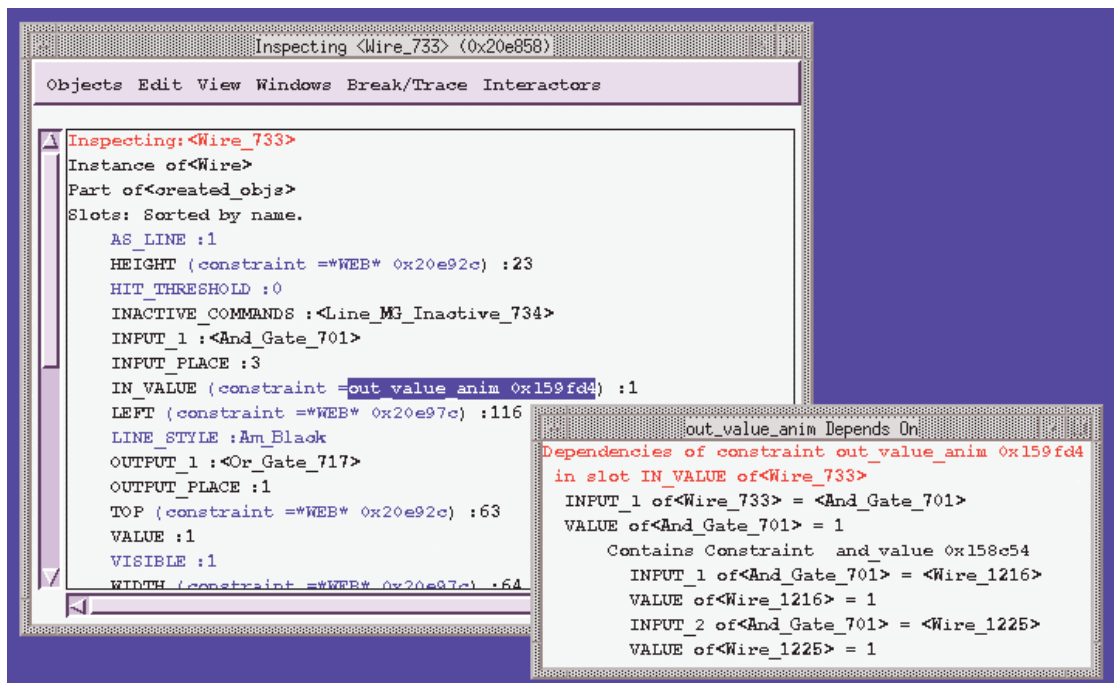


Figure 5: The Inspector in Amulet, viewing a wire object, and the dependencies of the constraint in its `IN_VALUE` slot. The constraint is named `out_value_anim` and it depends on two slots: `INPUT_1` of a wire and `VALUE` of an `And_Gate`. The value of the `And_Gate` in turn contains a constraint. In the main Inspector window, the slots which are inherited are shown in blue on the screen. Notice that the high-level names are shown for methods, constraints and objects.

6. Debugging

Debugging interactive applications requires more mechanisms than supplied with conventional development environments. Garnet and Amulet provide an interactive *Inspector* that displays the object's properties, traces the execution of interactive behaviors, and displays the dependencies of constraints (see Figure 5). From the Inspector, programmers can also set breakpoints or have messages printed whenever the value of a slot changes. The Inspector is also useful for development, since the developer can experiment with different values for colors, positions, and other parameters.

Furthermore, extensive error checking (when debugging is enabled) and helpful messages make applications easier to develop and debug. We try to make sure that programmers using Garnet and Amulet never see "Segmentation fault" or other common but unhelpful low-level error messages. The various macros and other mechanisms store the string names of objects, methods, and other types, so that the user can get useful, high-level information at run-time.

7. Embedding a Prototype-Instance Object System in C++

It was somewhat tricky to provide a dynamic prototype-instance system, dynamic slot typing, and constraints in C++ without using a preprocessor or a scripting language. To allow the same `Set` and `Get` to work for all types in C++, we provide accessing 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 type-checked can use Wrappers. Amulet can also handle memory management for Wrappers using reference counting.

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, as shown in many of the code examples above. This makes the C++ code look somewhat like our Garnet Lisp code.

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 test the type and value. `Am_Value` is a C++ type that consists of a type field and a union of types for the data. It has a set of conversion routines and a set of constructors and assignment operators that allow it to be automatically converted to and from any of the standard types.

8. Differences between Garnet and Amulet

Since Amulet was designed after Garnet, we fixed a number of problems we experienced with Garnet. Many of these were discussed above, 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,
- special forms of `Set` for when new slots are being created to help avoid errors,
- a flexible demon mechanism, and
- eager evaluation to support side effects.

We have also continued to add new features to Amulet, such as the new support for animation (section 5.4.1).

Another important difference is that 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.

9. Modularity

Some people claim that using methods is a better interface to objects because it supports better information hiding. The motivation is that the internal implementation of the object can be more easily changed if the interface is through methods. Therefore some object systems, such as Self [Ungar 1987], do not allow direct access to any object variables, but provide access to variables only through methods. Garnet and Amulet take an opposite approach, and the main interface is through the *data* of objects. The following sections discuss why this provides excellent modularity.

9.1 Data vs. Methods

In Garnet and Amulet, an object has specific input and output slots, and most objects of the same type use the same slots (for example, all graphical objects have left, top, width, height, filling-style, etc.). This corresponds to the exported methods in other object systems. In Garnet and Amulet, through the use of constraint formulas, objects can transform the parameter values in whatever way is desired. For example, when the user clicks on an object, the system sets a specific slot of the object, called `Am_SELECTED` in Amulet. It is up to the internal constraints in the selected object what this does, if anything. The color, position, or font of the object might have a formula depending on this slot. This interface is just as modular as if the system called a generic `Become_Selected` method.

Furthermore, the code that accesses a value of a slot does not know if the slot's value is constant or computed by a constraint. This can help with modularity and re-design since if an object is changed so that a value which was formerly a constant is now computed by a constraint, the external code that uses the value is not affected. This contrasts with some other object systems where there is a big distinction between accessing values stored in variables versus values that are computed, which can only be achieved by calling methods which return a value. In Garnet and Amulet, most of the interface is through accessing of the values of slots, and the external code cannot tell whether the value was constant or computed.

Although we do not currently provide mechanisms to declare which slots of an object can be used from outside and which are internal, this could easily be added. This would provide the same protection as class-instance models which have public and private methods.

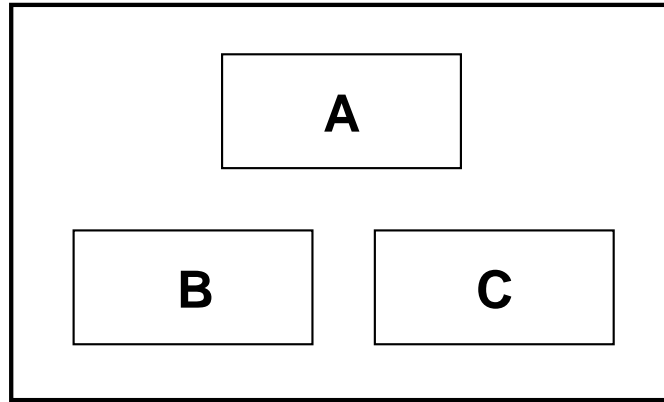


Figure 6: A box centered over two other boxes. If either box B or C moves, box A should move so that it stays centered over the boxes.

9.2 Constraints vs. Methods

Constraints also contribute to modularity in another way, by fixing a flaw in the conventional, imperative object-oriented model. In the conventional model, to achieve certain types of behavior, the programmer must either explicitly arrange the methods so they execute in the proper order, thus violating the modularity of objects, or else allow the methods to execute in an arbitrary order, thus evaluating methods more times than necessary, and possibly destroying the correctness of the program if the methods commit side-effects. For example, suppose that a programmer wants to keep a box called A centered above two other boxes called B and C (Figure 6). In a conventional system, the programmer might add a message to the move methods in B and C that calls a centering method in A. Later the programmer decides that C should always be 20 pixels to the right of B. The programmer thus expands the move method in B to send a message to the move method in C. Without proper sequencing, the centering method in A may be called twice, once by the move method in A, and once by the move method in B. However, the centering method in A should only be called once, after the methods in both B and C have terminated.

In this case, the programmer is faced with two equally unpalatable choices. The programmer can choose not to provide explicit sequencing, in which case the centering method in A may execute twice. This is both wasteful and potentially dangerous if the centering method commits side-effects (in this case it probably would not, but obviously there are situations where this could pose a problem). Alternatively, the programmer could rely on the fact that the move method in C calls the centering method in A, and thus not call the centering method itself. However, the implementation of the move method in

B now depend on the implementation of the move method in C, which violates the notion of modularity.

Notice that in either case the modularity principle is additionally violated because B and C have to know that A depends on them (and later B has to know that C depends on it). If the centering relationship between A, B, and C is later destroyed, not only must the centering method in A be deleted, but the move methods in B and C must be changed as well. In a constraint-driven language, neither of these problems arises since the constraint solver handles both communication between objects and the ordering of constraints. In the above example, the programmer would initially write a constraint that centered A above B and C. Later the programmer would add an additional constraint placing C 20 pixels to the right of B. The constraint solver would automatically ensure that the constraints were evaluated in the proper order. Thus the programmer would not have to worry about sequencing. In addition, the move methods for B and C would not have to know about the relationships among the three objects (the constraint solver would be responsible for propagating the change information), so they would simply modify the local state of their object. If one or both of the constraints were later deleted, the move methods would not have to be modified. Thus, in some cases, constraint-driven programming better preserves the modularity of objects.

10. Evaluation

We have now been programming with prototype-instance object systems and constraints for almost 10 years. This section discusses some of our opinions about the advantages and disadvantages.

- The ability to dynamically create new slots in objects at run time is very convenient, and allows the system to add annotations and other new information to built-in objects. One disadvantage of this is dealing with the problem of name clashes: we have occasionally had two different parts of the system try to use the same slot name in an object for different purposes.
- The ability to query all properties of all objects at run-time is crucial to making the systems, and applications created using them, easier to debug. Keeping around sufficient information to find out the list of all slots of an object, all instances created of a prototype, all parts of a group, and the names of everything, makes most of the properties and data visible to the programmer. However, this takes up a great deal of space, and also makes programs run slower. Therefore, we have

provided mechanisms to compile the libraries without the debugging information, for delivering applications.

- Having the primary interface to objects be a set of values which are set and accessed, rather than a set of methods that are called, makes many operations much easier. Constraints can be used to compute those values or to use them in other calculations. The objects can be automatically animated by simply setting the slots with a sequence of values, without any need for the object to even know it is being animated. Furthermore, Amulet's general undo facility takes advantage of this feature by remembering the old values of each changed object, and then just restoring the values if the user asks for an Undo [Myers 1996a].
- An interesting disadvantage of the prototype-instance object system is that it is harder to see the specification of an object from the code. Slots can be added anywhere, and there is no clear declaration of the internal versus external slots in Garnet or Amulet. In the exported header files, the main object names are just listed, with no indication of what slots control them. Therefore, discovering which slots are in each object, and what each slot does, usually requires carefully reading the documentation.
- Although we feel that the performance of Garnet and Amulet are adequate, the prototype-instance object system is still quite a bit slower than the "native" object systems (CLOS and C++). This is due to the dynamic look up of slots, and other overhead for supporting querying and constraints. The space overhead of objects is still quite large, with a typical object, containing 20 slots with a dozen formulas, is about 1K bytes. Thus, it would not be appropriate to use Garnet or Amulet objects if 100,000 were needed. Typical large Garnet and Amulet applications today would have 2000 to 5000 objects. We are investigating ways to provide "glyphs" [Calder 1990] which are very small objects for these situations [Myers 1994].
- We have worked hard to provide a reasonable syntax for objects in C++, but we are still not entirely happy. Beginners often have trouble with the syntax, and small slips and typos can result in unintelligible compiler error messages, or even worse, code which compiles but does unexpected things. Designing a language from scratch, as was done for Self, would be much more appealing, but then the resulting system might not be able to be used by as many people.

11. Related Work

Garnet and Amulet build on many years of work on user interface toolkits (see [Myers 1995] for a survey). The main other research project investigating the prototype-instance model is Self [Ungar 1987][Chambers 1989]. 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 first system with constraints was probably SketchPad [Sutherland 1963]. Many systems have used constraints as part of an object system [Borning 1986], but none is as general-purpose or fully-integrated as Garnet and Amulet. The first integrated constraint and object system was ThingLab [Borning 1981], which supported multi-way constraints. ThingLab was also a prototype-instance object system. EVAL/vite [Hudson 1993a] 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 [Sannella 1994b] 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 [Hill 1994] was designed to help create multi-user applications in Lisp. Like Amulet, Rendezvous allows multiple one-way constraints to be attached to a variable, but Rendezvous requires that variables be explicitly declared and uses a different implementation algorithm. Also, Rendezvous requires that all side-effects from constraints be deferred until the constraints have been reevaluated. In contrast, Amulet executes side effects as the constraint is evaluated. The Arkit toolkit [Hudson 1993b] 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. SubArctic [Hudson 1996] supports an efficient implementation of a few simple layout constraints in Java, but does not have a general-purpose constraint solver.

12. Conclusions

The style of programming in the Garnet and Amulet object systems is quite different from other object systems: the programmer collects together graphical objects, and then writes constraints to define the relationships among them. Much of the design of user interfaces can be done with graphical, interactive tools, rather than by writing code. Even when not using interactive tools, programmers rarely write methods when creating Garnet

and Amulet code. Our experience suggests that using a prototype-instance object system is very effective for graphical user interfaces, and we continue to investigate ways to enhance its advantages and overcome its weaknesses.

Acknowledgments

We want to thank the many developers and users of Garnet and Amulet over the years. For help with this chapter, we would like to thank Randy Smith.

References

- [Avrahami 1989] Gideon Avrahami, Kenneth P. Brooks and Marc H. Brown. "A Two-View Approach To Constructing User Interfaces," *Computer Graphics, Proceedings SIGGRAPH'89*. Boston, MA, Jul, 1989. pp. 137-146.
- [Borning 1981] Alan Borning. "The Programming Language Aspects of Thinglab; a Constraint-Oriented Simulation Laboratory," *ACM Transactions on Programming Languages and Systems*. *ACM Transactions on Programming Languages and Systems*. 1981. **3**(4). pp. 353-387.
- [Borning 1986] Alan Borning and Robert Duisberg. "Constraint-Based Tools for Building User Interfaces," *ACM Transactions on Graphics*. *ACM Transactions on Graphics*. 1986. **5**(4). pp. 345-374.
- [Calder 1990] Paul R. Calder and Mark A. Linton. "Glyphs: Flyweight Objects for User Interfaces," *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'90. Snowbird, Utah, Oct, 1990. pp. 92-101.
- [Chambers 1989] Craig Chambers, David Ungar and Elgin Lee. "An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes," *Sigplan Notices*. *Sigplan Notices*. 1989. **24**(10). pp. 49-70. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'89.
- [Giuse 1989] Dario Giuse. "Efficient Frame Systems," *Lecture Notes in Artificial Intelligence, EPIA 1989*. *Lecture Notes in Artificial Intelligence, EPIA 1989*. 1989. **390**
- [Gleicher 1993] Michael Gleicher. "A Graphics Toolkit Based on Differential Constraints," *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'93. Atlanta, GA, Nov, 1993. pp. 109-120.
- [Hashimoto 1992] Osamu Hashimoto and Brad A. Myers. "Graphical Styles for Building User Interfaces By Demonstration," *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'92. Monterey, CA, Nov, 1992. pp. 117-124.
- [Hill 1994] Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F. Patterson and Wayne Wilner. "The Rendezvous Architecture and Language for Constructing

Multiuser Applications,” *ACM Transactions on Computer-Human Interaction*. 1994. **1**(2). pp. 81-125.

[Hudson 1993a] Scott E. Hudson. *A System for Efficient and Flexible One-Way Constraint Evaluation in C++*. Graphics Visualization and Usability Center, College of Computing, Georgia Institute of Technology. April, 1993, 1993a.

[Hudson 1996] Scott E. Hudson and Ian Smith. “Ultra-Lightweight Constraints,” *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'96. Seattle, WA, Nov, 1996. pp. 147-155.
http://www.cc.gatech.edu/gvu/ui/sub_arctic/.

[Hudson 1993b] Scott E. Hudson and John T. Stasko. “Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions,” *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'93. Atlanta, GA, Nov, 1993b. pp. 57-67.

[JavaSoft 1996] JavaSoft. *JavaBeans*. Sun Microsystems. JavaBeans V1.0. December 4, 1996. <http://java.sun.com/beans>.

[Myers 1990a] Brad A. Myers. “A New Model for Handling Input,” *ACM Transactions on Information Systems*. 1990a. **8**(3). pp. 289-320.

[Myers 1991a] Brad A. Myers. “Graphical Techniques in a Spreadsheet for Specifying User Interfaces,” *Human Factors in Computing Systems*, Proceedings SIGCHI'91. New Orleans, LA, Apr, 1991a. pp. 243-249.

[Myers 1991b] Brad A. Myers. “Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs,” *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'91. Hilton Head, SC, Nov, 1991b. pp. 211-220.

[Myers 1992] Brad A. Myers. “Demonstrational Interfaces: A Step Beyond Direct Manipulation,” *IEEE Computer*. 1992. **25**(8). pp. 61-73.

[Myers 1995] Brad A. Myers. “User Interface Software Tools,” *ACM Transactions on Computer Human Interaction*. 1995. **2**(1). pp. 64-103.

[Myers 1998] Brad A. Myers. “Scripting Graphical Applications by Demonstration,” *Human Factors in Computing Systems*, Proceedings SIGCHI'98. Los Angeles, CA, Apr, 1998. pp. 534-541.

[Myers 1990b] Brad A. Myers, *et. al.* “Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces,” *IEEE Computer*. 1990b. **23**(11). pp. 71-85.

[Myers 1994] Brad A. Myers, Dario A. Giuse, Andrew Mickish and David S. Kosbie. *Making Structured Graphics and Constraints Practical for Large-Scale Applications*. Carnegie Mellon University Computer Science Department. CMU-CS-94-150. May, 1994. Also appears as CMU-HCII-94-100.

- [Myers 1996a] Brad A Myers and David Kosbie. "Reusable Hierarchical Command Objects," *Proceedings CHI'96: Human Factors in Computing Systems*, Vancouver, BC, Canada, April 14-18, 1996a. pp. 260-267.
- [Myers 1997] Brad A. Myers, *et. al.* "The Amulet Environment: New Models for Effective User Interface Software Development," *IEEE Transactions on Software Engineering*. 1997. **23**(6). pp. 347-365.
- [Myers 1996b] Brad A. Myers, Robert C. Miller, Rich McDaniel and Alan Ferreny. "Easily Adding Animations to Interfaces Using Constraints," *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'96. Seattle, WA, Nov, 1996b. pp. 119-128. <http://www.cs.cmu.edu/~amulet>.
- [Myers 1989] Brad A. Myers, Brad Vander Zanden and Roger B. Dannenberg. "Creating Graphical Interactive Application Objects by Demonstration," *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'89. Williamsburg, VA, Nov, 1989. pp. 95-104.
- [Sannella 1994a] Michael Sannella. *Constraint Satisfaction and Debugging for Interactive User Interfaces*. Seattle, WA, Dept. of Computer Science and Engineering, University of Washington. 1994a. Ph.D. Thesis. TR 94-09-10.
- [Sannella 1994b] Michael Sannella. "SkyBlue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction," *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'94. Marina del Rey, CA, Nov, 1994b. pp. 137-146.
- [Stein 1989] Lynn Andrea Stein, Henry Lieberman and David Ungar. "A Shared View of Sharing: The Treaty of Orlando," *Object-Oriented Concepts, Databases, and Applications*, New York, NY, ACM Press, Addison-Wesley. 1989. pp. 31-48.
- [Sutherland 1963] Ivan E. Sutherland. "SketchPad: A Man-Machine Graphical Communication System," *AFIPS Spring Joint Computer Conference*, 1963. pp. 329-346.
- [Ungar 1987] David Ungar and Randall B. Smith. "Self: The Power of Simplicity," *SIGPLAN Notices*. 1987. pp. 247-241. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications; OOPSLA'87.
- [Vander Zanden 1995a] Brad Vander Zanden. *An Incremental Algorithm for Satisfying Hierarchies of Multi-way, Dataflow Constraints*. Computer Science Department, University of Tennessee. Mar, 1995, 1995a.
- [Vander Zanden 1990] Brad Vander Zanden and Brad A. Myers. "Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces," *Human Factors in Computing Systems*, Proceedings SIGCHI'90. Seattle, WA, Apr, 1990. pp. 27-34.
- [Vander Zanden 1995b] Brad Vander Zanden and Brad A. Myers. "Demonstrational and Constraint-Based Techniques for Pictorially Specifying Application Objects and Behaviors," *ACM Transactions on Computer-Human Interaction*. 1995b. **2**(4). pp. 308-356.

[Vander Zanden 1994] 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*. 1994. **1**(2). pp. 161-213.

[Wilson 1990] David Wilson. *Programming with MacApp*. Reading, MA, Addison-Wesley Publishing Company. 1990.