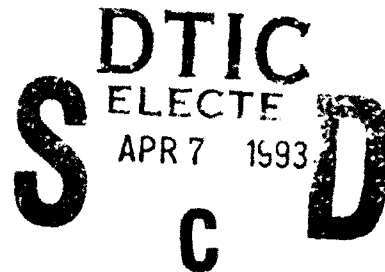
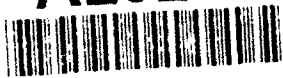


AD-A262 696



The Second Garnet Compendium: Collected Papers 1990-1992

edited by
Brad A. Myers

February 1993
CMU-CS-93-108

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

DISTRIBUTION STATEMENT A

Approved for public release
Distribution Unlimited

Abstract

This technical report brings together nine papers about various aspects of the Garnet project. It is a sequel to Computer Science Technical Report CMU-CS-90-154 which contained articles about Garnet from 1989-1990.

Copyright © 1993 Carnegie Mellon University

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. Additional support for Garnet has been provided by NEC Corporation, Apple Computer, Inc., Adobe Systems, Inc., and the General Electric Company.

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 NEC, Apple, Adobe, GE, or the U.S. government.

93 4 06 075

93-07193



139128

Keywords: Gamet, user interface development environments, user interface management systems, toolkits, constraints, interface builders, object-oriented programming, direct manipulation

Forward

The Garnet User Interface Development Environment contains a comprehensive set of tools that make it significantly easier to design and implement highly-interactive, graphical, direct manipulation user interfaces. The lower layers of Garnet provide an object-oriented, constraint-based graphical toolkit that allows properties of graphical objects to be specified in a simple, declarative manner and then maintained automatically by the system. The higher layers of Garnet include a number of interactive tools that allow much of the user interface to be created by demonstration without programming.

This technical report collects together a number of recent papers about Garnet, some of which have been or will be published elsewhere. Also included are a collection of pictures of applications created using Garnet. A previous technical report (CMU-CS-90-154) contained the papers from 1989 through 1990, including an introductory article describing all of Garnet which appeared in *IEEE Computer* in November, 1990. If you are not familiar with Garnet, it is best to read that article first. There is also a complete reference manual for Garnet that is revised about every six months. The current version of the manual is CMU-CS-90-117-R3.

The first four papers discuss the toolkit level of Garnet. The underlying object system is the topic of the first paper, page 1. Next, is a paper describing the constraint system (page 17). These and other features contribute to a unique programming style in Garnet, as described on page 27. The next paper summarizes some reasons that Garnet is good for creating interactive design tools (page 45). The last five articles discuss the higher-level tools in Garnet. First, a chapter from an upcoming book summarizes all the "demonstrational" aspects of Garnet (page 69). The two papers on the Gilt interface builder describe its technique for increasing application and user interface separation (page 89) and achieving consistent look-and-feel across applications (page 99). The C32 spreadsheet system (page 107) helps implement constraints when the simple icons in Lapidary are insufficient. The new Marquise tool allows the overall behavior of the interface to be defined (page 115). Finally, a collection of pictures of systems created using Garnet starts on page 123.

As mentioned in the articles, Garnet is available for anonymous FTP. To retrieve the system, ftp to a.gp.cs.cmu.edu (128.2.242.7). When asked to log in, use anonymous, and your name as the password. Then change to the garnet directory (note the double garnet's) and get the README explanation file:

```
ftp> cd /usr/garnet/garnet/
ftp> get README
```

Now, follow the directions in the README file.

DTIC Q111111

Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By <i>Rec Ltr.</i>	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
<i>A-1</i>	

Table of Contents

Forward	iii
Efficient Implementation of an Integrated Prototype-Instance and Object System.....	1
The Importance of Indirect References in Constraint Models	17
Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods	27
Environment for Rapid Creation of Interactive Design Tools	45
Garnet: Use of Demonstration.....	69
Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs	89
Graphical Styles For Building User Interfaces by Demonstration.....	99
Graphical Techniques in a Spreadsheet for Specifying User Interfaces	107
Marquise: Creating Complete User Interfaces by Demonstration ...	115
Screen Shots from Selected Garnet Applications.....	123

Efficient Implementation of an Integrated Prototype-Instance Object and Constraint System

Dario A. Gluse

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
dzg@cs.cmu.edu

Brad A. Myers

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
bam@cs.cmu.edu

Brad Vander Zanden

107 Ayres Hall
Computer Science Department
University of Tennessee
Knoxville, TN 37996-1301
bvz@cs.utk.edu

Abstract

KR is a portable object-oriented system with an integrated constraint maintenance mechanism. The object system implements the prototype-instance model and supports dynamic redefinition of prototypes. Constraints express relationships among values, and are specified using arbitrary Lisp expressions. The constraint system transparently keeps constraints up to date. For maximum performance, it is closely integrated with the object system. Several mechanisms, such as constraint value caching and copy-down inheritance, are used to improve performance. The close integration of object-oriented programming with flexible constraint maintenance makes the system well suited for a variety of application programs, including highly interactive graphical applications. KR is the basic building block of the Garnet user interface development toolkit.

1. Introduction

KR [Giuse 90] is a portable object-oriented system with an integrated constraint maintenance mechanism, written in Common Lisp. KR implements the prototype-instance model [Lieberman 86], and supports completely dynamic redefinition of prototypes with automatic change propagation.

The system began as an attempt to implement the best ideas found in frame systems without incurring their typical performance penalty [Giuse 89]. Constraint maintenance was first implemented as a separate layer on top of the existing language. After that experiment proved successful, we integrated constraints with the basic language to provide efficient performance. The three components (object representation, object-oriented programming, and constraint maintenance) are now closely integrated and present a smooth interface to the application programmer.

KR provides the object-oriented model and constraint maintenance system for Garnet [Myers 90], a comprehensive user interface development environment for X/11. In addition to being used to implement Garnet itself, KR is the implementation language for all Garnet applications. Such applications range from simulations of computer security constraints [Tygar 87] to graphical editors such as Lapidary [Myers 89], to visual programming applications. Other applications include speech-understanding research at Carnegie Mellon University [Young 89] and an intelligent language tutoring system for Chinese [Giuse 88].

Performance has remained a central goal throughout the evolution of the system. We have striven to achieve excellent performance for common operations, while supporting advanced features usually associated with experimental systems. KR does not attempt to implement every possible operation. Object-oriented programming, in particular, is simpler than in CLOS [DeMichiel 89], and is carefully tailored for typical Garnet applications.

KR is the only widely-used system that implements the prototype-instance model of inheritance. This model provides maximum flexibility, because any object can be used as a prototype for other objects. The notion of class, in particular, is absent. For emphasis, we will use the terms 'prototype' and 'instance' in the remainder of this paper. It should be remembered, however, that in KR there is no conceptual difference between those two terms. The same object can function as both a prototype and an instance.

Changes to an object used as a prototype are always reflected in instances, including existing ones. Consider, for example, a prototype for rectangles. The prototype typically supplies default values for such parameters as border thickness and filling color. Rectangles created as instances of the prototype will then inherit those parameters, unless the programmer explicitly overrides them. The prototype-instance model allows the values in the prototype to be modified at any time. All rectangle instances are then automatically modified to reflect the changes. It is even possible to change dynamically the prototype used for an instance. For example, an object with a rectangle prototype could be modified to become an instance of a circle instead. All necessary changes happen automatically.

The complete integration of KR objects and constraints allows the application programmer to use constraints for operations that would typically be implemented as (combinations of) methods in traditional object-oriented systems. This results in a highly declarative, rather than procedural, programming style [Myers 92].

2. Related Work

KR integrates ideas from several areas, including object-oriented systems, frame systems, and constraint systems. The resulting combination provides a unique mix of object-oriented programming and a declarative style of programming with constraints.

Historically, the first influence on KR came from frame systems. Systems such as KL-ONE [Brachman 77] and, especially, SRL [Fox et al. 84, Wright and Fox 83] were the primary influences. Unlike those systems, however, KR limits the number of features in order to achieve excellent performance. The system, in fact, began as an attempt to implement the best ideas from frame systems without incurring the performance penalty usually associated with them [Giuse 89, Giuse 90]. In order to achieve good performance, KR omits user-controlled inheritance paths, path grammars, slot specifiers, and multiple contexts. However, it retains several of the dynamic features associated with frame systems, such as user-defined inheritance slots, multiple inheritance, and the ability to add any slot to any object. Unlike SRL, KR allows the user to set slots even without a prior declaration: setting a slot's value creates the slot, if none previously existed.

The object-oriented component was also partially influenced by SRL. Like that system, KR does not distinguish between methods and ordinary values: any slot can contain a value or a method. As in SRL and Flavors [Weinreb and Moon 81], methods are invoked by sending a message. KR does not support generic functions, as found in both C++ [Stroustrup 86] and CLOS [Bobrow et al. 89]. The primary reason for this choice is that much of the functionality usually associated with generic functions is implemented as constraints in KR.

The constraint maintenance component was initially modeled after Coral [Szekely and Myers 88], an experimental constraint system written in CLOS. Coral, however, provided only a small portion of the indirect reference constraints functionality found in KR. Whereas in KR a constraint can reference arbitrary objects either directly or indirectly (by referring to slots that contain pointers to objects), Coral only allowed the user to provide fixed lists of object names. The KR approach is much more flexible, because it supports dynamic redefinition of the path leading to a desired value through any number of objects, and dynamic changes to objects.

Because much of what programmers do is exploratory programming, KR does not force any compile-time typing, but rather relies on Lisp's runtime type checking. This approach is similar to that used in other object-oriented systems such as Smalltalk-80 [Tesler 81, Goldberg and Robson 83] and SELF [Ungar and Smith 91], but unlike the C++ compile-time typing system.

Several object-oriented systems advocate restricting access to object slots via a method-based interface. This approach, already found in Flavors, is best exemplified by SELF, in which message passing is considered the fundamental operation and objects access state solely by passing messages. KR takes the opposite approach. Given the combination of object-oriented programming and constraint maintenance, most KR programs use constraints as the primary abstraction, and therefore access state directly via slots.

3. The Object Model

KR supports both named and unnamed objects. Objects in KR are collections of attribute/value pairs. Each attribute, known as a *slot*, has a name, which is unique within an object. KR objects are typically created using the macro `CREATE-INSTANCE`, which allows the user to specify a prototype for the object as well as a series of slots/values. It is possible to specify `NIL` as the prototype, in which case the new object does not have any prototype (at least initially). The following call creates a new object named `RECT-1` with two slots, `:LEFT` and `:HEIGHT`, and no prototype:

```
(create-instance 'RECT-1 nil (:left 20) (:height 25))
```

A slot contains a single value, but the value can be of any Lisp type (such as a list or an array). Lisp functions can be stored in ordinary slots, and can then be invoked as methods. The system does not require methods to be defined or installed specially, although the macro `DEFINE-METHOD` is provided for convenience. A special type of value, known as a *formula*, is used to implement constraints, as explained below. A formula specifies how to compute a value based on other values. When a depended value changes, the formula is (logically) reevaluated.

Slot names in KR begin with a colon. Any slot can be dynamically added to or removed from an object, whether or not the slot is defined in any prototype. Setting an object's slot with a value automatically creates the slot, if needed. This makes it extremely easy to associate any piece of information with any object, since slot names do not have to be predefined. For example, one can set the value of slot `:PERIMETER` in object `RECT-1` by typing:

```
(s-value RECT-1 :perimeter 125.5)
```

The macro `S-VALUE` is used to set the value of a slot of an object, creating the slot if needed and replacing any value that was there previously. The same function is also used to install a constraint on a slot.

Internally, slots can be of two types. The first type, known as a system-defined slot, is used for slots that are common to most objects. Examples include the `:IS-A` slot, which points to the prototype of an object, and the `:LEFT` slot, which indicates the leftmost edge of a graphical object. System-defined slots provide highly optimized access. KR macros such as `G-VALUE` (which retrieves the value of a slot) and `S-VALUE` expand into simple array references for system-defined slots. System-defined slots also use less storage than ordinary slots, because their position is known at compile time. System-defined slots are hard-wired into KR, and the application programmer cannot define new ones at run time. Adding system-defined slots at the KR level, however, is extremely simple (although a recompilation is necessary), making it easy to accommodate the evolving needs of the Garnet system.

The second type of slot are user-defined slots. Any object can have as many user-defined slots as needed. Such slots are slightly less efficient than system-defined ones, both in terms of performance and of storage. Note that the distinction between system-defined and user-defined slots is invisible to the user. Note also that there is no distinction in KR between class variables and instance variables; KR slots can be used either way, and their status can be changed dynamically.

The slots of an object are stored in a variable-length array. In addition to a value, each slot also contains information which is used internally by KR, as shown in Figure 1. KR objects are represented as Lisp structures with two structure slots. The first entry is the name of the object, or `NIL` for unnamed objects. The second entry contains the variable-length array of slot descriptors.

Each system-defined slot is represented by three entries in the array. User-defined slots take four entries, because the slot name needs to be stored as well. The first piece of information in a slot is the slot's current value (25 in the example). A special marker indicates slots that have no value.

The second piece of information associated with each slot is a collection of bits, encoded as an integer, which determine the characteristics of the slot. Currently, three bits are used. The first bit, *inherited-bit*, indicates whether the value in the slot was inherited from a prototype. The second bit, *is-parent-bit*, indicates whether any instance of the object has inherited the value from this slot. The third bit, *constant-bit*, indicates whether the slot is declared constant or was inferred constant (see below).

The third piece of information in a slot is the list of formulas that depend on the value in the slot. In Figure 1 this is shown as a list of three formulas (unnamed formulas are printed as "F" followed by a unique integer). This list is used to notify all formulas that are potentially affected whenever the value of the slot is modified. Slots that have only one dependent do not store a list, but just the formula itself.

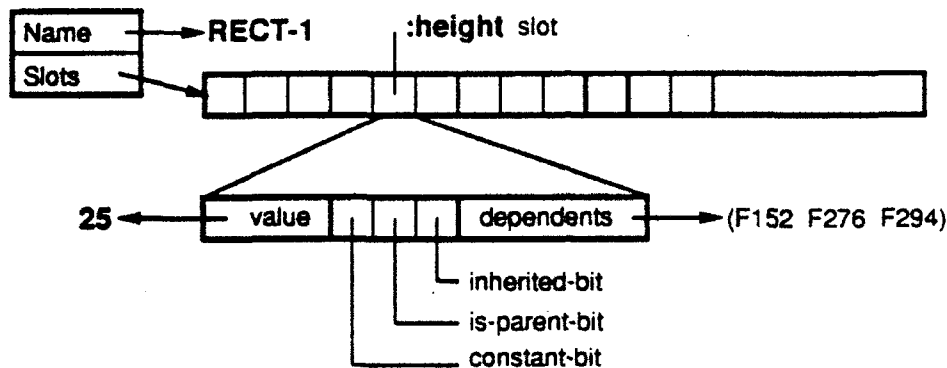


Figure 1: Internal representation of KR object RECT-1 with slot :HEIGHT

3.1 Inheritance

Inheritance in KR occurs through inheritance slots, i.e., slots that have been declared to the system using the macro `CREATE-RELATION`. The only system-defined inheritance slot is `:IS-A`. Its automatically maintained inverse slot, `:IS-A-INV`, lists all instances of an object, and is used to propagate changes automatically. The user may also declare new inheritance slots, with or without inverse slots. User-defined inheritance slots provide the same inheritance behavior as the `:IS-A` slot.

Inheritance slots (such as `:IS-A`) may contain one or more values. Thus, KR implements dynamic multiple inheritance, both through multiple values in the `:IS-A` slot, and through multiple inheritance slots in the same object. The `:IS-A` hierarchy is searched first for inheritance. If no values are inherited through the `:IS-A` hierarchy, and other inheritance slots are present, the other slots are then searched in turn. Multiple inheritance is not currently used in Garnet, but some non-Garnet applications use it.

The following code creates the two objects A and B, and connects B to A via the `:IS-A` slot:

```
(create-instance 'A nil (:left 10))      ; create top object, A
(create-instance 'B A (:top 15))         ; B :is-a A
```

Printing objects A and B shows the following:

```
{A  :LEFT = 10
   :IS-A-INV = B}      ; automatically created inverse slot

{B  :IS-A = A          ; inheritance slot
   :TOP = 15}
```

Note that slot `:IS-A-INV` is automatically set by the system. Because `:IS-A` is an inheritance slot, asking for the value of slot `:LEFT` in B would return the value 10, inheriting it from A.

Access to inherited slots is a potential performance bottleneck. KR implements a careful tradeoff between access time and storage requirements. When an instance is first created, only slots that are specifically defined by the user are actually created as local slots. In the example above, slot `:LEFT` in object B is not created, because it is not mentioned explicitly. When the value of a slot is requested and is not present locally, the system examines the inheritance hierarchy, looking for a prototype which supplies a value for the slot. If one is found, the system copies the value into the instance and all intervening

prototypes (if any). The *is-parent-bit* is set in the slot from which the value is inherited, and in all the intervening prototypes.

When an inherited value is copied down, it is marked as inherited via the *inherited-bit* of each slot into which it is copied. A subsequent request for the slot will then find the inherited value locally, and thus will be just as efficient as a local slot access. Inheritance, therefore, uses a lazy copy-down mechanism, since no copy is made until a value is actually requested. This solution represents an effective tradeoff between storage (which is only allocated when needed) and access time (which, except for the very first time, is just as fast as local access).

Implementing inheritance via lazy copying of values from prototypes to instances requires special care when changes are made to the hierarchy, or to one of the prototypes. In the example above, changing slot :LEFT in object A must also change the copy of the value in slot :LEFT of object B, since the latter inherited its value from A. Similarly, a change in the inheritance hierarchy (such as changing slot :IS-A in object B) would also cause old inherited values to become invalid.

KR handles such situations by recursively eliminating from instances all the values that have been inherited from their prototypes. When an inherited value is eliminated from an instance, it is replaced by a special *no-value* marker which can never appear as a user-defined value. Additionally, the *inherited-bit* is cleared. The slot, then, looks exactly like a newly created slot that has never been accessed. The change is recursively propagated down the hierarchy, stopping whenever an empty slot is reached, or whenever a slot is found for which a local value was defined. The advantage of this scheme is that if the slot in the prototype is changed again, it is not necessary to revisit the entire subtree of instances.

It might seem that during this recursive down-propagation one could simply set slots to the new value in the prototype, rather than to the special no-value marker. However, this is undesirable for two reasons. First of all, if the new value is a formula, a new copy of the formula would be required for each slot which had inherited the value, because formulas contain local information for each slot. Second, this approach would fail in the case of multiple inheritance, because some of the instances might in fact have inherited values from a different prototype. In the presence of multiple inheritance, the current approach ensures that when a value is requested it will be inherited from the correct prototype.

An interesting case of inheritance is structural inheritance, which refers to composite objects (i.e., objects that contain other objects as their components). When an instance of a composite object is created, the system arranges for the entire structure to be copied. In addition to an instance of the object, instances of each component are also created, and the structural connections among the various objects are set appropriately. Structural inheritance is an extremely powerful abstraction, because it frees the user from having to know whether an object being instanced is simple or composite. Even more importantly, KR's prototype-instance model means that any later change to the original composite object (the prototype) will be automatically reflected in its instances. It then becomes possible to alter a prototype dynamically and have the modifications propagate immediately to any instance.

4. Object-Oriented Programming

Object-oriented programming in KR is implemented via methods. Methods are procedural attachments that can be associated with any object using `DEFINE-METHOD`, or simply by setting a slot to a function value using `S-VALUE`. Internally, methods are stored in slots, just like any other KR value; the system does not limit the number or types of methods that can be defined. Methods can be created or modified dynamically, and any object can redefine any of its methods as needed. If a locally redefined method is eliminated, the prototype-defined method is automatically reused. This fully dynamic approach to method handling is more reminiscent of full-fledged frame systems such as SRL than of traditional object-oriented systems.

A method in KR is invoked by sending a message to an object via the macro KR-SEND. In most cases, methods are not actually defined at the level of an individual object, but are inherited from its prototype(s). Of course, the normal copying down mechanism is used for methods that are inherited from some prototype. KR provides facilities to affect method combination, specifically a function that allows a method to invoke a less specific method defined by some prototype of the current object. This part of the language is not as complex as in other Lisp-based object-oriented systems such as CLOS, however, because usually KR programs use a declarative approach based on constraints [Myers 92] instead of methods.

In addition to supplying default values, methods, and constraints, KR prototypes are also used to control the initialization of instances. Immediately after an instance is created, the system looks for an :INITIALIZE method (which is typically inherited). If one is defined, the system invokes it with the new instance as a parameter. This initialization mechanism, akin to the one provided by many class-based object systems, allows instances to be initialized using arbitrarily complex user-defined methods.

The other component of the KR object model is a *demon* mechanism. It is possible to associate with each object a demon, i.e., a procedural attachment that is invoked when certain slots in the object are modified. Using demons, KR applications can effectively implement active values. Two separate demons are executed at different stages in the value modification cycle (currently, Garnet only uses the first one). The first demon is the *invalidate demon*. It is executed when a slot is invalidated, typically because the slot contains a formula which depends on a value that changed. The invalidate demon is executed with the old value still in the slot, allowing the demon to record the old value, if so desired. The second demon is the *pre-set demon*. It is invoked immediately before the value in a slot is actually set. The primary difference between the two demons is the timing: the invalidate demon is called immediately after a formula becomes invalid, but the pre-set demon is not called until the value of the formula is demanded. If the value of a formula is never demanded, the pre-set demon may never be called, but the invalidate demon is always called. Note that the invalidate demon is called only when a formula first becomes invalid; slots that contain already invalid formulas are not affected.

When a slot in an object is modified, KR checks whether the slot requires a demon. This is specified by storing the list of slot names in the system-defined slot :UPDATE-SLOTS. In most cases, the list of slot names is actually inherited from some prototype. In Garnet, this list includes all the slots that control the graphical appearance of objects. If the changed slot is in :UPDATE-SLOTS, KR looks for a demon for the object. If one is found, it is called with the object and the slot as parameters.

Because demons are application-defined, they can perform any desired action. In the Garnet graphical object system, for example, the invalidate demon is used to support the update algorithm, which ensures that incremental changes to objects are reflected in the display. The Garnet demon records the fact that the object was modified by adding it to a list of "dirty" objects. At the next display update cycle, all dirty objects are redisplayed with their new graphical features.

5. The Constraint Model

A constraint allows the value in a slot (the dependent value) to be computed from the values in other slots. In doing so, a constraint specifies that the dependent value must be recomputed when any of the other values change. The current version of KR implements lazy constraint evaluation, which means that recomputation does not take place until needed, i.e., until the dependent value is actually demanded. An eager-evaluation version of constraint evaluation is currently under development.

Constraints are specified by a special type of object, called a *formula*, created by the macro FORMULA (and its variant O-FORMULA, which is used for compiled constraints). Internally, formulas are represented as Lisp structures. To illustrate the representation of formulas, consider executing the following KR

code:

```
(create-instance 'CIRCLE-2 nil
  (:left 4)
  (:top (o-formula (+ 6 (gv :SELF :left)))))

(g-value CIRCLE-2 :top)
```

The call to `CREATE-INSTANCE` creates an object named `CIRCLE-2`, sets its `:LEFT` slot to 4, and sets the `:TOP` slot to contain a formula which adds 6 to the contents of slot `:LEFT`. The call to `G-VALUE` requests the value of slot `:TOP`, causing the formula to be evaluated, and returns the value 10. The internal structure of the formula at this point is illustrated in Figure 2.

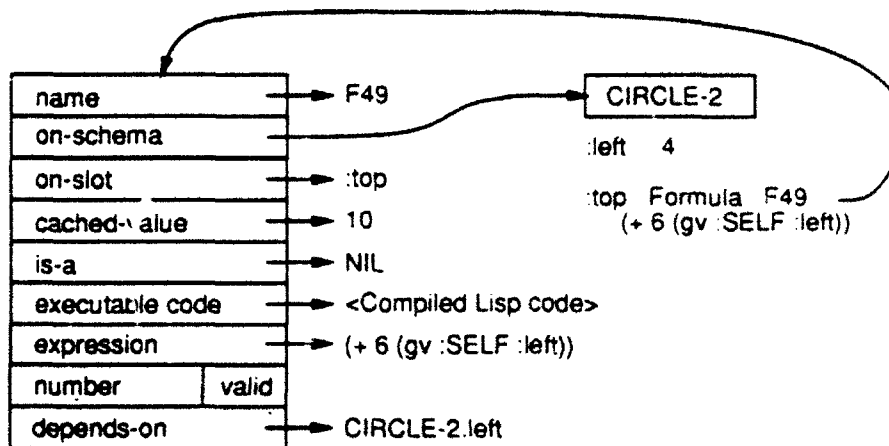


Figure 2: Internal structure of a KR formula.

Two structure slots, *on-schema* and *on-slot*, are used to point to the object and slot on which the formula is installed. The structure slot named *cached-value* holds the formula's current cached value. Instead of re-evaluating a formula every time its value is requested, KR caches the computed value in this structure slot. A single *valid* bit (shown near the bottom of the structure) indicates whether the cached value is valid. If any of the depended values change, the bit is reset, indicating that the cached value has become stale and the formula will need to be re-evaluated when its value is requested.

The *is-a* structure slot is used to point to the formula's parent; in our example, it contains `NIL` because the formula does not have any. The following two structure slots contain the executable code for the formula, and the original expression that was used when the formula was created. The next structure slot contains an integer which encodes the *valid* bit and the so-called cycle number. KR uses this number for two separate purposes. First, the number is set to 0 when the formula is created, indicating that the formula was never evaluated. Second, KR uses the number to detect constraint circularities.

Constraint circularities arise when two formulas depend on each other's value. To detect such cases, at the beginning of each evaluation KR increments a global number. Before each formula is evaluated, KR sets its cycle number to the global number. Before evaluating each formula, however, the system checks whether its cycle number equals the global number. Normally, cycle numbers should be strictly lower than the global number. If a formula is found whose cycle number equals the global number, the formula must have already been visited during the current evaluation cycle, and hence must be a part of a

constraint circularity. KR then breaks the loop and returns the current cached value of the formula where the circularity was detected. The values of the other formulas in the cycle are recomputed accordingly.

The final structure slot in a formula contains a list of dependencies, which is needed when the formula is destroyed. This list allows the system to remove the formula from the dependents-list of all slots whose values were requested by the formula. Without this list, stale pointers would be left around after a formula is destroyed.

Unlike most constraint systems, in which constraints can only be specified using a restricted language [Vander Zanden 89, Borning 79], constraints in KR can be arbitrary Lisp expressions. It is common for application programmers to define complex constraint expressions that use conditionals, loops, local variables, and the full range of Lisp functionality. The only limitation is that constraint expressions should not have side effects, because the system does not guarantee the precise order of evaluation (or re-evaluation) of constraints.

Constraints are closely integrated with the object model. First, a constraint is placed on a slot simply by setting the value of the slot to be a formula object. From the user's viewpoint, this is identical to setting a slot with a regular value, except that the value is wrapped in a formula macro call. Second, a slot that contains a constraint is accessed exactly like any other slot; constraint evaluation is transparent. Third, change propagation is also transparent. When a slot is changed using S-VALUE, KR automatically checks whether the slot was used in computing the value of any existing formula. All dependent formulas are then recursively invalidated, and will be recomputed as needed. Several optimizations are used to make this process efficient. If the slot is set to the same value it had before, no invalidation needs to happen. Also, invalidation stops as soon as an invalid formula is reached, since the algorithm guarantees that all formulas that depend on that formula will already have been invalidated. Note that the entire process is invisible to the user: all the user does is to set a value in a slot.

The advantages of allowing arbitrary Lisp expressions in constraints are clear. Programmers can express very complicated behavior through constraints, e.g., determine the exact layout of composite objects such as trees or variously-aligned lists of similar objects. Such complex constraints are typically supplied by system implementors and inherited by all user-defined gadgets. For example, the Garnet system provides composite lists of gadgets that are formatted using constraints. User-defined gadgets, such as menus, that use these lists automatically inherit the system-defined constraints.

Conventional constraint systems typically need to parse constraint expressions and determine what values are being referenced. The fact that KR supports arbitrary Lisp expressions in constraints, however, makes it impossible to precompute the list of all depended values. A constraint, for example, may invoke user-defined functions, which might contain references to arbitrary values. KR solves this problem by arranging for dependencies to be computed dynamically, as the constraint expression is being evaluated. This is done via the macro GV. Like G-VALUE, this macro retrieves the value from a slot. In addition, however, GV also records the dependency, if needed. Consider the following formula expression, which computes the right edge of an object by adding the object's width to the left edge of object RECT-3:

```
(+ (gv RECT-3 :left) (gv :SELF :width))
```

When this expression is evaluated, the first GV adds the current formula to the list of dependents of slot :LEFT in object RECT-3 (the list of dependents is one of the three components of a slot, as shown in Figure 1). The second GV adds the formula to the list of dependents of slot :WIDTH in the current object. Once these dependencies are set up, KR will invalidate the formula whenever one of the depended values is modified.

As shown in the example, :SELF can be used instead of an object name to indicate the object on which the formula is installed. In addition, GV allows more than one slot name to be specified. For example, the

following formula expression can be used to fetch the value of slot :LEFT from the object contained in slot :PARENT of the current object: (gv :SELF :parent :left).

In this case, GV is used to specify a *path*. Slot :PARENT is accessed to yield an object. The resulting object is accessed and the value of its :LEFT slot is returned. Because (gv :SELF) is such a common idiom, KR provides the equivalent macro GVL. The expression above could have been written as (gvl :parent :left).

An advantage of computing dependencies dynamically is that fewer dependencies may be set up. Imagine an expression that contains a conditional. As long as only one branch of the conditional is taken, there is no need to establish dependencies to values that might be needed in the other branch. This prevents unnecessary work, since a change to one of those values could not affect the final value anyway. Eliminating the need to parse constraint expressions, therefore, results in greater flexibility.

An important distinguishing feature of KR constraints is that they allow fully indirect references [Vander Zanden 91]. Most existing constraint systems, by comparison, only allow hard-wired object references [Vander Zanden 89, Myers 88]. In KR, constraints may refer to slots of other objects indirectly. A typical example of an indirect reference is shown in Figure 3.

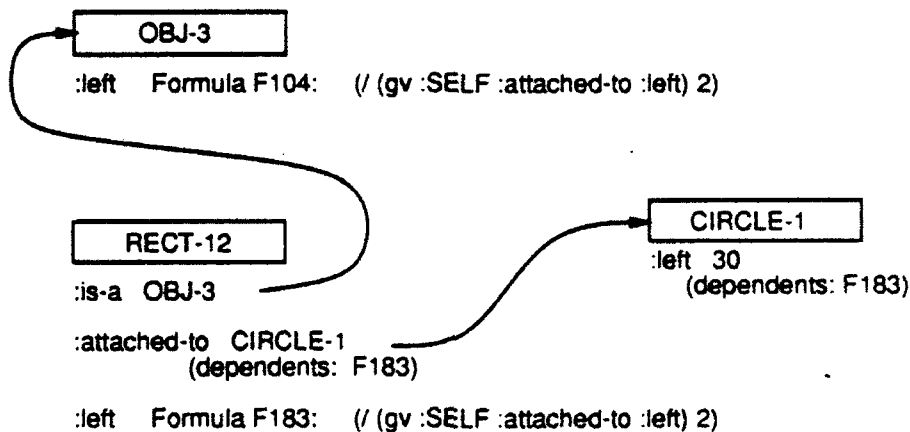


Figure 3: An inherited constraint which uses indirect references.

The formula in slot :LEFT of object RECT-12, F183, was inherited from prototype OBJ-3. Slot :LEFT in object RECT-12 is computed by referring to slot :LEFT in object CIRCLE-1. The current value is 15, obtained by dividing 30 (the value in CIRCLE-1) by 2, as specified in the formula. The name of that object is stored in slot :ATTACHED-TO. Note that the constraint expression of the formula refers to the target object only indirectly, using the contents of slot :ATTACHED-TO in the current object (i.e., :SELF). The constraint is effectively parameterized. Changing the value of slot :ATTACHED-TO in object RECT-12 will automatically cause the value of slot :LEFT to be recomputed. The primary advantage of indirect references is that they allow generic formulas to be defined by an object's prototype. Generic formulas can be used verbatim in the instances, because they use relative paths without hard-wired object names.

Indirect references in KR constraints are possible because the system records the dependency of such constraints on all the intervening links of the chain of references. In Figure 3, for example, both the :ATTACHED-TO slot and object CIRCLE-1's :LEFT slot list formula F183 as their dependent. If slot

:ATTACHED-TO is set to a different value, say object NEW-OBJ, formula F183 is invalidated. When the value of object RECT-12's :LEFT slot is requested, the formula will recompute a new value using the left slot of object NEW-OBJ.

It is clear from the previous discussion that constraints interact with other parts of KR. A first interaction is between constraints and values, because slots can contain either ordinary Lisp values or formulas. This potential problem is avoided by letting all slot-accessing macros, such as G-VALUE and GV, work on either type of slot. If a slot contains an ordinary Lisp value, the macros simply return the value. If a slot contains a formula, the macros evaluate the formula first (if its cached value is not valid) and then return the value. This operation may actually trigger a complex chain of nested evaluations, because the formula may depend on other formulas that also need to be re-evaluated. All of this, however, happens transparently.

A second interaction is between constraints and inheritance. Because inherited values are actually copied from the prototype into the instance, a change to a prototype may modify the (inherited) value in an instance. Consider, for example, the case where object B (an instance of object A) inherits the value of its :LEFT slot from A, as shown in Figure 4.

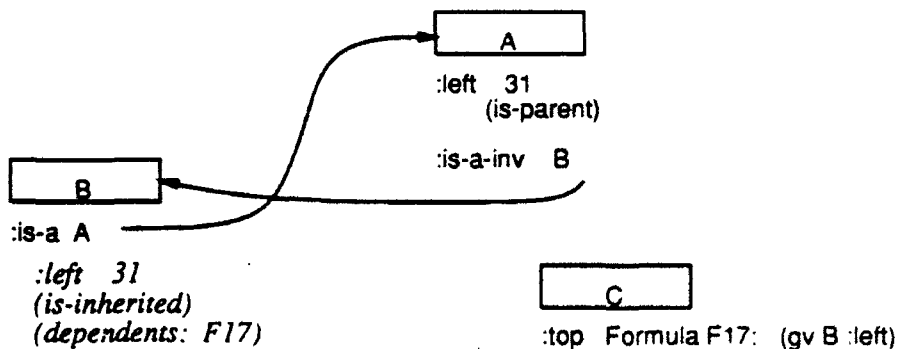


Figure 4: Modifying a slot in prototype A causes formula F17 to be invalidated.

Slot :TOP of object C is constrained to have the same value as object B's :LEFT slot. Now, imagine that slot :LEFT in object A is changed. The value in slot :LEFT of instance B was inherited, and therefore it changes (more precisely, its value is removed and replaced by a non-value, as explained in section 3.1). Consequently, formula F17 in slot :TOP of object C must be invalidated, since the value on which it depends has changed.

To address this problem, the inheritance mechanism described earlier is modified as follows. When a slot is changed, KR checks whether its previous value had been inherited by some other object; this information is contained in the *is-parent-bit* of the slot. If so, the instances of the object are recursively visited. If any instance had inherited the value, the inherited value is removed from the slot. If, in addition, some formula depends on the instance's slot (as described by the slot's list of dependents), the formula (and the formula's own dependents) are recursively invalidated. This process involves two distinct graphs: the inheritance hierarchy, through which values may have been inherited from prototypes into instances, and the constraint graph, which determines how values depend on other values.

6. Special Efficiency Features

KR improves the efficiency of the constraint system by caching computed values in formulas. After a formula is evaluated, its value is cached locally and can then be used over and over until the formula is invalidated. Access to a cached value in a formula is only slightly slower than access to a locally defined, regular value.

Another feature that is essential for performance is that user-specified constraints are compiled when the file they are in is compiled. This is possible because the O-FORMULA macro expands into a LAMBDA form containing the formula expression. The resulting code is extremely efficient and takes advantage of all the built-in optimizations. Inside compiled constraints, for example, slot accessor macros are expanded into array references.

KR contains several user-controlled features that make it easier to generate very efficient applications. First, much of the error-checking code in the system is conditionally compiled. Once an application program has been thoroughly tested, the user can simply run the system with the "no-debug" version of KR, which eliminates most error checks. A second mechanism allows the application programmer to specify that certain paths in indirect reference constraints are immutable. For well-debugged object hierarchies, this declaration allows constraint re-evaluation to become considerably faster by avoiding unnecessary path traversals. It also saves storage by preventing unnecessary dependencies from being created. A third mechanism, which is nearing completion, allows the programmer to declare that certain slots in an object are "constant", i.e., their value will never change after the object is created. This information is stored in a slot's *constant-bit*. Formulas that only depend on constant slots can then be eliminated automatically, further improving efficiency and storage requirements.

7. Conclusions

The current implementation of KR is the result of our collective experience in using the system over the past several years. Whenever a choice existed, we have opted for solutions that could be implemented efficiently without making the programming interface unnecessarily complex.

KR is a very flexible object system. It supports multiple inheritance on both system-defined and user-defined inheritance slots, automatically maintained inverse slots, and the prototype-instance model of inheritance. Prototypes may be modified dynamically, and the results are immediately propagated to instances. Any object may be used as a prototype; no compile-time declarations are necessary. Slots may be added and removed from objects at will. It is also possible to change the prototype of any instance from one object to another, causing all values that were inherited from the old prototype to change appropriately.

KR also provides a very flexible constraint system. New constraints may be created dynamically, and may be installed on any slot. Constraints can be specified as arbitrarily complex Lisp expressions. Moreover, constraints can use indirect references, which offer maximum flexibility by allowing values to be obtained from different objects at runtime. Constraints and inheritance interact properly, and changes in the inheritance hierarchy are propagated through the constraint hierarchy as well.

KR demonstrates that object-oriented programming and constraint maintenance can be effectively integrated. This combination results in great flexibility and a unique programming style in which much of the functionality usually associated with method invocation is performed by constraints. The system is the only widely used object-oriented system in which the prototype-instance model is combined with copy-down multiple inheritance. We are currently experimenting with extensions that will further improve performance, such as user-specified declarations of constant slots and eager constraint evaluation.

Acknowledgements

This research was sponsored by the 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.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Bibliography

- [Bobrow et al. 89] Bobrow, D.G.; DeMichiel, L.G.; Gabriel, R.P.; Keene, S.E.; Kiczales, G.; Moon, D.A.
Common Lisp Object System Specification.
LISP and Symbolic Computation 1(3/4):245-394, January, 1989.
- [Borning 79] Alan Borning.
Thinglab: A Constraint-Oriented Simulation Laboratory.
Technical Report SSL-79-3, Xerox Palo Alto Research Center, July, 1979.
- [Brachman 77] Brachman, R.J.
A structural paradigm for representing knowledge.
PhD thesis, Harvard University, May, 1977.
- [DeMichiel 89] DeMichiel, L.G.
Overview: The Common Lisp Object System.
LISP and Symbolic Computation 1(3/4):227-244, January, 1989.
- [Fox et al. 84] Fox, M.S.; Wright, J.M.; Adam, D.
Experiences with SRL: an analysis of a frame-based knowledge representation.
In First International Workshop on Expert Database Systems. 1984.
- [Giuse 88] Giuse, D.A.
LISP as a rapid prototyping environment: the Chinese Tutor.
LISP and Symbolic Computation 1(2):165-184, September, 1988.
- [Giuse 89] Giuse, D.
Efficient Frame Systems.
Lecture Notes in Artificial Intelligence - EPIA 89.
In J.P. Martins and E.M. Morgado,
Springer-Verlag, Berlin, 1989, pages 39-50.
- [Giuse 90] Giuse, D.A.
Efficient Knowledge Representation Systems.
Knowledge Engineering Review 5(1):35-50, 1990.
- [Goldberg and Robson 83] Goldberg, A.; Robson, D.
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, Reading, MA, 1983.
- [Lieberman 86] Lieberman, H.
Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems.
Sigplan Notices 21(11):214-223, November, 1986.
ACM Conference on Object-Oriented Programming Systems Languages and Applications; OOPSLA '86.

- [Myers 88] Brad A. Myers.
Creating User Interfaces by Demonstration.
Academic Press, Boston, 1988.
- [Myers 89] Myers, B.A.; Vander Zanden, B.; Dannenberg, R.B.
Creating Graphical Interactive Application Objects by Demonstration.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages
95-104. Proceedings UIST'89, Williamsburg, VA, Nov, 1989.
- [Myers 90] Myers B.A., Giuse D.A., Dannenberg R.B., Vander Zanden B., Kosbie D.S., Pervin
E.C., Mickish A., Marchal P.
Gamet: Comprehensive Support for Graphical, Highly Interactive User Interfaces.
IEEE Computer 23(11):71-85, Nov, 1990.
Also appeared in *Japanese in Nikkei Electronics*, vol. 3-18 no. 522, 187-203.
- [Myers 92] Brad A. Myers, Dario Giuse, and Brad Vander Zanden.
Declarative Object-Oriented Programming; How to Program in a Prototype-Instance
System Without Methods.
1992.
Submitted for Publication.
- [Stroustrup 86] Stroustrup, B.
The C++ Programming Language.
Addison Wesley, 1986.
- [Szekely and Myers 88] Szekely, P.A. and Myers, B.A.
A User Interface Toolkit Based on Graphical Objects and Constraints.
Sigplan Notices 23(11):36-45, November, 1988.
- [Tesler 81] Tesler, L.
The Smalltalk Environment.
BYTE 8:90-147, August, 1981.
- [Tygar 87] J. D. Tygar and J. M. Wing.
Visual Specification of Security Constraints.
In *1987 Workshop on Visual Languages*, pages 288-301. Visual Language '87,
Linkoping, Sweden, Aug, 1987.
- [Ungar and Smith 91] Ungar, D.; Smith, R.B.
SELF: The Power of Simplicity.
Lisp and Symbolic Computation 4(3):187-205, July, 1991.
- [Vander Zanden 89] Vander Zanden, B.
Constraint Grammars-- A New Model for Specifying Graphical Applications.
In *Human Factors in Computing Systems*, pages 325-330. Proceedings SIGCHI'89,
Austin, TX, April, 1989.
- [Vander Zanden 91] Vander Zanden, B.; Myers, B.A.; Giuse, D.A.; Szekely, P.
The Importance of Pointer Variables in Constraint Models.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages
155-164. Proceedings UIST'91, Hilton Head, SC, Nov., 1991.

[Weinreb and Moon 81]

Weinreb, D. and Moon, D.

Lisp Machine Manual

Fourth Edition edition, Symbolics, Inc., Cambridge, MA, 1981.

[Wright and Fox 83]

M. Wright, M. Fox.

SRL: Schema Representation Language.

Technical Report, Carnegie-Mellon University, December, 1983.

[Young 89]

Young, S.R.; Hauptmann, A.G.; Ward, W.H., Smith, E.T.; Werner, P.

High-level Knowledge Sources in Usable Speech Recognition Systems.

Communications of the ACM 32(2):183-194, February, 1989.

Reprinted from *ACM Symposium on User Interface Software and Technology*
Hilton Head, SC, Nov. 11-13, 1991. pp. 155-164

The Importance of Pointer Variables in Constraint Models

Brad Vander Zanden

Computer Science Department
University of Tennessee
Knoxville, TN 37996
bvz@cs.utk.edu

*Brad A. Myers
Dario Giuse*

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
brad.myers@cs.cmu.edu
dzig@cs.cmu.edu

Pedro Szekely

USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
szekely@isi.edu

Abstract

Graphical tools are increasingly using constraints to specify the graphical layout and behavior of many parts of an application. However, conventional constraints directly encode the objects they reference, and thus cannot provide support for the dynamic runtime creation and manipulation of application objects. This paper discusses an extension to current constraint models that allows constraints to indirectly reference objects through pointer variables. Pointer variables permit programmers to create the constraint equivalent of procedures in traditional programming languages. This procedural abstraction allows constraints to model a wide array of dynamic application behavior, simplifies the implementation of structured object and demonstrational systems, and improves the storage and efficiency of highly interactive, graphical applications. It also promotes a simpler, more effective style of programming than conventional constraints. Constraints that use pointer variables are powerful enough to allow a comprehensive user interface toolkit to be built for the first time on top of a constraint system.

Keywords: Constraints, development tools, incremental algorithms

1 Introduction

User interface toolkits, particularly graphical layout tools, are increasingly adopting the constraint model of computation. The constraint model uses equations to denote relationships between two or more objects. For example, a designer might write the following equation to position a circle 10 pixels to the right of a rectangle:

```
left = my-rect.right + 10
```

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1991 ACM 0-89791-451-1/91/0010/0155...\$1.50

The advantage of the constraint model is that changed data is automatically propagated to the appropriate places and relationships are automatically maintained. Thus, if a user drags the rectangle around the screen, a constraint solver continuously resatisfies the above equation, causing the circle to follow the rectangle around the display.

Having a constraint solver automatically maintain a set of relationships is obviously advantageous, since it saves the programmer from having to manually write code to accomplish the same task. However, conventional constraints directly encode the objects they reference. For example, `my-rect` is hardcoded into the above constraint. Thus they cannot support the dynamic runtime creation of objects, since the constraints in the newly created objects will reference the wrong objects. These constraints also cannot model any application behavior which involves objects that continuously change their relationships to other objects. For example, a feedback object must highlight any item in a menu, an arrow might point at different boxes in a boxes-and-arrows editor, and a truck switches streets as it navigates through a city.

These shortcomings can be remedied by adding pointer variables to constraints that allow objects to indirectly reference other objects (these constraints are called *indirect reference constraints*). For example, the above constraint could be rewritten as¹:

```
left = self.obj-over.right + 10
obj-over = my-rect
```

where `self` refers to the object containing the variable `left`, in this case, the circle. By changing the value of the

¹To improve readability, we are expressing constraints in conventional infix notation rather than Lisp's prefix notation. In Garnet the constraint would actually be written as `(+ (gvl :obj-over :right) 10)` where `gvl` stands for *get value through link*. The `''` that goes before variable names is also dropped in the notation used in this paper.

variable `obj-over`, the circle can be positioned to the right of any object. With this extension, constraints can support the runtime creation of objects and express the dynamic behaviors that occur inside an application window, as well as the static layout relationships that occur around the application window.

Pointer variables allow the programmer to define the constraint equivalent of procedures in traditional programming languages. Generalizing direct references into indirect references is akin to defining the parameters of a procedure. The advantages of procedural abstraction are well known, but the implementation of procedural abstraction in constraint systems is still quite novel. Constraint procedural abstractions greatly simplify the implementation of many interface features and enable the implementation of new ones that would have been unwieldy without procedural abstraction:

- Feedback, in which objects, such as checkmarks or inverted rectangles, may appear with any item in a set of objects;
- Prototype-Instance models, in which instances of constraints must be inherited from prototypes and references must be adjusted so that they point to the instance rather than the prototype;
- Programming by example, in which constraints that are demonstrated for example objects must be converted to general constraints that work with any object;
- Abstract specification of layouts, in which generic objects are laid out using constraints, and the specific widgets are filled in later, based on such parameters as the availability of screen space;
- Simulations, in which objects are frequently constrained to new and different objects, for example objects moving between the machines on a factory floor.

Over the last couple years, we have gained considerable experience using indirect reference constraints in the Garnet project [11]. We have found that they are crucial for implementing the insides of application windows, which is the hardest and most time-consuming portion of an interface to construct. In Garnet, constraints consist of arbitrary pieces of Lisp code and consequently, they are used to specify more than just graphical layouts. For example, they are used to communicate information between multiple threads of a dialog, to compute the attribute values of objects, and to monitor the states of various objects. The procedural abstraction provided by indirect reference constraints is so powerful that Garnet implements its toolkit on top of the constraints [11]. No other constraint-based toolkit does this.

Indirect reference constraints also provide an entirely new style of programming that seems much simpler and more effective than conventional constraints. It will become apparent how indirect reference constraints lead to far simpler implementations as this paper describes many of the important applications of indirect reference constraints. This paper will also discuss how indirect reference constraints can enhance the performance of an application while decreasing its storage demands. Finally, various implementation strategies for indirect reference constraints will be discussed.

2 Related Work

While pointer variables are commonly incorporated in programming languages, they have only recently been incorporated in their full generality in constraint systems. A restricted version of indirect reference constraints first appeared in Coral [17]. Coral permitted a designer to provide a list of objects that a constraint could reference. For example, a designer could provide a list of menu items and a feedback object would be able to appear over any of them. However, Coral did not allow constraints to reference arbitrary objects through variables, and thus did not provide the full generality of indirect reference constraints.

Thinglab [2] also provides a limited form of indirect reference constraints. Designers can construct pathnames that allow a constraint to traverse a structure hierarchy to find an object. If one of the components in the structure hierarchy changes, the new object will be automatically referenced by the constraint. However, arbitrary references to objects through pointer variables are not supported. Penguins [7] supports a model of indirect reference constraints that is similar to the one described in this paper but it uses a different constraint solving algorithm. Many other systems, such as Grow [1], Apogee [5], Peridot [9] and CONSTRAINT [19], allow constraints to directly reference objects but do not allow indirect references.

Kaleidoscope supports a different type of abstraction—constraint abstraction rather than procedural abstraction—in which procedures consist of a set of parameterized constraint statements and produce as output a set of constraints instantiated with the parameters passed to the procedure [3].

Finally, a number of researchers have developed models that allow constraints to have variables, but not pointer variables [16, 15, 8, 14]. For example, a programmer could write a constraint such as `feedback.position = item1.position + offset`, initially assign the value of 10 to `offset`, and later assign the value of 20 to `offset`. However, a programmer could not write a constraint of the form `feedback.position = self.obj-over.position + offset`, where `obj-over` is a pointer variable that points to an arbitrary object.

3 Applications of Indirect Reference Constraints

Indirect reference constraints can be used to implement many parts of an application that are difficult or infeasible to implement with direct reference constraints. These include feedback, copying and instancing of composite objects with constraints in them, programming by example, abstract specification of layouts, and simulations.

3.1 Feedback

Most direct manipulation interfaces provide feedback to the user while performing an operation. For example, a rectangle may highlight the item that the user is currently pointing at in a menu (Figure 1.a). While it is generally impractical to handle feedback objects using direct reference constraints, they are easily handled using indirect reference constraints. For example, the feedback object in Figure 1.a must be able to highlight any of the menu items, but a direct reference constraint will only allow it to highlight one of these items. In contrast, indirect reference constraints allow the feedback object to reference any of these menu items through a variable, such as *obj-over*. This technique works equally well for feedback objects that highlight a fixed set of objects, such as the objects in a menu, or a dynamic set of objects, such as the objects in a drawing window (Figures 1.a and 1.b).

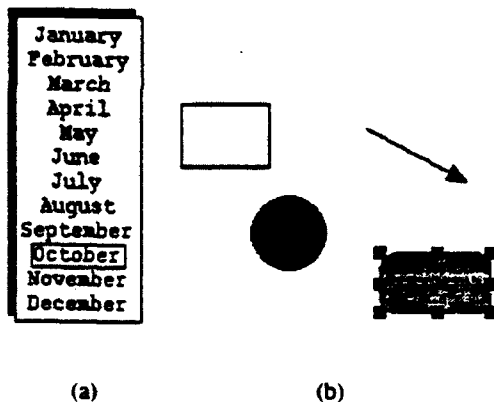


Figure 1.

The rectangular feedback object in the menu and the selection handles in the drawing editor use constraints to center themselves over the selected items and to change their dimensions to the dimensions of the selected item. By indirectly accessing a selected item through the variable *obj-over*, the feedback objects are able to appear both over any item in a static set of objects, such as the menu items (a), or any item in a dynamic set of objects, such as the objects in the drawing editor (b).

3.2 Structured Objects

Pointer variables simplify the integration of constraints into a structured object system. A structured object consists of several parts, such as the labeled box in Figure 2, which consists of a rectangle and a piece of text. Typically these parts are mutually constrained. For example, the label is centered inside the box and the size of the box depends on the size of the label.

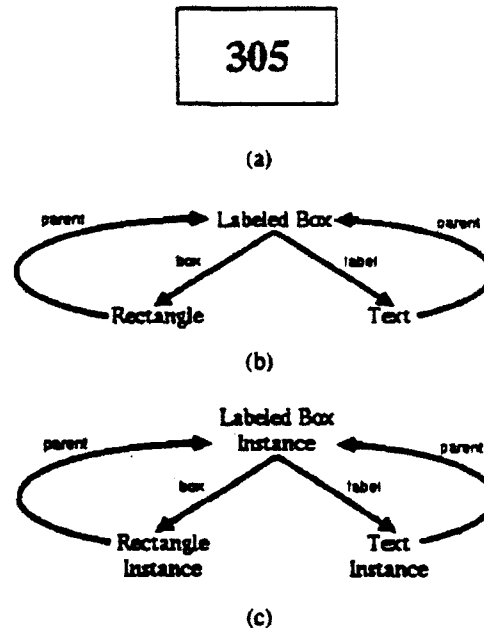


Figure 2.

Structured objects, such as this labeled box (a), are built up from other objects, such as this rectangle and number (b). Each object maintains pointers to its parent and its children, so that constraints can indirectly reference one another through pointers. This facilitates the copying and instancing of objects, since the object system simply sets the pointers in the new objects, and the constraints automatically reference the appropriate objects (c).

Interactive applications need to make copies or instances of these objects at runtime (e.g., creating new objects in a drawing program, creating new circuit elements in a circuit simulation program). These operations can be easily implemented using indirect reference constraints, but are quite difficult to implement in regular constraint systems.

In an indirect reference constraint system, each object maintains a pointer to its parent, and a set of pointers to its

children (Figure 2.b). Constraints reference objects by following the appropriate pointers. For example, if the label's parent pointer is contained in the variable `parent` and the labeled box keeps pointers to its children in the variables `label` and `box`, then the label can be centered inside the box using the following constraints:

```
center-x = self.parent.box.center-y
center-y = self.parent.box.center-y
```

To create an instance of an object, the object system creates instances of each of the object's components and sets the pointer variables (Figure 2.c). The object system also creates instances of each of the constraints in the prototype's components and stores them in the appropriate places in the new instance's components. No changes are needed to the constraint expression. The constraints in the newly created objects will automatically reference the appropriate objects since they will follow the pointers in the instance objects rather than in the prototype objects. For example, the constraint that computes the value for `center-x` in the label instance will follow the parent and box pointers in the labeled box structure hierarchy and retrieve the `center-x` value of the rectangle instance.

In a direct reference system, constraints must use hardcoded references to objects. For example, the label in Figure 2 could be centered inside the box using the following direct reference constraints:

```
center-x = box.center-x
center-y = box.center-y
```

When a new instance of labeled-box is created, the object system will have to replace all references to box with references to the newly created instance of box.

The object system will have to track down the references by manually traversing the prototype's hierarchy to find where box is in relation to label, (the relation is go to label's parent, which is labeled box, then to labeled box's first child, which is box), then use the same traversal in the instance's hierarchy to find the appropriate reference to the newly created instance of box. Thus it is much simpler and more efficient to implement copying and instancing operations in indirect reference systems than in direct reference systems.

3.3 Programming by Example

Indirect reference constraints make it easier to implement systems that employ demonstrational programming, such as the graphical interactive design tool Lapidary [10]. In a demonstrational system, a user draws an example picture or demonstrates an example behavior, and then the system creates a prototype object or behavior by generalizing the picture or demonstrated behavior. If the demonstrational system uses indirect reference constraints, then it is easy to generalize these examples. In fact, the example that the user draws or demonstrates is already a prototype, since the

object can be instanced or copied using the scheme described in the previous section.

In Figure 3, a designer is using Lapidary to create a boxes-and-arrows editor. The designer has drawn an example picture in which the arrows are attached to the center of the boxes they connect. Lapidary represents the constraints of the line internally as indirect reference constraints:

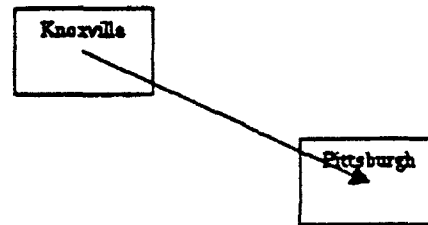


Figure 3.

An example picture demonstrating that the endpoints of an arrow should be attached to the centers of the boxes it connects. An interface builder will generalize this arrow into a prototype that can connect any pair of boxes.

```
endpt1 = self.from-obj.center
endpt2 = self.to-obj.center
from-obj = box1
to-obj = box2
```

The designer can save this arrow and the application can use it as a prototype. When the boxes-and-arrows editor creates instances of this arrow, it stores pointers to the appropriate boxes in the `from-obj` and `to-obj` variables, and the constraints automatically attach the endpoints of the arrow instance to the centers of the boxes. The application does not need to know anything about the constraints, structure, or graphics of the line. The constraints on the endpoints could connect centers to centers, right sides to left sides, or even use a complex formula that computes the nearest sides and tries to avoid crossing other lines.

3.4 Abstract Specification of Layouts

Indirect reference constraints facilitate the specification of layouts, independently of the objects to be layed out. For example, a designer might want to specify that a generic feedback object should appear over a selected object. The actual type of feedback used might depend on the size of the selected object and the type of the selected object. As another example, Humanoid [18] and Jade [20] allow a designer to define constraint-based rules that describe the general layout of dialog boxes in terms of the generic parts

of a dialog box, such as a title, a body that contains the items of the dialog box, an OK button, and a cancel button. One can then apply the rules to different dialog boxes, irrespective of the widgets that fill the roles of the different parts. For example, storing the following constraint on the x coordinate of the OK button will force the button to be placed 10 pixels to the right of the dialog box's title, regardless of which widget is used for the title or the OK button:

```
left = self.parent.title.right + 10
```

3.5 Simulations

Simulations often require objects to move smoothly between various points of the display. For example, sort animations show objects moving around in linked lists or arrays, navigation systems move objects around transportation corridors, and manufacturing systems route objects through the machines on a factory floor. Indirect reference constraints model this motion by using variables to reference the beginning and target positions.

For example, suppose we want the carton in Figure 4 to glide from station A to station B as if it were on a conveyor belt. This could be done by writing a set of constraints that interpolate the carton's position based on a timer and the stations' positions:

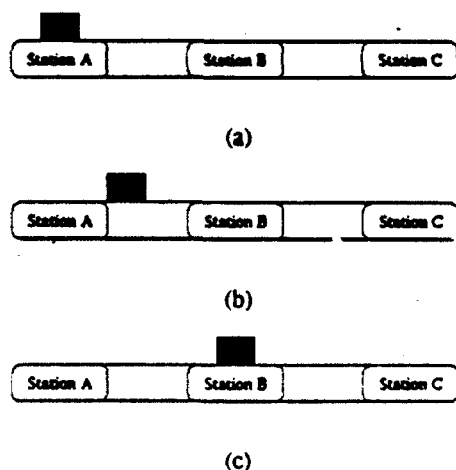


Figure 4.

An assembly line with stations connected by a conveyor belt. A carton should be centered above the station that is currently processing it (a), and cartons should move smoothly from one station to the next, (b) and (c).

```
time = 0
x-distance = self.to-station.center-x
```

```
- self.from-station.center-x
center-x = self.from-station.center-x
+ (self.x-distance * self.time)
to-station = station-b
from-station = station-a
```

x-distance computes the distance between the old and new stations, and center-x contains the current x position of the carton. As the application program increments time from 0 to 1, the carton moves smoothly between its old and new assembly station. To prepare for the next step of the animation, the application resets the timer to 0, stores station B in from-station, and stores a new station, C, in to-station.

4 Performance and Implementation Advantages of Indirect Reference Constraints

The generalization of constraints using pointer variables can improve the efficiency of an application by reducing the number of constraints and objects it uses, reducing the size of the constraints it uses, and reducing the number of constraints that it must dynamically create and delete. Indirect reference constraints also make it easier for the constraint system to maintain one rather than multiple copies of a constraint and make it easier to statically compile the constraints.

Storage improvements come in two forms. First, by allowing objects to be constrained to many different objects, indirect reference constraints may significantly decrease the number of objects which an application creates. For example, suppose a feedback object should highlight the currently selected item in a menu, as in Figure 1.a. If direct reference constraints are the only constraints available, the designer may create a separate feedback object for *each* menu item, since the constraints will bind each feedback object to exactly one item. However, as noted in Section 3.1, indirect reference constraints allow a feedback object to highlight any menu item, and thus one feedback object suffices. Second, indirect reference constraints can be written much more compactly and elegantly than direct reference constraints. Returning to the feedback example, a clever designer who is working with direct reference constraints might be able to use only one feedback object by defining constraints which reference every object in a menu and describe how the feedback object should highlight each menu item. For example, to implement the feedback object in Figure 1.a, the designer might write the following constraint to define the left side of the feedback object:

```
feedback.left = case month
"January": Jan.left
"February": Feb.left
"December": Dec.left
```

where month is a string variable containing the currently selected month.

However, this solution has four drawbacks:

- Non-modular and Inelegant: If the designer

adds a new menu item, the designer must also remember to modify the constraints in the feedback object.

- **Space:** The constraint must have twelve separate conditions and actions, which causes the code to occupy a considerable amount of space at runtime. Also, 12 dependency pointers, one for each object, must be maintained by the constraint system.
- **Efficiency:** The constraint depends on all twelve objects. If one object changes, even if it is not the currently selected object, the constraint must be reevaluated.
- **Dynamic Sets:** This technique only works for static sets of objects, since the objects must be hardcoded in the constraint. It cannot be used to describe dynamic sets of objects, such as the objects in the drawing window in Figure 1.b.

Indirect reference constraints suffer from none of these disadvantages. The corresponding indirect reference constraint would be:

```
feedback.left = self.obj-over.left
```

where `obj-over` is a pointer to the selected menu item.

This constraint is both compact and modular. The designer can add or delete items from the menu without worrying about the effect of the change on the feedback object. It occupies less space than the corresponding direct reference constraint and requires only two dependency pointers, one to the variable `obj-over` and one to the selected menu item. It depends only on the `obj-over` variable and the currently selected menu item, so it will only be reevaluated when absolutely necessary. Finally, this type of constraint can handle dynamic sets of objects. If additional items are added to the menu, the constraint automatically deals with them without having to be rewritten.

The efficiency advantage of indirect reference constraints derives in part from their storage advantage. Fewer constraints means fewer constraints to solve, and thus, less work for an equation solver. For example, a constraint solver that employs eager evaluation might take only one fifth the time to set up the feedback for a five item menu using indirect reference constraints instead of direct reference constraints, because there are only one fifth as many constraints to solve.

Efficiency advantages also arise because 1) an object system does not have to locate and replace direct references; and 2) fewer constraints have to be dynamically created and destroyed. The search and replace issue was discussed in section 3.2. To illustrate the reduction in dynamically created and destroyed constraints, consider the construction of a menu using direct reference constraints. It may be

impractical from a storage standpoint to maintain one feedback object for each item. Thus, the application may maintain only one feedback object and destroy the old constraints and create new constraints each time the feedback moves to a new item. The overhead involved in destroying and creating these constraints can be avoided if indirect reference constraints are used.

Indirect reference constraints also simplify the construction of the constraint system. First, the formula for an indirect reference constraint can be stored in a prototype and instances of the prototype can maintain pointers to this formula. Thus, many instances of a prototype constraint can be created, but the formula is created only once. Second, the parameters to a constraint are implicitly declared by pointer variables, so the constraint system can statically compile constraints by wrapping a function header around them. This considerably simplifies implementing a constraint system in an existing general-purpose language. For example, Garnet constraints can be arbitrary Lisp code. Direct reference systems typically also maintain only one copy of a constraint and statically compile it. However, to accomplish this, they require the user to write the constraint as a function, complete with parameters denoting the direct references, or else parse the constraint to determine the direct references. However, we have discovered that users find it irritating and cumbersome to have to define parameters for constraints. For example, it is much more elegant and compact to write

```
feedback.left = self.obj-over.left
```

than to write

```
constraint left-align (obj) ( obj.left )
feedback.left = left-align(self.obj-over)
```

Similarly, it can be quite difficult to write a parser to search through each constraint and locate the direct references.

5 Implementation

The algorithms for implementing indirect reference constraints build on the algorithms for implementing direct reference constraints.

5.1 Lazy Evaluation

A variation of nullification/reevaluation algorithms can be used to handle indirect reference constraints. Nullification/reevaluation algorithms represent the constraints as a directed graph with nodes representing constraints, and edges (called dependencies) representing data flowing from one constraint to another (Figure 5.a). When the value of a node changes (typically a "leaf" node such as *e* or *f*), all nodes that directly or indirectly depend on this changed node are marked out of date. When the value of a node is requested, the constraint that computes its value starts demanding the values of other nodes on which it depends. If these nodes are out of date, they will recursively demand the values of the nodes they depend on, until eventually nodes are reached whose values are up to date, at which point the constraints can compute their value and

return [13, 6]. For example, suppose that node *e* is changed in Figure 5. The lazy evaluator will mark the nodes *b*, *d*, and *a* as out of date (Figure 5.b). If the value of node *a* is then requested, *a* will demand the value of *d*. *d* is out of date so it demands the values of *e* and *f*, both of which are up to date, computes its own value, marks itself up to date, and returns its value to *a*. *a* then demands the value of *c* which is up to date, computes its own value, marks itself up to date, and returns.

Nullification/reevaluation algorithms were originally constructed with the assumption that the edges in the graph remain static while the constraint solver is evaluating the graph. However indirect reference constraints can cause the graph to dynamically change as the constraints are being evaluated, because the pointer variables may change, causing a constraint to access information from a different set of nodes. For example, when the constraint on node *a* is being evaluated, it may start referencing node *b* rather than node *c* (Figure 5.c).

To handle this situation, we have extended the algorithm so that dependencies can be dynamically created and deleted as the constraints are being evaluated. Dependencies are timestamped so that if they are not used by a constraint in a subsequent evaluation, they become stale and are discarded. When a constraint demands a variable, the constraint solver either creates a new dependency between the constraint and the variable if such a dependency did not already exist, or else updates the time stamp on the dependency so that it matches the timestamp on the constraint (a constraint is timestamped each time it is evaluated). The constraint solver removes stale dependencies as it invalidates constraints. Before following a dependency, it checks whether the dependency's timestamp matches the timestamp of the constraint it points to. If the two timestamps disagree, the dependency is discarded. A beneficial side effect of this scheme is that constraints which involve conditionals depend only on the variables that make up the condition and the branch of the condition that is executed. Thus the number of dependency pointers and unnecessary evaluations are minimized.

To see that this scheme works, note that a constraint will dynamically add or delete dependencies only if it contains pointers or conditionals. If a constraint depends on pointer variables, the constraint will be marked out of date when the pointer variables change and the constraint will be reevaluated when its value is next requested. At this point, the constraint solver will add edges to this constraint from the new set of nodes it references (Figure 5.c). The dependencies to variables that are not requested by the constraint on this evaluation will become stale and be removed the next time these dependencies are examined. Thus the constraint will demand the values of the correct set of nodes and will obtain the correct result.

In the case of a conditional, the branch or branches of the conditional that were ignored during the previous evalua-

tion of the constraint will only have to be evaluated if the condition itself changes. Since the constraint depends on the variables in this condition, it will be marked out of date when one of these variables changes and will be automatically reevaluated (of course it will also be reevaluated if one of the variables in the branch that was last executed is changed). Again, the constraint solver will add dependency edges to this constraint from the new set of variables it references in whichever branch is executed and remove edges that emanate from variables in the previously executed branch. Thus constraints with conditionals will always be evaluated correctly.

5.2 Eager Evaluation

Our eager evaluation algorithm uses a variation of an eager evaluator developed by Roger Hoover [4]. Like the lazy evaluator, this algorithm makes use of dataflow graphs. However, it assigns priorities to the nodes in the graph, indicating the nodes relative position in topological order (Figure 6.a). When a node changes value, all its immediate successors are added to a priority queue based on their priorities. When the evaluator starts executing, it removes the lowest priority node from the queue and evaluates it. By evaluating the lowest priority node in the queue, the evaluator ensures that the values of all nodes that the constraint associated with this node may request are up to date.

In the Hoover algorithm, the priorities are maintained in an ordered list and each node in the dataflow graph points to one of these priorities. Comparisons between priorities can be performed in $O(1)$ time and insertions of new priorities can be accomplished in amortized $O(1)$ time. When an edge is added to a dataflow graph, the algorithm checks whether the priority of the source node is greater than or equal to the priority of the destination node (data flows from the source node to the destination node; for example *e* is the source node and *h* is the destination node for the edge that connects these nodes). If the priorities are out of order, the algorithm follows the successors of the destination node transitively until it reaches nodes whose priority numbers are greater than the priority of the source node (the Hoover algorithm will also follow predecessors of the source node; however, to save space we do not maintain backpointers and thus we cannot search backward from nodes). These nodes are termed boundary nodes. The algorithm works back from these boundary nodes and assigns to the intermediate nodes new priority numbers that are between the priority numbers of the source and boundary nodes. If it runs out of existing priority numbers, it creates new ones by inserting records into the ordered list directly after the record associated with the priority number of the source node.

For example, suppose a dependency from node *d* to *f* is added in Figure 6.b. Node *d* has priority 2 while node *f* has priority 1 so the nodes are out of order. The algorithm goes to node *g*, which has a priority of 2, and then to node *h*, which has a priority of 3. Since this is greater than node *d*'s

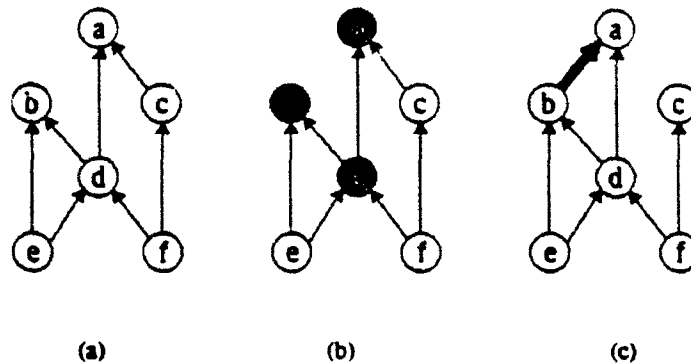


Figure 5.

(a) Constraints are represented as nodes in a directed graph. The edges represent data computed by a constraint that another constraint uses. (b) The gray nodes represent nodes marked out of date when node *e* is changed. (c) Node *a* now depends on node *b* rather than node *c*.

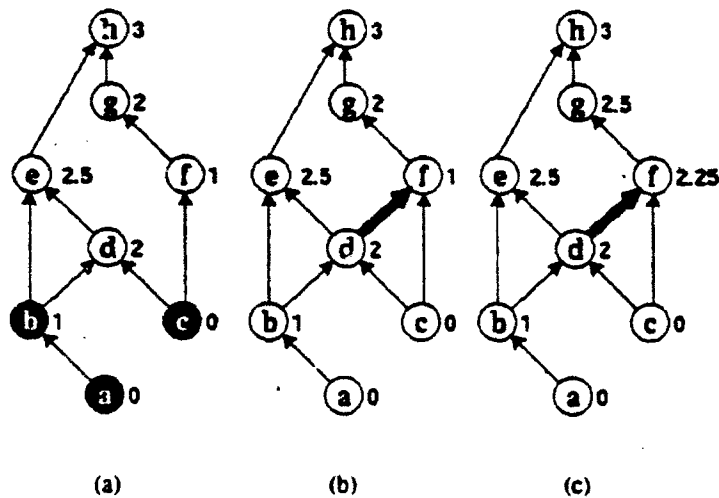


Figure 6.

(a) Numbers are assigned to nodes according to the order in which they are evaluated. Nodes cannot be evaluated until all their predecessors have been evaluated. Darkened nodes represent evaluated nodes. Nodes *d* and *f* are ready for evaluation. (b) Node *f* now depends on node *d* as well as node *c*. (c) Nodes *f* and *g* must be renumbered to make their priorities agree with their position in topological order.

priority, the search stops here and node *h* becomes a boundary node. In this case there is one priority, 2.5, between 2 and 3 in the priority list, so the algorithm assigns this priority to node *g*.² At this point the algorithm runs out of

existing priorities, so it inserts a new one in the ordered list and assigns it to node *f* (Figure 6.c).

The Hoover algorithm assumes that dataflow graphs cannot change once constraint evaluation begins, so the reordering scheme and the evaluator can be invoked in sequence. However indirect reference constraints may cause the edges of the graph to change *during* constraint evaluation. Thus

²We are using rational numbers for illustrative purposes only. The actual algorithm for maintaining the ordered list uses only integers and does some reordering to ensure that only integers are used.

the numbers assigned to the nodes may become incorrect and force an equation to be evaluated prematurely. To overcome this difficulty, we have taken the approach of dynamically updating the topological order each time the graph changes, and evaluating nodes according to this revised topological order. In other words, the reordering algorithm and the constraint evaluator are interleaved. Dependencies are dynamically created and deleted in the same way as in the lazy evaluation algorithm.

The time complexity of this algorithm depends on the number of reorderings and the time required by each reordering. Assuming a bounded number of variables per equation, a reasonable assumption for constraints in graphical interfaces, a reordering requires $O(k)$ time where k is the number of nodes that must be reordered (Hoover's algorithm has an additional log factor in the running time since it uses a priority queue to guide its forward and backward searching; however this forward and backward searching may reorder fewer nodes than our algorithm, thus offsetting the log factor). Assuming there are n nodes in the graph, the worst case running time of the algorithm is $O(dn)$ where d is the number of dynamically added edges. As with most incremental algorithms, this worst case running time is misleading, since most node evaluations do not trigger a reordering and most reorderings do not visit all n nodes. Indeed, results based on preliminary testing of the algorithm suggest that pointer variables typically do one of two things: 1) they shift between nodes whose priority numbers are identical, thus causing no reordering to occur; or 2) they shift between a fixed set of nodes, and once they have shifted to the highest number node, reordering never occurs again. The former case arises frequently in simulations where an object is typically moving between independent but fairly similar objects that have roughly the same number of constraints and the latter case arises frequently in menus where the last item has the constraints with the highest priority number (because it is the last item laid out). Thus in practice, the algorithm appears to fairly rapidly quiesce to a state where very few reorderings occur during constraint evaluation.

Other Implementation Issues

Each time a constraint is evaluated, its value is cached so that the next time the constraint's value is requested, the constraint will not be reevaluated unless one of its parameters has changed. Similarly the values of paths can be cached to improve efficiency. For example, in the labeled box example presented in Section 3.2, the label accessed the position of the box using the path (`self.parent.box`). The first time this path is evaluated, the constraint solver can cache the resulting pointer to the box, so that as long as the variables comprising the path do not change, the constraint behaves as a direct reference constraint. The variables on this path still maintain dependency pointers to the constraint, so that if one of these variables changes, the path can be reevaluated and a new value cached for it.

Another implementation issue that arises is what to do with constraints containing variables that are nil or which reference deleted objects. The two options considered in Garnet were 1) to destroy the constraint, keeping its previously computed value; or 2) to keep the constraint and return its previously computed value. Under option two, the constraint would again be evaluated once all its variables point at valid objects. We settled on the second option since, in many cases, the constraint will be used again. For example, feedback objects that are invisible may have their `obj-over` variables set to nil, yet the constraints should be maintained so that they can correctly position the feedback object once it is made visible and its `obj-over` variable is set to a newly selected item.

6 Status and Future Work

Indirect reference constraints have been completely implemented at a very low level in Garnet. Every layer in Garnet is implemented on top of the constraint system using indirect reference constraints, except for the lowest-level untyped object system. This includes the graphical object system, the handling of the input, and all the widget libraries. In addition, Garnet has approximately 150 users who have used indirect reference constraints to generate hundreds of applications.

Garnet currently uses lazy evaluation and a modified user-controlled version of caching that evaluates a path the first time the constraint is evaluated and then ignores it if the user assures the constraint solver that the path will never change. On a SUN Sparcstation 1+ running Lucid Common Lisp, an indirect reference to an object through a variable (e.g., `self.obj-over.left`) requires 170 microseconds, whereas a direct reference (e.g., `menu-item1.left`) or a reference that uses a cached path requires 54 microseconds. If a constraint does not have to be reevaluated, its previously computed value can be accessed in 19 microseconds, regardless of whether it is a direct reference or indirect reference constraint. Garnet's constraint solver can solve indirect reference constraints quickly enough to allow feedback objects to track the mouse in real time or to perform smooth, realtime animations, even in large, constraint-based applications. For example, the Lapidary interactive design tool [10] consists of 16000 lines of Lisp code and 23500 constraints, all of which are indirect reference constraints, and is fast enough to provide instantaneous feedback to the user.

We have a working version of an eager evaluator that we believe is more efficient than the current lazy evaluator and which should be implemented in Garnet in the near future. We also have a design for two-way indirect reference constraint systems. Finally, we are examining graphical means of tracing these constraints so that designers can debug them more easily [12].

7 Conclusions and Future Work

Indirect reference constraints allow procedural abstraction to be added to constraint systems. This significantly extends the potential uses of constraints in interactive applications by allowing constraints to express the dynamic behavior that occurs inside an application's window. These constraints can be used to specify animations and feedback that operate over dynamic sets of objects, implement copying and instancing of structured objects in prototype-instance systems, simplify the creation of prototype objects from example objects in demonstrational systems, and abstractly specify layouts. In addition, their programming style is simpler and more effective than conventional constraints, they improve the efficiency of applications, and they decrease an application's storage demands. Because of their flexibility and ease of use, indirect reference constraints have permitted a comprehensive user interface toolkit to be built for the first time on top of a constraint system. This represents an important step toward the development of a general-purpose, constraint-based, interactive programming language.

References

1. Paul Barth. "An Object-Oriented Approach to Graphical Interfaces". *ACM Transactions on Graphics* 5, 2 (April 1986), 142-172.
2. Alan Borning and Robert Duisberg. "Constraint-Based Tools for Building User Interfaces". *ACM Transactions on Graphics* 5, 4 (Oct. 1986), 345-374.
3. Bjorn N. Freeman-Benson. Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming. OOPSLA/ECOOP'90 Conference Proceedings, 1990, pp. 77-88.
4. R. Hoover. *Incremental Graph Evaluation*. Ph.D. Th., Department of Computer Science, Cornell University, Ithaca, NY, 1987.
5. Scott E. Hudson. Graphical Specification of Flexible User Interface Displays. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'89, Williamsburg, VA, Nov., 1989, pp. 105-114.
6. Scott E. Hudson. Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update. Tech. Rept. TR89-12, The University of Arizona, 1989.
7. Scott E. Hudson. An Enhanced Spreadsheet Model for User Interface Specification. Tech. Rept. TR90-33, The University of Arizona, 1990.
8. J. Jaffar and J. Lassez. Constraint Logic Programming. Proceedings of the Principles of Programming Languages Conference, ACM, Munich, Germany, Jan., 1987, pp. 111-119.
9. Brad A. Myers. *Creating User Interfaces by Demonstration*. Academic Press, Boston, 1988.
10. 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.
11. Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. "Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment". *IEEE Computer* 23, 11 (Nov. 1990), 71-85.
12. Brad A. Myers. Graphical Techniques in a Spreadsheet for Specifying User Interfaces. Human Factors in Computing Systems, Proceedings SIGCHI'91, New Orleans, LA, April, 1991, pp. 243-249.
13. T. Reps, T. Teitelbaum, and A. Demers. "Incremental Context-Dependent Analysis for Language-Based Editors". *ACM TOPLAS* 5, 3 (July 1983), 449-477.
14. V. A. Saraswat. *Concurrent Constraint Programming Languages*. Ph.D. Th., School of Computer Science, CMU, Pittsburgh, PA, 1989.
15. Guy L. Steele, Jr. *The Definition and Implementation of A Computer Programming Language based on Constraints*. Ph.D. Th., Department of Computer Science, MIT, Boston, MA, 1980.
16. Ivan E. Sutherland. SketchPad: A Man-Machine Graphical Communication System. AFIPS Spring Joint Computer Conference, 1963, pp. 329-346.
17. Pedro A. Szekely and Brad A. Myers. "A User Interface Toolkit Based on Graphical Objects and Constraints". *Sigplan Notices* 23, 11 (Nov. 1988), 36-45. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'88.
18. Pedro Szekely. Template-Based Mapping of Application Data to Interactive Displays. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'90, Snowbird, Utah, Oct., 1990, pp. 1-9.
19. Brad T. Vander Zanden. Constraint Grammars—A New Model for Specifying Graphical Applications. Human Factors in Computing Systems, Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 325-330.
20. 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, April, 1990, pp. 27-34.

Reprinted from *Proceedings OOPSLA'92: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. October 18-22, 1992. Vancouver, BC, Canada. *SIGPLAN Notices*, vol. 27, no. 10. pp. 184-200.

DECLARATIVE PROGRAMMING IN A PROTOTYPE-INSTANCE SYSTEM: OBJECT-ORIENTED PROGRAMMING WITHOUT WRITING METHODS

Brad A. Myers

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
bam@cs.cmu.edu

Dario A. Giuse

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
dzg@cs.cmu.edu

Brad Vander Zanden

Computer Science Department
University of Tennessee
107 Ayres Hall
Knoxville, TN 37996-1301
bvz@cs.utk.edu

ABSTRACT

Most programming in the Garnet system uses a declarative style that eliminates the need to write new methods. One implication is that the interface to objects is typically through their data values. This contrasts significantly with other object systems where writing methods is the central mechanism of programming. Four features are combined in a unique way in Garnet to make this possible: the use of a prototype-instance object system with structural inheritance, a retained-object model where most objects persist, the use of constraints to tie the objects together, and a new input model that makes writing event handlers unnecessary. The result is that code is easier to write for programmers, and also easier for tools, such as interactive, direct manipulation interface builders, to generate.

KEYWORDS: Object-Oriented Programming, Prototype-Instance Model, Toolkits, Declarative Programming, Constraints, Input, Garnet.

1. Introduction

Over the last three years of using the Garnet system to create dozens of large-scale user interfaces, we have observed that the style of programming in Garnet is quite different from that in conventional object-oriented systems. In Garnet, programmers combine pre-defined objects into collections, use constraints to define the relationships among them, and then attach pre-defined "Interactor" objects to cause the objects to respond to input. The result is a declarative style of programming where the programmer rarely writes methods. Furthermore, the interface to objects is usually through direct accessing and setting of data values, rather than through methods.

The features of the Garnet object system have been motivated by the overall goal of the project: to provide high-level, interactive, mouse-based tools for rapidly prototyping and creating graphical, highly-interactive, direct manipulation programs. Because of the emphasis on rapid creation and easy editing, we have chosen to make the object system completely flexible and dynamic. Since the interactive tools need to be able to generate code for the interface and then read the code for later editing, it is easier to generate high-level, declarative specifications. Because much of the look and the dynamic behavior in Garnet can be specified by supplying parameters to pre-defined objects, it is easier for interactive tools to display these options in dialog boxes or intelligently guess them using demonstrational techniques.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1992 ACM 0-89791-539-9/92/0010/0184...\$1.50

In order to achieve these goals, we have made a number of interesting design decisions which contribute to Garnet's unique programming style. First, Garnet uses a prototype-instance model rather than the more popular class-instance model. In a prototype-instance model, there is no distinction between instances and classes; any instance can serve as a "prototype" for other instances. Garnet's model is unique in that it supports structural-inheritance. This means that when a prototype object is a collection (or "aggregate") of other objects, Garnet creates instances of all components when the aggregate is instanced. Therefore, the programmer can construct complex graphical objects by declaratively listing the primitive component objects. It is not necessary to write creation or drawing methods.

Second, the objects in Garnet are usually persistent and long-term. For example, the graphics model requires that there be an object in memory corresponding to each object on the screen. This means that the programmer does not have to deal with object refresh, and allows the toolkit to contain high-level support, like selection handles. In many other systems, a single object can be used like a stamp-pad and drawn in multiple places on the screen.

Third, constraints can be used to declare the relationships among the objects. Constraints in Garnet are tightly integrated with the object system, so that any slot of any object can have a constraint which calculates its value. The result is that the interface to objects is usually through data values which are directly accessed and set, rather than through methods. Constraints are used to propagate the changes appropriately.

Fourth, Garnet incorporates a novel input model, which provides standard objects called "Interactors" to handle the most popular direct manipulation behaviors. This is based on the Model-View-Controller idea from Smalltalk [8], where the Interactors correspond to the controllers. In Garnet, however, unlike in Smalltalk and other implementations of this idea, the programmer rarely writes new Interactor methods. Instead, the programmer attaches an instance of a pre-existing Interactor object to the graphical objects using constraints, and declaratively specifies any necessary controlling parameters for the Interactor.

This paper discusses these aspects of Garnet, and shows the advantages of the Garnet style of programming. Even though conventional wisdom for object-oriented programming is that writing methods is "good" and exposing the objects' data is "bad," we show that the Garnet style is just as modular and provides just as much information hiding. Furthermore, there is some evidence that, at least for user interface programming, it is more effective.

Garnet is a comprehensive user interface development environment in Lisp for X/11.¹ It is in the public domain and is freely available. Currently, over 30 projects around the world are using the system regularly.² The system contains a number of features that make it well-suited for creating graphical user interfaces. Unlike other toolkits which primarily supply widgets, Garnet is specifically designed to cover *all* aspects of user interface programming, especially the insides of application windows. While there have been a number of papers about Garnet [17] and its components [23, 15, 24, 19], this is the first paper about the programming style. For a complete discussion of programming in Garnet, see the reference manual [20].

2. Related Work

In the terms of the "Treaty of Orlando" [22], the Garnet object system is 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. These features make Garnet much like SELF [5] and other prototype-instance systems [9]. However, unlike SELF, Garnet rarely uses multiple inheritance (although it is allowed), and we have integrated a constraint solving mechanism with the

¹Display Postscript and Macintosh versions are in progress.

²You can get Garnet by anonymous FTP from `a.gp.cs.cmu.edu`. Change to the directory `/usr/garnet/garnet/` (note the double garnet's) and retrieve README for instructions. Or you can send electronic mail to `garnet@cs.cmu.edu`.

object system. Another important difference is that Garnet encourages programmers to directly access and set slots of objects, whereas SELF prevents this and only provides methods.

Many systems have used constraints as part of an object system [3], but none is as general-purpose or fully-integrated as Garnet. Garnet is also the first system to introduce pointer variables into constraints (where the objects referenced by the constraint can change). The first integrated constraint and object system was ThingLab [2], which supported multi-way constraints. ThingLab was also a prototype-instance object system. Apogee [7] and Grow [1] are more closely related to Garnet in goals, since they are user interface toolkits. Also, like Garnet, they implement one-way constraints. Neither, however, uses constraints as the primary mechanism for information passing, so they both make extensive use of methods.

As was mentioned, Garnet's input model is based on the Model-View-Controller idea from Smalltalk [8]. Other attempts to capture interactive behaviors include the model used by graphics standards, such as PHIGS, GKS, CGI, CORE, etc., which identifies five or six basic input types (e.g., locator, stroke, valuator, choice, pick and string for PHIGS). This is based on a model by Foley and Wallace [6]. Unfortunately, this model has proven unusable for modern user interfaces [12].

The current object and constraint system is a complete redesign and rewrite of the Coral system [23]. Coral was implemented in CLOS, but was abandoned because it was too slow and inflexible in practice. Like Garnet, Coral provided a declarative syntax for objects and constraints, but it was not possible to modify objects once they had been created. Coral used a conventional class-instance model, rather than the prototype-instance model we now use. It also required that the constraints be parsed to search for object references, which limited the kinds of constraints that could be written. The current Garnet constraint system does not need to parse constraints because it dynamically determines the dependencies when the constraint is evaluated. Coral did not provide for arbitrary pointer variables in constraints like Garnet does now, and it used active values, which we have found to be unnecessary in Garnet with fully func-

tional constraints. Coral had a special-purpose mechanism for constraints over lists of objects, such as the items of a menu. For example, you could specify a constraint for the top of the first item and a different constraint for the rest of the items. This is not needed in Garnet due to the support for arbitrary code in constraints (you can just use Lisp's looping facilities). The create routines in Coral were specific to each class, rather than a generic function that would work for all classes. Other important problems with Coral were that the declarative technique did not support changing objects after they were created, and it was not available to interactive editors. Therefore, a separate procedural mechanism was supplied. In the current Garnet, the declarative and procedural mechanisms have equivalent power.

3. The Prototype-Instance Object Model

The Garnet object system implements the prototype-instance model [9], and supports completely dynamic redefinition of prototypes with automatic change propagation. There is no distinction between instances and classes; any instance can serve as a "prototype" for other instances. All data and methods are stored in "slots" (sometimes called "fields" or "instance variables"). An instance can add any number of new slots, and slots that are not overridden in an instance inherit the values from its 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 in Common Lisp, a function 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 Smalltalk. The ability to dynamically add, delete, and modify methods has proven important in graphical interface builders since they need to temporarily insert their own methods during "build" mode, and then retract them during "test" mode.

All objects are created with the standard function `create-instance` which takes an optional name of the new object, an optional object to be used as a prototype, and a list of slots and values that should have local values. Slots that are not mentioned start out using the inherited, default value from the

prototype (which can be changed later). Slot names start with colons, and can contain any number of printable characters (e.g., :left, :interim-selected, :obj-over). In the following example, the rectangle named my-rect will inherit the :top, :width, and :height from the prototype rectangle:

```
; create an object named rectangle inheriting from nothing.
(create-instance 'rectangle NIL
  (:top 10) (:left 10) ; specify values for some slots.
  (:width 20) (:height 25) (:color black))
```

```
; create my-rect inheriting from rectangle.
(create-instance 'my-rect rectangle
  (:left 45) (:color blue)) ; override two slots.
```

Setting an object's slot with a value automatically creates the slot, if needed. This makes it extremely easy to associate any piece of information with any object, since slot names do not have to be predefined. For example, the following will create a new slot in rectangle:

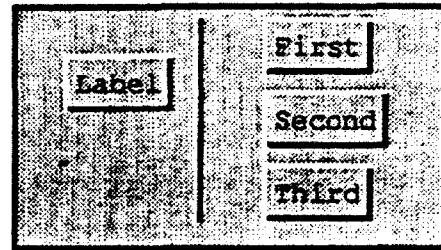
```
(s-value rectangle :perimeter 90)
```

Because my-rect inherits from rectangle, it will also now have the new slot.

There is a special kind of object in Garnet called an "aggregate" which is a collection of other objects. A unique feature of Garnet is that whenever an instance is made of an aggregate, Garnet automatically creates instances of all its components, and links them together appropriately. This "structural inheritance" is an extremely powerful abstraction, because it frees the user from having to know whether an object being instanced is a primitive object like rectangle or a composite like button; the create-instance call is the same.

For example, a button might be composed of three rectangles and a text object. The programmer can declaratively list these as part of a button, as shown in Figure 1. Then, when the user creates my-button1 using button as the prototype, Garnet automatically creates instances of the three rectangles and the text. Of course, any of the parts could themselves be aggregates, and the instancing would be applied recursively. Constraints (described below) are used to declare how the properties of the components are connected.

An important innovation in Garnet is that edits made to the prototype are automatically reflected in all instances. For example, if the color of fill-inside



(a)

```
(create-instance 'button aggregate
  (:parts
    ((:top-edge rectangle ...) ; white left & top edges
     (:bottom-edge rectangle ...) ; black right & bottom
     (:fill-inside rectangle ...) ; grey 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"))
```

(b)

Figure 1:

(a) A button (shown on the left) and some instances created from it. (b) The outline of the button's aggregate structure and the code to create the instances.

were changed in button, it would automatically also change in my-button1 and all the other instances (see Figure 2). More significantly, if a part is added or removed from the prototype, then Garnet will add or remove the corresponding object from all instances. 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 stores pointers in each prototype to all instances to support these operations.

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. For example, the programmer could have left the existing button prototype of Figure 1 unmodified, and created a new type of button that looks like Figure 2 by specifying:

```
(create-instance 'new-button button
  (:parts
    ((:top-edge :omit) ; don't want the top-edge rectangle.
     (:fill-inside :modify
       ; just change the filling-style property.
       (:filling-style light-gray))
     (:bottom-edge and :label are unchanged.
    )))
```

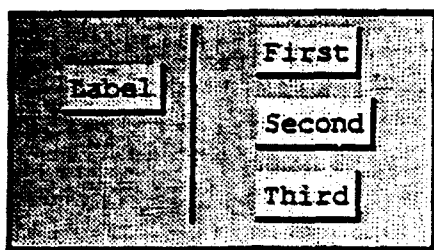


Figure 2:

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 1, these changes propagate automatically to the instances.

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). In Garnet, 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 [15] 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. Because all objects have the same structure, Garnet provides a built-in routine that will save the objects to a file [21]. The contents of the file is simply the declarative code to create the objects, as in Figure 1-b. Therefore, this file can be compiled using the standard Lisp compiler, and the standard Lisp load routine is all that is needed to read in the objects.

Therefore, Lapidary can simply call the standard save routine to write the created objects to a file, instead of having to generate code for the methods to create, draw, and erase the objects, and for handling input events. When the application wants these graphical objects to appear at run-time, it only needs to load the file, and create instances of the prototypes supplying

the appropriate parameters. Note that unlike other interactive interface builders, such as the NeXT Interface Builder, Lapidary allows the designer to define *entirely new* objects, not just choose pre-defined objects from a palette. The various features of Garnet's object system make this much easier to implement [21].

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 Lapidary or a similar tool changes the prototype, all of the instances are updated immediately, even if they appear inside of an application that is currently running. This helps achieve the goal of making Garnet useful for rapid prototyping of interfaces, since the designer can see the results of the edits in context. In a class-instance model or any method-based object system, it would usually be necessary to stop and recompile to see the results of edits.

One claimed disadvantage of the prototype-instance model is speed, since every slot access and setting might require a search up the inheritance hierarchy to find the slot. However, through implementation techniques such as caching, we have significantly improved the performance of Garnet. Thus, even though Garnet offers dynamic inheritance, constraints, and automatic constraint elimination (explained below), it only takes 17.9 microseconds to access a slot (on a SPARCStation 1, using Allegro Common Lisp v4.0.1).

4. Retained Objects

Another important feature of Garnet's object system is that most objects are "long-term." Unlike other object systems, it is rare in Garnet to repeatedly allocate and dispose of objects. Most objects are used to represent application information, graphical displays, or interactive behaviors which persist.

For example, all graphics use a "retained-object model" (sometimes called "structured graphics" or a "display list"). This means that for every graphical object on the screen, there is a corresponding object in memory. Therefore, to make something appear on the screen, the programmer creates instances of graphical objects and adds them to a window. A significant

difference from other systems that supply structured graphics, such as CLIM [11] and InterViews [10] is that there is no way to avoid using the structured graphics in Garnet: *all* graphics must be displayed by attaching instances of objects to windows.

As an example, to display `my-button1` or `my-rect`, the programmer can create an instance of a window and add these objects:

```
(create-instance 'my-window window)
(add-components my-window my-button1 my-rect)
```

This will cause the objects to be displayed. Prototypes can also be displayed, since there is no distinction between prototypes and instances. Therefore, the button that says "Label" in Figure 1 is the actual prototype for the instances.

In a similar way, Interactor objects which control behaviors (described below) are also allocated and attached to graphics. Furthermore, the data that describe the information and state of the application are often stored as Garnet objects. Thus, our techniques are not just limited to the graphical user interface part of the application.

In order to change any property of an object, it is only necessary to set the appropriate slot, and Garnet will propagate the change appropriately. For example, to change the string of `my-button1`, you could use:

```
(set-value my-button1 :string "New Label")
```

This is implemented using a special demon procedure that can be associated with each object. This demon will be called whenever any slots of the object change. For graphical objects in Garnet, a built-in demon is used which automatically insures that the appropriate graphical objects on the screen are redrawn. The graphical update algorithm attempts to minimize the number of objects that are redrawn by first determining all objects that change and all objects that intersect them, and then drawing only those objects (from back to front) using an appropriate clipping region. A different demon is used for Interactor objects, and applications can supply their own demons for application-specific objects, if necessary.

The advantage of the retained-object model is that programmers are freed from many of the maintenance tasks they would have in most other systems. There is never a need to write or call `create`, `initialize`, `draw`, or `erase` methods. When a

complex application-specific graphical object is desired, the programmer uses the declarative syntax to list all the component parts, and then creates instances and adds them to the appropriate window. Of course, the prototypes themselves can also be created dynamically at run time. When objects are to be changed, Garnet automatically determines what must be redrawn. Using the same mechanism, Garnet handles window scrolling and refresh automatically.

Of course, the primitive graphical objects, such as rectangles, lines and text, use `draw` methods internally to display themselves on the screen. Other internal methods are used for handling refresh and for asking objects whether they are under the mouse. However, since Garnet supplies a primitive object for each kind of drawing operation in the X Window System, anything that can be drawn in X can be created by combining instances of Garnet's graphical objects. Therefore, the programmer can simply combine the built-in graphical objects, and never needs to write new methods.

Another important advantage of the retained model is that the toolkit can provide built-in utilities for many of the common functions, since all data uses a standard structure. For example, Garnet provides a widget which displays the popular square "handles" around graphical objects for selection, moving, and growing them. This works because the handles can reference the retained graphical objects to know what is on the screen, and how to modify them. Similarly, there are built-in routines for creating, duplicating, deleting, moving, growing, and printing objects. Thus, application developers do not need to write code for any of this.

The primary problem with the retained object model is the potential for enormous space inefficiencies. If there are 10,000 objects on the screen, there must be 10,000 objects in memory to represent them. We have taken a number of steps to overcome this problem. As with the Glyphs in InterViews [4], we remove unneeded information from objects. For example, we can remove large numbers of unnecessary constraints (see below). However, unlike Glyphs, each object in Garnet still keeps information about where it is located on the screen. Second, if there are a large number of nearly identical objects, such as the

squares in a bitmap editor ("fat bits"), the lines in a map or mesh (Figure 3), or the dots in a graph, then a "virtual aggregate" can be used that just pretends to create an object for each graphic. The programmer provides a prototype object, and the virtual aggregate simulates creating an instance for each data value, but actually does not allocate any objects in memory. It still appears to the rest of the code, however, that there is an object for each value. Using these techniques, people have created quite large applications using Garnet.

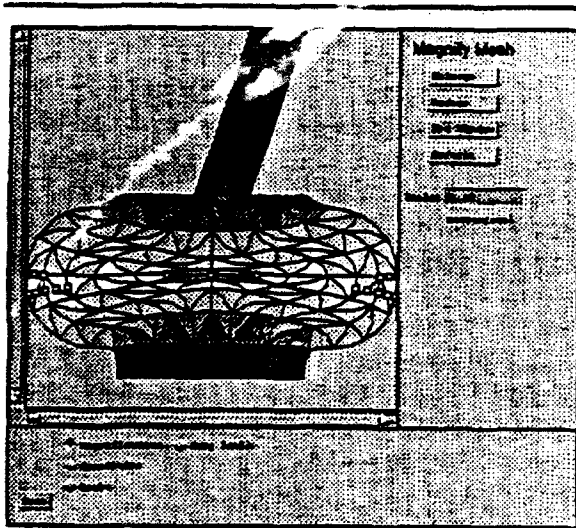


Figure 3:

A mesh created using a virtual aggregate for the polygons and another virtual aggregate for the square knobs. For the polygons, the virtual aggregate is passed a prototype for a polygon, and an array containing the list of points and the color for each polygon. The virtual aggregate then pretends to allocate an object for each element of the array, but actually just draws the prototype object repeatedly. (Picture courtesy of Kenneth Meltner of General Electric [13].)

5. Constraints

An important feature of Garnet is that any slot of any object can contain a *constraint* instead of a normal value. A constraint is a relationship that is declared once and then maintained automatically by the system. For example, instead of making one endpoint of a line be (10,45), a programmer can define it to be the same as the center of the left edge of a rectangle. Then the system will change the value of the endpoint automatically whenever the rectangle moves. The syntax for referencing slots of objects in Garnet is

(gv object slot), where gv stands for "get-value."

Although many other research systems have provided constraints, Garnet is the first to truly integrate them with the object system and to make them general purpose. Constraints in Garnet can be any Lisp expression. An important result of these design decisions is that constraints are used throughout the system in many different ways. For example, 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, contains 16,700 constraints. Of course, many of these are only evaluated once, and may be eliminated, as will be discussed later.

Since they can contain arbitrary code, constraints might be thought to be like methods, and, in fact, they serve a similar purpose: to define the operation of objects. However, 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, and do not have side effects. Finally, by focusing on data values, constraints make programming more data oriented, rather than procedure oriented. Section 8 discusses why constraints provide more information hiding than conventional methods.

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. Garnet provides a declarative syntax so the programmer can simply list the relationships of the parts. An innovation of the Garnet constraint system is that the objects can be referenced through pointer variables [25]. This is used to allow 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 1, the bottom-edge

rectangle can refer to the width of the text object using:

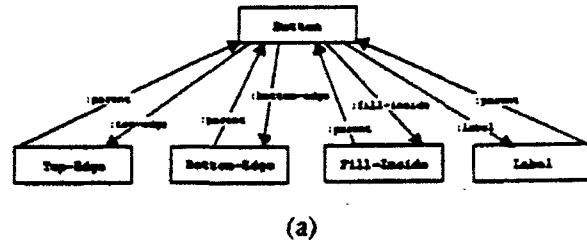
```
(gv :SELF :parent :label :width)
```

As shown in Figure 4-a, this starts from the bottom-edge rectangle, goes up to the parent aggregate, down to the label part, and gets the :width from there. 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 sets pointers to the appropriate objects into the slots :parent, :fill-inside, :bottom-edge, etc. This makes it easy for Garnet to create instances of the entire aggregate (including the constraints), since Garnet does not need to edit the constraint code. Because this style of constraint is quite common, we provide an abbreviation of (gv :SELF) as gvl. Figure 4-b shows the constraints used to tie together the parts of the button of Figure 1.

These constraints are fairly simple, and are representative of the majority of the constraints used in Garnet. However, some objects have quite long and complex constraints. For example, the aggregate object 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.

Another important use of constraints is to copy values and parameters around. For example, the Motif button 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 aggregate. 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. In Garnet, the standard create-instance routine is used for all objects, and it can be used to set an arbitrary number of slots.

Although the slots which serve as parameters are in the top-level button aggregate, for these values to actually take effect they must be copied down to the



(a)

```
(create-instance 'button aggregate
(:left 20)           ; These are the
(:top 20)            ; parameters to
(:string "label")    ; 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))))))
```

(b)

Figure 4:

(a) The structure of the objects in the button of Figure 1 showing the references. (b) The complete code used to produce the button. This shows the constraints which put the graphics in the correct places and copy the parameter values to the parts.

appropriate places in the components. For example, the string value is specified at the top level in Figures 1 and 4-b, 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 Lisp 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.

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 provides some iconic menus for defining the most popular constraints (Figure 5). We have found that these are sufficient for most graphical applications. For more complex constraints, a spreadsheet-like interface, which is called C32, provides a number of features to help programmers who do not know the exact syntax [19]. 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. Of course, it also balances parentheses. In the future, we will explore automatic inferencing of constraints, as was done in Peridot [14]. We envision that when "guessing" mode is turned on, the system will try to find a likely constraint between the newly drawn object and the neighboring objects.

The performance of constraints in Garnet is quite fast. Evaluating constraints is not much slower than the calculations the programmer would have to perform anyway. On a Sun SPARCstation 1, a simple constraint evaluation (in Lisp) takes 110 microseconds. This means that objects tracking the mouse can afford to have dozens of constraints being re-evaluated for each incremental mouse movement. The system caches old values for constraints, so ones that do not change value are not re-evaluated. We have discovered that the primary performance problem with constraints is not speed, but rather space. For each constraint there must be pointers from slots that are

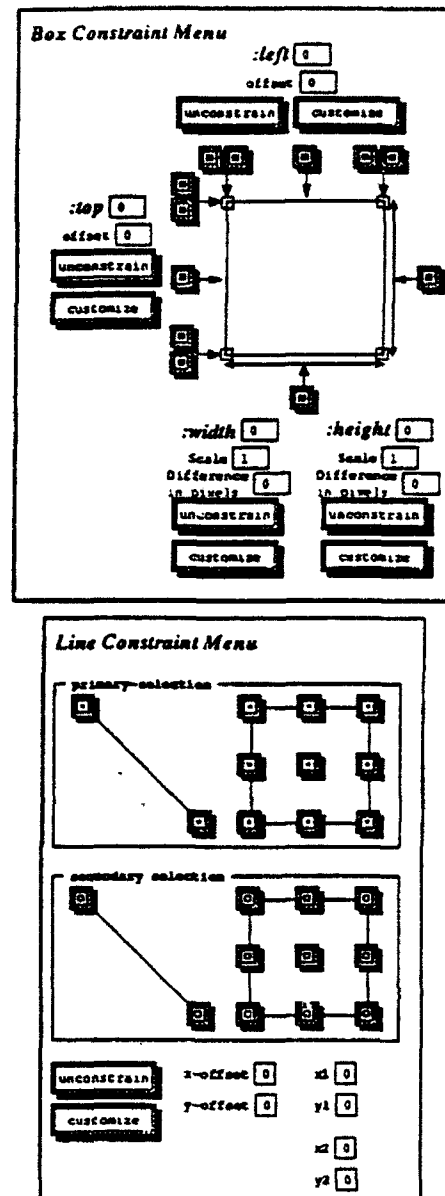


Figure 5:

The dialog boxes from Lapidary [15] that allow the most common constraints to be set. The menu on the top is for rectangular objects (which includes circles and aggregates), and the one on the bottom is for attaching lines to each other or to rectangular objects. For the box constraints, the column of buttons labeled *:top* will cause the dependent object to be: on top of the other object, just inside the other object, centered vertically in the object, just above the other object, or just below the other object. Similarly, the row of buttons labeled *:left* determine the horizontal relationship. The button on the bottom constrains the width, and the one on the right constrains the height. The text fields, like *offset* and *scale* are used to supply parameters to the constraints. For lines, either end of the line can be attached to various positions of a box-like object or of another line.

referenced to the constraints that use them, and from constraints to the slots they reference. We observed that many of the constraints are only used once when the object is initially placed, so we devised a technique where no memory is allocated for these constraints. This has been enormously effective, and decreases the total run-time storage requirements of applications on average by about 50%. For some dialog boxes, like the color selection palette, 1500 constraints are reduced to only 100. As an example of a large scale application, 6690 constraints (which is over 40%) are eliminated from the Lapidary graphical editor.

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. More relevant to this paper, however, is that constraints 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.

6. Input Model

Virtually all toolkits, graphics packages, and window managers use the same input model: a stream of input event records is sent to the appropriate window. The application program is expected to de-queue these events and interpret them. Garnet uses an entirely different model, based on encapsulating input behaviors separately from the graphics [16, 18]. This handles all input so objects never need event-handling methods.

Garnet provides seven basic "Interactor" objects that handle all of the most common direct manipulation behaviors. The Interactor objects in Garnet are completely independent of any graphical representation,

and are purely input filters.³ The seven types of Interactors currently in Garnet are:

Menu-Interactor - Used to select one or more from a set of objects. This can be used for menus, radio buttons, check boxes, simple push buttons, and the arrows on scroll bars. In addition, this can be used to cause application objects to become selected in a graphics editor.

Move-Grow-Interactor - This is used to move an object or one of a set of objects using the mouse. There may be feedback to show where the object will be moved, or the object itself may move with the mouse. This Interactor can be used to implement the indicator for one-dimensional or two-dimensional scroll bars, and also for moving application objects in a graphics editor.

New-Point-Interactor - This is used when one, two or an arbitrary number of new points are desired from the mouse.

Angle-Interactor - This is used to get the angle the mouse moves around some point. It can be used for circular gauges or for rotating objects.

Trace-Interactor - This is used to get all of the points the mouse goes through between start and end events, for use in free-hand drawing.

Text-String-Interactor - This is used to edit text and supports single-line, or multi-line and multi-font strings. A key translation table allows arbitrary mappings of editing operations.

Gesture-Interactor - This supports freehand gesturing, like drawing an "X" on top of an object to delete it.

Unlike other implementations of the Model-View-Controller idea, in Garnet the programmer never needs to create new kinds of "controllers." It is only necessary to create an instance of a pre-defined Interactor and to supply a few parameters. An important reason that this works is that we have carefully chosen the parameters so that they support the full range of direct manipulation interfaces. For example, the designer can specify which mouse button or keyboard key causes the Interactor to start operating, and which causes it to stop. Menu-interactors can be told whether single or multiple selections are desired. The most important parameters, however, are the

³Note that this use of the term "Interactor" is different from some other systems that use the term for an entire widget (graphics plus behaviors). In Garnet, Interactors have no graphics, only behavior.

graphics that the Interactors operate over. We have observed that although direct manipulation interfaces vary widely in their "look," they are mostly identical in their "feel" or behavior. Therefore, by separating the behavior from the graphics, and including parameters for the most popular options, virtually all behaviors can be provided without requiring new code.

For example, to create an interactor which moves around any of the objects which are components of an aggregate called `my-agg`, the following is all that is needed:

```
(create-instance 'my-mover Move-Grow-Interactor
  (:feedback-obj my-feedback-rect)
  (:start-where '(element-of my-agg)))
```

The rest of the properties of `my-mover` will use the default values (start on left button down, move the object rather than grow it, etc.). After it is created, `my-mover` will continuously watch for a mouse leftbutton press over any of the objects in `my-agg`. When this happens, it will make the feedback object (`my-feedback-rect`) visible and begin moving it to follow the mouse until the mouse button is released. At that point, the `my-feedback-rect` will become invisible and the actual object will be moved. (If no feedback object had been supplied, then the element of `my-agg` would be directly dragged by the mouse.)

There is a standard protocol through which the Interactors interface to the graphical objects. The `move-grow-interactor` sets the `:box` slot of objects, and the `:left` and `:top` slots would be tied to the `:box` slot with constraints. This allows there to be arbitrary filtering without the Interactor having to know about it. To find which object is under the mouse, the Interactor sends a message to the aggregate. This will, in turn, send messages to each of the components. However, the programmer never has to write methods for these, since all graphical objects are created by combining the Garnet primitives which supply the appropriate methods.

The `Menu-Interactor` has two protocols: it can take a separate feedback object as a parameter, or it will directly modify the object that becomes selected. If there is a feedback object, then its `:obj-over` slot is set to the object that becomes selected. The feedback object is expected to have constraints that

will cause the position and size to depend on whatever object is set into the `:obj-over` slot. For example, the left formula might be `(gvl :obj-over :left)`, which will make the feedback object have the same left position as whatever object is selected. Notice that the Interactor does not need to know whether the feedback object is a simple XOR rectangle or an aggregate containing squares that serve as selection handles.

If there is no feedback object, then the `menu-interactor` sets the `:selected` slot of the object itself. There might be constraints that change position, color or font based on whether the object is selected or not. For example, to implement a Motif-like pushed-in appearance for the button of Figure 6, the color of the `:top-edge` might be computed by the constraint:

```
(if (gvl :parent :selected)
    black ; then case
    white ; else case)
```

The formula on the `:bottom-edge` would be the opposite, and the color of the fill-inside would choose between gray and dark-gray. Note that this is all performed without methods: the parameters to the Interactors are values in slots, and the interface between the Interactors and the graphical objects is through setting well-defined slots in the graphics.

It is always legal in Garnet to set a slot's value (the slot does not have to be pre-defined). Therefore, if the programmer does not want anything to happen when the object becomes selected, he or she can simply not attach any constraints to the slots. There is never a worry of a "Message-not-understood" error as in a conventional class-instance system, where the programmer would have to define an appropriate method at the root class (e.g., `object`), to make sure that there would never be a run-time error if arbitrary objects could be selected.

Since Interactors can be specified by filling in parameters, it is easy to create them in interactive editors. For example, Lapidary provides a dialog box for each Interactor type that allows graphics to be attached and parameters to be set. This is how Lapidary allows arbitrary behaviors to be connected to application-specific graphics interactively, without requiring the programmer to write code. Interactors can be added to aggregates, so the single

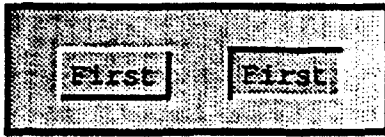


Figure 6:

The button of Figure 1 can be made to look like it moves in 3-D by changing the colors of the parts. The Interactor does not need to know how the button responds to becoming selected.

create-instance call will create the graphics and Interactors necessary for an object to behave correctly.

We have found the Interactor model to be extremely effective. This model makes it much easier to program direct manipulation interfaces. However, we have found a few cases where the built-in parameters are not sufficient. In this case, it is possible for the programmer to write methods to filter the data. Typically, these are used when custom processing is needed when the Interactor starts, stops or aborts. Even when this is required, however, the interface the programmer sees is still higher-level than conventional event-handling. Details are available elsewhere [20].

7. Example and Comparisons

To give an example of the style of programming in Garnet, we will sketch the implementation of the toy graphics editor in Figure 7 and compare it with the implementation in conventional object-oriented languages. Here, every time the user clicks with the right mouse button in the drawing window, a new box and arrow is created using the current line style (which is shown on the left). The arrows always go to the previously-created box. The user can click with the left mouse button to select objects, and the handles appear. Dragging a handle moves or grows the selected object. The Delete button deletes the selected object, and pressing on a new line style while an object is selected causes the object to change. Of course, much of this program could be created using the Lapidary graphical editor without writing any code, but we will assume here that Lapidary is not being used, and the programmer wants to write everything by hand.

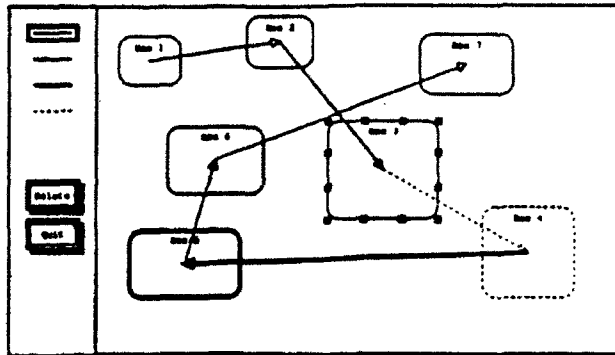


Figure 7:

A simple editor. Box 3 has been selected by the user, and the current line-style (shown on the left) is a thin line.

To implement this in Garnet, the programmer would first create prototypes for the two kinds of objects that can be created: an arrow, and an aggregate containing a rounded-rectangle and a text object. The aggregate will contain constraints that keep the text centered at the top of the rounded rectangle. Then, a main window would be created containing the buttons for delete and quit, and four line objects to serve as a palette. A rectangle would be added to show which line style is selected, and a menu-interactor would be attached to the four lines with the rectangle as the feedback.

To allow new objects to be created, a New-Point-Interactor would be added to the right part of the window which starts on the right button. A parameter to this interactor is the prototype from which instances will be created. Here, this slot will contain an aggregate containing the prototype box and arrows. Formulas in the prototypes will cause the arrows to have the appropriate end points and the string to have the appropriate value.

To make the objects selectable, it is only necessary to include the pre-defined selection-handle-widget, which displays the squares around the objects and allows objects to be resized and moved. Internally, this widget contains many formulas that cause the squares to be attached to the objects at the appropriate places (it works for both boxes and lines). The value of the selection-handle widget is the selected object, which will be accessed by the callback functions for changing the line-style and delete.

To compare the implementations, we asked a number of people to implement the same editor in different object systems and toolkits. Most of these people were the designers of the toolkits.

One implementation was in GINA++, a research toolkit in C++ from the German National Research Center for Computer Science.⁴ The implementation defines classes for the line-style palette items, for the commands for creating and deleting objects, for the graphical objects, and for the editor and its panes. Methods on the graphical objects include setting and accessing the "to" and "from" objects (for the arrows), drawing, and drawing with dashed outline to serve as a feedback object. Methods for the editor include creating the windows, and accessing and setting the current line-style and the selected object. GINA++ provides a retained-object model, so the programmer does not need to write erase or redisplay methods. Support for selection handles around a rectangular object is built in, but the programmer overrode the selection draw-method for lines to only show handles at the end points. To handle creating new objects, when GINA++ sends the `button_press` message to the background window, the `CreateBox` object is created. This special command object defines methods to handle the incremental feedback when dragging out a new box, and then creating a new rounded-rectangle and a new arrow when the mouse button is released.

CLIM [11] is a popular commercial Lisp toolkit that uses CLOS, the standard Common Lisp object system. Like Garnet, CLIM supplies a retained object model with incremental redisplay (which they call "streams") and high-level input handling (called "translators"). Also, CLIM provides a declarative mechanism for defining the window layout, but not for object definitions, so the programmer wrote draw-methods for the objects and selection handles. The programmer also had to write an event handler for the creation of objects, since there is not an appropriate "translator."

In both GINA++ and CLIM, methods are needed for

drawing objects, since they cannot be specified declaratively. How the rectangle the text is displayed is hard-wired into the draw method of the box class, and thus it might be harder to modify than in Garnet, especially by interactive programs. Because they do not have constraints, the code must explicitly redraw the lines and the text label when the box is moved, whereas in Garnet this is handled automatically.

As a small measure of whether the Garnet technique is more effective, Figure 8 shows the coding time and size information for seven implementations of the editor in Figure 7. All but the MacApp one was implemented by one of the designers of the toolkit, so you can expect that they knew the systems well. The MacApp implementor was also an expert with his system. Zdrava is an experimental, unfinished system, so the times for it are simply estimates from the designer. Of course, these numbers do not constitute a scientific study, and the other programmers did not know that they were participating in a time test. Furthermore, the example was chosen by the Garnet designer. Still, the data does suggest that graphical programs can be smaller and written faster in Garnet.

System	Language	Time	Lines of Code
Garnet	Common Lisp	2.5 hrs	183 lines
CLIM+Zdrava	Common Lisp	2.5 hrs (est.)	190 (est.)
CLIM	Common Lisp	4.5 hrs	331 lines
MacApp	Object Pascal	9 hrs	1026 lines
GINA++	C++	16 hours	550 lines
LispView	Common Lisp	2 days	500 lines
CLM, GINA	Common Lisp	2 to 3 days	273 lines

Figure 8:

Times and code size to create the editor of Figure 7 using various systems. CLIM and GINA are discussed in the article. MacApp is a commercial product of Apple and LispView is a commercial product of Sun.

8. 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 [5], do not allow direct access to any object variables, but only provide access

⁴For more information on GINA++ or CLM/GINA for Lisp, contact Mike Spenke, P.O. Box 1316, D-W-5205 St. Augustin 1, Germany, +49 2241 14-2642, spenke@gmd.de.

through methods. Garnet takes an opposite approach, and the main interface is through the data of objects. However, this can be just as modular.

8.1 Data vs. Methods

In Garnet, an object advertises its 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 advertising the exported methods in other object systems. In Garnet, through the use of constraint formulas, objects can transform the parameter values in whatever way is desired. For example, the `Menu-Interactor` sets the `:selected` slot of objects. 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 the `:selected` slot, and the `Interactor` does not care. This interface is just as modular as if the `Interactor` called a generic `Become-Selected` method.

Although Garnet does 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.

8.2 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 9). 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.

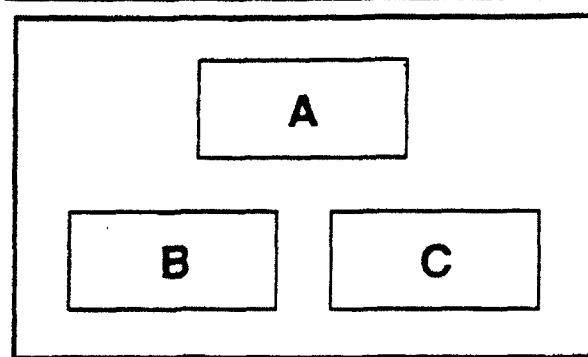


Figure 9:

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.

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 depends 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. (A similar situation arose in the example of Figure 7, where the conventional systems put code in the methods of the boxes to maintain the lines.)

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 constraint-driven programming better preserves the modularity of objects.

8.3 Interactors vs. Methods

The Garnet input model also provides better modularity than found in other systems. The graphics are entirely independent of the behaviors, and they can be developed and modified separately. In other systems, models, views and controllers have always been tightly coupled, so they all had to be carefully modified together.

8.4 Re-use

Another key feature of Garnet is that it provides better software re-use than most other toolkits. The programmer does not have to re-program new event handlers, since the built-in Interactors are sufficient. The programmer does not need to deal with window refresh or maintaining relationships among objects, since the object system and constraint solver handle this. In addition, since we can be sure that there is an object in memory for every object on the screen, it is possible to provide higher-level widgets, such as the selection-handles. The handles contain constraints that reference the selected object. Toolkits without retained objects cannot supply selection handle widgets because they would need to access the application's internal data structure to know where objects are and how to move and grow the objects.

Another feature of Garnet is that, if the programmer wants to make a slight modification of an existing ob-

ject, it is only necessary to specify the specific changes to the graphics, rather than having to write completely new draw methods.

9. Conclusion

The style of programming in the Garnet object system is quite different from other object systems: the programmer collects together graphical objects, writes constraints to define the relationships among them, and then attaches instances of pre-defined Interactor objects to cause the objects to respond to the user. Usually, much of the "programming" 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 code. Our experience suggests that this style of programming is much more effective for graphical user interfaces. It would be interesting to see which other types of programming it works well for. For example, object-oriented data bases seem like a good candidate, since they clearly use a "retained-object model," and a primary use of methods there is to update objects and to maintain consistency among various objects. Many other application areas might also benefit from this style of programming.

Acknowledgements

For help with this paper, we would like to thank Chris Laffra, Francesmary Modugno, Andrew Mickish, Scott McKay, Jade Goldstein, James Landay, and Bernita Myers. Thanks also to Hans Muller, Scott McKay, Christian Beilken, Markus Sohlenkamp, and John Pane for implementing the example application in different systems and for helping me understand their code.

This research was sponsored by the 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.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

1. Paul Barth. "An Object-Oriented Approach to Graphical Interfaces". *ACM Transactions on Graphics* 5, 2 (April 1986), 142-172.
2. Alan Borning. "The Programming Language Aspects of Thinglab; a Constraint-Oriented Simulation Laboratory". *ACM Transactions on Programming Languages and Systems* 3, 4 (Oct. 1981), 353-387.
3. Alan Borning and Robert Duisberg. "Constraint-Based Tools for Building User Interfaces". *ACM Transactions on Graphics* 5, 4 (Oct. 1986), 345-374.
4. 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.
5. Craig Chambers, David Ungar, and Elgin Lee. "An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes". *Sigplan Notices* 24, 10 (Oct. 1989), 49-70. ACM Conference on Object-Oriented Programming: Systems Languages and Applications; OOPSLA'89.
6. James D. Foley and Victor L. Wallace. "The Art of Natural Graphic Man-Machine Conversation". *Proceedings of the IEEE* 62, 4 (April 1974), 462-471.
7. Scott E. Hudson and Shamim P. Mohamed. "Interactive Specification of Flexible User Interface Displays". *ACM Transactions on Information Systems* 8, 3 (July 1990), 269-288.
8. Glenn E. Krasner and Stephen T. Pope. "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system". *Journal of Object Oriented Programming* 1, 3 (Aug. 1988), 26-49.
9. Henry Lieberman. "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems". *Sigplan Notices* 21, 11 (Nov. 1986), 214-223. ACM Conference on Object-Oriented Programming: Systems Languages and Applications; OOPSLA'86.
10. Mark A. Linton, John M. Vlissides and Paul R. Calder. "Composing user interfaces with InterViews". *IEEE Computer* 22, 2 (Feb. 1989), 8-22.
11. Scott McKay. "CLIM: The Common Lisp Interface Manager". *Comm. ACM* 34, 9 (Sept. 1991), 58-59.
12. Jon Meads. "The Standards Factor". *SIGCHI Bulletin* 19, 1 (July 1987), 34-35.
13. Kenneth J. Meltsner. "A Metallurgical Expert System for Interpreting FEA". *Journal of Metals* 43, 10 (Oct. 1991), 15-17.
14. Brad A. Myers. *Creating User Interfaces by Demonstration*. Academic Press, Boston, 1988.
15. 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.
16. Brad A. Myers. Encapsulating Interactive Behaviors. Human Factors in Computing Systems, Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 319-324.
17. Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. "Gamet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces". *IEEE Computer* 23, 11 (Nov. 1990), 71-85.
18. Brad A. Myers. "A New Model for Handling Input". *ACM Transactions on Information Systems* 8, 3 (July 1990), 289-320.
19. Brad A. Myers. Graphical Techniques in a Spreadsheet for Specifying User Interfaces. Human Factors in Computing Systems, Proceedings SIGCHI'91, New Orleans, LA, April, 1991, pp. 243-249.
20. Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, James A. Landay, Richard McDaniel, and Vivek Gupta. *The Gamet Reference Manuals: Revised for Version 2.0*. Tech. Rept. CMU-CS-90-117-R2, Carnegie Mellon University Computer Science Department, May, 1992.
21. Brad A. Myers and Brad Vander Zanden. "Environment for Rapid Creation of Interactive Design Tools". *The Visual Computer; International Journal of Computer Graphics* 8, 2 (Feb. 1992), 94-116.
22. Lynn Andrea Stein, Henry Lieberman, and David Ungar. A Shared View of Sharing: The Treaty of Orlando. In Won Kim and Frederick H. Lochovsky, Ed., *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Addison-Wesley, New York, NY, 1989, pp. 31-48.

23. Pedro A. Szekely and Brad A. Myers. "A User Interface Toolkit Based on Graphical Objects and Constraints". *Sigplan Notices* 23, 11 (Nov. 1988), 36-45. ACM Conference on Object-Oriented Programming: Systems Languages and Applications; OOPSLA '88.

24. 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, April, 1990, pp. 27-34.

25. Brad Vander Zanden, Brad A. Myers, Dario Giuse and Pedro Szekely. The Importance of Pointer Variables in Constraint Models. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'91, Hilton Head, SC, Nov., 1991, pp. 155-164.

Visual Computer

Environment for rapidly creating interactive design tools

Brad A. Myers¹ and
Brad Vander Zanden²

¹ School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213 USA

e-mail: brad.myers@cs.cmu.edu

² Computer Science Department, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301, USA

e-mail: bvz@cs.utk.edu

The Garnet toolkit was specifically designed to make highly interactive graphical programs easier to design and implement. Visual, interactive, user-interface design tools clearly fall into this category. At this point, we have used the Garnet toolkit to create three different interactive design tools: Gilt, a simple interface builder for laying out widgets; Lapidary, a sophisticated design tool for constructing application-specific graphics and custom widgets; and C32, a spreadsheet interface to constraints. The features of the Garnet toolkit that made these easier to create include use of a prototype-instance object system instead of the usual class-instance model, integration of constraints with the object system, graphics model that supports automatic graphical update and saving to disk of on-screen objects, separation of specifying the graphics of objects from their behavior, automatic layout of graphical objects in a variety of styles, and a widget set that supports such commonly used operations as selection, moving and growing objects, and displaying and setting their properties.

Key words: User-interface design environments - Toolkits - Object-oriented systems - Constraints - Input handling - Garnet

Offprint request to: B.A. Myers

1 Introduction

Creating visual interactive design tools can be a difficult task when the appropriate support is not available. The Garnet toolkit (Myers et al. 1990) was specifically designed to make the construction of interactive, graphical, direct manipulation programs, including interactive user-interface builders, significantly easier. A *toolkit* is a collection of interaction techniques (sometimes called widgets or gadgets), such as menus, scroll bars, and buttons, along with a programming mechanism to create them. The Garnet toolkit has allowed us to quickly create a number of interactive user-interface design tools. It also provides an appropriate platform on which to investigate novel types of graphical tools. This article concentrates on those aspects of the Garnet toolkit that make it effective for creating interactive design tools. It uses examples from Garnet's own design tools to illustrate how the various features of the Garnet toolkit simplified the tools' implementation. A number of previous articles have described Garnet (Myers et al. 1990) and the design tools (Myers et al. 1989; Vander Zanden and Myers 1990; Myers 1991a), but none have described in-depth the specific features of the toolkit that were designed to support interactive design tools. Neither have any of the previous Garnet papers discussed how the components of the Garnet toolkit simplify implementing the unique functionality required by interactive design tools.

2 The Garnet project

Garnet, which stands for Generating an Amalgam of Real-time, Novel Editors and Toolkits, is a research project at Carnegie Mellon University. An important goal is to investigate appropriate foundations for the toolkits of the future so they will be able to more effectively support highly interactive graphical user-interface software. In addition, Garnet aims to create high-level interactive design tools to make user-interface software significantly easier to design and implement by both programmers and non-programmers.

Garnet contains both programming and interactive design tools. The programming tools are called the Garnet Toolkit, which is divided into two layers. The lower layer, also called the toolkit intrinsics, provides functionality that allows widgets and application-specific graphics to be created. This includes the object system, constraints, input handling, and graphical object support. The Garnet toolkit intrinsics are "look-and-feel indepen-

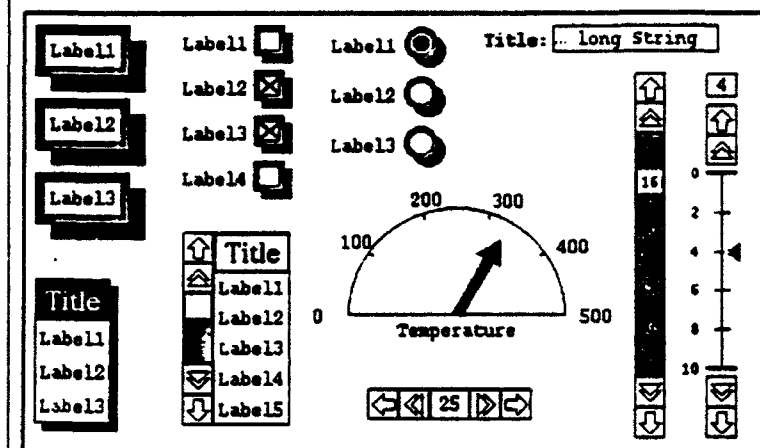


Fig. 1. Some of the gadgets in the Garnet toolkit with the Garnet look-and-feel, which has the buttons floating above the shadow, and moving in "simulated 3-D" towards the shadow when pressed

dent," which means that arbitrary graphics and behaviors can be implemented. The intrinsics also provide a machine- and window-manager-independent interface, so that software written using the Garnet toolkit intrinsics will run without change on any platform for which Garnet has been implemented. Currently, Garnet runs on the X Window System, but Macintosh and Display Postscript versions are planned.

The next layer of the toolkit contains a "widget set," which is a large and growing collection of menus, buttons, gauges, scroll bars and sliders (Fig. 1).

The interactive Garnet design tools include Gilt, Lapidary, C32, and a hybrid interactive-programming tool, Jade. Gilt (Myers 1991b) is an *interface builder*, which is a program that lets the designer graphically place user-interface components in a window, and thereby create menus, palettes, and dialog boxes. Gilt provides most of the functionality found in other interface builders, but was created in only about two man-months and contains about 2700 lines of Lisp code. Lapidary (Myers et al. 1989; Vander Zanden and Myers 1991) is a research vehicle for investigating how to provide new functionality in visual interactive design tools. For example, it lets the user-interface designer draw pictures of what the user interface should look like and then demonstrate how it should change in response to user input. Lapidary is the only interactive design tool that allows the *behavior* of application-specific graphical objects to be defined without writing code. C32 is a spreadsheet interface for defining constraints (Myers 1991a).

Jade automatically creates menus and dialog boxes from a textual list of their contents (Vander Zanden and Myers 1990). A graphics artist can then modify his layout using a drawing editor, and Jade will remember the changes, even if the original textual specification is changed.

Garnet is implemented in Common Lisp and currently uses the X window manager. Garnet is therefore portable and runs on various machines and operating systems. For example, Garnet runs on CMU, Lucid, Allegro, TI, and Harlequin Common Lisps on hardware such as Sun, DECStation, TI and HP. Garnet does *not* use the Common Lisp Object System (CLOS) or any Lisp or X toolkit (such as InterViews (Linton et al. 1989), CLUE, CLIM, or Xtk (McCormack and Asente 1988)).

The toolkit layer of Garnet (Myers et al. 1991) and Gilt have been released for general use, and there are over 150 licensed universities and companies. We expect to release other parts shortly.¹

3 Related work

The Garnet system as a whole is a user interface development environment, which is sometimes called a user interface management system (UIMS). These tools have been surveyed in various places (Hartson and Hix 1989; Myers 1989a; Brown and Cunningham 1989).

There are many different toolkits available today.

¹ Garnet is available free of charge. Contact the first author for more information about obtaining Garnet, or send electronic mail to garnet@cs.cmu.edu

Visual Computer

The most famous are probably Macintosh Toolbox, Windows and Presentation Manager toolkits for the PC, and various X toolkits based on Xtk (McCormack and Asente 1988) including Motif and OpenLook. All of these toolkits provide libraries of widgets, but no support for handling input and output in the main application window. The NeXTStep toolkit for the NeXT computer provides an object-oriented set of widgets, as well as a well-defined method for subclassing the existing objects to create application-specific objects. However, these objects still have to deal with all input and output directly. None of these toolkits provide special features to make it easy to create interactive design tools.

Some of the features found in the Garnet toolkit have been used in previous systems, but Garnet is the first to integrate them all. The prototype-instance model for objects (Lieberman 1986) has been used in SELF (Chambers et al. 1989). Constraints, which are relationships that are specified once but maintained automatically by the system, have been widely used by research systems (Borning and Duisberg 1986). Constraints can be either *one-way* or *multi-way*. One-way constraints allow the constraint solver to change only one of the objects in the constraint in order to satisfy it; multi-way constraints allow any of the objects in the constraint to be changed. Early multi-way constraint systems include Sketchpad (Sutherland 1963), which pioneered the use of graphical constraints in a drawing editor in the early 1960s, and Thinglab (Borning 1981), which used constraints for graphical simulation. More recently, Thinglab has been refined to aid in generating user interfaces (Freeman-Benson et al. 1990). CONSTRAINT (Vander Zanden 1989) provided a user-interface development environment that employed multi-way constraints, but introduced a new constraint-solving algorithm that made multi-way constraints efficient enough to be solved in real time. Most systems that have been designed for developing user interfaces use one-way constraints because of their simplicity and efficiency. Grow (Barth 1986), Peridot (Myers 1988), and Apogee (Henry and Hudson 1988) are three examples. Grow was perhaps the first user-interface development system that employed constraints, Peridot was the first to try to infer constraints, and Apogee was the first to employ lazy evaluation. Garnet currently uses one-way constraints, but a more powerful model is being investigated.

The event-handling part of Garnet categorizes behaviors into a few "interactor" object types and is based on the Model-View-Controller idea from Smalltalk (Krasner and Pope 1988). However, the Views and Controllers tend to be so tightly linked in Smalltalk that the programmer must program new Controllers for most new objects, whereas in Garnet programmers rarely need to program new interactors. In addition, Garnet's interactors are built into the toolkit, whereas in Smalltalk the programmer must code the event handlers. Finally, the interactors provide a rich set of parameters for customizing their behavior so that it is unusual for a programmer to create subclasses of the built-in interactor types (although it is possible).

Gilt, the Garnet Interface Builder, is very similar to other interface builders, such as Menulay (Buxton et al. 1983), Trillium (Henderson 1986), Dialog-Editor (Cartelli 1988), vu (Singh and Green 1988), NeXT's Interface Builder, the Prototyper from Smethers Barnes for the Macintosh, and UIMX from Visual Edge for X. Lapidary extends the ideas in these to support the creation of widgets, based on ideas in Peridot (Myers 1990a; Myers 1988). Lapidary can also create application-specific graphics.

C32 is based on financial spreadsheets like VisiCalc, Lotus 1-2-3, and Microsoft Excel. Other systems that have used spreadsheets for defining user interfaces include NoPumpG and NoPumpII (Wilde 1990). The NoPump systems were standalone spreadsheets that were not integrated with an toolkit, so they had to invent a constraint solver and a way to handle input, which the Garnet toolkit provides to C32.

4 Garnet interactive design tools

This section describes several of the interactive design tools built by the Garnet group. Some external users have also built interactive design tools on top of Garnet [for example, Humanoid (Szekely 1990)]. Later sections will illustrate how various features of the Garnet toolkit simplified the construction of these tools.

4.1 Gilt

Gilt, the Garnet Interface Layout Tool, is a simple widget layout tool patterned after the NeXT inter-

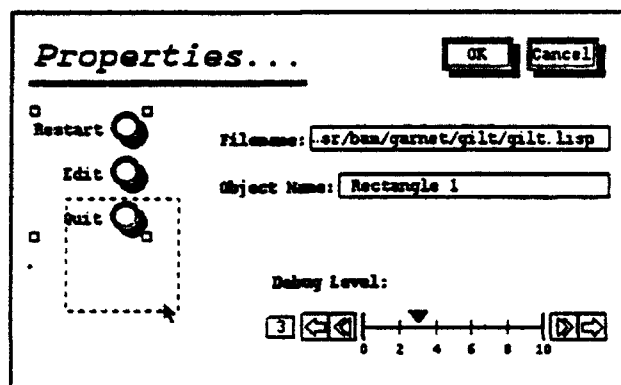


Fig. 2. Gilt work window showing a sample dialog box being created

face builder. It is intended to help create dialog boxes that do not change size and whose contents do not change dynamically (although the widgets can "grey out" when not valid). Gilt supplies a palette of the built-in Garnet widgets (either with the Garnet or Motif look-and-feel) plus rectangles, lines, text labels, and bitmaps to be used as decorations. The user can select objects in the palette and place them in a workspace window (Fig. 2). Objects in the workspace window can be selected, moved or changed in size, their text labels changed, and their other properties set. Commands for duplicating and deleting objects are provided. There is also an Align command, which adjusts objects' position and spacing.

If the user needs a more dynamic interface, for example to have one object's property tied to another's, the code generated by Gilt can be edited using a text editor, or read into Lapidary (because all the tools use the same file format).

4.2 Lapidary

Lapidary is a graphical interactive design tool that allows users to pictorially specify all graphical aspects of an application, including the widgets that surround application windows and the objects and behaviors that go inside application windows (Myers et al. 1989; Vander Zanden and Myers 1991). Lapidary is an acronym that stands for *L*isp-based *A*ssistant for *P*rototyping *I*nteractive *D*esigns *A*llowing *R*emarkable *Y*ield.

Many of the windows that comprise the Lapidary interface are shown in Fig. 3. Lapidary provides a direct-manipulation drawing editor that allows

users to create objects from scratch, such as the floating button menu in Fig. 3, or edit instances of objects created from prototypes in libraries. A set of iconic constraint menus allow users to position objects in a scene. If users cannot find the constraint they need, the customize button can be hit to gain access to the C32 spreadsheet described in Sect. 4.3. Users can specify behaviors either through dialog boxes or via demonstration.

4.3 C32

C32 is a spreadsheet interface for defining complex constraints (Myers 1991a). C32 stands for *C*leaver and *C*ompelling *C*ontribution to *C*omputer Science in *C*ommonLisp, which is *C*ustomizable and *C*haracterized by a *C*omplete *C*overage of *C*ode and *C*ontains a *C*ornucopia of *C*reative *C*onstructs, because it *C*an *C*reate *C*omplex, *C*orrect *C*onstraints that are *C*onstructed *C*learly and *C*oncretely, and are *C*ommunicated using *C*olumns of *C*ells that are *C*onstantly *C*alculated so they *C*hange *C*ontinuously and *C*ancel *C*onfusion. The intention is that the Lapidary icons or a similar mechanism can be used for simple constraints, but sometimes the user will need more complex relationships. C32 provides the same benefits for constraint definition as conventional spreadsheets provide for financial operations. Some features of C32 are:

- The current values of the constraints are visible and are updated immediately if the associated graphical object changes, either due to the user interacting with the graphics or some program changing a value. This makes the results of the constraints visible and therefore more understandable.
- Menus provide the most common functions that might be inserted into constraints, so users do not have to know the function syntax.
- Object and slot references are automatically generated, so users do not have to know the syntax for references or the particular form the reference should take (objects can be referenced directly by name or indirectly through their position in the aggregate hierarchy and C32 figures out which is more appropriate). The user can select the referent either by pointing to a C32 spreadsheet cell or just by pointing to the actual graphical object in a Garnet window.

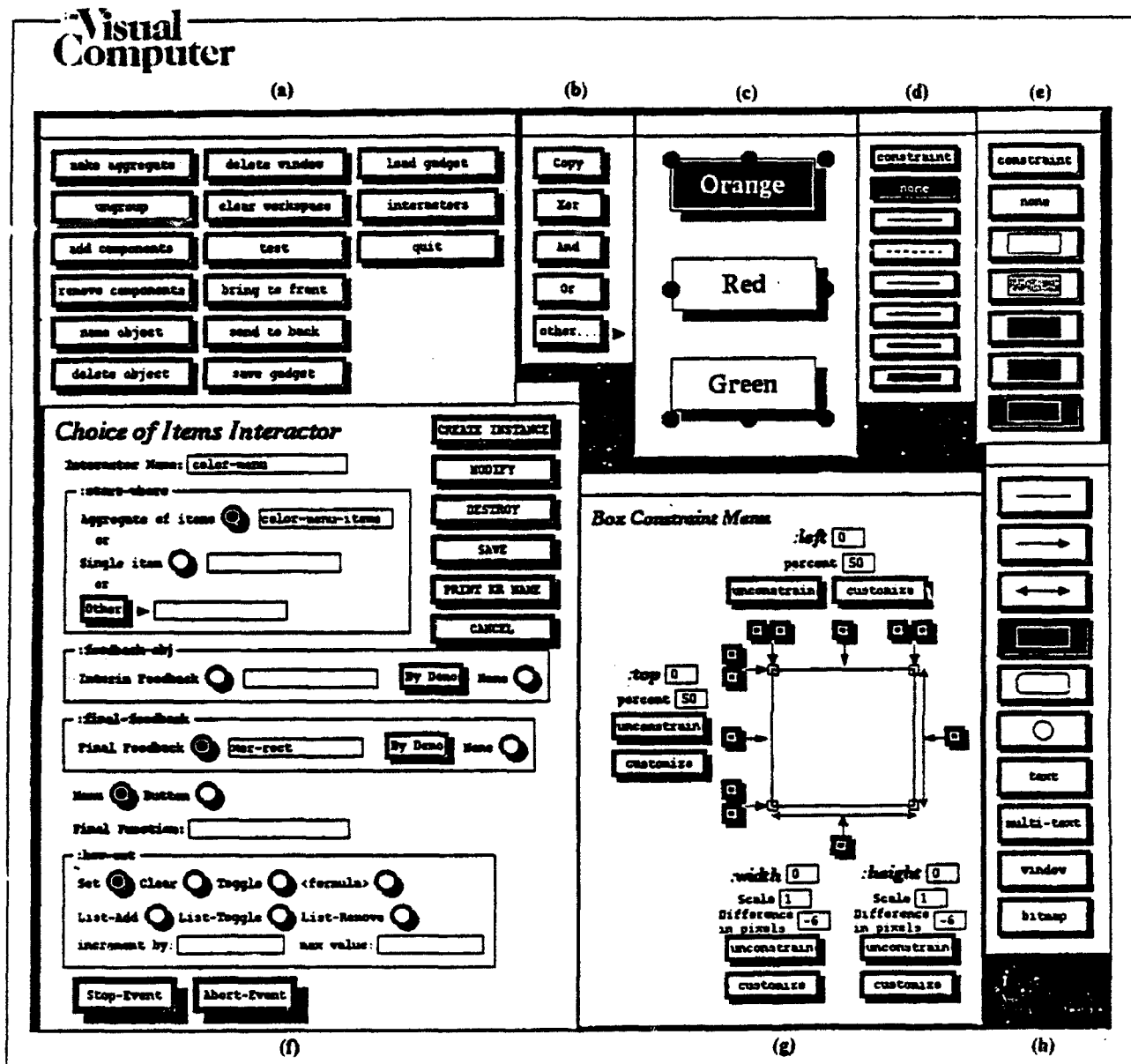


Fig. 3a-h. Various Lapidary windows. The drawing window (c) contains floating buttons for a color menu. The constraint menu below it (g) has been used to align the objects within each button, the items within the menu, and the black xor rectangle, which is the final feedback (currently over "Orange"). The dialog box in the lower-left corner (f) defines the menu behavior by

indicating that the behavior will apply to the objects in *color-menu-items* and that the final feedback will be the black xor rectangle. The window in the upper-left corner (a) contains the main Lapidary commands, and the remaining windows contain geometric properties that may be set in Lapidary

- C32 guesses how to parameterize constraints when they are copied from one place to another or generalized into procedures, so abstract and reusable constraints can be constructed *by example*.
- Graphical techniques are used to help trace and debug constraints.

Each object has its own column in the spreadsheet, with the rows showing the values of each of the slots. Icons are used to show whether a slot is in-

herited and whether it is calculated using a constraint. Figure 4 shows an example.

4.4 Jade

While Jade is only partially interactive, it is interesting how it uses the features of the Garnet toolkit. Jade (a Judgement-based Automatic Dialog Editor) takes a textual specification of a dialog and

MY-RECTANGLE		STRING1		ARROW	
Left	10	String	"C32"	X1	
Top	10	Font	OPAL-DEFAULT-F...	Y1	
Width	50	Left	25	X2	
Height	50	Top	28	Y2	
Visible	⊕⊕T	Width	⊕⊕21	Left	
Line-Style	OPAL-DEFAULT-L...	Height	⊕⊕14	Top	
Filling-Style	NTL	Visible	⊕⊕T	Width	
Draw-Function	COPY	Line-Style	OPAL-DEFAULT-L...	Height	
Window	W	Fill-Backcolor	NTL	Visible	
Parent	A	Actual-Height	NTL	Line-Style	
Is-A	OPAL-RECTANGLE	Draw-Function	COPY	Filling-Style	
		Window	W	Draw-F...	

Fig. 4. C32 viewing three objects. The scroll bars can be used to see more slots or columns. Changing the window's size will change the number of slots and objects displayed (the number of rows and columns). Field values are clipped if they are too long, but can be scrolled using editing commands. The ⊕ icon means that the slot value is computed with a formula (constraint). All inherited slots are shown in italics and marked with the ⊕ icon. When a formula is inherited, the icon is next

to the formula icon and the value is shown in a regular font. This is because the value is usually different from the prototype's. For example, the :width slot of a text object usually contains a formula that computes the width based on the object's font and string value. Most text objects inherit this formula, but still have a different current value, because their string and font values are different

automatically generates a dialog box (Vander Zanden and Myers 1990). It is typically used when it is too inconvenient to use Gilt or Lapidary, for example when an interface has dozens of dialog boxes. A graphics artist can use a direct manipulation editor to add decorations or modify the layout created by Jade, and Jade will remember the changes, even if the underlying textual specification is modified. A sample dialog and its textual specification is shown in Fig. 5. In laying out the dialog, Jade consults designer-provided look-and-feel and rule libraries. The look-and-feel libraries help Jade decide which gadgets should be associated with various elements of the textual specification. For example, Jade has selected a Garnet-style radio button in Fig. 5a and an OpenLook-style widget in Fig. 5b to represent a single-choice behavior. The gadgets for the look-and-feel libraries can be created using Lapidary or C32. Jade uses the rule libraries to help it lay out the dialog box. In the future, we plan to develop an interactive design tool that allows graphics artists to define rules by example, perhaps by applying constraints to an example object.

5 Toolkit properties

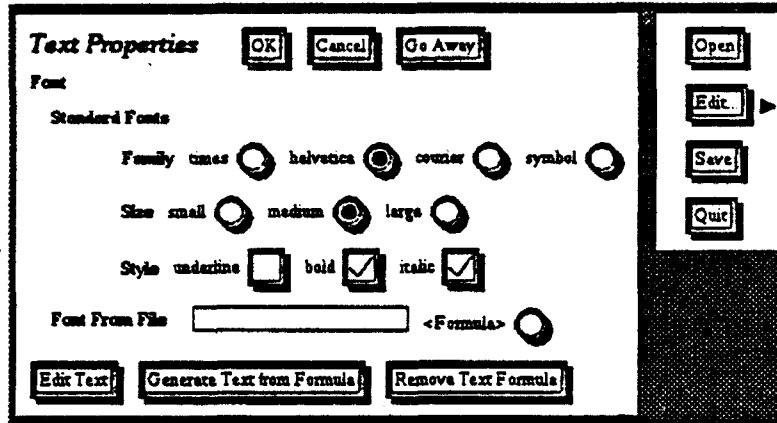
Many features of the Garnet toolkit were added specifically to make it easier to create highly inter-

active user interfaces, such as the interfaces of visual interactive design tools. These include:

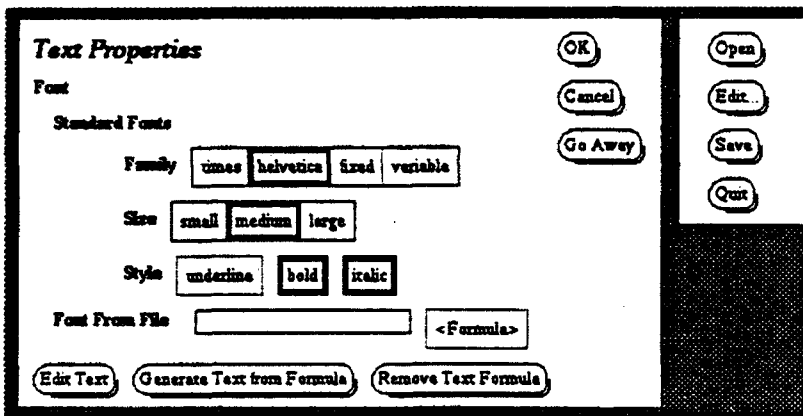
- Use of a prototype-instance object system instead of the usual class-instance model.
- Integration of constraints with the object system.
- Graphics model that supports automatic graphical update.
- Separation of specifying the graphics of objects from specifying the behavior.
- Support for saving objects on the screen or in memory to disk as text files in a way that they can be read back into the system.
- Automatic layout of graphical objects in a variety of styles, such as rows, columns, tables, trees, and graphs.
- Widget set that supports such commonly used operations as selection, moving, and growing objects and displaying and setting their properties.

Several of the features contain innovations, such as structural inheritance in the prototype-instance system, pointers in the constraint system, an integration of constraints with an automatic graphical update system, interactors model for separating graphics from behavior, and a method for minimizing the amount of information written to disk when saving objects. Other features, such as automatic layout and widget set are not novel but they are useful in creating interactive design tools. The following sections discuss these features in detail.

Visual Computer



a



b

```
(create-dialog
  (("Font"
    (("Standard-Fonts"
      (("Family" ("times" "helvetica" "courier" "symbol"))
      ("Size" ("small" "medium" "large"))
      ("Style"
        ("underline" "bold" "italic")
        (:behavior :multiple-choice)))
      ("Font From File"
        (:behavior :text))
      "<Formula>"))
    ("Edit Text" "Generate Text from Formula"
      "Remove Text Formula")
    (:behavior :command)
    (:stop-action text-handler)))
  (:stop-group (("OK" "Cancel" "Go Away")
    (:stop-action 'font-stop-action))))))

(create-dialog
  (("open" ("edit" ("copy" "cut" "paste" "delete"))
    "save" "quit")
    (:behavior :command)
    (:stop-action control-menu-handler)))
  c
```

Fig. 5a-c. Alternative Garnet a and OpenLook b style dialogs for specifying the properties of a text object and controlling an application. Both dialogs' layout and look-and-feel were generated automatically by Jade from the same textual specification c

5.1 Prototype-instance model

5.1.1 General description

The Garnet object-oriented programming system supports a prototype-instance model, rather than the conventional class-instance model used by most other object systems, such as Smalltalk, C++, and CLOS. In a prototype-instance model, there is no distinction between instances and classes; any instance can serve as a "prototype" for other instances (even an object that is being displayed on the screen can be a prototype).

All data and methods are stored in "slots" (sometimes called fields or instance variables). Slots that are not overridden by a particular instance inherit their values from their prototypes. There is no distinction between data and method slots in Garnet. Any slot can hold any type of value, and in Common Lisp a function is just a type of value. Slot names can contain any number of printing characters (e.g., left, interim-selected, obj-over).

An instance can also add any number of new slots. Unlike conventional class-instance models, this means that the number of slots in each object is variable - each object can have a different number of local slots, depending on which properties it wants to inherit defaults for and which it wants to override. Moreover, the number of slots of an object can even change dynamically. Slots can be explicitly removed from objects at any time. Also, if a program sets the value of a slot that does not exist, then the slot is created automatically. If a program asks for the value of a slot that does not exist, NIL is returned (there is a function that tests whether a slot exists). All slots are untyped.

The efficiency of Garnet's object system is actually better than other class-based Lisp object systems, such as CLOS. For example, on a Sun SPARCstation 1 running Allegro Common Lisp, the simplest slot accessor function takes two- and one-half times as long in CLOS (23.7 microseconds), than in Garnet (9.5 microseconds). To create an instance in CLOS (3486 microseconds) takes about eight times longer than in Garnet (433 microseconds).

A unique feature of the Garnet object system is the support for *structural inheritance*. Objects can be grouped into an "aggregate" object. Then this aggregate can be used as a prototype for new objects in the same way as a primitive object can be. When an instance is made of an aggregate, Gar-

net automatically creates instances of all the parts and links them together in the appropriate manner. Edits made to the prototype are automatically reflected in all instances. For example, if objects are added or removed from the prototype, the same edit is made to all the instances of that prototype (see Fig. 6).

The function that adds a new object to an aggregate takes an optional locator argument (e.g., front, back, before *obj*, after *obj*) to determine where to insert the object with respect to the other children of that aggregate (which is important, because the order determines the back-to-front drawing order and therefore which objects are covered by other objects). After the object is inserted into the aggregate, Garnet checks a special slot, which contains a list of all the instances of the aggregate. For each instance of the aggregate, Garnet creates an instance of the inserted object and inserts the new instance into the instance aggregate. Instances of the inserted object that are propagated to an aggregate's instances are also positioned using the locator argument. If the locator specified that the added object should be inserted before or after a child, Garnet finds the instance of this child in each of the aggregate's instances and then inserts an instance of the added object in the appropriate spot. When an object is removed from an aggregate, Garnet first removes the object from the aggregate's components list, then removes instances of the object from each of the aggregate's instance-components list.

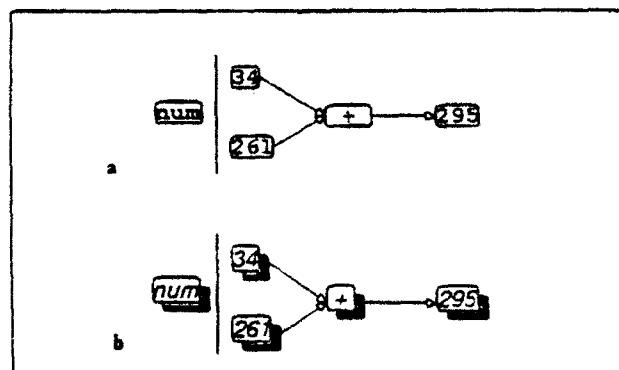


Fig. 6a, b. Before a and after b editing a prototype. When the prototype (shown on the left) is edited to change the font to italic and add a shadow, the modifications are immediately reflected in all of the instances (shown on the right). Note that the sizes of the boxes change in b because the font width is smaller

Visual Computer

If the programmer wants the new object to be independent of the prototype (so that edits to the prototype do *not* affect the instance), then the built-in copy function can be used instead of the instancing function. The resulting object will look the same as an instance, but there will not be any back pointers.

5.1.2 Saving and restoring objects

Another very important feature of the Garnet object system is that there is built-in support for saving and restoring objects to the disk. A single Garnet function takes an in-memory or on-screen object and writes it to a disk file in a format that can later be read in again. If the object is an aggregate, then all the parts are written to the file also. Direct pointer links among objects are automatically converted to indirect references so that no actual pointers will be written to the disk. Constraints (see Sect. 5.2) are also output in an appropriate form.

In addition to the convenience of having saving and restoring code provided to applications, providing a central mechanism simplifies the implementation of interactive design tools. The tools can create whatever structures they want without having to worry about converting these structures to a suitable external structure, because the built-in save-restore mechanism can write out *all* Garnet objects.

Due to the tree structure of aggregates (aggregates contain primitive objects and other aggregates, which themselves may contain objects and aggregates), it is economical to save them using a tree-structured format, listing only parts and slots that override defaults defined by the prototype. First, the differences between each object and its prototype are computed. Inasmuch as an object can add, remove, or even override the children it inherits from its prototype, these differences may include structural differences as well as simple value differences. Only these differences and the name of the prototype are needed to reconstruct a copy of the object. If there are structural differences, Garnet writes out commands that tell the object-creation mechanism how to create additional children, override existing children, or omit certain children. By writing out only the differences rather than the complete structure of each instance, the programmer is able to understand and change

the generated code more easily, and the code is shorter and more efficient.

The output file is simply the Lisp code to create the objects. Therefore, when the code is loaded by an application or by an interactive design tool, the objects will be created. The prototype-instance object system allows a tool to modify the objects in memory, which means that it is never necessary for the tool to look at or parse the generated file.

Generated files are in exactly the same form as the code that a programmer would write by hand to create the same objects. Therefore, no special mechanism is needed to read objects in again: the standard Lisp loading function is sufficient. Neither special formats for output, such as UIL for the Motif toolkit, nor associated parsers, are needed. Programmers of design tools do not need to write code generators, as in most other systems, because the built-in Garnet function can be used. The standard Lisp compiler can also be used to make the generated files more efficient.

In addition, because the generated object descriptions are standard Lisp using the conventional Garnet style and functions, it is very easy for programmers to read the code to make sure that it is correct. It is also possible to directly modify the code using a text editor if necessary. Inasmuch as Garnet requires no special mechanism to read the code, the programmer is free to make arbitrary edits and the file will still be loadable (assuming the edits do not introduce errors, of course). For example, the programmer could add or subtract children from an aggregate, add new slots for special processing, write complex constraints that cannot be expressed using the design tools, and even add or delete new objects. In addition, because the programmer's changes will be reflected in the in-memory versions of the objects after the code is loaded into a design tool, these edits will be preserved when the files are written out again.

Because interactive design tools are only capable of specifying certain types of objects and behaviors, there are cases when a programmer will need to edit the code created by the design tool in order to achieve the desired functionality for an interface. Thus, it is important that the programmer be able to modify the generated code and still be able to operate on the objects and behavior using the design tools. Garnet's ability to create code in the language used by the programmer without having to resort to special or binary representations is therefore an important advantage when creating

new interactive design tools. Because the code can be compiled, there is also no efficiency penalty for this.

Both Lapidary's and Gilt's implementation are substantially simplified by this save-restore facility. When either tool wants to save an object, behavior, or window to disk, they first write some standard header information to the file, and then call the Garnet object-writing function on the entire window. As long as programmers do not change the header (which they have no reason to do), both tools can read a file simply by using the Lisp "load" function.

Another advantage of the simple format for the generated code is that it is possible to write compilers that optimize the generated objects, for example by removing extraneous slots or constraints that a tool may have inserted for its own use. This can improve the efficiency of the applications that use the objects.

5.1.3 Uses of the prototype-instance model in design tools

The use of a prototype-instance model has a large number of significant advantages for interactive design tools.

5.1.3.1 Dynamic editing. In conventional class-instance object systems, it is very expensive to modify classes. If there are existing instances of the classes in memory, then modification is often not even allowed (for example in C++). Instead, the instances must be destroyed, classes recompiled, and then new instances must be created. Even in systems that permit class evolution, the data structures representing the instances are semi-compiled and must be redone if classes are allowed to change. Thus, something as simple as adding a new instance variable to all objects in a class while the objects are being viewed is either very expensive or impossible in most systems. This makes it very difficult to create an editor that will allow dynamic prototyping and editing of objects and their structure.

The dynamic editing capabilities of the Garnet prototype-instance model clearly makes it ideal for prototyping systems. An interactive design tool can simply provide mechanisms for the user to add and remove slots, objects, and properties from objects. If the user happens to modify a prototype object, then Garnet insures that all instances are updated

appropriately. The very same editing operations can be used on prototypes and on instances, both from the user's and the tool implementer's point of view, and Garnet takes care of the bookkeeping.

As an example, Lapidary uses the prototype-instance model to allow users to dynamically create and modify objects. Initially, objects are created with only a basic set of graphical slots, methods, and components. Users can then attach constraints and behaviors and add or subtract objects. If the components of a prototype are modified, the modifications will be immediately propagated to all instances, thus allowing the user to immediately see the new look in context. This form of structural inheritance is unique among interface builders. For example, users can make the edits shown in Fig. 6 using Lapidary.

5.1.3.2 Creation of prototypes. The lack of distinction between prototypes and instances also means that interactive design tools can easily provide a mechanism for users to make libraries of objects. The tool can allow the user to select an object (which, of course, may be an aggregate of other objects) and use that object as a prototype. The prototypes might be collected together in a window to form a "style sheet." The user can then use instances of the prototype in application windows. If the user decides that the prototype does not look or operate correctly, the prototype can be edited (which will also change all the instances) so the user can see immediately how all the instances will look in context. In a similar way, a palette for the final program can be created interactively in the design tool. The palette will contain prototype objects created using the design tool and the program will only need to create an instance of the appropriate one when the end user wants a new object.

It is this ability to create prototypes on the fly that allows the Lapidary tool to create application-specific graphical objects. If the user is designing a graph editor with nodes and arcs, some graphical objects can be drawn to represent a node, these can be collected into an aggregate, and then that aggregate used as a prototype for all the nodes. The application program then only needs to create instances of the node prototype, and does not need to know anything about its internal graphical appearance or structure (it might be a single rectangle or a whole collection of objects).

Visual Computer

5.1.3.3 Adding new slots. The ability to add new slots to existing objects is quite useful for interactive design tools. For example, the tool might want to store information about how the objects were created, determine how the object's properties were set, or save the previous values of properties to support undo. Even if the objects are primitive objects, such as rectangles and text, or instances of predefined widgets from a library, the tool can simply create new slots in the objects using slot names that are meaningful to the tool. Because messages are simply function values in slots, new messages can just as easily be added to pre-existing objects. No new "classes" are needed. Thus, a design tool can add tool-specific methods, such as a custom destroy method.

These new slots that the tool creates can be written out to the disk with the rest of the objects. This is very useful if the slots contain status or identification information that the tool needs when reading the objects in again². Alternatively, the tool implementer can declare that these extra slots should not be written out with the objects.

Lapidary makes use of the ability to add and delete slots both to support itself and to support behaviors added to the objects. For example, it adds the `objover` slot to feedback objects so that they can indirectly reference the object they should highlight. Lapidary also adds slots to support indirect references to objects and offsets in its positioning and sizing constraints. For example, to support a constraint that aligns the left side of an object with the side of another object, Lapidary adds an `objover` slot to reference the other object and a `left-offset` slot to reference the size of the offset. The use of indirection through these slots allows Lapidary to support the fine-tuning of a constraint without having to destroy and recreate it each time a user changes an offset or a target of the constraint. Before saving a set of objects, Lapidary removes its support slots so they will not clutter up the generated code. Similarly, Lapidary reinstalls these slots when the objects are loaded later.

5.1.3.4 Dynamic loading. Another important feature of the object system is that single objects or

² In this case, the programmer would, of course, have to be careful to preserve this information if hand-editing of the generated code was necessary

groups of objects can be dynamically loaded at any time. Garnet requires that the prototype for objects be loaded before any instances of that prototype are used, but it is easy to insure that the files that create instances automatically load the prototypes first, if they are not already loaded.

Dynamic loading can also be used to reduce the memory size of applications. For example, Gilt uses bitmap pictures to represent the widgets that can be loaded. The first time that a particular type of widget is needed, Gilt simply loads the prototype and creates an instance of it. Many of the large and complicated widgets are not needed by most users, and so they do not need to ever be loaded.

5.2 Constraints

Garnet provides constraints that allow programmers to specify relationships between objects that are automatically maintained by a constraint solver. The relationships expressed by constraints may be graphical, for example, to position a checkmark next to a set of menu items, or non-graphical, for example, to make the selected item in a property menu (e.g., a line style) match the corresponding property in the selected object. The contents of an object's slot can be an ordinary value, such as a number or string, or they can be a constraint that calculates the value. When the value of the slot is requested, Garnet will automatically evaluate the constraint and return the calculated value.

Garnet constraints are arbitrary pieces of Lisp code. They may be thought of as functions that take a set of slots as parameters and return a result. Hence, they are one-way constraints. The constraint-solver is responsible for detecting changes to the slots referenced by a constraint and re-evaluating that constraint automatically. Both eager and lazy algorithms for implementing the constraint solver are presented in Vander Zanden et al. (1991).

A novel feature of Garnet is that programmers can write constraints that indirectly reference other objects through pointer variables and that these variables can be changed under program control at runtime. For example, suppose a checkmark should be able to appear next to *any* item in a menu. A programmer could create this behavior

by inserting the following constraint in the left slot of checkmark:³

```
checkmark.left=self.objover.right+10
```

The reference `self.objover.right` causes the constraint solver to consult the objover slot in the current object (in this case, the check mark). The constraint solver will then request the value of the right slot in the object contained in the slot named objover.

The use of pointer variables considerably simplifies the conversion of example objects to prototypes as well as several other features of interactive design tools (see Sect. 5.2.1). It also provides the full power of procedural abstraction in constraints. Each constraint is equivalent to a procedure that may be called with a new set of arguments on each invocation. Previous constraint systems have allowed regular variables, but not pointer variables (Vander Zanden et al. 1991). For example, the 10-pixel offset could be made into a variable called `offset`, but the reference `objover` would not be permissible in other systems. If the programmer wanted the 10-pixel offset to be a variable in Garnet, he could rewrite the constraint as `self.objover.right+self.offset`, which would cause the constraint solver to look for the value of `offset` in the object that contained the constraint.

Constraints are first-class objects, just like any other object in Garnet. A constraint object contains slots that contain a pointer to the constraint's lisp code, the value last computed by the constraint, an indication of whether this value is up-to-date, and the object and slot to which this constraint is attached. A design tool is free to add other slots to this constraint object that will assist the design tool in determining the type of this constraint or other relevant information. Like other objects, constraints may serve as prototypes. An instance of a constraint will inherit the pointer to its function, thus allowing multiple constraints to use the same piece of code.

Representing constraints as objects and allowing pointer variables simplifies the integration of constraints with the prototype-instance system. When Garnet makes an instance of a prototype, it exam-

ines each of the slots in the prototype, and if a slot contains a constraint, places an instance of that constraint in the corresponding slot of the new instance object.

If the prototype is an aggregate, Garnet creates pointers in the aggregate to each child and back-pointers in each child to the aggregate. For example, the thermometer aggregate in Fig. 7 has pointers to its bulb, shaft, and mercury stored in the slots `bulb`, `shaft`, and `mercury` respectively. Similarly, each of the thermometer's children have a pointer to the thermometer stored in their parent slot. Constraints in any object in an aggregate can therefore reach any other object by traversing the aggregate hierarchy using the appropriate set of pointers. For example, the mercury needs to know the position and size of the shaft in order to position itself in the shaft. It can access these values through the parent and shaft pointers (e.g., `mercury.left=`

```
self.parent.shaft.left).
```

By referencing other objects in the aggregate hierarchy indirectly via pointers, the programmer can ensure that the constraints in instances of prototypes will automatically reference the appropriate information. Thus, Garnet is able to implement the instanting of aggregate objects simply by creating instances of each of the components of the aggregate hierarchy, creating instances of constraints in constrained slots and creating the appropriate set of pointers. It is not necessary to modify the code of the constraints themselves.

5.2.1 Advantages of constraints

Constraints are appealing, because they declaratively describe relationships between a program's entities. A constraint solver automatically keeps the constraints satisfied, thus propagating changed data to the appropriate locations. In constructing interactive design tools, constraints can be used to specify the graphical layout of the tools' objects and the dynamic graphical behavior of these objects. In addition, they can be used to support many of the services that these tools provide. Finally, the constraint system makes it easy for interactive design tools to provide constraints to the interface designers. For example, Lapidary allows users to attach constraints to objects that graphically position the objects or control their dynamic behavior.

³ For the sake of readability, we are expressing constraints in a more conventional infix notation rather than Lisp's prefix notation. In Garnet, this constraint would actually be written as `(+(gv :self :objover :right) 10)` where `gv` stands for *get value*.

Visual Computer

The support provided by constraints is covered in the following sections.

5.2.1.1 Prototypes. Interactive design tools can use pointer variables in constraints to easily convert example objects to prototypical objects. Whenever a designer attaches a constraint to an example object, the design tool can use pointer variables to indirectly reference the objects to which this object is constrained. Instances of this example object will inherit these constraints, and by setting the pointer variables appropriately, the instances can be constrained to other sets of objects. Thus, pointer variables automate part of the process of converting the example object to a prototype.

However, this conversion is not complete until the design tool has identified which slots in the example should be parameters that are set at instance-creation time. Part of this identification can be automatically performed when the designer saves the object. The design tool can check to see whether there are any slots in the saved object which reference objects that are not being saved. If so, the design tool can infer that these slots point to example objects and replace these example objects with null pointers. This process converts the pointer variables to parameters, because instances of this prototype can instantiate the pointer slots with the appropriate objects and exhibit the same behavior or layout relationship that the prototype did. For example in Lapidary, suppose the designer has drawn the two boxes and arrow shown on the screen in Figure 8a. The designer wants arrows that are used in the application to be able to attach themselves to the sides of objects, so the designer attaches the ends of the arrow to the sides of the two boxes using alignment constraints (Fig. 8b). Internally, Lapidary represents the constraints using pointer variables:

```
arrow:
  endpt1: self.from-obj.right-center
  endpt2: self.to-obj.left-center
  from-obj: box1
  to-obj: box2
```

If the designer then saves the arrow without saving the boxes, Lapidary notices that the slots `from-obj` and `to-obj` do not point to objects being saved, infers that these slots point to example objects, and replaces their values with null pointers. When an application creates an instance of this arrow, it can instantiate the `from-obj` and `to-`

`obj` slots with the appropriate objects and the instance arrow will attach itself to the desired objects in the correct way.

Other parameters, such as the label of a labeled box cannot be automatically identified without assistance from the designer. However, once the designer identifies these slots, the design tool can construct a constraint that retrieves the value of the slot from the root of the prototype. This constraint will use backpointers in the aggregate hierarchy to climb from the object that owns the slot to the top-level object in the prototype's aggregate hierarchy. For example, suppose the color of the mercury in the thermometer in Fig. 7 should be a parameter. Once the designer identifies the color as a parameter, the design tool can insert the following constraint into the color slot for the bulb and mercury objects:

```
color=self.parent.color
```

This constraint goes to the parent of either the bulb or mercury object, which is the thermometer, and retrieves the color set by the programmer.

5.2.1.2 Spreadsheet tools. Spreadsheets have proven to be a popular design interactive design tool (Wilde and Lewis 1990; Myers 1991a). One of the main difficulties in constructing a spreadsheet is building a constraint solver for the spreadsheet's equations. Garnet automatically provides such a constraint solver and a powerful set of constraints that will be adequate for most spreadsheet interface-design tools.

Of course, C32 uses the built-in constraint solver to evaluate the constraints that the user creates. In addition, C32 makes extensive use of constraints in its own implementation. For example, each value display has a constraint that ties it to the actual value in the associated object. Therefore, if the value changes as a result of user interaction with the interface, the cell's value will be automatically updated. Similarly, the visibility of the icons that show whether the slot is inherited and whether the slot contains a constraint is controlled by constraints to the actual cells. The font of the cell label and value is also constrained to the inheritance flag, so that italics is used if the slot is inherited.

5.2.1.3 Demonstration. Constraints help support several forms of *demonstrational programming*

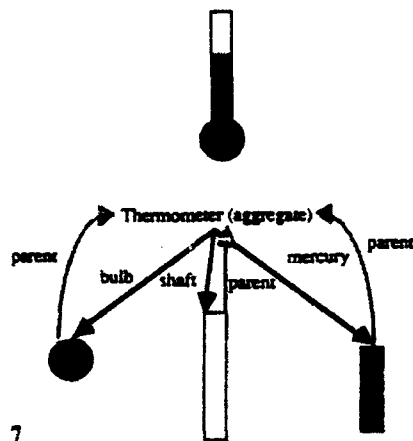
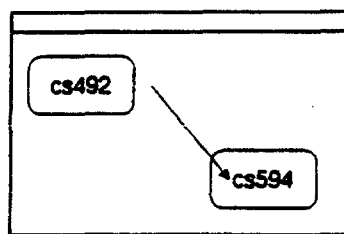


Fig. 7. A thermometer and its aggregate hierarchy. References from one object to another use paths through the hierarchy. Objects that are part of the thermometer have programmer-assigned names, such as bulb and mercury, and references to the thermometer from a part use the standard parent slot

Fig. 8a, b. The designer wants to create a prototype arrow where one endpoint should be connected to the right side of a box and the other endpoint should be connected to the left side of a box. To do this, the designer draws the arrow and two boxes a and uses a line-constraint menu to attach the endpoints of the arrow to the example boxes b. When the arrow is saved, the references to the boxes in the arrow's constraints will be replaced with null pointers, thus converting the example arrow to a prototype

Fig. 9. A menu that displays selections by moving them to the right column. A designer can demonstrate this behavior by using constraints to place an unselected item in the left column and a selected item in the right column. Lapidary will notice that different constraints control the position of a selected and unselected item's left side, and generate code to select the appropriate constraint based on the item's selection status



8a

Choice:	
History	Math
Chemistry	Computer Science
Biology	Basket Weaving
English	

9

Line Constraint Menu

primary-selection

secondary selection

unconstrain x-offset 0 x1 0

customize y-offset 0 y1 0

x2 0

y2 0

8b

Visual Computer

(Myers 1990b). In demonstrational programming, the designer manipulates objects under the observation of the design tool. The design tool then tries to infer the general form of the behavior from this specific example. Peridot (Myers 1990a) and Meta-Mouse (Maulsby and Witten 1989) are two other examples of demonstrational systems.

A simple form of demonstrational programming is one in which a designer specifies a "before" state for an object, edits it, and presents the design tool with an "after" state. For example, a "3-D" button might be in one position normally, and move when the mouse is pressed on it. To support this, the design tool would allow the user to draw the two states. It would then figure out differences, determine how to implement the changes, and generalize the behavior so that it applies to any of a related group of objects, such as a set of items in a menu.

If the designer edits the objects using constraints, then the differences are fairly easy to determine. The design tool can check for differences in constraints on the same slots, differences in offsets or scaling factors, or differences in pointer variables. Based on these differences, the design tool can synthesize a constraint that incorporates the before and after values and makes the selection based on the value of an indicator, such as a Boolean variable.

For example, Lapidary supports this form of demonstrational programming by creating a copy of the object to be demonstrated and allowing the designer to edit the copy. When the designer is finished, Lapidary compares slots in the copy (the "after" state) with slots in the original (the "before" state). For any slots that are different, Lapidary creates a constraint that simply chooses between the values, based on a controlling parameter.

For instance, suppose a designer wanted to demonstrate that items should move to the right column of the menu in Fig. 9 when the user clicks on them. This behavior could be demonstrated by selecting an item in the left column and declaring that the item is in its "before" state. The designer could then change the constraint on the item's left slot, so that the item is now positioned in the right column. The before and after states would look as follows:

	<i>Before</i>	<i>After</i>
left:	self.objover. left	self.objover. right - self.width
objover:	left-column	right-column

Because the "before" and "after" values of the objover and left slots differ in this example, Lapidary synthesizes constraints that choose between the differing values, based on the value of a variable, such as selected. For example, the constraints might be:

```
objover: if self.selected
         then right-column else left-column
left:    if self.selected
         then self.objover.right - self.width
         else self.objover.left
```

Lapidary then creates instances of these new constraints and installs them in the other items in the menu, thus generalizing the behavior from the example item to all items in the menu.

The constraints in Garnet can support more general forms of demonstrational programming as well. To guess which kinds of layouts the user is trying to achieve, as in Peridot (Myers 1990a), the creator of a design tool can derive the set of constraints that enforce the desired types of layouts. These constraints will comprise a formal, rigorous basis for the demonstrational system. When a user manipulates objects, the design tool can use various metrics to measure how well the various constraints fit the demonstrated behavior and choose those constraints that best fit the behavior. The advantage of using constraints is that quantitative metrics can be developed that rigorously assess the closeness with which various constraints match a demonstrated behavior. Because the demonstrational system can explain its inferences in terms of the metric, the designer of the demonstrational system can improve the system's inferences by modifying the metrics, based on feedback received from users.

5.2.1.4 Annotation of specifications. Some systems, such as Jade and Chisel (Singh and Green 1989), produce a rough cut of an interface from a textual specification. An interactive design tool can then be used to polish the generated interface, for example, by adding decorations or repositioning objects. If the interface designer later changes the textual specification, the changes made by the design tool should be remembered. Thus, the design tool should annotate the textual specification in some fashion.

In a Garnet-generated design tool, constraints are used when adding decorations or repositioning objects. For example, if a rectangle is drawn that en-

closes a group of radio buttons, it will typically be attached to the radio button group using constraints. In turn, the radio buttons are tied to various entities in the textual specification. By keeping track of the objects to which constraints are applied, the tool can determine to which entities in the textual specification the decorations apply. Thus, even if a designer changes the underlying textual specification, the interactive design tool can still remember the graphical changes that were made and faithfully reproduce them when the textual specification is run through the interface generator. For example, the rectangle in the above example would be constrained to surround the set of radio buttons, so the rectangle will grow automatically if new items are added to the set.

Jade takes advantage of Garnet's annotation capabilities when a graphics artist uses a direct manipulation editor to either change the layout or add decorations to a Jade-created dialog box. To annotate the textual specification, Jade keeps track of which objects are being repositioned or decorated and then maps these objects to their underlying entities in the textual specification. When the graphics artist is satisfied and saves the dialog box, Jade uses Garnet's writing facility to save the decorations, constraints used to reposition the objects, and references to the appropriate items in the textual specification.

5.2.1.5 Rule-based systems. Many systems that generate interfaces from textual specifications use rules in order to layout the various scenes in the interface (Vander Zanden and Myers 1990; Wiecha et al. 1989; Bennett et al. 1989). One way to implement the rules is to have them generate a set of constraints from a prototypical set of constraints. The use of pointer variables makes it easier for rules to create instances of these constraints, because the pointer variables can be made to point to the object or set of objects to which a rule is applied and the constraints will automatically enforce the rule. For example, a rule that places a set of buttons at the top right of another group of buttons might be expressed as:

```
at-top-right-rule:
  left: self.buttons.right+10
  top: self.buttons.top
```

A tool could apply this rule by generating instances of these constraints in the appropriate slots in a button group and setting the buttons pointer.

uses rules similar to this one in laying out objects in a dialog box.

5.3 Automatic graphical update

The graphical object system in Garnet is different from many other systems in two respects. First, it uses a *retained object model* that allows it to automatically update the screen when objects change or a part of the window becomes uncovered. Most other toolkits force the application to manually handle redisplay by determining which objects are changed and which objects they overlap and then issuing erase and draw commands that cause the display to be appropriately updated. A second difference from other systems is that the display system is integrated with the constraint system, so the constraint system automatically notifies the display system whenever an object changes.

The retained object model is somewhat similar to a display list in that each graphical object on the screen corresponds to an object in memory. However, the objects are at a higher level than the objects in a display list, because they are integrated with the constraint system, can be accessed by an application, and are used by Garnet to determine which portions of the display to update. For example, to move a rectangle to a new position, the application sets the left and top slots of the object and Garnet automatically takes care of erasing the object at its old position and drawing it at the new position. In addition, the constraint system propagates the changes to other objects in the system, and these objects, as well as any other objects that overlap the changed objects, are also redrawn. If the window manager needs part of the window to be redrawn (for example, because the user has uncovered it), Garnet can handle this automatically without involving the application.

The algorithm used by Garnet always tries to minimize the number of objects that are erased and redrawn, rather than simply redrawing the entire window, which can be important for complex scenes. Garnet keeps track of all objects changed by either the application or the constraint system. When asked to update the screen, it finds the bounding rectangles of the changed objects in their old and new positions and then redraws all objects that intersect those regions. Clipping regions, which are supported by the underlying window managers, are used so that other objects will not

Visual Computer

be affected. As an example of the resulting performance, moving one object through a window containing 200 other objects takes 14.9 milliseconds per move on a Sun SPARCStation (67 moves per second) rather than the 188 milliseconds (5.32 moves per second) it would take if Garnet simply redrew all the objects in the window each time.

The advantage of the retained object model for tool builders is that they do not have to have to build the elaborate data structures required to refresh the screen and they do not have to handle the complex task of interfacing the constraint solver with the display manager. When a property of an object should change (e.g., to have a new color or position), the tool can simply set the appropriate slot of the object. If other objects are affected by the change, their slots will be automatically changed as well. If an object should be deleted, it can simply be removed from the window's list of objects. Garnet handles the rest.

Another advantage is that often tools do not need to create their own representation of the data. Each window contains a list of the objects in it and applications are free to add their own slots to objects to hold any necessary extra information (as described above in Sect. 5.1.3.3). Therefore, the window's object list can often be used by applications as their description of the current state. All the application-specific slots will be written out automatically to the file along with the standard graphical slots.

5.4 Behaviors

In Garnet, the graphical objects do not respond to input events. Instead, separate objects, called *interactor* objects, handle all input (Myers 1989b; Myers 1990c). The interactors encapsulate the common interactive behaviors found in direct manipulation interfaces. Each type of interactor handles a different kind of behavior. Currently, the interactor types are:

Menu- Interactor:	to choose one or more items from a set or for a single, stand-alone button. This interactor can be used for menus, radio buttons, and making selection "handles" appear over objects in a graphics editor.
----------------------	--

Move-Grow- Interactor:	to move or change the size of an object or one of a set of objects using the mouse. This interactor can be used for one-dimensional or two-dimensional scroll bars, horizontal and vertical gauges, and for moving or growing application objects in a graphics editor.
---------------------------	---

New-Point- Interactor:	to enter one, two, or an arbitrary number of new points using the mouse, for example for creating new lines or rectangles in an editor.
---------------------------	---

Angle- Interactor:	to calculate the angle that the mouse moves around some point. This can be used for circular gauges or for rotating objects.
-----------------------	--

Trace- Interactor:	to get all of the points the mouse goes through between start and end events, as is needed for free-hand drawing.
-----------------------	---

Text-String- Interactor:	to input a small (optionally multiline) string of text.
-----------------------------	---

Each interactor is parameterized in various ways, so the programmer can control the mouse or keyboard events that cause it to start and stop as well as the optional application procedures to be called on completion. The most significant parameters, however, are the objects that are used as the places where the interactor should operate and the (optional) objects that will handle feedback. For example, the programmer might create a set of text objects to be the domain of selection in a menu and a black XOR rectangle to be the feedback. Each type of interactor has a well-defined protocol with which it controls the graphics. This protocol is explained in depth in (Myers 1990c).

The interactors are first-class objects, and they can be included in prototypes (e.g., a prototype scroll bar). Interactors in prototypes also will be saved to disk and read back in automatically.

Another feature of interactors is that they can be easily turned on and off. Each interactor has an active slot, which can contain a constraint that determines whether the interactor should run or not. This makes it easy for design tools to implement the "Build" vs "Run" modes: when in build-mode, the interactors in the interface under construction are inactive, and when in run-mode, they are active. Similarly, the interactors that handle selection and editing for the design tool use the

reverse constraint. Gilt and Lapidary use this feature to disable widgets unless the user hits the "Run" button.

5.4.1 Advantages of interactors for writing graphical programs

The interactors paradigm helps programmers create graphical programs in a number of ways. First, by separating out the graphics from the behavior, the code of the tool itself is more modular. Second, it makes it easier to investigate different looks and feels. Third, because each interactor provides a high-level of built-in functionality, many otherwise complex behaviors can be added to interfaces easily. For example, a common way to handle selection is for the user to press on an object and have "handles" appear (see Fig. 10). When the user presses on a handle, the object underneath moves or changes size. This behavior can be provided in Garnet by using a menu-interactor with the handles as a feedback object for the selection and a move-grow-interactor with the handles as the starting position. The programmer only needs to create instances of these two types of interactors and provide the appropriate parameters; no event loops need to be coded and no methods need to be written.⁴ Also, in-place text editing is very easy to support, simply by attaching a text-editing interactor to any text string in the interface. For example, it was easy in the Gilt interface builder to support editing of the menu and button labels directly in the graphics window using a text-interactor (rather than requiring the new labels to be entered in a property sheet). The sizes of the graphics surrounding the label are automatically adjusted as letters are typed, due to the constraints built into the widgets (for example, the rectangles around a labeled-button will expand and shrink).

Fourth, the interactors package supports dragging objects among windows in the same way that they are moved inside a single window. This can be used to move or copy objects from one window to another rather than using the more clumsy cut-and-paste style.

Fifth, the interactors package supports the window manager cut buffer to allow easy copying of text from one application to another. Extensions to support copying of graphics are planned.

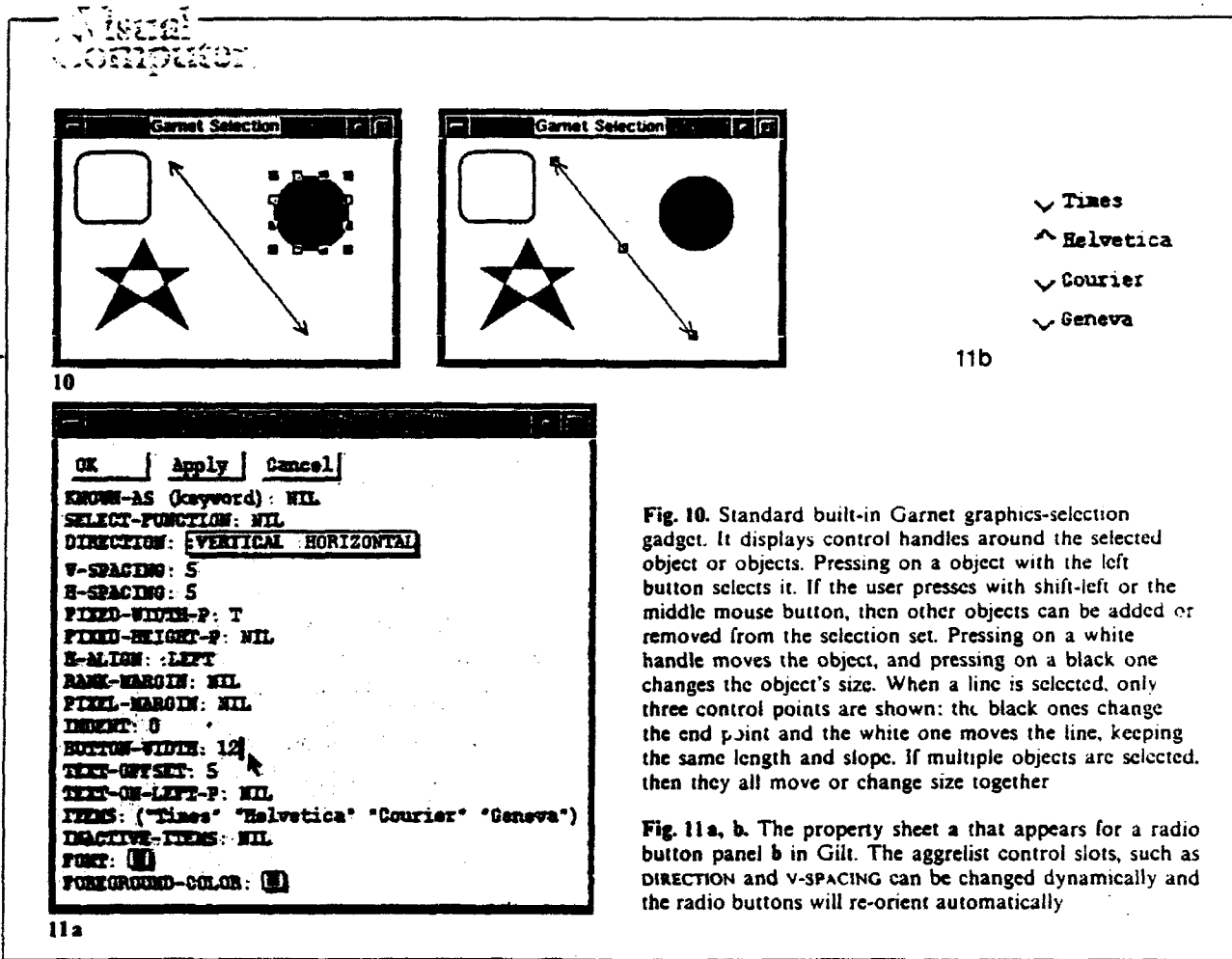
⁴ However, see Sect. 5.6, where a widget that handles this automatically is explained

5.4.2 Advantages of interactors for IDTs

In addition to the general advantages of interactors, there is a particular advantage for the designer of an interactive design tool. If the tool wants to allow the user to define the interactive *behaviors* of the objects being designed, it must first have a model of those behaviors. In many systems, this model consists of a set of pre-defined interaction techniques that are tightly bundled with the graphics, making it difficult, if not impossible, to attach behaviors to *application* objects. In contrast, the interactors model separates behavior from the graphics, so that behaviors may be attached to widget objects (e.g., menus, scroll bars, buttons) as well as application objects. Indeed, several of the interactors, such as the move-grow, new-point, and trace interactors, were specifically designed for application objects, while the menu, text-string, and angle interactors all have application uses (the menu interactor, for example, can be used to select one or more application objects).

The interactors model also provides a rich set of parameters that allow designers to specify a wide variety of behaviors without having to drop down to the programming-language level. Because there are only six types of interactors and the parameters are well-defined, an interactive design tool can easily provide dialog boxes for the user to fill in the desired values or can attempt to infer the behavior and its parameters from the user's actions. Thus, the interactors make it much easier to implement interactive design tools that allow a designer to specify the behavior of application objects as well as the behavior of new widget objects.

Lapidary is a good example of a design tool that takes advantage of these features of interactors. It provides dialog boxes for each of the interactors and allows the user to attach graphics to an interactor by selecting graphical objects and pointing to the appropriate graphical parameter in the dialog box (e.g., the start-where parameter, which controls which objects the interactor operates on, or the interim and final feedback parameters, which control what type of feedback the user sees as the behavior is executing). For example, if the user is creating a new kind of menu, he can pull up the choice-of-items dialog box (see Fig. 3), create an instance of a menu interactor, and attach it to the graphics that the user has created. The behavior can then be interactively tested by putting Lapidary in "test" mode.



Lapidary exploits interactors in other ways as well. Internally, interactors make it much easier to create all of Lapidary's complicated behaviors, such as the multiple ways of selecting objects and providing feedback or the different ways of selecting constraints in the iconic constraint menus. Externally, Lapidary uses the interactors to separate the editing of graphics and behaviors. Users creating objects from scratch can concentrate on defining their graphics and behaviors separately and integrating them when they are finished. Furthermore, they can later edit either the behaviors or graphics, thus modifying either the look or the feel without touching the other. Instances of widgets from prototype libraries can be similarly edited. This separation allows a user to rapidly prototype different looks-and-feels.

Currently, we are investigating ways to infer the parameters of the interactors from a demonstration of the behavior, as in the earlier Peridot system (Myers 1988). Because there are only six possible behaviors and a small number of parameters, the

inferences about the meaning of the user's demonstration have a good chance of being correct.

5.5 Automatic layout

There are many times when elements of a user interface need to be displayed in a regular fashion. For example, the items in a menu are often evenly spaced in a column. Garnet provides various special forms of aggregates that automatically lay out their components. Interactive design tools can easily provide automatic layout to users by simply creating instances of these special aggregates and allowing users to then add the components.

One such aggregate, called an "aggrelist," will arrange the elements in a row, column, or table. The programmer can specify the spacing of the elements and whether they are centered or justified to the left, right top, or bottom. Each element can be individually created by a program and added to the aggregate. Alternatively, a single prototype can be

supplied along with a list of strings or other values and the aggregate will automatically create an instance of the prototype for each string or value.

Aggrelists are used throughout the Garnet widget set to control the layout of menus and buttons. Because the layout parameters to an aggrelist can be changed dynamically (e.g., to change the elements to be centered or left-justified or to put them in multiple rows or columns), this allows the toolkit user to customize the layout of the elements. For example, the Gilt interface builder only needs to set the value of the orientation field of an aggrelist to change a set of radio buttons from horizontal to vertical. No special code is needed in Gilt to adjust the layout. The controlling slots are provided to the user as fields in a property sheet (see Fig. 11). Aggrelists are also helpful in the C32 implementation, where they are used to lay out the fields in columns.

Another special type of aggregate will arrange the elements in a tree or graph. A default layout algorithm is supplied, but the programmer can supply a different one if desired. There are also default prototypes for graphics for the nodes and arcs, but again the programmer can supply different prototypes.

5.6 Widgets

The Garnet widget set contains many widgets that help create interactive programs, such as visual interactive design tools. It contains the standard widgets found in other toolkits, such as radio buttons, check boxes, scroll bars, sliders, text-entry fields, and various forms of fixed, pop-up, and pull-down menus. These come in two varieties: the Garnet look-and-feel shown in Fig. 1 and a Motif look-and-feel (Fig. 12). These can be used to make the dialog boxes and main command menus of an editor. There are also widgets to pop-up windows with error messages, confirmation requests, and prompts. It is interesting to note that implementing the Motif widget set (which does not use any of the Motif code that implements the Xtk C version) took only two man-weeks on top of the Garnet toolkit intrinsics.

In addition, however, the Garnet widget set also contains many high-level widgets that can help with the *insides* of the design tool or end-application windows. For example, one gadget supplies

a scrolling window facility that displays horizontal and vertical scroll bars and automatically handles refreshing parts that are scrolled onto the screen.

Another special Garnet widget supports selection of graphical objects. If the programmer creates an instance of a multiple-selection object in a window, then any of the objects in the window can be selected using the mouse and selection handles will appear around them (see Fig. 10). The objects then can be moved or changed in size. All this functionality is supplied by the multiple-selection widget, and the programmer only has to make sure that the objects to be edited understand the standard protocol so they can be modified. This widget is used extensively by Gilt.

Another useful widget for some design tools is a property sheet, which shows labels and current values (see Fig. 11). Each label is usually the slot name of the field of the object that specifies the property and the value is usually a textual representation of the value. However, the property-sheet widget allows an arbitrary widget to be used as the value. For example, in Fig. 11, a special widget is used for the DIRECTION slot, which allows the user to select one of a set of values with the mouse, and the FONT property contains an icon, which pops up a font-selection dialog box (which itself was created using Gilt).

An interesting feature of the Garnet system is that the built-in widgets can be easily used in interactive design tools even though they were hand-coded without any thought for their use in this way. The ability to dynamically load Garnet objects and dynamically change their properties makes any Garnet object usable in design tools. For example, the widgets displayed by Gilt are simply those that were already in the Garnet widget set. We did not have to create new widgets for use by Gilt.

Lapidary takes advantage of Garnet's button and type-in widgets in constructing its interactor dialog boxes, main editor menu, and iconic constraint menus. It also extensively uses the error gadgets to inform users of various mistakes. In the future, we plan to integrate the property-sheet widgets into Lapidary in order to allow the user to edit non-graphical properties of an object. Lapidary does not use the multiple-selection widget, because it implements a more complex selection model. For example, by using different keyboard keys and mouse buttons, the user can select the aggregate of the selected object or select the object hidden underneath the object.

Visual Computer

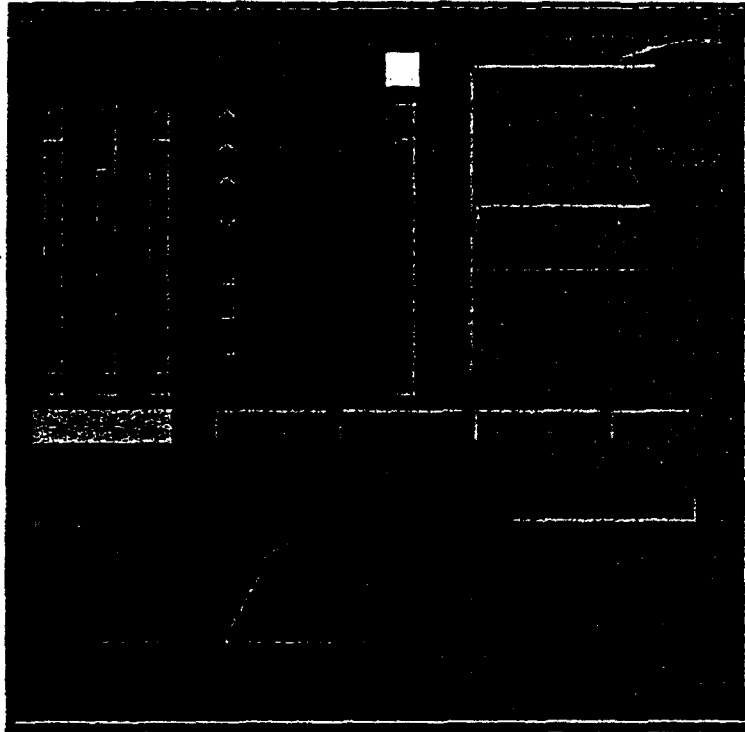


Fig. 12. Some of the widgets with a Motif look-and-feel implemented in Garnet

Finally, the C32 window uses scroll bars and the scrolling-window widget. Of course, the menus and buttons are Garnet widgets. The value of each slot uses a scrollable-text field widget (so if the value is too large, it can be scrolled left and right).

5.7 Other toolkit features

In addition to those discussed above, the Garnet system also provides a few other features that make it easier to build interactive design tools.

First, the object-oriented graphics package and the interactor objects hide the details of the window manager from application programs. Therefore, a programmer can create a tool that will run on different architectures. In addition, the interfaces created by the users of the interactive design tools will also run on multiple architectures.

Another feature stems from the decision to implement Garnet in Lisp. The built-in Lisp interpreter is available to the tool builder so that no additional interpreters are needed. This helps support the interactive loading of objects and interactive creation of constraints, as described above. In addition, the

interpreter can be used to support "Run Mode," where the tool allows the interface to be exercised to show how it will operate for the end user. The tool can simply create the actual objects, constraints, and interactors that will be present in the end-user interface and the Lisp interpreter allows the interactors and constraints to operate.

6 Future work

The Garnet project is on-going and we are constantly trying to improve the toolkit and high-level tools. Current work on the toolkit is focusing on increasing the functionality and efficiency of the object system and the constraints. We will also add gesture recognition as a new primitive interactor type, so that applications and interactive design tools can experiment with gestural interfaces.

Most future work, however, will concentrate on the interactive design tools themselves. The tools described here need to be completed and released and more functionality needs to be added. In addition, we plan to explore new forms of tools that will allow even more of the application-specific be-

havior of objects to be specified. The emphasis will be on specifying the behavior by demonstration rather than through dialog boxes or hand-coding.

In addition to new specific tools, we want to look at more comprehensive "frameworks" for interactive design tools and other interactive applications. For example, almost all applications have a palette of choices and a workspace window in which the user creates instances of objects in the palette. Objects in the workspace window can then be selected and processed further. Whereas the toolkit provides primitives that make all these steps easy, the programmer still has to put them together. A framework like Unidraw (Vlissides and Linton 1989) or MacApp (Schmucker 1986) would make this easier. Thus, we will be developing one for Garnet. The planned framework will also support Undo, Help, and other high-level operations.

Another focus will be on how to provide toolkit-level support for demonstrational interfaces to make it easier for design tools and applications to implement a demonstrational interface.

7 Conclusions

By specifically designing the underlying toolkit intrinsics to support the *insides* of application windows, Garnet is able to make the creation of visual interactive design tools significantly easier than with conventional toolkits. Features such as the use of a prototype-instance object system, constraints, automatic graphical update, automatic object saving, automatic layout, the use of interactor objects, and high-level widgets that support graphical selection, property sheets, and error reporting have allowed us to quickly create a variety of innovative interactive design tools. In addition, these tools are able to help build the application-specific, highly-interactive graphical parts of user interface and not just layout the widgets that go around the main application window or in dialog boxes.

Acknowledgements. For help with this paper, we would like to thank Bernita Myers and the referees.

This research was sponsored by the Avionics Lab, 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. The views and conclusions contained in this document are those of the authors and should not be interpreted as repre-

senting the official policies, either expressed or implied, of the U.S. government.

Additional support for Garnet was supplied by Apple Computer, General Electric, and NEC.

References

- Barth P (1986) An object-oriented approach to graphical interfaces. *ACM Trans Graph* 5(2):142-172
- Bennett WE, Boies SJ, Gould JD, Greene SL, Wiecha CF (1989) Transformations on a dialog tree: rule-based mapping of content to style. *Proc ACM SIGGRAPH Symposium on User Interface Software and Technology*, Williamsburg, VA, pp 67-75
- Borning A (1981) The programming language aspects of thing-lab; a constraint-oriented simulation laboratory. *ACM Trans on Progr Lang Syst* 3(4):353-387
- Borning A, Duisberg R (1986) Constraint-based tools for building user interfaces. *ACM Trans Graph* 5(4):345-374
- Brown JR, Cunningham S (1989) *Programming the user interface: principles and examples*. John Wiley & Sons, New York
- Buxton W, Lamb MR, Sherman D, Smith KC (1983) Towards a comprehensive user interface management system. *Proceedings SIGGRAPH'83*, Detroit. *Comput Graph* 17(3):35-42
- Cardelli L (1988) Building user interfaces by direct manipulation. *Proc ACM SIGGRAPH Symposium on User Interface Software*, Banff, pp 152-166
- Chambers C, Ungar D, Lee E (1989) An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. 49-70. *ACM Conference on Object-Oriented Programming, Systems Languages and Applications*. Sigplan Notices, 24(10)
- Freeman-Benson BN, Maloney J, Borning A (1990) An incremental constraint solver. *Commun ACM* 33(1):54-63
- Hartson HR, Hix D (1989) Human-computer interface development: concepts and systems for its management. *Comput Surv* 21(1):5-92
- Henderson Jr DA (1986) The trillium user interface design environment. *Human Factors in Computing Systems*. *Proc SIGCHI'86*, Boston, pp 221-227
- Henry TR, Hudson SE (1988) Using active data in a UIMS. *Proc ACM SIGGRAPH Symposium on User Interface Software*, Banff, pp 167-178
- Krasner GE, Pope ST (1988) A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *J Object Oriented Progr* 1(3):26-49
- Lieberman H (1986) Using prototypical objects to implement shared behavior in object oriented systems. *ACM Conference on Object-Oriented Programming, Systems Languages and Applications*. Sigplan Notices 21(11):214-223
- Linton MA, Vlissides JM, Calder PR (1989) Composing user interfaces with InterViews. *IEEE Comput* 22(2):8-22
- Maulsby DL, Witten IH (1989) Inducing procedures in a direct-manipulation environment. *Human Factors in Computing Systems*. *Proc SIGCHI'89*, Austin, pp 57-62
- McCormack J, Asente P (1988) An overview of the X toolkit. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, Banff, pp 46-55
- Myers BA (1988) *Creating user interfaces by demonstration*. Academic Press, Boston

Visual Computer

- Myers BA (1989a) User interface tools: introduction and survey. *IEEE Software* 6(1): 15-23
- Myers BA (1989b) Encapsulating interactive behaviors. *Human Factors in Computing Systems. Proc SIGCHI'89, Austin*, pp 319-324
- Myers BA, Vander Zanden B, Dannenberg RB (1989) Creating graphical interactive application objects by demonstration. *Proc ACM SIGGRAPH Symposium on User Interface Software and Technology, Williamsburg*, pp 95-104
- Myers BA (1990a) Creating user interfaces using programming-by-example, visual programming, and constraints. *ACM Trans Progr Lang Syst*, 12(2): 143-177
- Myers BA (1990b) Demonstrational interfaces: a step beyond direct manipulation. Tech Rep CMU-CS-90-162, Carnegie Mellon University Computer Science Department
- Myers BA (1990c) A new model for handling input. *ACM Trans Inf Syst* 8(3): 289-320
- Myers BA, Giuse DA, Dannenberg RB, Vander Zanden B, Kosbie DS, Pervin E, Mickish A, Marchal P (1990) Garnet: Comprehensive support for a graphical, highly-interactive user interfaces. *IEEE Computer* 23(11): 71-85
- Myers BA (1991a) Graphical techniques in a spreadsheet for specifying user interfaces. *Human Factors in Computing Systems. Proc SIGCHI'91, New Orleans*, pp 243-249
- Myers BA (1991b) Separating application code from toolkits: eliminating the spaghetti of call-backs. *Proc ACM SIGGRAPH Symposium on User Interface Software, Hilton Head*
- Myers BA, Giuse D, Dannenberg RB, Vander Zanden B, Kosbie D, Marchal P, Pervin E, Mickish A, Kolojejchick JA (1991) The Garnet Toolkit reference manuals: support for highly-interactive, graphical user interfaces in Lisp. Tech Rep, CMU-CS-90-117-R, Carnegie Mellon University Computer Science Department
- Schmucker KJ (1986) MacApp: an application framework. *Byte* 11(8): 189-193
- Singh G, Green M (1988) Designing the interface designer's interface. *Proc ACM SIGGRAPH Symposium on User Interface Software, Banff*, pp 109-116
- Singh G, Green M (1989) A high-level user interface management system. *Human Factors in Computing Systems. Proceedings SIGCHI'89, Austin*, pp 133-138
- Sutherland IE (1963) SketchPad: a man-machine graphical communication system. *AFIPS Spring Joint Computer Conference* 23: 329-346
- Szekely P (1990) Template-based mapping of application data to interactive displays. *Proc ACM SIGGRAPH Symposium on User Interface Software, Snowbird*, pp 1-9
- Vander Zanden B (1989) Constraint grammars - a new model for specifying graphical applications. *Human Factors in Computing Systems. Proceedings SIGCHI'89, Austin*, pp 325-330
- Vander Zanden B, Myers BA (1990) Automatic, look-and-feel independent dialog creation for graphical user interfaces. *Human Factors in Computing Systems. Proceedings SIGCHI'90, Seattle*, pp 27-34
- Vander Zanden B, Myers BA (1991) Creating graphical interactive application objects by demonstration: the Lapidary interactive design tool. 12-minute videotape. *SIGGRAPH Video Review Issue* 64
- Vander Zanden B, Myers BA, Giuse D, Szekely P (1991) The importance of indirect references in constraint models. *Proc ACM SIGGRAPH Symposium on User Interface Software, Hilton Head*
- Vlissides JM, Linton MA (1989) Unidraw: a framework for building domain-specific editors. *Proc ACM SIGGRAPH Symposium on User Interface Software and Technology, Williamsburg*, pp 158-167
- Wiecha C, Bennet W, Boies S, Gould J (1989) Generating user interfaces to highly interactive applications. *Human Factors in Computing Systems. Proc SIGCHI'89, Austin*, pp 277-282
- Wilde N, Lewis C (1990) Spreadsheet-based interactive graphics: from prototype to tool. *Human Factors in Computing Systems. Proceedings SIGCHI'90, Seattle*, pp 153-159



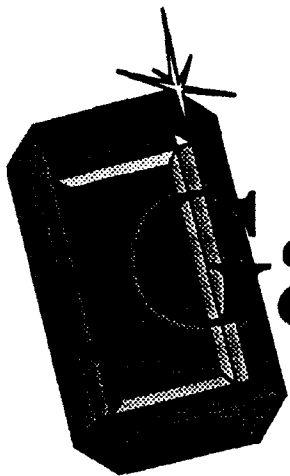
BRAD A. MYERS is a research computer scientist at Carnegie Mellon University, where he is the principal investigator for the Garnet User Interface Development Environment. From 1980 until 1983, he worked at PERQ Systems. Myers received his PhD in computer science at the University of Toronto, where he developed the Peridot UIMS. He received his MS and BSc degrees from the Massachusetts Institute of Technology, during which time he was a research intern at Xerox PARC.

His research interests include user-interface development systems, user interfaces, programming by example, visual programming, interaction techniques, window management, programming environments, debugging, and graphics. He belongs to SIGGRAPH, SIGCHI, ACM, IEEE, and the IEEE Computer Society.

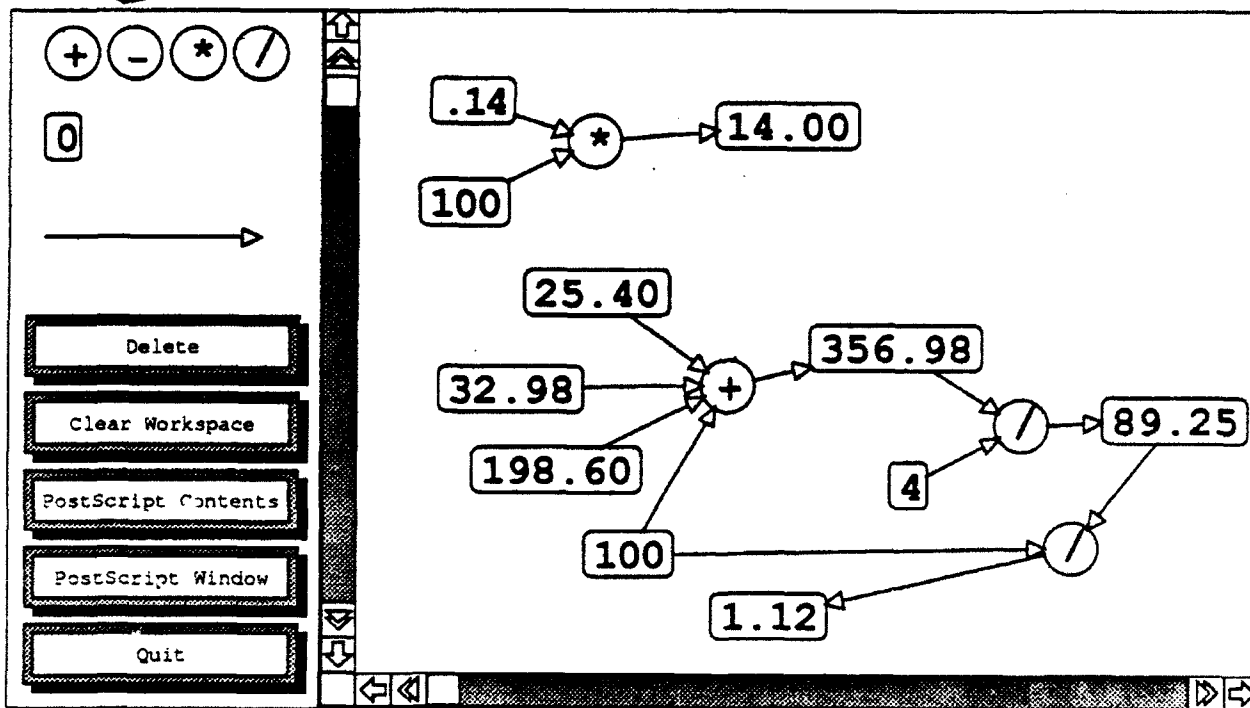


BRAD VANDER ZANDEN is assistant professor at the University of Tennessee and an active participant in the Garnet project. His work has focused on developing new paradigms for creating interactive visual environments and on creating new incremental algorithms for constraint satisfaction. More generally his research interests include user-interface development systems, program visualization and animation, constraint systems, programming environments, and graphics. Dr

Vander Zanden received his bachelors degree from Ohio State, and his MS and PhD from Cornell. He also spent two years as a postdoctoral fellow at Carnegie Mellon. He is a member of the ACM and IEEE.



arnet



Reprinted from *Watch What I Do: Programming by Demonstration*,
Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman,
David Maulsby, Brad A. Myers and Alan Turransky, eds.
Cambridge, MA: The MIT Press, 1993. pp. 219-236.

Garnet: Uses of Demonstrational Techniques

Chapter
10

by
Brad A. Myers
Carnegie Mellon University

Garnet is a comprehensive user interface development environment in Lisp for X/11 (Display Postscript and Macintosh versions are in progress). It helps create graphical, highly-interactive, direct manipulation user interfaces. Garnet contains many high-level tools, including the Gilt interface builder [Myers 91d], the Lapidary interactive design tool [Myers 89b], the C32 spreadsheet system [Myers 91a], the Jade dialog box system [Vander Zanden 90], and more to come. Garnet also contains a complete toolkit, which uses constraints [Vander Zanden 91a], a prototype-instance object model, and a new model for handling input [Myers 90c]. The toolkit also contains two complete widget sets, one with the Motif look and feel.

Typical applications created with Garnet include: drawing programs similar to Macintosh MacDraw, user interfaces for expert systems and other AI applications, box and arrow diagram editors, graphical programming languages, game user interfaces, simulation and process monitoring programs, user interface construction tools, CAD/CAM programs, etc. Garnet is in the public domain and is freely available. As of fall, 1992, over 30 projects around the world are using the system regularly. You can get Garnet by anonymous FTP from `a.gp.cs.cmu.edu`. Change to the directory `/usr/garnet/garnet/`

Introduction

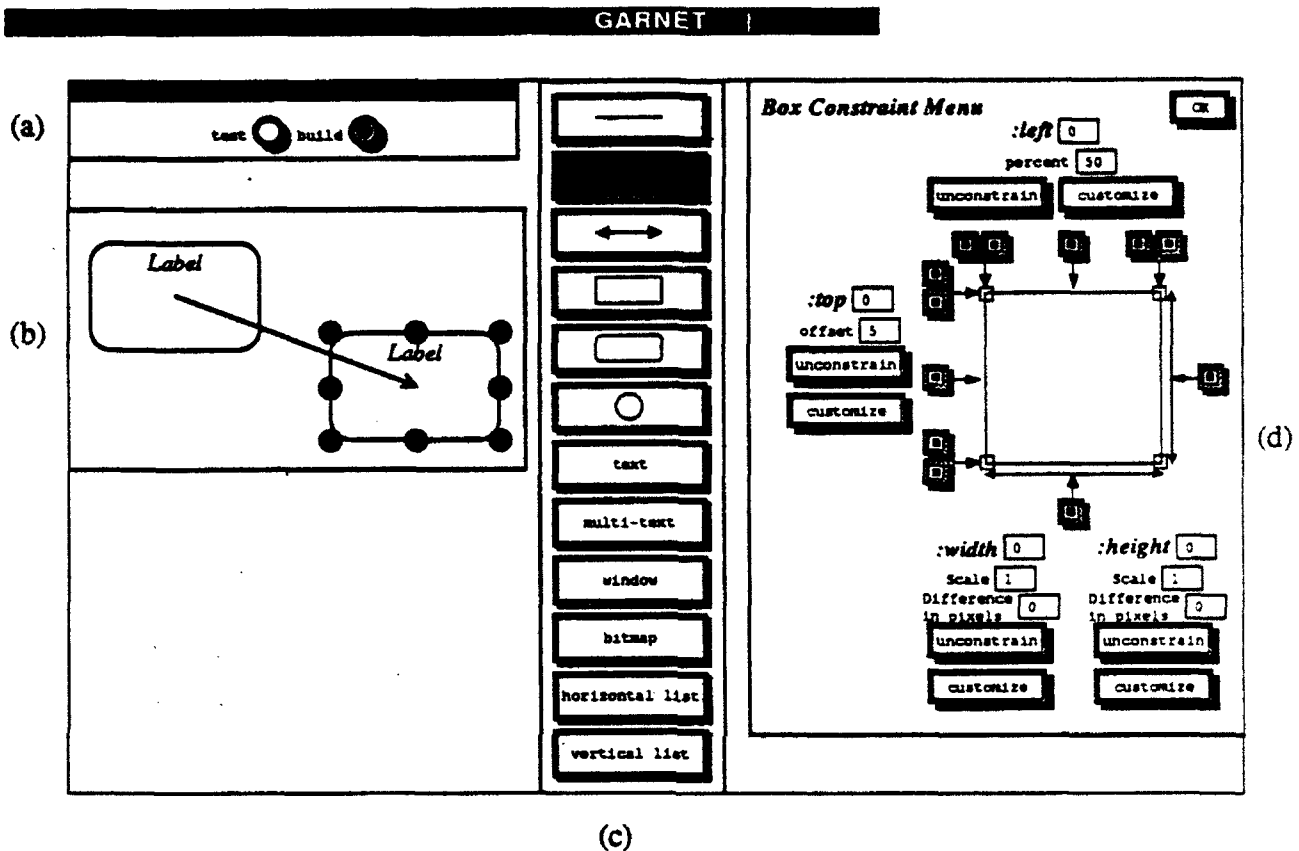
BRAD MYERS

and retrieve the README file for instructions. Or you can send electronic mail to `garnet@cs.cmu.edu`. Garnet stands for Generating an Amalgam of Realtime, Novel Editors and Toolkits.

One of the important goals of the Garnet project is to allow *all* aspects of the user interface to be created without conventional programming. In particular, we want to allow the user to *draw* example pictures to show what the user interface will look like, and then *demonstrate* how the user interface will respond to inputs from the end user. As a result, demonstrational techniques are widely used in Garnet, mainly in the various higher-level tools. This chapter discusses some of these. Other papers about Garnet discuss the overall design [Myers 90d], the components, the programming style [Myers 92a] [Myers 92f], and there is a complete reference manual [Myers 92b].

Lapidary

The Lapidary user interface tool allows the pictorial aspects of programs to be specified graphically [Myers 89b] [Vander Zanden 91b]. A "Lapidary" is a workman who cuts, polishes and engraves precious stones, and here is a Lisp-Based Assistant for Prototyping Interface Designs Allowing Remarkable Yield. In addition, the behavior of these objects at run-time can be specified using dialogue boxes and by demonstration. In particular, Lapidary allows the designer to draw pictures of application-specific graphical objects which will be created and maintained at run-time by the application. This includes the graphical entities that the end user will manipulate (such as the components of the picture), the feedback that shows which objects are selected (such as small boxes around an object), and the dynamic feedback objects (such as hair-line boxes to show where an object is being dragged). Lapidary is a direct descendent of Peridot (chapter 6) and extends a number of Peridot's ideas.



In addition, like Peridot, Lapidary supports the construction and use of "widgets" (sometimes called interaction techniques or gadgets) such as menus, scroll bars, buttons and icons. Lapidary therefore supports *using* a pre-defined library of widgets, and *defining* a new library with a unique "look and feel." The run-time behavior of all these objects can be specified in a straightforward way using constraints and abstract descriptions of the interactive response to the input devices. Lapidary generalizes from the specific example pictures to allow the graphics and behaviors to be specified by demonstration.

Graphical objects can be created in a number of different ways using Lapidary. As shown in Figure 1, the standard menus provide the usual range of graphical primitives, so objects can be created from scratch.

Figure 1: The workspace window of Lapidary (b), where a node of a graph editor is being created, along with the standard commands (a), object menus (c), and a dialog box for setting constraints on rectangles (d).

BRAD MYERS

Constraints

A central feature of Lapidary that makes it appropriate for creating run-time application graphics is the use of *constraints*. Constraints allow the designer to specify a relation between a graphic object and other objects in the scene, and have that relation maintained at run-time by the system. If a constraint is one of a standard set, then it can be specified easily using the Lapidary menus (see Figure 1-d). These menus support having objects be connected on their edges or in the middle, with optional offsets. The sizes of objects can also be related. There are different windows showing the constraints for lines and a few other objects. Experience with Peridot demonstrates that these simple types of constraints make up the vast majority of those needed in typical user interfaces.

Sometimes, designers want to use relationships that cannot be created out of these simple choices. In that case, the *Custom* option is selected, and the designer is allowed to type in an arbitrary Common Lisp expression specifying the constraint using the C32 system (discussed below).

Unlike Peridot, Lapidary currently does not try to infer the graphical constraints. Instead, they must all be specified explicitly using the dialog boxes. With Lapidary, we wanted to concentrate on creating a practical tool that extends the range of interfaces that can be produced, and the constraint inferencing in Peridot was felt to be too risky for the first version. Future Garnet tools will revisit this issue.

In order for the graphical objects to be useful at run-time, the specific constraints must be *generalized* to work on run-time objects, rather than on the specific example objects used in the editor. For example, in Figure 1, the label on the nodes should change, but still stay centered, as the node is replicated. Thus, the constraints need to be generalized to reference objects indirectly through variables, rather than by using specific object names. To do this, the reference to the object is replaced with an expression that calculates the desired object, and stores it in a special slot. The constraint system then automatically ensures that the constraints change whenever the slot is set with a different object.

It is important to emphasize that Lapidary makes these transformations automatically. The user interface designer never sees any of the code. Even if the designer created custom constraints by typing Lisp code, the references in the expression can be to example objects (selected by pointing at them with the

GARNET I

mouse), and the system will convert these references to be general variables where appropriate.

Another way that Lapidary generalizes from the examples is to automatically make copies of objects at run-time. For example, to show the selection in a drawing editor, the designer might draw a single set of selection handles around an example object. However, at run time, multiple objects might be selectable, so Lapidary arranges for the selection handles to be duplicated at run-time if necessary.

Interactive behavior

Although it is useful to prototype the graphic appearance of user interfaces, it is much more useful if the interactive behavior can also be specified easily. Lapidary therefore provides this capability. In order to edit the behavior of objects, or to add behavior to new objects, we have encapsulated a number of kinds of interactive behaviors into "interactor" objects [Myers 90c], each of which has its own dialogue box for specifying properties. For example, to change which mouse button operates a menu, it is only necessary to change the button indicated in the dialogue box.

Often, there will be a specific object that serves as the feedback for an operation. For example, a reverse-video rectangle might move over the items in the menu to show which is the current selection. In other cases, the objects *themselves* should change to be the feedback. For example, the currently selected item in a menu might be shown in italics. Another use is to have buttons move to cover their shadows (and therefore look more "3-D"), as in the Motif and Garnet look and feels. In this case, the desired changes can be shown *by demonstration*. To specify the changes by demonstration, first the designer selects the objects that will change, and then hits a button in the dialogue box. The full current state of the selected objects is remembered. Next, the designer edits the objects in whatever way desired, for example to make the string be italic. Then, another button is hit, and Lapidary creates a constraint that will choose between the two values based on whether the object is selected or not. Changes can be made to as many properties as desired, and correct constraints will be created for all of them.

Summary

Through the use of demonstrational techniques, Lapidary is able to allow the designer to interactively create far more of the user interface than any other tool.

BRAD MYERS

In particular, new widgets and application-specific objects can be created. Demonstrational techniques are crucial since these objects all are parameterized and will change dynamically at run time, so it is only possible to draw *examples*, not the actual objects to be used. These examples generalized into named prototypes which the applications can then make instances of at run time.

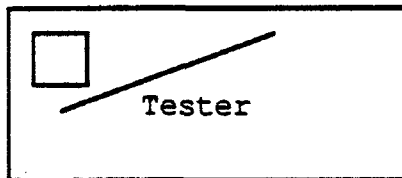
C32

When the iconic menus in Lapidary are not sufficient for specifying the desired constraints, Garnet provides the C32 spreadsheet program to help enter more complex constraints [Myers 91a]. C32 can also be used stand-alone. It displays and allows the user to edit any kind of object and constraint, no matter how they were created: by hand-coding, by using Lapidary, or by using C32. C32 stands for CMU's Clever and Compelling Contribution to Computer Science in Common Lisp which is Customizable and Characterized by a Complete Coverage of Code and Contains a Cornucopia of Creative Constructs, because it Can Create Complex, Correct Constraints that are Constructed Clearly and Concretely, and are Communicated using Columns of Cells that are Constantly Calculated so they Change Continuously and Cancel Confusion.

Figure 2 shows a typical instance of C32. Each column contains a separate object. Rows are labeled with the names of the slots, such as :left, :top, :width, :height, :visible, etc. Since different objects can have different slots, the slot names are repeated in each column. For example, lines have slots for the endpoints (:x1, :y1, :x2, :y2) but rectangles do not. Also, each object's display can be scrolled separately, so each has its own scrollbar. This makes the spreadsheet look somewhat like a multi-pane browser as in Smalltalk.

GARNET		
C32::R		
:Left	10	
:Top	10	
:Width	20	
:Height	20	
:Visible	T	
:Line-Style	OPAL:DEFAULT-LI	
:Filling-Style	NIL	
:Draw-Function	:COPY	
:Window	C32::W	
:Parent	C32::DEMO-AGG	
:Is-A	OPAL:RECTANGLE	
C32::LINE1		
:X1	20	
:Y1	40	
:X2	100	
:Y2	10	
:Left	20	
:Top	10	
:Width	80	
:Height	30	
:Visible	T	
:Line-Style	OPAL:BLUE-LINE	
:Filling-Style	NIL	
:Draw-Function	:COPY	
C32::		
:Str		
:Font		
:Left		
:Top		
:Width		
:Height		
:Visible		
:Line		
:Fill		
:Act		
:Draw		
:Wind		

(a)



(b)

The spreadsheet cells show the current values of the slots. If a value changes, then the display will be immediately updated. If the user edits the value in the spreadsheet cell, the object's slot will be updated. The "F" icon by some slots in Figure 2 means that the slot value is computed from a formula. Pressing the mouse on the icon causes the constraint expression to appear in a different window. The expression itself can be edited by typing or other techniques.

Use of Inferencing

It is sometimes not convenient to read an object into a spreadsheet column just to generate a reference to it. Therefore, a command will place into the current formula a reference to any object in a Garnet window. However, selecting a graphical object does not specify which *slot* of the object should be referenced. In one mode, the user must type this directly or select a slot from a menu. However, the other mode uses heuristics to guess the slot from the example by looking at the slot being filled and where the mouse is pressed in the selected object. For example, if the slot is :left, and the mouse is pressed at the right of an object, then the reference will be to the right of the object. For the :width

Figure 2: (a) C32 viewing three objects (b). The scroll bars can be used to see more slots or columns. Changing the window's size will change the number of slots and objects displayed (the number of rows and columns). Field values are clipped if they are too long, but can be scrolled using editing commands. The "F" icon means that the slot value is computed with a formula. All inherited slots are shown in italics and marked with the "I" icon. When a formula is inherited the value is shown in a regular font since it is usually different from the prototype's. The inherited icon is also shown next to the formula icon rather than next to the value.

BRAD MYERS

slot, however, the same press would generate a reference to the width of the object. Unlike Peridot, C32 does not try to confirm any of the inferences, but rather simply inserts the text into the formula. If the guess is incorrect, it is easy for the user to delete the text and type the correction.

Once a complex formula is created, it will often be needed in a slightly different form for a different slot or a different object. As an example, suppose the user has constructed a constraint that centers an object horizontally with respect to two other objects. Now, suppose the programmer wants to center the object vertically also. The formula could be copied to the :top slot, but all the slot references need to be changed (:left to :top and :width to :height). Therefore, when a formula is copied, C32 tries to guess whether some slot names should be changed. This uses a few straightforward rules based on the slot names of the source and destination slots. Currently, these rules are hard-wired into the code. If it appears that slot names should be changed, the user is queried with a dialog box, and if the answer is OK, then the formula is modified automatically. Since this is a more radical change than the inferred slots discussed in the previous section, it seems prudent to require confirmation.

Automatic generalization

Another possibility is that the references in the formula should be *generalized* into variables. C32 therefore provides a command that will change the entire formula into a function that takes the objects and/or slots as parameters. The user can choose the names for the function and for the variables. C32 will generalize objects, slots, or both.

The intelligent copying and generalizing in C32 helps the user generate correct constraints *by example*. Without these aids, it is quite common to forget to change one or more of the references when formulas are copied. Generalizing also helps the programmer decrease the size of the code by promoting the reuse of existing formulas.

Gilt

Gilt is an interface builder that allows dialog boxes and other windows to be created interactively by choosing widgets from a palette and putting them into a window using a mouse [Myers 91d]. Gilt is similar to many other interface

GARNET

builders, including the NeXT Interface Builder, Prototyper from SmethersBarnes for the Macintosh, etc. Gilt stands for the Garnet Interface Layout Tool.

Demonstrational techniques have been added to Gilt in two places: to infer graphical styles from examples, and to infer transformations of data and dependencies to minimize the number of call-back procedures.

Graphical styles in Gilt

In most toolkits, the widgets have many properties that the designer can set, such as the color, font, label string, orientation, size, the minimum and maximum values of a range, etc. Many widgets in the Motif widget set, for example, have nearly 50 different properties that can be set. Most interface builders, including Gilt, provide "property sheets" that allow the designer to specify the desired values. However, it can be quite difficult and time consuming to find and set all of the appropriate properties. To show the magnitude of the problem, many applications contain over 2000 widgets, and the properties for each must be set in a consistent manner. A study has shown that achieving consistency in an interface is a frequently cited problem [Myers 92c].

Another problem for interface designers is *laying out the widgets in the window*. When the designer places widgets with the mouse, they tend to be uneven and look sloppy. Therefore, most builders provide grids and alignment commands. However, these can be clumsy to use, and they do not ensure that different dialog boxes will have a consistent alignment (for example, that the titles are always centered at the top of the window).

To help solve these problems, Gilt introduces the notions of *Graphical Tabs* and *Graphical Styles* into an interface builder, which are more completely described in [Hashimoto 92]. These are based on the styles and tabs in text editors such as Microsoft Word. A "graphical tab" is simply a horizontal or vertical position in the graphics window to which objects can be aligned. A "graphical style" is a named set of properties, which can be applied to widgets. The designer can edit a widget so it has the desired properties, select it, and then define a named style based on it. The values of the properties and the positions of the widgets will be associated with that style name. The style can then be applied to other widgets.

Furthermore, Gilt will try to automatically guess when to apply a style, so the designer does not have to. By guessing the appropriate properties and layout,

BRAD MYERS

Gilt makes the user interface design process significantly faster, since users can quickly and imprecisely place widgets, and the system will automatically neaten them. Since the inferencing is based on the styles the user has defined, rather than based on global, default rules, as in earlier systems like Peridot and Druid [Singh 90], the inferred properties and positions are more likely to be correct.

These features in Gilt are classified as "demonstrational" because the user defines a style by example on a particular widget, but the style is automatically generalized so it will work on any of a set of widget types.

A graphical style includes a set of widget properties, and optionally some position information as well. To create a new style, the designer modifies a widget to the desired appearance using the conventional property sheets, selects that widget, and then issues the `Define Style` command. The designer must then type a style name into the Style editing window that will appear. Gilt compares the widget's current properties with the default values for that widget and copies all that are different. Styles can also include position information. For example, a designer might specify that objects with the `Main-Title-Style` should use a large bold font, and be centered at the top of the window. The position information for styles can either be with respect to a graphical tab stop, or relative to a previously created object.

Inferring styles

Although the styles mechanism as described above is already quite useful, Gilt goes further and tries to automatically determine when a particular style is appropriate. The style control window (Figure 3) provides three options: no inferencing of styles, styles applied immediately when they are inferred, or a prompt-first mode where the designer is asked if the style should be applied, as in Peridot and Druid [Singh 90]. If the system usually infers the correct style, then the immediate mode will be the most efficient.

GARNET |

Filename: /usr/bam/cmu-styles		Read	Save	Clear
Guessing:	☒ ON	☒ Immediate		
	☐ OFF	☒ Prompt First		
Set Style	Define Style	Edit Style	Edit TabStop	
Try Again	Undo			
Style of Selected Object: Main-Title-Style				

When inferencing is on, Gilt tries to infer a new style whenever a widget is created or moved. The algorithm looks for styles that affect the same type as the widget, and, if the style has a position component, then it checks how close the widget matches the style's position. The types that styles are associated with include strings, button objects (including radio buttons and check boxes), numeric sliders (including both sliders and scroll bars), text input fields, etc. A list is created of all the styles that match, sorted from most likely to least likely.

Any inferencing system will sometimes guess wrong. Thus, it is important to provide appropriate feedback so the users are confident that they are in control and know what Gilt is doing. In immediate mode, the first style on the style list is immediately applied to the graphics, and the name of the style is shown at the bottom of the style control window (Figure 3). The widget will also jump to the inferred position and change appearance. If the inferred style is not correct, the designer can hit the "Try Again" button, which will remove the guessed style and instead apply the next style in the sorted list. This can be repeated until there are no more styles in the list. The "Undo" button can also be hit to remove the guessed style, and return the widget to its original position and properties. In prompt-first mode, the sorted list of all the inferred styles is presented in a window, with the most likely selected. The designer can select a different style, if necessary, and then hit OK or Cancel. When a style is defined, it immediately becomes a candidate for inferencing. This is very useful when a number of widgets will all be created using the same style.

Figure 3: The main style control window. This allows styles to be read and written to a file, and style guessing to be turned on and off. Also, the style of the selected object is always echoed at the bottom of the window.

BRAD MYERS***Editing styles***

When a style is applied to a widget, either explicitly or inferred, Gilt sets up appropriate pointers and back pointers so that if the style is ever edited, all widgets using that style are immediately updated.

Styles can be edited in two ways. A property sheet can be displayed which shows the current values of the properties for the style, and this can be edited directly. This property sheet has the same format as the ones for the standard widgets. The positions associated with the style can be edited using the appropriate dialog boxes.

Alternatively, the designer can edit the styles in the same way as they were created: by working on example widgets. Whenever a widget is edited that has already been defined to be of a particular style, Gilt pops up a dialog box asking if the edit should change the style itself. The other alternatives are to make the widget no longer belong to the style, or to cancel the change and return the object to its appearance before the edit was attempted.

In the future, we plan to add the ability to have objects use a particular style with exceptions, but this is a complex problem [Johnson 88]. Some of the issues are whether to copy the attributes or retain the link to the original style, what to do to a style when the style it inherits from is changed, and whether to save the inheritance links in the style files, or write out all the style information to each file.

Minimizing call-back procedures in Gilt

Conventional toolkits today require the programmer to attach *call-back* procedures to most buttons, scroll bars, menu items, and other widgets in the interface. These procedures are called by the system when the user operates the widget in order to notify the application of the user's actions. Unfortunately, real interfaces contain hundreds or thousands of widgets, and therefore many call-back procedures, most of which perform trivial tasks, resulting in a maintenance nightmare. Gilt allows the majority of these procedures to be eliminated [Myers 91d]. The user interface designer can specify by demonstration many of the desired actions and connections among the widgets, so call-backs are only needed for the most significant application actions. In addition, the call-backs that remain are completely insulated from the widgets, so that the application code is better separated from the user interface.

GARNET |

We have observed that many of the call-back procedures are actually used to filter the values from widgets and connect widgets to each other, rather than to perform real application work. By identifying some common tasks that call-backs are used for, and providing other methods for handling the tasks, we have been able to eliminate the need for most call-backs. The tasks can be classified into the following categories:

Preparing the data for applications. Often, call-backs are used to convert the values that the widgets return into a form that the application wants. This may involve converting the type of a value, for example from a string to an enumerated type, or it may involve combining the values from multiple widgets into a single record structure.

Error checking. Before the data is passed to the application, some error checking of it is often needed, along with appropriate messages when there is an error.

Preparing data to be shown to the user. Another set of procedures is usually needed to set the widgets with appropriate default values, which are often dynamically determined by the application. For example, when a color dialog box is displayed, the widgets in it will usually need to be set to the color of the currently selected object. In some cases, it may even be necessary to change the *number* of widgets in the dialog box each time it is displayed, for example, if a button is needed for each application data value.

Internal control. Many call-backs are used to control connections between user interface elements, which require little application intervention. For example, these procedures might cause a widget to be disabled (gray) when a radio button is selected, or cause one dialog box to appear when a button in another is hit.

Gilt provides a standard style of window that allows the filter expressions to be entered. The goal is to minimize the amount of code that needs to be typed to achieve the required transformation. Therefore, much of the filter expression is generated automatically when the designer *demonstrates* the desired behavior. Other parts can be entered by selecting items from menus. As a last resort, the designer can type the required code. If a call to an application function is necessary in a filter expression, Gilt makes sure that the procedure is called with ap-

BRAD MYERS

appropriate high-level parameters, rather than such things as a widget pointer or the string labels. The `Th`, `At` call-backs that remain are completely insulated from the user interface.

Gilt tries to automatically pick the appropriate transformation. There are two techniques used to guess what is appropriate. First, the designer can type an example value into the `Resulting Filtered Value` field at the bottom of the `Exported Value Control` window (Figure 4-a). In this case, Gilt will try to guess a transformation that will convert the current unfiltered value into the specified value. If none of the built-in transformations is appropriate, then Gilt creates a case statement. The designer can then operate the widget to put it into different states (and therefore to change the unfiltered value), and type the desired filtered value for each case. This allows arbitrary transformations (e.g., converting the German "Fettdruck" or the French "Gras" to `:BOLD`). The resulting code for the filter is shown in the `Filter Expression` window.

The second option is used when the designer enters a procedure into the filter expression, and then selects a widget to supply the value to a parameter of the procedure. Here, Gilt tries to find an appropriate transformation so that the widget value will be filtered into the required type of the parameter. A `Value Control` window will pop up to confirm each transformation, and also to request the designer to specify the transformation if Gilt cannot infer it.

The user can check that the filter expression is achieving the desired result in two ways. First, the interface can be exercised to test the code. Second, the `Filter Expression` field shows the Lisp code that is being used. In the future, we will be investigating other techniques for showing the transformations that will be usable by non-programmers. For example, the filter expressions might use normal arithmetic expressions, or we might create a special graphical programming language.

GARNET |

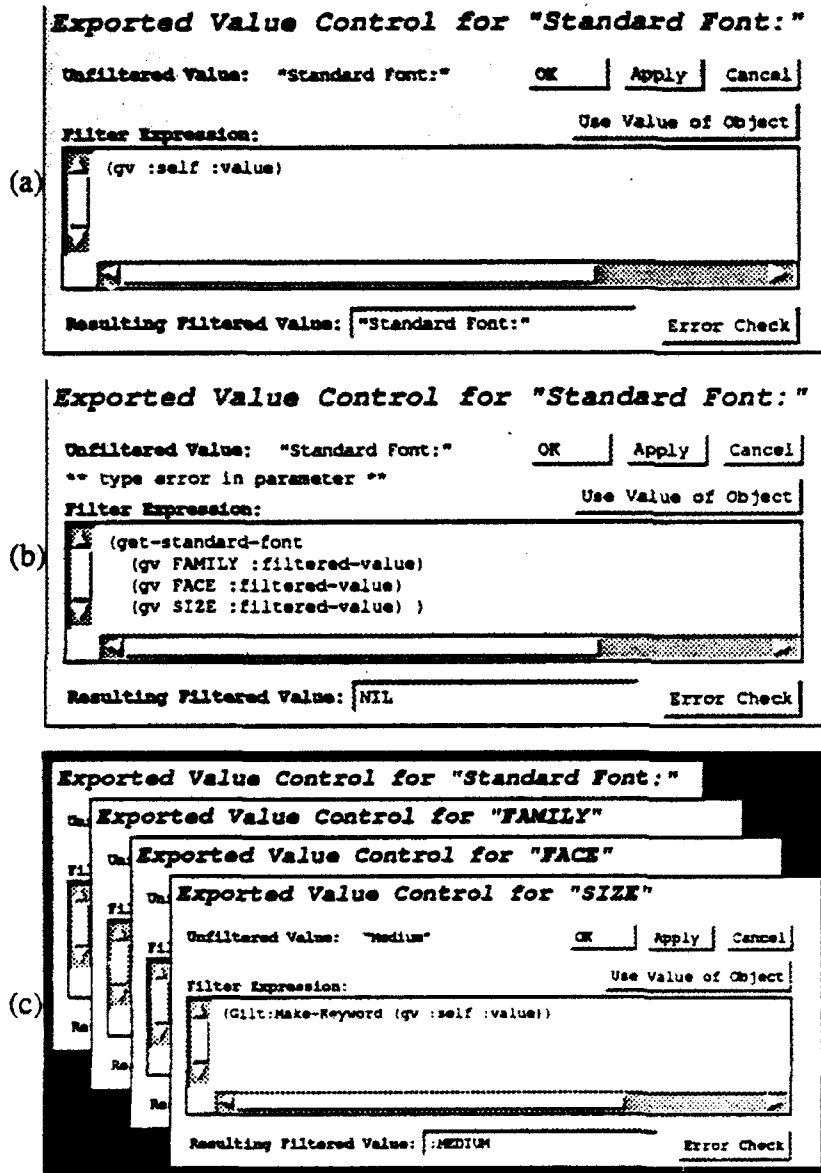


Figure 4: (a) The Gilt window that allows the designer to control how values for widgets are filtered. Many of the fields are filled in by Gilt as the designer demonstrates the desired behavior. The Unfiltered Value shows the value as currently provided by the widget before any filtering. The Filter Expression is the Lisp expression to filter the value. The designer can hit the Use Value of Object button to insert a reference to the value of a selected object. The default filter simply copies the original value. The Resulting Filtered Value field shows the final value after the filtering. This field can be edited to show the transformation for the current widget by example. (b) shows the filter expression after a function has been selected from a menu and the widget references have been filled in. (c) shows the additional windows that appear to confirm the transformations that are inferred for the widgets that are referenced in (b).

BRAD MYERS

For enabling and disabling widgets, similar techniques are used. One of the most common dependencies is to enable and disable widgets based on the values of other widgets. To specify this, the designer can operate a widget to have the appropriate value, then enable or disable the dependent widget, and Gilt will fill in the values for the `Change my Enable` expression. In trying to guess appropriate control expressions for dependent slots, Gilt knows about check boxes and radio buttons being on or off, text fields being empty or having a value, and numbers being zero or non-zero. In addition, if the `Change my Enable` window is for a set of selectable items (such as a menu or a panel of buttons), the controlling widget can return a list of values, each element of which controls an item.

All the other properties of widgets can be controlled in the same way as enabling. Widgets can be made to be visible and invisible by bringing up a `Change my Visible` window. Similar windows control properties such as color and font.

To edit the value of any of the filter expressions for a widget, the designer can simply select the widget and bring up the appropriate `Control...` or `Change my...` window. The designer can then edit the text of the expression. Alternatively, if the user demonstrates new transformations, these will replace the existing ones as appropriate.

Jade

Jade creates dialog boxes from just a list of their contents [Vander Zanden 90]. It uses general rules from graphic design as well as look-and-feel-specific rules in order to create a pleasing presentation. Jade is useful when an application contains a large number of dialog boxes (so that using Gilt would be inconvenient), or when the contents of a dialog box is not known in advance so the dialog box needs to be dynamically generated (so using Gilt would be impossible). Jade stands for the Judgment-based Automatic Dialog Editor.

The demonstrational aspect of Jade is that it will automatically generate the rules that control the layout from examples of the desired picture. An interactive editor is being created that will allow the designer to show the system how the interface should look. This part of the system is still under development.

GARNET |

Garnet contains built-in support for interfaces including *gestures*. A gestural interface uses the path that the mouse goes through in order to determine the command. For example, the user might draw an "X" over an object to cause it to be deleted, or an "L" shaped motion might mean to create a new rectangle, whereas a circular motion might mean create a new circle. The gestural mechanism in Garnet uses an algorithm that is trainable [Rubine 91a]. This means that the designer gives *examples* of the desired gestures, and the system uses statistical techniques to match the end-user's gestures against the examples to decide which gesture the end-user is giving.

Unlike Peridot, the goal in Garnet is not specifically to investigate demonstrational techniques, but rather to create a usable and efficient collection of tools to create user interfaces. However, we have found that demonstrational techniques are very effective for extending the boundaries of what can be accomplished by direct manipulation. Inferencing is used in most of our demonstrational systems, but it is not particularly sophisticated. In all cases, just one or two rules are needed to decide how and when to generalize. All the techniques are application-specific and ad-hoc, which suggests that some general-purpose toolkit containing demonstrational techniques would probably not be helpful. As with other demonstrational interfaces, the primary problems in Garnet have been how to provide appropriate feedback for the generalizations so the users are comfortable with them, and how to allow editing. In most cases, the technique used currently is just to show the Lisp code that was inferred, and require that the user directly edit the code, but we plan to investigate more sophisticated methods in the future.

The Garnet research is sponsored by the Avionics Lab, 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.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Gestures**Lessons learned****Acknowledgments**

| BRAD MYERS

Bibliography

- [Hashimoto 92] Hashimoto, O. and Myers B., "Graphical Styles For Building User Interfaces by Demonstration," *ACM Symposium on User Interface Software and Technology*, Monterey, CA, Nov. 16-18, 1992. pp. 117-124. Reprinted in this technical report.
- [Johnson 88] Johnson J. and Beach R., "Styles in Document Editing Systems," *IEEE Computer*, Vol. 21, No. 1, IEEE, January 1988, pp. 32 - 43.
- [Myers 89b] Myers B., Vander Zanden B. and Dannenberg R., "Creating Graphical Interactive Application Objects by Demonstration," *Proceedings of the Symposium on User Interface Software and Technology*, ACM SIGGRAPH, Williamsburg, November 1989, pp. 95 - 104.
- [Myers 90c] Myers B., "A New Model for Handling Input," *ACM Transactions on Information Systems*, Vol. 8, No. 3, ACM, July 1990, pp. 289 - 320.
- [Myers 90d] Myers B., Giuse D., Dannenberg R., Vander Zanden B., Kosbie D., Pervin E., Mickish A. and Marchal P., "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces," *IEEE Computer*, Vol. 23, No. 11, IEEE, November, 1990, pp. 71 - 85.
- [Myers 91a] Myers B., "Graphical Techniques in a Spreadsheet for Specifying User Interfaces," *Proceedings of CHI '91*, ACM, New Orleans, April 1991, pp. 243 - 249. Reprinted in this technical report.
- [Myers 91d] Myers B., "Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs," *Proceedings of the Symposium on User Interface Software and Technology*, ACM SIGGRAPH, Hilton Head, November 1991, pp. 211 - 220. Reprinted in this technical report.
- [Myers 92a] Myers B. and Vander Zanden B., "Environment for Rapid Creation of Interactive Design Tools," *The Visual Computer: International Journal of Computer Graphics*, Vol. 8, No. 2, February 1992, pp. 94 - 116. Reprinted in this technical report.

GARNET I

- [Myers 92b] Myers B., Giuse D., Dannenberg R., Vander Zanden B., Kosbie D., Marchal P., Pervin E., Mickish A., Landay J., McDaniel R. and Gupta V., "The Garnet Reference Manuals: Revised for Version 2.1," Technical Report CMU-CS-90-117-R3, Department of Computer Science, Carnegie Mellon University, May 1992.
- [Myers 92c] Myers B. and Rosson M., "Survey on User Interface Programming," *Proceedings of CHI '92*, ACM, Monterrey, May 1992, pp. 195 - 202.
- [Myers 92f] Myers B., Giuse D. and Vander Zanden B., "Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods," *Proceedings of the Conference on Object-Oriented Programming: Systems Languages and Applications*, October 1992. pp. 184-200.
Reprinted in this technical report.
- [Rubine 91a] Rubine D., "Specifying Gestures by Example," *Proceedings of SIGGRAPH '91*, Vol. 21, No. 4, ACM, Las Vegas, July 1991, pp. 329 - 337.
- [Singh 90] Singh G., Kok C. and Ngan T., "Druid: A System for Demonstrational Rapid User Interface Development," *Proceedings of the Symposium on User Interface Software and Technology*, ACM SIGGRAPH, Snowbird, October 1990, pp. 167 - 177.
- [Vander Zanden 90] Vander Zanden B. and Myers B., "Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces," *Proceedings of CHI '90*, ACM, Seattle, April 1990, pp. 27 - 34.
- [Vander Zanden 91a] Vander Zanden B., Myers B., Giuse D. and Szekely P., "The Importance of Pointer Variables in Constraint Models," *Proceedings of the Symposium on User Interface Software and Technology*, ACM SIGGRAPH, Hilton Head, November 1991, pp. 155 - 164. **Reprinted in this technical report.**
- [Vander Zanden 91b] Vander Zanden B. and Myers B., "Creating Graphical Interactive Application Objects by Demonstration: The Lapidary Interactive Design Tool," *SIGGRAPH '91 Video Review*, Vol. 64, No. 1, ACM, 1991.

Reprinted from *ACM Symposium on User Interface Software and Technology*
Hilton Head, SC, Nov. 11-13, 1991. pp. 211-220.

SEPARATING APPLICATION CODE FROM TOOLKITS: ELIMINATING THE SPAGHETTI OF CALL-BACKS

Brad A. Myers

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

ABSTRACT

Conventional toolkits today require the programmer to attach *call-back* procedures to most buttons, scroll bars, menu items, and other widgets in the interface. These procedures are called by the system when the user operates the widget in order to notify the application of the user's actions. Unfortunately, real interfaces contain hundreds or thousands of widgets, and therefore many call-back procedures, most of which perform trivial tasks, resulting in a maintenance nightmare. This paper describes a system that allows the majority of these procedures to be eliminated. The user interface designer can specify by demonstration many of the desired actions and connections among the widgets, so call-backs are only needed for the most significant application actions. In addition, the call-backs that remain are completely insulated from the widgets, so that the application code is better separated from the user interface.

KEYWORDS: Call-Back Procedures, Dialog Boxes, UIMSs, Interface Builders.

1. Introduction

The Gilt Interface Builder allows dialog boxes and similar user interface windows to be created by selecting widgets from a palette and laying them out using a mouse. More interestingly, Gilt provides a variety of mechanisms to reduce the number of *call-back* procedures that are necessary in graphical interfaces. A "call-back" is a procedure defined by the application programmer that is called when a *widget* is operated by the end user. A "widget" is an interaction technique such as a menu, button or scroll-bar. A collection of widgets is called a *toolkit*. Examples of toolkits are the Macintosh Toolbox, the Motif and Open-Look toolkits for X windows, and NeXTStep. Most

toolkits today require the programmer to specify call-backs for almost every widget in the interface, and some widgets even take more than one call-back. For example, the slider widget in Motif has two call-backs, one for when the indicator is dragged and one for when it is released.

A typical user interface for a moderately complex program will contain hundreds or even thousands of widgets. For example, the VUIT program from DEC uses over 2500 widgets. This means that the programmer must provide many call-back procedures. To add to the complexity, each type of widget may have its own protocol for what parameters are passed to the call-back procedures, and how the procedures access data from the widget.

The use of all of these call-backs means that the user interface code and the application code are not well separated or modularized. In particular:

- The call-backs closely tie the application code to a particular toolkit. Since each toolkit has its own protocol for how the call-backs are called, moving an application from one toolkit to another (e.g., from Motif to Open-Look) can require recoding hundreds of procedures.
- The call-backs make maintaining and changing the user interface very difficult. Changing even a small part of an interface often requires rewriting many procedures. Even if a graphical interface builder is used to change the widgets, the call-backs must be hand-edited afterwards if widgets are added, deleted, or modified.
- The call-backs often are passed the text labels shown to the user, so if the natural language used in the dialog box is changed (e.g., from English to French), the values passed to the call-backs will change, requiring the application code to be edited.

We have observed that many of the call-back procedures are actually used to filter the values from widgets and connect widgets to each other, rather than to perform real application work. By identifying some common tasks that call-backs are used for, and providing other methods for handling the tasks, we have been able to eliminate the need for most call-backs. The tasks can be classified into the following categories:

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1991 ACM 0-89791-451-1/91/0010/0211...\$1.50

Preparing the data for applications. Often, call-backs are used to convert the values that the widgets return into a form that the application wants. This may involve converting the type of a value, for example from a string to an enumerated type, or it may involve combining the values from multiple widgets into a single record structure.

Error checking. Before the data is passed to the application, some error checking of it is often needed, along with appropriate messages when there is an error.

Preparing data to be shown to the user. Another set of procedures is usually needed to set the widgets with appropriate default values, which are often dynamically determined by the application. For example, when a color dialog box is displayed, the widgets in it will usually need to be set to the color of the currently selected object. In some cases, it may even be necessary to change the *number* of widgets in the dialog box each time it is displayed, for example, if a button is needed for each application data value.

Internal control. Many call-backs are used to control connections between user interface elements, which require little application intervention. For example, these procedures might cause a widget to be disabled (grey) when a radio button is selected, or cause one dialog box to appear when a button in another is hit.

Gilt provides a standard style of window that allows the filter expressions to be entered. The goal is to minimize the amount of code that needs to be typed to achieve the required transformation. Therefore, much of the filter expression is generated automatically when the designer *demonstrates* the desired behavior. Other parts can be entered by selecting items from menus. As a last resort, the designer can type the required code. If a call to an application function is necessary in a filter expression, Gilt makes sure that the procedure is called with appropriate high-level parameters, rather than such things as a widget pointer or the string labels. Thus, the call-backs that remain are completely insulated from the user interface.

Gilt is a part of the Garnet system [8]. Garnet is a comprehensive user interface development environment containing many high-level tools, including Gilt, the Lapidary interactive design tool [7], the C32 spreadsheet system [9], etc. Garnet also contains a complete toolkit, which uses constraints [15] and a prototype-instance object model. Gilt stands for the Garnet Interface Layout Tool, and it supports interfaces built using either the Garnet look-and-feel widget set or the Motif look-and-feel widget set.¹ Gilt uses CommonLisp, but the ideas presented here are applicable to interface builder tools using conventional compiled languages.

¹The Motif-style widgets in Garnet are implemented on top of the Garnet Toolkit intrinsics and do not use any of the Xtk code in C. Although they look and behave like the standard Motif widgets, they have the same procedural interface as the Garnet widget set.

2. Related Work

Of course, there are a large number of commercial and research interface builders that lay out widgets, including DialogEditor [3], the Prototyper for the Macintosh [13], the NeXT Interface Builder, UIMX for Motif, and Druid [12]. However, these only have limited mechanisms for reducing call-backs. Many of them support transitions from one dialog box to another, and NeXT allows the output value of one widget to be connected to the input of another, if no filtering is needed. Druid adds the ability to set the initial values for widgets (but only statically, not application data dependent), and to collect values of widgets for use as the parameter to a procedure. It allows the designer to specify some of these by demonstration. However, in Gilt, significantly more of the user interface can be specified without requiring call-backs, the call-backs are more independent of the widgets, and a uniform framework is used for all filtering.

A primary influence on Gilt is the Peridot UIMS [6]. Peridot was the first system to allow the designer to specify the behavior of the interface by demonstration. Gilt uses some of the techniques in Peridot to guess the appropriate transformations based on the example values.

The filter expressions that the designer specifies in Gilt are implemented using *constraints*. A constraint is a relationship that is declared once and then maintained by the system. Constraints have been used by many systems, starting with Sketchpad [14] and Thinglab [Thinglab/Toplas]. Uses of constraints within user interface toolkits include GROW [1], Peridot [6], and Apogee [5].

Other systems have allowed the designer to specify the connections between the user interface and the application procedures at a high level. The Mickey system [11] uses special comments in the procedure definition to describe the connection to the user interface. The UIDE system [4] allows the application procedures to be defined in advance, and generates the interface partially from these. Unlike these systems, Gilt requires the designer to specify the graphics, and then explicitly attach the graphics to the procedures, but it infers the mapping between the values returned by the widgets and the values desired by the procedures.

3. Example

To show how easy it is to define dependencies without writing call-backs, we will first present an example of creating the dialog box of Figure 1. There are a number of dependencies in this relatively simple interface. The return value of the dialog box is a font object. If one of the standard fonts is selected, then the corresponding built-in font object should be returned. Otherwise, the return value will be a font specified by name, so the specified file should be opened, and a new font object created for that file.

First, the user would create the graphics for the dialog box by selecting the widgets from a palette and typing in the correct labels, in the conventional direct manipulation man-

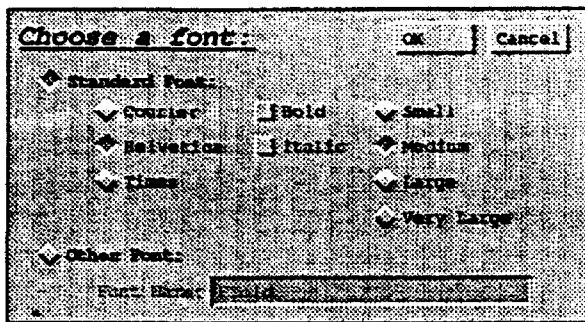


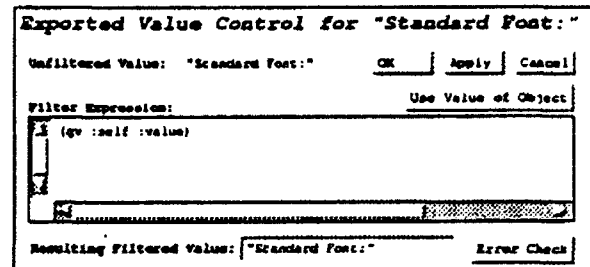
Figure 1:

A font selection dialog box being created in Gilt. When the Standard Font radio button is pressed, the Font Name field is disabled (grey), and when the Other Font radio button is selected, the three sets of buttons under Standard Font (for the family, face and size of the font) become disabled.

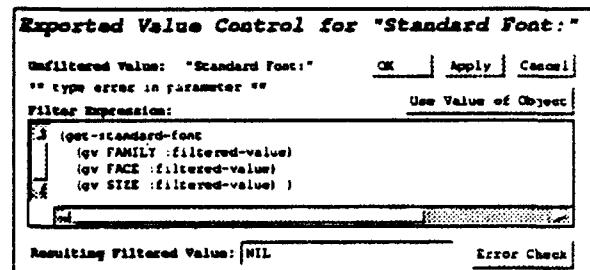
ner as with other interface builders. Next, the designer gives meaningful names to the widgets (e.g., the Bold/Italic radio buttons are called "face").

The default value of a set of radio buttons is the string label of the selected button. We will now override this and make the value of the Standard Font branch instead be the appropriate font object. To do this, we bring up the Gilt window in Figure 2-a. When it is brought up, it initially shows that the resulting exported value from the widget is the same as the widget's value. To get the appropriate font object, we need to call the Gilt function `get-standard-font`, so we choose this from a menu. This inserts the function call into the filter expression. The procedure should be passed the values of the three sets of widgets under Standard Font. Therefore, we select the three widget sets and hit the Use-Value-of-Object button in the window. This inserts references to all the selected objects into the filter expression, resulting in Figure 2-b. The references are inserted in the order the objects were selected. These references will be to the *filtered* values of the widgets, which so far are the same as the default values: the string names of the labels. However, Gilt knows that `get-standard-font` expects Lisp keywords as arguments rather than strings (a "keyword" is an atom prefixed by a colon, such as `:bold`). Therefore, Gilt can tell that there is a mismatch, so it tries to determine a possible transformation. Another Exported Value Control window pops up for each of the selected widgets, and the designer can check that the inferred transformations are correct (Figure 2-c). If not, the designer can give additional examples or explicitly edit the generated code. In this example, however, the system guesses all cases correctly, so the designer simply hits "OK" on all of the windows. This will assert constraints so that the filtered values of the widgets will be keywords, as required.

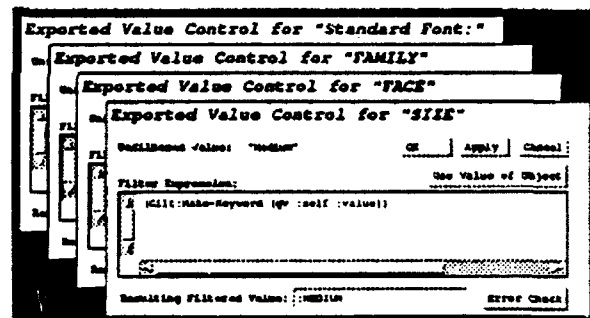
Now, the value for the other branch must be set. The designer selects the Other Font radio button and brings up the Exported Value Control window for it. By selecting the `get-font-from-file` function from a



(a)



(b)



(c)

Figure 2:

(a) The Gilt window that allows the designer to control how values for widgets are filtered. Many of the fields are filled in by Gilt as the designer demonstrates the desired behavior. The Unfiltered Value shows the value as currently provided by the widget before any filtering. The Filter Expression is the Lisp expression to filter the value. The designer can hit the Use Value of Object button to insert a reference to the value of a selected object. The default filter simply copies the original value. The Resulting Filtered Value field shows the final value after the filtering. This field can be edited to show the transformation for the current widget by example. (b) shows the filter expression after a function has been selected from a menu and the widget references have been filled in. (c) shows the additional windows that appear to confirm the transformations that are inferred for the widgets that are referenced in (b).

Figure 3:

This window allows the designer to specify the handling of error values. When Other Font's filtered value is NIL, the first error string is printed, and when Other Font is the special value :NOT-FONT, the second string is printed. The Use Value of Object button is used to insert a reference to a selected object, here, the value of the Font Name widget, which contains the current file name. The Another Error Check button causes another If value is and Error String pair to appear.

Figure 4:

The Gilt window that allows the designer to specify that the enable property of a widget depends on other widgets. When the Expression returns NIL, the widget is shown "greyed-out."

menu, then selecting the Font Name widget, and finally hitting the Use Value of Object button, the designer can specify the appropriate dependencies. Since get-font-from-file expects a string, no further transformations are needed. If the font is not found, the get-font-from-file func. on returns error values, so the Error Check window is used to specify the handling of this (Figure 3). The designer types the appropriate error return values and response strings into this window.

Next, the value of the entire dialog box is specified as the value of the pair of radio-buttons Standard Font and Other Font, and now the dialog box will return a single value, computed based on the settings of the widgets.

Finally, the designer needs to specify when the various widgets should be disabled (greyed out). First, the designer selects the Font Name text field, and then brings up the Change my Enable window (see Figure 4). Note that this window has the same general form as the Value Control window, but simply controls a different property of the widgets (the enable flag rather than the value). Next, the

designer selects the Other Font radio button and hits the Use Value of Object button. This makes the Font Name enabled (not grey) when Other Font is chosen. Similarly, the family, face and size buttons under Standard Font are enabled when Standard Font is selected.

4. Filtering

Each widget in Garnet will always first compute its default value, which is then assigned to the widget's slot (instance variable) called :VALUE.² This value can then be filtered to derive the value seen by application programs, which is set into the slot called :FILTERED-VALUE. This is implemented as a constraint that sets the value of the :FILTERED-VALUE slot whenever the value of the :VALUE slot changes. The default constraint simply copies the value. Experience has shown that most filter expressions are rather short, often only one or two lines. Sometimes, it will be necessary to have longer, complex transformations or access to application-specific functionality and data. Here, a conventional text editor would be used to create a function which will then be called by the filter expression entered with Gilt. However, the function will be independent of the particular widgets used since Gilt provides transformations of the arguments and return values from the function.

As was shown in the example, Gilt provides a number of ways to specify the appropriate filtering of data and control in the user interface, so the application code is independent of the particular widgets used and the label strings shown to the user. All of these transformations use the same, standard Control windows shown in the previous examples. The following sections show how the various tasks that require call-backs in other toolkits are performed in Gilt.

4.1 Preparing Data for Applications

Many call-backs in widgets simply filter the output value to convert it to a form needed by the application program. For example, for Figure 1, you might need as many as 13 different call-backs in other toolkits to generate the single font value to be returned. In Gilt, the value of the dialog box is available in a variable, without requiring a call-back.

Unlike most toolkits, Garnet provides values for *groups* of widgets. For example, the default value of a radio button set is the name of the radio button that is selected, or NIL if none are. For a set of check boxes (that allows multiple selections), the value is a list of the selected buttons. The innovation in Gilt is that the designer can specify alternative values for widgets. In the example, the value of the pair of radio buttons Standard Font/Other Font will be a font object.

Many of the desired transformations of the values can be achieved by simple type conversions: from strings to keywords, atoms, numbers, etc. Therefore, Gilt provides a

²All slot names in Garnet start with a colon.

number of built-in data transformations:

- String to Lisp atom (e.g. "Bold" to 'BOLD).
- String to Lisp keyword (e.g. "Bold" to :BOLD).
- String to index of item in the set of buttons (e.g. "Bold" to 0).
- String to number (e.g. "10" to 10).
- Integer range to a different integer or float range.

Similar transformations would be appropriate for a builder generating other computer languages, like C or Pascal, which might automatically create enumerated types, sets, bit vectors, or named constants.

Gilt tries to automatically pick the appropriate transformation. There are two techniques used to guess what is appropriate.

First, the designer can type an example value into the Resulting Filtered Value field at the bottom of the Exported Value Control window (Figure 2-a). In this case, Gilt will try to guess a transformation that will convert the current unfiltered value into the specified value, using the above rules. If none of the built-in transformations is appropriate, then Gilt creates a case statement. The designer can then operate the widget to put it into different states (and therefore to change the unfiltered value), and type the desired filtered value for each case. This allows arbitrary transformations (e.g., converting the German "Fettdruck" or the French "Gras" to :BOLD). The resulting code for the filter is shown in the Filter Expression window.

The second option is used when the designer enters a procedure into the filter expression, and then selects a widget to supply the value to a parameter of the procedure. Here, Gilt tries to find an appropriate transformation so that the widget value will be filtered into the required type of the parameter. This is the technique used in the example. A Value Control window will pop up to confirm each transformation, and also to request the designer to specify the transformation if Gilt cannot infer it.

A number of standard procedures are provided in a pop-up menu, so the designer can often select a procedure for the filter expression rather than typing it. The provided routines will transform a string into a file pointer, a string into a font pointer, numbers or a string into a color, keywords into a font, etc. If one of these is selected from the menu, the appropriate code is entered into the Filter Expression field. Because these routines take abstract values as parameters, and return a value of the appropriate type (such as a font object), the implementation of the routines is entirely independent of the widgets. In fact, standard, built-in routines, such as the Lisp function probe-file, can be used in many cases.

Gilt can execute the filter expressions, including any procedures entered by the designer, by using the Lisp interpreter. Therefore, when Gilt is put in "run-mode" the actions will happen just as they will for the end user. Gilt first checks to make sure that all procedures are defined, in case the designer has entered an application-specific procedure that is not implemented yet. In this situation, Gilt

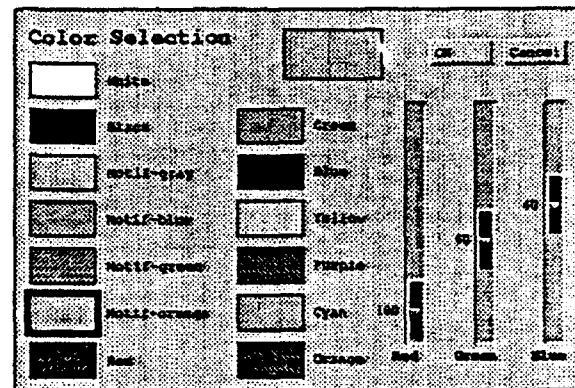


Figure 5:

The color selection dialog box created using Gilt (naturally, this is in color on the screen). There are a number of dependencies among the widgets that were defined by demonstration. If one of the color buttons on the left is selected, the sliders adjust to the appropriate position for that color. If the sliders are moved, the highlight in the color buttons (here shown around Motif-orange), goes to the appropriate color or goes off. The rectangle in the upper center always shows the current color. The filtered value of the rectangle is its color, and the value of the dialog box is defined as the filtered value of the rectangle.

requests the designer to give an example of the value the function would return.

Sometimes, the value of a widget might be computed based on the values of *multiple* other widgets. In the example of section 3, the value of the Standard Font radio button is computed based on the values of three sets of buttons. The default expression creates a list out of the values, but by editing the filter expression, it is easy to create a record or structure instead of a list, or to process the values in various ways. In Figure 2-b, the get-standard-font routine is called on the values of three widgets to return a single font object.

Gilt allows decorations to be added to the dialog boxes, such as rectangles, lines and labels. These normally do not have a value, but they can be given one using a Value Control window. For example, the rectangle at the upper center of Figure 5 shows the current selected color. The value of this rectangle should be its color. To achieve this, the designer can type (gv :self :COLOR) into the Filter Expression field.³ To make this a little easier, the designer can choose the desired field of the selected object from a pop-up menu.

The user can check that the filter expression is achieving the desired result in two ways. First, the interface can be exercised to test the code. Second, the Filter Expression field shows the Lisp code that is being used.

³gv stands for "get value" and it looks in the specified object for the specified slot.

In the future, we will be investigating other techniques for showing the transformations that will be usable by non-programmers. For example, the filter expressions might use normal arithmetic expressions, or we might create a special graphical programming language.

4.2 Error Handling

Call-back procedures in other toolkits are often used to check for error values, especially in text input fields. Gilt provides a standard error-filtering mechanism that minimizes the connections between the error checking code and the widgets. The designer can bring up the Error Check window (Figure 3), and type a value into the if-value-is field. If the filtered value for the widget is ever equal to the if-value-is value, then an error has occurred. If the Error String field contains a string, then a error dialog box is popped-up showing that string. The string can embed references to other widgets using the Use-Value-of-Object button, for example, to show the incorrect value. Alternatively, if the Error String field contains an expression or function call, then it is executed.

Alternatively, an expression using the value of the widget can be entered into the if-value-is field, which should return T if an error should be reported. For example, to report an error if an input number is odd, the designer could simply enter `(oddp (gv :self :FILTERED-VALUE))`. If the filter expression itself returns an error message string, then the if-value-is might just test if the filtered value is a string, and the Error String would just be `(gv :self :FILTERED-VALUE)`.

There can be multiple if-value-is and Error String pairs, which would be useful, for example, for a font finding routine that returned different values to tell if the file was not found, or if the file was not a valid font, as in Figure 3. The get-font-from-file filter will return a font, or NIL if the file is not found, or :NOT-FONT if the file is found, but it is not a font.

4.3 Preparing Data to be Shown to the User

Most toolkits require that the designer create additional procedures to set the widgets based on application-specific data. For example, when many dialog boxes are made visible, the values of some widgets should be set to a particular value. If a widget should *always* have the same value when the dialog box appears, then the designer can simply supply this value by example, as in other interface builders like Druid [12]. However, it is very common for the initial values for widgets to depend on application-supplied data. For example, when the font dialog box is made visible by an application, it should reflect the font of the selected object, or if there is no object selected, then the current global default. The next sections discuss how Gilt allows this to be specified easily.

4.3.1 Defining Parameters to the Dialog Box

When a window is designed in Gilt, parameters to the window can be specified, along with an example current value

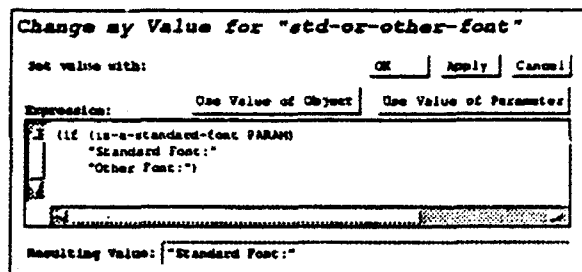


Figure 6:

The Gilt window to cause the displayed value of a widget to change based on other widgets. Here, the Standard Font/Other Font radio buttons of Figure 1 are set based on the value of the parameter. The designer only had to select the is-a-standard-font procedure from a menu, the rest of the expression was entered by Gilt as the widgets in Figure 1 were operated.

for the parameter. If an application wants to display a window designed in Gilt, it can simply call⁴

```
(Show-Dialog dialog-name param1 param2 ...)
```

For example, the font dialog box of Figure 1 would take a single font object as a parameter. Thus, the application causes the dialog box to appear while still being independent of how the parameters are used to set the widgets.

For "modal" dialog boxes (that require the user to say OK or CANCEL before doing other operations), the Show-Dialog routine will return the value of the dialog box. The designer can specify the value of the dialog box using a Value Control window, as was shown in the example. For non-modal dialog boxes, Show-Dialog will return immediately, and the designer can attach a call-back procedure to the OK button. Of course, this call-back will be passed the filtered value of the window, so it will be independent of the widgets that are used in the window to enter the value.

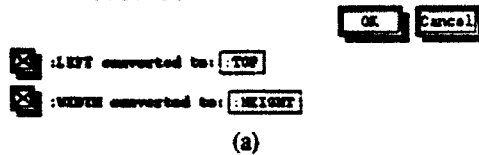
4.3.2 Using the Parameters

To set the value of a widget based on the parameters, the designer uses the Change my Value window (see Figure 6). The primary difference from the Value Control window shown earlier is that here we are changing the value shown to the user, rather than simply filtering the value returned by the widget. However, this window is very similar to the Value Control window, and the interface to the designer is essentially the same.

The result of the expression should be an appropriate value for the widget. For example, Figure 6 calculates the string

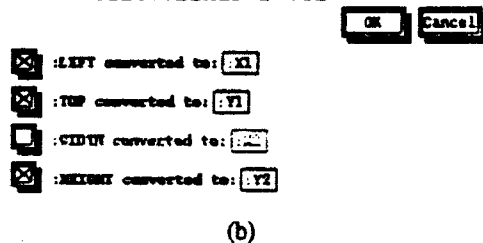
⁴In a language that does not support functions with a variable number of arguments, a Gilt-like builder could create a different `show-<dialog-name>` routine for each window designed.

Copying C32::Line1's :X1 to
C32::Line1's :Y1



(a)

Copying User::R1's :Left to
Gilt::Line9's :Y1



(b)

Figure 7:

This dialog box (which uses the Garnet widget set instead of the Motif widget set used by the other figures in this paper), repeats the check box, the label and the text type-in field. The controlling expression for (a) might be ((T :LEFT :TOP) (T :WIDTH :HEIGHT)), where the T controls the radio button, the second element is used in the label, and the third is used as the default for the text input field. The user can then turn on and off the desired slots using the check boxes, or type a new name.

name of the branch of the radio button to be selected. Of course, designers can simply type in the appropriate code, but Gilt provides demonstrational techniques to make this easier. The designer can operate the widgets to put them into the appropriate state, and then give the expression that will determine when that state is to be used. For example, for the font dialog box, the designer could select the Standard Font/Other Font widget, and bring up a Change my Value window (Figure 6). Then, the Standard Font radio button would be pressed, and the designer could hit the Use Value of Parameter button. Then, the designer would have to edit the expression to return T when the font was a standard font using the is-a-standard-font procedure. By default, the other value of the radio buttons will be used otherwise, so nothing is needed for that case. Next, the designer would bring up Change my Value windows for the other widgets, such as Font Name, and write expressions to extract the appropriate information from the font object parameter.

4.3.3 Dynamic Creation of Widgets

Sometimes, a parameter might specify the *number* of widgets that need to be created. In this case, the designer can show by example the set of widgets to be replicated, select them, and bring up a Replicate Control window, which is similar to the Change my Value win-

dow. The expression in this window is expected to return an integer to tell how many copies of the widgets are desired. Alternatively, the expression can return a list of values, in which case, the number of copies depends on the length of the list. Here, each copy is assigned the appropriate element from the list. For example, in Figure 7, the application might supply as a parameter to the dialog box a list of slot names to control how many times the check box, the label and the text input field are repeated.

4.4 Internal Control

In other toolkits, another set of call-back procedures are often needed to control the setting of the value or other property of one widget based on the value of another, or to bring up a new dialog box when a button is pressed. The next sections discuss how Gilt allows these to be specified using filter expressions.

4.4.1 Value Dependencies

Sometimes, when a widget is operated, this should cause a different widget to change its value. For example, when the user hits on a color button in Figure 5, the sliders should move to show the appropriate values for that color. Gilt provides a convenient mechanism for specifying this using the same Change my Value window as for having a widget's value depend on a parameter (Figure 6).

The designer selects the widget that should change (for example, the red slider of Figure 5), and brings up a Change my Value window. Next, the widget that it should depend on (here, the color button set) is selected, and the Use-Value-of-Object button is hit. This will generate the expression

```
(gv Color-buttons :FILTERED-VALUE)
```

but for the red slider, only the red component of the color should be used, so the designer would edit the expression to be

```
(gv Color-buttons :FILTERED-VALUE :RED)
```

Now, whenever the color buttons are operated, the red slider will be set correctly. The other two sliders would be fixed similarly.

Sometimes, widgets may need to be replicated based on the value of another widget. In the Xerox Star and Viewpoint, menus only show legal values, rather than greying out illegal values. For example, in a font-choice dialog box, if different fonts have different sizes available, the components in the menu of sizes must be dynamically changed. The Replicate Control window discussed in section 4.3.3 is used to control this.

4.4.2 Specifying Other Dependencies

The previous sections discussed how the *value* of a widget can be controlled. In many cases, however, other properties of widgets may need to be set, such as whether it is enabled or not (greyed-out). This is handled in a uniform way, using a window similar to the Change my Value window. The designer selects the widget to be controlled, specifies the desired property from a menu, and the appropriate window is brought up.

4.4.2.1 Enabling

One of the most common dependencies is to enable widgets based on other widgets. As shown in the example of section 3, the designer can operate a widget to have the appropriate value, then enable or disable the dependent widget, and Gilt will fill in the values for the Change my Enable (Figure 4). In trying to guess appropriate control expressions for dependent slots, Gilt knows about check boxes and radio buttons being on or off, text fields being empty or having a value, and numbers being zero or non-zero. In addition, if the Change my Enable window is for a set of selectable items (such as a menu or a panel of buttons), the controlling widget can return a list of values, each element of which controls an item. For example, in Figure 8, the menu of font sizes will have a Change My Enable expression that computes the list of valid font sizes based on the selected font in the left menu. Although an application function is needed, the function will be independent of the particular widgets used, since it will take a font object and return a list of valid sizes. Gilt will automatically create an expression to enable the items that correspond to the values in the list and disable the others.

4.4.2.2 Other Properties

All the other properties of widgets can be controlled in the same way as enabling. Widgets can be made to be visible and invisible by bringing up a Change my Visible window. Most widgets also have additional properties which can be set, such as their color or font. To change the color of an object, the Change my Color window is used. For example, to change the color of the red slider based on the value it returns, the designer could simply select the red slider, bring up the Change my Color window, select the slider again, hit the Use-Value-of-Object button, and then edit the expression to be⁵

```
(Make-Color (gv :self :FILTERED-VALUE)
  0 0)
```

Using the dependency control on various properties is also useful for decorations such as rectangles and labels. For example, the color of the rectangle in the center of Figure 5 can be made to depend on the three sliders in this way.

4.4.3 Sequencing of Dialogs

Another common internal control action that sometimes requires call-backs is for a button to cause another dialog box to appear. Gilt, like other interface builders, allows this to be demonstrated, by simply operating the button, and showing which dialog box should appear. However, unlike other systems, Gilt also allows the initial values of widgets in the sub-dialog to be set. Windows similar to the Change my ... windows appear that allow the values of the parameters to the sub-dialog to be specified based on the values of the parent dialog box. Gilt will automatically create the code to call Show-Dialog in the appropriate way. If the sub-dialog is modal (which is the usual case), then the value of the sub-dialog is assigned by default as

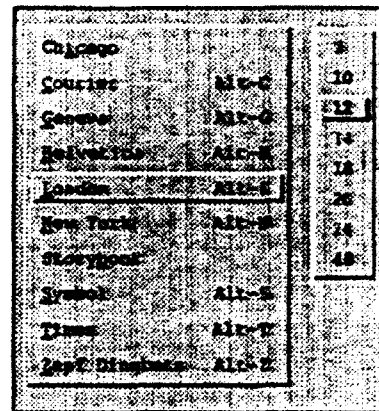


Figure 8:

In these Motif-style menus, the various font sizes in the menu on the right become enabled or disabled depending on the sizes available for the font that is selected in the menu on the left.

the value of the button that caused the sub-dialog to be displayed. Of course, the designer can control this using the Value Control for the button.

If the sub-dialog is not modal, then the end user will be allowed to operate widgets in both windows. Gilt supports cross-window dependencies, so that a value in one dialog box can depend on a value in another dialog box.

5. Editing and Saving

To edit the value of any of the filter expressions for a widget, the designer can simply select the widget and bring up the appropriate Control... or Change my... window. The designer can then edit the text of the expression. Alternatively, if the user demonstrates new transformations, these will replace the existing ones as appropriate.

Gilt provides a special feature to make it easier to convert an interface to a different natural language. After a value transformation has been specified, the next time the designer edits the displayed label names, Gilt will pop-up a question to ask if the corresponding exported values should change also. If the designer says "no", then the value filter function is automatically changed so that all the new label strings will still produce the same old values, so any code that uses the values will not need to be changed.

Other special features make editing the widgets easier. Gilt provides a "Replace widget" command, which allows, for example, a set of buttons to be replaced by a menu. As many of the properties as possible are retained, including the label names and the filter expressions. In addition, the filter expressions can be copied from one widget to another. Finally, because the more complex filter procedures and application-specific call-backs are called with abstract parameters (such as keywords), they usually will not need to be changed when the widgets are edited. We will be investigating other techniques for editing in the future.

⁵The slider's value is a number, but we need a color object for the color property. Make-Color is a standard routine that takes numbers representing the red, green and blue values and returns a color object.

As was mentioned previously, the expressions are implemented as constraints attached to the appropriate properties of objects. Garnet has a built-in mechanism for saving any object as a Lisp code file, including all of its constraints [10], and this is used by Gilt. Therefore, all the filter expressions are output automatically along with the user interface definition. Since the output is textual Lisp code, it is possible for programmers to edit the file directly, but we expect this to not be necessary.

6. New Kinds of Widgets

The techniques that have been described are not limited to only the built-in widgets in the Garnet toolkits. If the user wants a new kind of widget, then it can be created either by coding it by hand or using the Lapidary design tool [7]. The new widget can then be dynamically loaded into the Gilt palette, and used like any built-in widget.

All of the widgets in the Garnet toolkit are controlled through the same protocol, which includes a specification of what the properties of the widget are and the types of the properties (string, boolean, integer, list, etc.). This allows the appropriate Control windows to be created. For custom widgets, the designer will need to conform to the standard protocol. Lapidary has built-in mechanisms to help with this for widgets created using it. The inferring of the filter expressions is based on the type of the properties, so the demonstrational techniques described in this paper can be used for designer-created widgets as well. As an example, the color selection buttons on the left of Figure 5 are not a standard widget, but were partially coded by hand and then read into Gilt for the dependencies to be specified.

Another interesting feature is that a set of widgets can be saved, along with their interdependencies defined in Gilt, and used as a prototype in other interfaces. For example, the Standard Font group from Figure 1 could be read into the Gilt palette, and then placed in other dialog boxes. Due to the prototype-instance object model in Garnet, no extra mechanisms are needed in Gilt to support this.

7. Status and Future Work

An earlier version of Gilt has been released to all Garnet users.⁶ The version described here has been mostly implemented, and is expected to be finished and released in the next few months.

In the future, in addition to releasing this version of Gilt for general use, we would like to investigate combining some of the features of Lapidary with Gilt, so that the designer can specify constraints on the widgets, for example to make decorations or the entire window grow if a widget gets bigger. It has been suggested that a wiring diagram approach to specifying the interdependencies among widgets might be easier to use. We will investigate allowing the designer to draw wires among the widgets to show the flow

of values and enabling. This might also be helpful as a debugging tool to show where the dependencies are. Other debugging and maintenance aids will also be added, such as browsers to show all the filter expressions, and the procedures and global variables used in them. Finally, we will add some of the demonstrational techniques from Peridot and Druid that neaten the display as widgets are drawn.

8. Conclusion

The Gilt interface builder contains a number of innovations that significantly improve the separation of application code from toolkits. By identifying the most common tasks that call-backs are used for, Gilt is able to supply built-in mechanisms to handle them. Using a standard style of window, the designer can enter short filter expressions. Because many of the tasks involve straightforward filtering, Gilt can often infer appropriate transformations from examples of the desired output or actions. Even when more complex transformations are required, and which use application-specific procedures, the application code is completely independent of the actual widgets and the names used in the user interface. Although Gilt is implemented in Lisp, which makes the dynamic execution of the entered code much easier, the general techniques are appropriate for conventional compiled languages and for interface builders for conventional toolkits. Therefore, the techniques could be readily applied to today's user interface tools.

The mechanisms that are described here make it much faster to build dialog boxes with interdependencies among the widgets. However, we expect their main advantage to be the improved maintainability of the resulting code. For example, it should be much easier with Gilt than most other interface builders to convert a user interface to a different natural language or switch between different forms of widgets (e.g., from menus to buttons), or even different widget sets (e.g., from Motif to OpenLook). We will be exploring the effects of these features as Gilt becomes widely used by the Garnet community.

Acknowledgements

Andrew Mickish implemented the features described in this article. Osamu Hashimoto also contributed to the design and implementation of Gilt. Brad Vander Zanden, David Kosbie, Andrew Mickish, Osamu Hashimoto, Bernita Myers, and the referees provided useful comments on this paper.

This research was sponsored by the Avionics Lab, 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.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

⁶The Garnet system is available for free from CMU, but you need to have a license. If you are interested in using Gilt and Garnet, please contact the author or send electronic mail to garnet@cs.cmu.edu.

References

1. Paul Barth. "An Object-Oriented Approach to Graphical Interfaces". *ACM Transactions on Graphics* 5, 2 (April 1986), 142-172.
2. Alan Borning. "The Programming Language Aspects of Thinglab; a Constraint-Oriented Simulation Laboratory". *ACM Transactions on Programming Languages and Systems* 3, 4 (Oct. 1981), 353-387.
3. Luca Cardelli. Building User Interfaces by Direct Manipulation. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'88, Banff, Alberta, Canada, Oct., 1988, pp. 152-166.
4. James D. Foley, Christina Gibbs, Won Chul Kim, and Srdjan Kovacevic. A Knowledge-Based User Interface Management System. Human Factors in Computing Systems, Proceedings SIGCHI'88, Washington, D.C., May, 1988, pp. 67-72.
5. Tyson R. Henry and Scott E. Hudson. Using Active Data in a UIMS. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'88, Banff, Alberta, Canada, Oct., 1988, pp. 167-178.
6. Brad A. Myers. *Creating User Interfaces by Demonstration*. Academic Press, Boston, 1988.
7. 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.
8. Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. "Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment". *IEEE Computer* 23, 11 (Nov. 1990), 71-85.
9. Brad A. Myers. Graphical Techniques in a Spreadsheet for Specifying User Interfaces. Human Factors in Computing Systems, Proceedings SIGCHI'91, New Orleans, LA, April, 1991, pp. 243-249.
10. Brad A. Myers and Brad Vander Zanden. "An Environment for Rapid Creation of Interactive Design Tools". *The Visual Computer: International Journal of Computer Graphics* (1991), to appear.
11. Dan R. Olsen, Jr. A Programming Language Basis for User Interface Management. Human Factors in Computing Systems, Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 171-176.
12. Gurinder Singh, Chun Hong Kok, and Teng Ye Ngan. Druid: A System for Demonstrational Rapid User Interface Development. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'90, Snowbird, Utah, Oct., 1990, pp. 167-177.
13. SmethersBarnes, P.O. Box 639, Portland, Oregon 97207, Phone (503) 274-7179. Prototyper 3.0.
14. Ivan E. Sutherland. SketchPad: A Man-Machine Graphical Communication System. AFIPS Spring Joint Computer Conference, 1963, pp. 329-346.
15. Brad Vander Zanden, Brad A. Myers, Dario Giuse and Pedro Szekely. The Importance of Indirect References in Constraint Models. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'91, Hilton Head, SC, Nov., 1991.

Reprinted from *ACM Symposium on User Interface Software and Technology*,
Monterey, CA, Nov. 15-18, 1992. pp. 117-124.

Graphical Styles for Building User Interfaces by Demonstration

Osamu Hashimoto
C&C Systems Research Labs.
NEC Corporation
4-1-1 Miyazaki, Miyamae-Ku,
Kawasaki, Kanagawa 216, Japan
osamu@ts1.cl.nec.co.jp

Brad A. Myers
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue,
Pittsburgh, PA 15213
brad.myers@cs.cmu.edu

ABSTRACT

Conventional interface builders allow the user interface designer to select widgets such as menus, buttons and scroll bars, and lay them out using a mouse. Although these are conceptually simple to use, in practice there are a number of problems. First, a typical widget will have dozens of properties which the designer might change. Insuring that these properties are consistent across multiple widgets in a dialog box and multiple dialog boxes in an application can be very difficult. Second, if the designer wants to change the properties, each widget must be edited individually. Third, getting the widgets laid out appropriately in a dialog box can be tedious. Grids and alignment commands are not sufficient. This paper describes *Graphical Tabs* and *Graphical Styles* in the Gilt interface builder which solve all of these problems. A "graphical tab" is an absolute position in a window. A "graphical style" incorporates both property and layout information, and can be defined by example, named, applied to other widgets, edited, saved to a file, and read from a file. If a graphical style is edited, then all widgets defined using that style are modified. In addition, because appropriate styles are inferred, they do not have to be explicitly applied.

KEYWORDS: User Interface Builder, User Interface Management System, Demonstrational Interfaces, Styles, Tabs, Garnet, Direct Manipulation, Inferencing

INTRODUCTION

The Gilt Interface Builder allows dialog boxes and similar user interface windows to be created by selecting widgets from a palette and laying them out by direct manipulation (see Figure 1). Two sets of extensions have been added to Gilt to make it significantly easier to create these user interfaces. The first set helps eliminate many of the call-back procedures which communicate to application programs. This was described in a previous paper [8]. The second set of extensions make it easier and faster for the designer to

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

achieve the desired appearance for the user interface, and is described here.

In most toolkits, the widgets have many properties that the designer can set, such as the color, font, label string, orientation, size, the minimum and maximum values of a range, etc. Many widgets in the Motif widget set, for example, have nearly 50 different properties that can be set. Most interface builders, including Gilt, provide "property sheets" that allow the designer to specify the desired values (see Figure 2). However, it can be quite difficult and time consuming to find and set all of the appropriate properties. To show the magnitude of the problem, many applications contain over 2000 widgets, and the properties for each must be set in a consistent manner. A study has shown that achieving consistency in an interface is a frequently cited problem [9].

Another problem for interface designers is laying out the widgets in the window. When the designer places widgets with the mouse, they tend to be uneven and look sloppy. Therefore, most builders provide grids and alignment commands. However, these can be clumsy to use, and they do not insure that different dialog boxes will have a consistent alignment (for example, that the titles will always be centered at the top of the window).

To help solve these problems, Gilt introduces the notions of *Graphical Tabs* and *Graphical Styles*. These are based on the styles and tabs in text editors such as Microsoft Word. A "graphical tab" is simply a horizontal or vertical position in the graphics window to which objects can be aligned. A "graphical style" is a named set of properties and layout information, which can be applied to widgets. The designer can edit a widget so it has the desired properties, select it, and then define a named style based on it. The values of the properties and the position of the widgets will be associated with that style name. The style can then be applied to other widgets.

Furthermore, Gilt will try to automatically guess when to apply a style, so the designer does not have to. By guessing the appropriate properties and layout, Gilt makes the user interface design process significantly faster, since users can quickly and imprecisely place widgets, and the system will

automatically neaten them. Since the inferencing is based on the styles the user has defined, rather than based on global, default rules, as in earlier systems like Peridot[5] and Druid[11], the inferred properties and positions are more likely to be correct.

A set of styles and tabs can be written to a file to form a *Graphical Style Sheet* which can be used to insure that multiple applications have a consistent appearance. If a style is edited, all widgets that are based on that style are automatically updated, so that the interfaces will continue to be consistent.

Gilt is a part of the Garnet system[7]. Garnet is a comprehensive user interface development environment containing many high-level tools, including Gilt, the Lapidary interactive design tool[6], and others. Garnet also contains a complete toolkit, which uses a prototype-instance object model, constraints, and a separation of the behaviors from the graphics. Gilt stands for the Garnet Interface Layout Tool, and it supports interfaces built using either the Garnet look-and-feel widget set or Motif look-and-feel widget set. (The Motif-style widgets in Garnet are implemented on top of the Garnet Toolkit intrinsics and do not use any of the Xtk code in C. Although they look and behave like the standard Motif widgets, they have the same procedural interface as the Garnet widget set.) If you are interested in getting Garnet, contact the second author. Gilt uses CommonLisp, but the ideas presented here are applicable to interface builder tools using conventional compiled languages.

RELATED WORK

Of course, there are a large number of commercial and research interface builders that lay out widgets, including the Prototyper for the Macintosh, UIMX for Motif, DialogEditor[1], the NeXT Interface Builder[14], Druid[11] and YUZU[12]. All of these have the same basic structure: there are two or more windows. One is the work window where the user interface is being created, and another is the widget window, sometimes called the "palette" containing the widgets that can be placed. (Typically, in addition to the standard interaction techniques like menus, radio buttons, check boxes, and scroll bars, there are also decorations like rectangles, lines and text labels that can be added to the picture. In this paper, these are all included when the word "widget" is used.) The designer selects a widget from the palette and places it in the work window using a mouse. Usually, the designer can change the position and size of widgets using the mouse, and edit other properties using dialog boxes or property sheets. The builders also provide many editing functions such as moving, copying, deleting, and aligning widgets, and reading and writing to a file.

Peridot[5] guessed alignment of graphical objects using global rules. Druid[11] applies a similar technique to widget alignment. When the designer adds a new widget in a window, Druid immediately tries to find other widgets in the window that the new widget might be aligned with. For example, when the designer creates a label below another existing label, Druid guesses that the new label and the

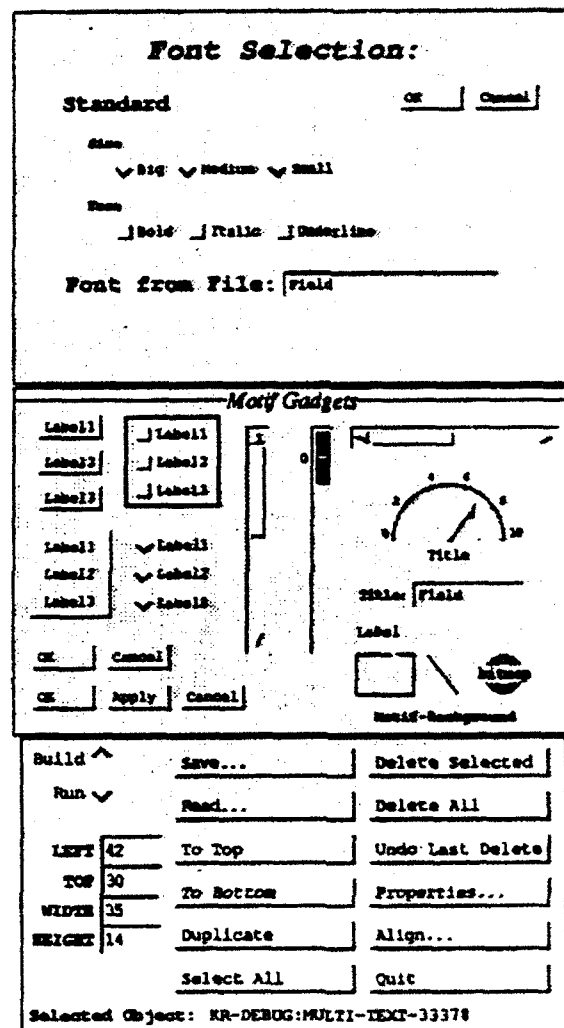


Figure 1: Gilt Main Windows

The top is the work window where a dialog box for a text editing application is being defined. The middle window is the palette of Garnet Motif gadgets that can be added to the work window. The bottom window is the main Gilt control panel containing the Gilt commands. The position and size of the selected widget is echoed in the text boxes at the left of this window.

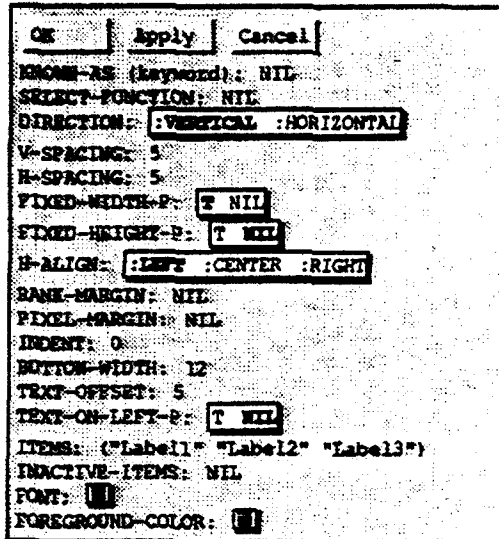


Figure 2: Gilt Property Sheet Window

The *Property Sheet* window for a set of check boxes. The designer can press with the cursor over any of the text fields, and type a new value. Pressing on the icon next to *Font* or *Foreground-Color* will bring up a sub-dialog box.

existing label should have the same left. It pops up a window so the designer can confirm the guess, and if the designer says yes, then Druid adjusts the objects automatically. However, Druid does not infer other properties of the objects, and the layout rules are hard-wired, rather than based on the user's preference, as in Gilt.

Many interface builders have provided interesting mechanisms for specifying the positions of widgets. For example, *FormsVBT*[2] and *ibuild* [13] use a "glue" model based on TeX. Glue has a varying stretch, and using the right kinds of glue between widgets causes the widgets to move appropriately when windows change size. In *Lapidary*[6], the designer can select two objects, and define arbitrary layout constraints between them. The most common constraints can be applied by using iconic menus. *OPUS*[3] shows the specified constraints as wires between the objects. We feel that the concept of tab stops will be more familiar to users and will be easier to use than these other approaches, while still providing most of the needed functionality. Also, no previous interactive builder has incorporated a notion of Graphical Styles, as used in Gilt.

The design for styles and tabs in Gilt is based on their use in text editors, in particular Microsoft Word for the Macintosh. This text editor allows the users to move a marker in a graphical ruler to set a tab stop, and if the TAB key is typed, the text cursor will move to the designated place. To define a style in Microsoft Word, the user formats some text in the

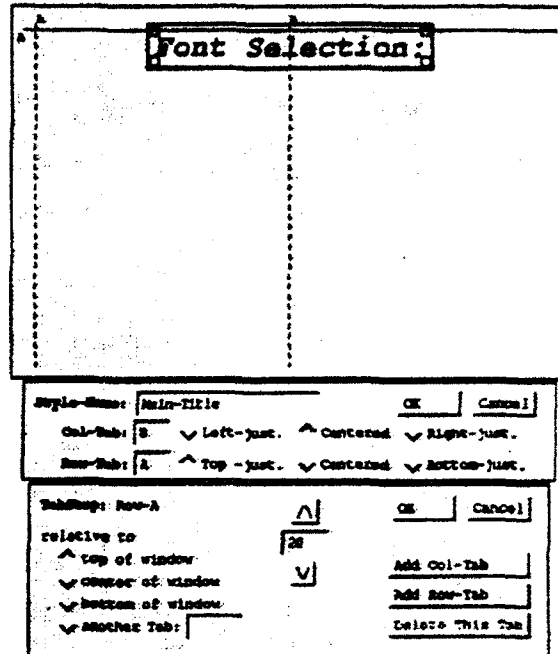


Figure 3: Style Editing Window and TabStop Window
Row-Tab A is selected in the work window (top), and is a horizontal tab that is 20 pixels from the top of the window, as shown by the *TabStop Editing* window at the bottom. The string "Font Selection:" is top-justified on Row-Tab A, and centered horizontally on Col-Tab B, which is centered in the window, so it will move if the window changes size. The *Style Editing* window (center) shows that the title is using the style *Main-Title* and Col-Tab B and Row-Tab A.

desired way, selects it, and then defines a new named style based on it. More general text styles are supported in [10].

GRAPHICAL TABS

An important graphic design principle is that widgets should be aligned evenly. This means that the edges or centers of the widgets should be the same, and that they should be evenly spaced. Furthermore, different dialog boxes should use the same alignments. For example, if in one dialog box a set of radio buttons is left justified under a title, and offset below it by 10 pixels, the same offset and alignment should generally be used in other dialog boxes.

Graphical tabs allow these kinds of relationships to be defined. A "graphical tab" is a horizontal or vertical position in a window. A horizontal tab position is specified relative to the top, bottom or center of a window. Similarly, a vertical tab is specified relative to the left, right or center. This allows the tabs to move appropriately if the window is resized. Just as with text editor tabs, the designer can specify whether the

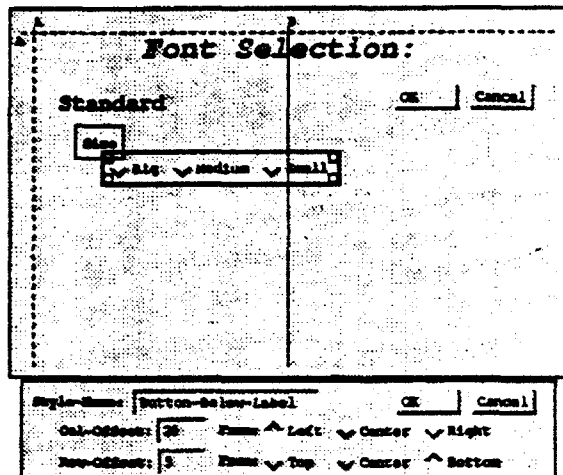


Figure 4: Style Editing Window for Relative Position
The position for the radio buttons is defined relative to the string "Size".

widgets will be left-justified, centered, or right-justified on the tab (or top-, centered, or bottom-justified for horizontal tabs). Since Garnet is implemented on X/11 which uses a pixel coordinate system, the offsets are specified in pixels. Gilt names the tabs with letters (although user-named tabs might be added in the future). Figure 3 shows a Gilt work window with a set of tabs visible. Whether the tabs are visible or not is controlled by a command.

New tab stops can be explicitly added by clicking on the "add tab" buttons in the *TabStop Editing* window shown at the bottom of Figure 3. Tabs can be selected by pointing on the label next to the line in the work window. The selected tab can then be deleted if no styles or widgets are using it. Tabs can be edited by entering new values into the tabstop editing window, or the tab stop labels in the work window act as handles and can be directly dragged with the mouse. When a tab is moved, all of the widgets defined using that tab are also moved.

GRAPHICAL STYLES

A graphical style includes a set of widget properties, and optionally some position information as well. To create a new style, the designer modifies a widget to the desired appearance using the conventional property sheets, selects that widget, and then issues the *Define Style* command. The designer must then type a style name into the *Style Editing* window that will appear. Gilt compares the widget's current properties with the default values for that widget and copies all that are different. The widgets used to define the style are surrounded with a dark outline rectangle in the work window while the style is being defined or edited ("Font Selection:" in the top window of Figure 3).

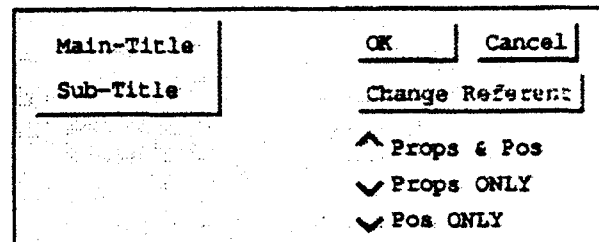


Figure 5: Set Style Window

This window allows designers to explicitly set a style. All the current defined styles are listed on the left, and the designer can choose one, and then specify whether the associated properties, position or both should be applied to the selected widget. If the selected style uses a relative position (Figure 4), then the *Change Referent* button is not grayed-out, and can be used to select the widget that the widget should be relative to.

Since all the widgets in the Garnet toolkit use the same names and values for similar properties, a style defined on one type of widget will often work on other types. For example, radio buttons, check boxes, and button sets all allow the designer to specify the orientation (horizontal or vertical) and fonts. In the top window of Figure 1, all the buttons have the same style properties. The types that styles are associated with include strings, buttons (which include radio buttons, check boxes and button sets), numeric sliders (which include both sliders and scroll bars), text input fields, etc.

Styles can also include position information. For example, a designer might specify that widgets with the *Main-Title* style should use a very large bold and italic font, and be centered at the top of the window. The position information for styles can either be with respect to a graphical tab stop, or relative to a previously created widget. For the first type, the appropriate tab name can be entered into the style editing window (see the center window of Figure 3). Either or both of the horizontal and vertical tab name fields can be blank, in which case no position information is recorded in that direction.

To specify that a style's position should be relative to another widget, the designer selects the referent widget after the style editing window is displayed. The style window will then change, as shown in Figure 4. When a style is relative, only the type of the referent widget is remembered. For example, in Figure 4, the style is defined as offset from any string. This will allow the *Button-Below-Label* style to be used relative to other strings, which can be in other parts of the window.

The *Set Style* window (see Figure 5) allows the designer to choose any of the defined styles, and also whether the position and properties aspects of the style should be applied. When setting the properties, Gilt checks each property associated with the style to see whether the widget accepts that property.

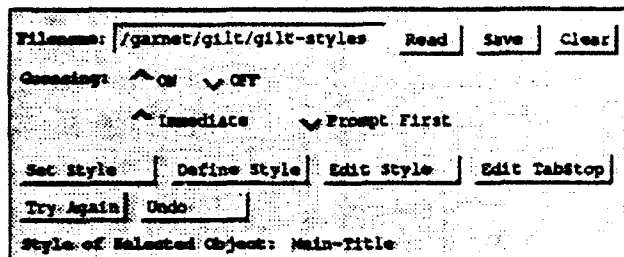


Figure 6: Style Control Window

This window allows styles to be read and written to a file, and style guessing to be turned on and off. Also, the style of the selected widget is always echoed at the bottom of the window.

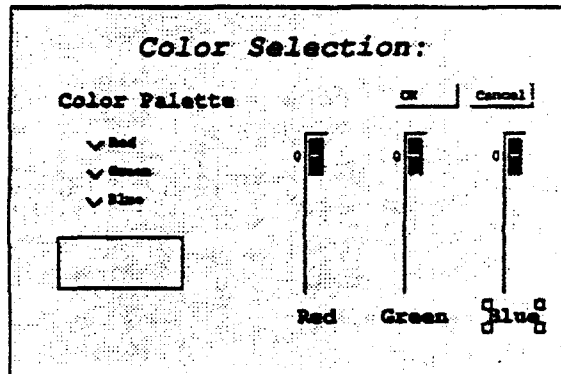


Figure 7: Inferring Styles

If not, then that property is ignored. For example, a style defined using radio buttons might have a value for the *Text-on-left-p* property, which determines which side the diamond is on. However, this is not relevant for push buttons (since their text is inside the button), so it would then be ignored. For styles with absolute positions, the widget simply moves to the correct tab stop. For relative positions, the user can specify the referent widget.

INFERRING STYLES

Although the styles mechanism as described above is already quite useful, Gilt goes further and tries to automatically determine when a particular style is appropriate. The *Style Control* window (Figure 6) provides three options: no inferencing of styles, styles applied immediately when they are inferred, or a prompt-first mode where the designer is asked if the style should be applied, as in Peridot and Druid. If the system usually infers the correct style, then the immediate mode will be the most efficient.

When inferencing is on, Gilt tries to infer a new style

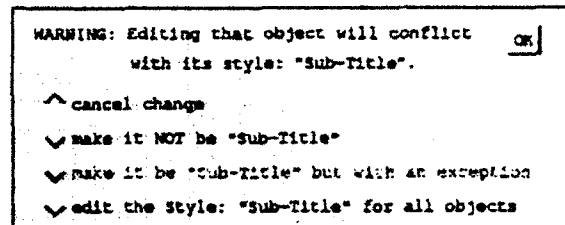


Figure 8: Warning Window

This window pops up when the designer edits a widget that has a style attached to it.

whenever a widget is created or moved. The algorithm looks for styles that affect the same type as the widget, and checks how close the widget matches the style's position. For a style with a relative position, in order to find its possible referent widgets Gilt checks all the widgets of the appropriate type near the new widget. A list is created of all the styles that match, sorted according to the distance to the tab stops or the referent widgets. For example, in Figure 1 the main-title and the sub-titles use different styles with different fonts and positions, and Gilt can infer the appropriate style from the position when the designer places the new string.

Any inferencing system will sometimes guess wrong. Thus, it is important to provide appropriate feedback so the users are confident that they are in control and know what Gilt is doing. In immediate mode, the first style on the style list is immediately applied to the widget, and the name of the style is shown at the bottom of the style control window (Figure 6). The widget will also jump to the inferred position and change appearance. If the inferred style is not correct, the designer can hit the *Try Again* button (Figure 6), which will remove the guessed style and instead apply the next style in the sorted list. The *Undo* button can also be hit to remove the guessed style, and return the widget to its original position and properties. In prompt-first mode, the sorted list of all the inferred styles is presented in a window, with the most likely selected. The designer can select a different style, if necessary, and then hit *OK* or *Cancel*.

When a style is defined, it immediately becomes a candidate for inferencing. This is very useful when a number of widgets will all be created using the same style. In Figure 7, after the designer defines a style which centers the text label below the first scroll bar, when the second scroll bar and label are created, the label will automatically be centered. This highlights an advantage of the style approach over a rule-based approach as used in Druid and Peridot. Those systems might have put the label left-justified under the second scroll bar if it was placed closer to that alignment, but Gilt only matches against previously demonstrated styles, so it is more likely to guess the designer's intentions. This will also help achieve a consistent design.

EDITING STYLES

When a style is applied to a widget, either explicitly or inferred, Gilt sets up appropriate pointers and back pointers so that if the style is ever edited, all widgets using that style are immediately updated.

Styles can be edited in two ways. A property sheet can be displayed which shows the current values of the properties for the style, and this can be edited directly. This property sheet has the same format as the ones for the standard widgets (Figure 2). The position associated with the style can be edited using the appropriate dialog boxes (Figure 3 and 4).

Alternatively, the designer can edit the styles in the same way as they were created: by working on example widgets. Whenever a widget is edited that has already been defined to be of a particular style, Gilt pops up a dialog box asking if the edit should change the style itself (Figure 8). The other alternatives are to make the widget no longer belong to the style, or to cancel the change and return the widget to its appearance before the edit was attempted.

In the future, we plan to add the ability to have widgets use a particular style with exceptions, but this is a complex problem[4]. Some of the issues are whether to copy the attributes or retain the link to the original style, what to do to a style when the style it inherits from is changed, and whether to save the inheritance links in the style files, or write out all the style information to each file.

WRITING AND READING STYLES

A set of styles can be written to a file using the buttons in the style control window (Figure 6). This file can then act as a "Style Sheet." Whenever a new dialog box is being created, the style sheet file can first be read. Then, the appropriate styles can be inferred or explicitly applied to the widgets. This will help insure that the new dialog box is consistent with previous dialog boxes created for this or other applications.

When the work window is saved to a file, Gilt will optionally include the style information in that file. In this case, the file is self-contained. Alternatively, the file can simply contain a pointer to the appropriate style file. Then, whenever the window is used by applications or read back into Gilt, the style file will be re-read, so any subsequent edits to the styles will be reflected. However, this can cause the window to look ugly (for example, if the style for a set of radio buttons changes from horizontal to vertical, the buttons are likely to overlap other widgets). Therefore, a version number is kept in the style file, so at least a warning can be issued when an old window is opened with an edited style file.

EVALUATION

As a small, informal experiment to see how quickly users could create interfaces, using graphical styles and tabs, four subjects were given two tasks. Each task has two similar dialog boxes. For the first box in each task (i.e. DBox1 and DBox3), the properties for all widgets were set using the

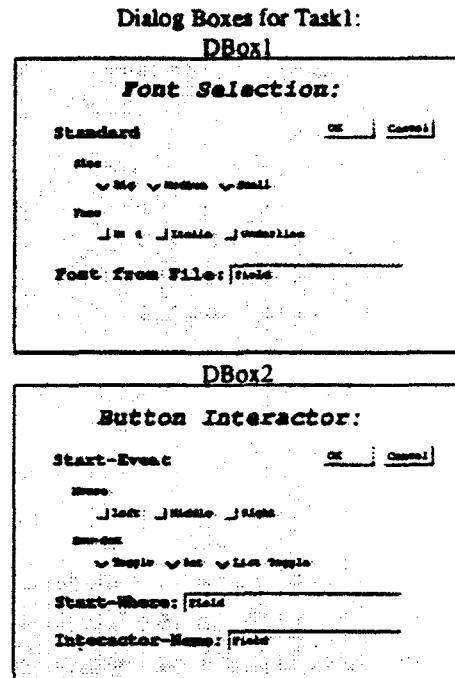


Figure 9: Dialog Boxes for Task1 and Task2

property sheets. Their positions were determined using tabs. Then, several styles were defined using these properties and positions. For the second box in each task (i.e. DBox2 and DBox4), the properties and positions for all widgets can be inferred automatically, using the styles defined in the first box (Table 1, Figure 9). The same four dialog boxes are created without any styles by a Gilt expert. The results were used to compare with above results (Table 2 and 3).

Table 1: Task Description

	Task1		Task2	
	DBox1	DBox2	DBox3	DBox4
Widgets	8	9	7	11
Defined TabStops	3	0	1	0
Defined Styles	6	0	4	0
Guessed Widgets	2	9	3	11

From Table 2 and 3, it is clear that the dialog box, where all widgets are inferred, is created in less than half the time for dialog boxes without styles: a 0.45 ratio for DBox2 and a 0.42 ratio for DBox4. In guessing the styles for users on DBox2 and DBox4, Gilt guessed correctly almost all the time. The over-heads for defining styles and tabs are small: a 1.73 ratio for DBox1 and a 1.28 ratio for DBox3. Note, however, that the longer time for DBox1 is mostly due to the learning time since this was the first time using Gilt and styles for almost all subjects. In addition, novices can learn Gilt styles and tabs quickly, because, in DBox1, they needed a 1.73 ratio, but, in DBox3, they needed only a 1.28 ratio.

The verbal protocols for these subjects indicated that they felt that Gilt style guessing was useful and comfortable. Two subjects said that the "Try Again" button is very good. Subject A, who took this test twice, using styles and without any styles, felt that defining relative styles was very useful, because the conventional layout mechanism did not support "offset" among widgets, so he often had to calculate "left + width + offset" values for the referent widget, in order to determine the left hand position for the new widget. He indicated that graphical tabs were good for aligning some widgets at particular fixed lines. However, all subjects claimed that they had to think about style names, whenever defining any styles. They indicated that it was difficult to give good names for all styles. Also, they said that sometimes they couldn't remember whether this name had already been used or not. Thus, we plan to prepare a default style name in the style editing window.

STATUS AND FUTURE WORK

An earlier version of Gilt has been released to all Garnet users. The version described here has been mostly implemented, and is expected to be debugged and released in the next few months.

In the future, we plan to investigate unifying tabs with the

Table 2: Task1 Results

[sec]	DBox1	DBox2	DBox1+2
Style: Subject A	420	160	580
Style: Subject B	1000	200	1200
Style: Subject C	910	300	1210
Style: Subject D	1060	270	1330
Style: Average	847	233	1080
No Style: Subject A	490	510	1000
Ratio	1.73	0.45	1.08

Table 3: Task2 Results

[sec]	DBox3	DBox4	DBox3+4
Style: Subject A	250	110	360
Style: Subject B	400	300	700
Style: Subject C	380	180	560
Style: Subject D	500	180	680
Style: Average	383	193	575
No Style: Subject A	300	460	760
Ratio	1.28	0.42	0.76

relative styles. It seems like there should be a convenient way to define "relative tabs" that will achieve the desired results. As discussed above, we would also like to investigate exceptions to the styles. There might be a way to copy just some values from one style into another, and ways to read just a few styles from a file. Further work is needed on ways for the system to automatically generalize styles, so that, for example, the font property or color defined on a radio button will be applied to a circular gauge, even though they have different types.

CONCLUSIONS

The *Graphical Styles* mechanism described in this paper can help designers more quickly create user interfaces, because many of the properties and alignments can be applied with a single specification, or even inferred automatically. In addition, the styles can help insure consistency across multiple dialog boxes in an application, and even across multiple applications, since *Style Sheets* can be developed and re-used. The *Graphical Tab* mechanism seems to be an easier-to-understand and easier-to-edit mechanism than other layout approaches. Finally, in addition to being useful for user interface builders, such as Gilt and YUZU, we feel that the graphical styles and graphical tab mechanisms would be useful for a wide range of graphical editors, including drawing programs and CAD/CAM.

ACKNOWLEDGEMENTS

Andrew Mickish helped to implement the features described in this article. Brad Vander Zanden and other members of

Garnet project provided useful advice and help with the design and implementation. For help with this paper, we want to thank Dave Kosbie, Richard McDaniel, Andy Mickish, Francesmary Modugno, Bernita Myers, Brad Vander Zanden, Tomonari Kanba, Hiroshi Yamada, Kin-ichi Hisamatsu, Kyoji Kawagoe, the YUZU development members and the reviewers.

This research was partially sponsored by NEC Corporation, and partially by the Avionics Lab, 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. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

- [1] Luca Cardelli. Building User Interfaces by Direct manipulation, In *Proceedings of UIST'88*, Alberta, Canada, 1988, pp.152-166.
- [2] Gideon Avrahami, Kenneth P.Brooks, and Marc H.Brown. A Two-View Approach To Constructing User Interfaces, In *Proceedings of SIGGRAPH'89*, Boston, 1989, pp.137-146.
- [3] Scott E.Hudson and Shamim P.Mohamed. Interactive Specification of Flexible Interface Displays, *ACM Transactions on Information Systems* 8,3, 1990, pp.269-288.
- [4] Jeff Johnson and Richard J.Beach. Styles in Document Editing Systems, *IEEE Computer* 21,1, 1988, pp.32-43.
- [5] Brad A.Myers. Creating User Interfaces by Demonstration, *Academic Press*, 1988.
- [6] Brad A.Myers, Brad Vander Zanden, and Roger B.Dannenberg. Creating Graphical Interactive Application Objects by Demonstration, In *Proceedings of UIST'89*, Williamsburgh, 1989, pp.95-104.
- [7] Brad A.Myers, Dario A.Giuse, Roger B.Dannenberg, Brad Vander Zanden, David S.Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive Support for Graphical Highly-Interactive User Interfaces, *IEEE Computer* 23,11, 1990, pp.71-85.
- [8] Brad A.Myers. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs, In *Proceedings of UIST'91*, Hilton Head, 1991, pp.211-220.
- [9] Brad A.Myers and Mary Beth Rosson. Survey on User Interface Programming, In *Proceedings of CHI'92*, Monterey, 1992, pp.195-202.
- [10] Brad A.Myers. Text Formatting by Demonstration, In *Proceedings of CHI'90*, New Orleans, 1991, pp.251-256.
- [11] Gurminder Singh, Chun Hong Kok, and Teng Ye Ngan. Druid: A System for Demonstrational Rapid User Interface Development, In *Proceedings of UIST'90*, Snowbird, 1990, pp.167-177.
- [12] Takahiro Sugiyama *et al.* CANAE User Interface Builder: YUZU (In Japanese), In *Proceedings of the 45th National Convention of Information Processing Society of Japan*, Tokushima, 1992, 5Q-3.
- [13] John M.Vlissides and Steven Tang. A Unidraw-Based User Interface Builder, In *Proceedings of UIST'91*, Hilton Head, 1991, pp.201-210.
- [14] Bruce F.Webster. The NeXT Book, *Addison-Wesley Publishing*, 1989.

Reprinted from *Proceedings SIGCHI'91: Human Factors in Computing Systems*.
New Orleans, LA. April 28-May 2, 1991. pp. 243-249.

GRAPHICAL TECHNIQUES IN A SPREADSHEET FOR SPECIFYING USER INTERFACES

Brad A. Myers

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

ABSTRACT

Many modern user interface development environments use *constraints* to connect graphical objects. Constraints are relationships that are declared once and then maintained by the system. Often, systems provide graphical, iconic, or demonstrational techniques for specifying some constraints, but these are incapable of expressing all desired relationships, and it is always necessary to allow the user interface designer to write code to specify complex constraints. The spreadsheet interface described here, called C32, provides the programmer with the full power of writing constraint code in the underlying programming language, but it is significantly easier to use. Unlike other spreadsheets tools for graphics, C32 automatically generates appropriate object references from mouse clicks in graphics windows and uses inferencing and demonstrational techniques to make constructing and copying constraints easier. In addition, C32 also supports monitoring and debugging interfaces by watching values in the spreadsheet while the user interface is running.

KEYWORDS: Constraints, Spreadsheets, User Interface Development Tools.

INTRODUCTION

C32 is a tool that helps users construct *constraints*. A "constraint" is a relationship among objects that is declared once and maintained automatically by the system. Typically, a constraint is expressed as an expression (or "formula") that is stored in a slot of an object. The expression is re-evaluated whenever any other values change that are referenced in the formula. Constraints are used to control the graphical objects in many modern user interface toolkits.

C32 uses a spreadsheet model and provides the programmer with the full power of writing constraint code in the

underlying programming language. However, it is significantly easier to use, and provides many of the advantages for graphics programming that financial spreadsheets provide for business.

C32 is different from previous spreadsheet systems for user interface construction because it uses a wide array of visual and inferencing techniques so the user does not have to write the entire constraint by hand. In particular:

- C32 automatically generates appropriate references to graphical objects when the user clicks on the object in a user interface window.
- It uses demonstrational techniques to guess which properties of objects should be used.
- It guesses how to parameterize constraints when they are copied from one place to another or generalized into procedures, so abstract and reusable constraints can be constructed *by example*.
- It incorporates graphical techniques to help trace and debug constraints.
- It is integrated with an existing prototype-instance system in which constraints can be inherited.
- It is one of a suite of tools built on top of an existing, successful constraint system, rather than providing the only interface to the constraints, so users can choose other tools when they are more appropriate.

Many systems provide a graphical, direct manipulation technique for specifying some constraints. Unfortunately, such techniques are incapable of expressing every constraint the user may want. C32 provides a convenient and easy-to-use technique for constructing constraints when the graphical techniques are inappropriate. For example, C32 pops up when the user asks for a custom constraint in the Lapidary interactive design tool [8]. C32 can also be used stand-alone when a graphical front end is not available. We have found C32 to be significantly easier to use than constructing the constraints by typing in code.

C32 has been implemented as part of the Garnet system [10]. It is an acronym, and stands for CMU's Clever and Compelling Contribution to Computer Science in CommonLisp which is Customizable and Characterized by a Complete Coverage of Code and Contains a Cornucopia of Creative Constructs, because it Can Create Complex, Correct Constraints that are Constructed Clearly and

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1991 ACM 0-89791-383-3/91/0004/0243...\$1.50

Concrete, and are Communicated using Columns of Cells that are Constantly Calculated so they Change Continuously and Cancel Confusion.

RELATED WORK

Spreadsheets have been used for financial calculations for a long time, starting with VisiCalc in about 1984, and most systems have the same form: an array of cells, with each column labeled with a letter and each row with a number. Some extensions to the spreadsheet idea include using it for logic programming [12] and a technique for adding procedural abstraction [1].

The NoPumpG and NoPumpII systems [15] use spreadsheets to define graphical user interfaces, but they have a number of important differences from C32. The most important is that the NoPump systems provide many different types of cells. In C32, all the cells are the same type: slots of objects. To program the user interface, NoPump provided special cell types that reported low-level mouse positions and clock ticks, whereas C32 uses Garnet's high-level Interactor objects [9] to handle behaviors, and slots of Interactors can be specified and viewed in C32 cells, just like any other object. In NoPump, the cells are free floating whereas C32 uses a tabular organization, like a conventional spreadsheet. Thus, all the cells about a particular object are always in one place in C32. There are additional small differences. The cells in NoPump are typed and the formulas use a special language. In C32 the cells can hold any kind of value, and formulas are expressed in a standard language (Common Lisp). Also, NoPump provides few facilities for object referencing, and none for formula generalization.

Constraints have been used by many systems, starting with Sketchpad [13] and Thinglab [3]. Uses of constraints within user interface toolkits include GROW [2], Peridot [7], Apogee [5], and CONSTRAINT [14].

Graphical interfaces to constraints include the "wiring diagrams" in Thinglab [4], the iconic interfaces in Juno and Lapidary [8], and the reference lines in Apogee [6]. The Peridot system automatically infers constraints from example drawings [7]. The wiring diagrams are hard to use for complicated constraints, and the other techniques cannot even handle complex constraints. The spreadsheet interface described here could be used as a *supplement* to these other techniques when they cannot generate the desired relationship.

CONSTRAINT EXPRESSIONS

Objects in Garnet have instance variables or fields, called *slots*. The content of each slot is either a normal value, such as a number or string, or a *formula* that computes the value. References to other objects in formulas use a special form: (gv other-object slot-name), where gv stands for "get value."

Because each slot can contain at most one formula, only *one-directional* constraints are supported. We are exploring multi-way constraints for the future, in which case C32 will be changed appropriately.

Through extensive experience with the many projects that have hand-coded Garnet constraints, we have discovered that people have trouble generating correct constraints. Although most constraints in interfaces are very simple, there are a reasonable number of five to ten line code fragments used as constraints. Even the simple constraints can be tricky to enter, since the user must reference the appropriate objects and slots. Also, given a set of constraints that are not working correctly, users have difficulty finding the bugs. C32 was designed to address these problems.

OVERVIEW

C32 can display and allow the user to edit any kind of object and constraint, no matter how they were created: by hand-coding, by using Lapidary (a Garnet interactive design tool), or by using C32.

Figure 1 shows a typical instance of C32. Each column contains a separate object. Rows are labeled with the names of the slots, such as :left, :top, :width, :height, :visible, etc.¹ Since different objects can have different slots, the slot names are repeated in each column. For example, lines have slots for the endpoints (:x1, :y1, :x2, :y2) but rectangles do not. Also, each object's display can be scrolled separately, so each has its own scroll bar. This makes the spreadsheet look somewhat like a multi-pane browser as in Smalltalk.

The spreadsheet cells show the current values of the slots. If a value changes, then the display will be immediately updated. It is important to emphasize that the user interface being constructed will operate normally (albeit a little slower) even while the spreadsheet is displaying objects in that user interface. The underlying constraint mechanism is used internally by the spreadsheet to maintain this connection. Monitoring the values as they change can help the programmer debug objects, and makes the constraints much more "visible" and understandable. If the user edits the value in the spreadsheet cell, the object's slot will be updated.

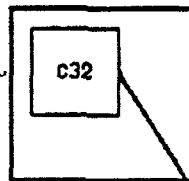
The  icon by some slots in Figure 1 means that the slot value is computed from a formula. Pressing the mouse on the icon causes the constraint expression to appear in a different window (see Figure 2). The expression itself can be edited by typing or other techniques (discussed later). Note that unlike cells in conventional financial spreadsheets, C32 allows a slot to have a value different from what the formula would calculate. Therefore, the user can edit the value of slots with formulas without affecting whether there is a formula there or not. This can be useful when trying to find the correct value for a slot while debugging. To remove a formula, the user simply deletes the entire string in the formula display window.

One novel feature of the underlying object system is that new slots can be added to objects at any time. Using C32, the user can create a new slot by simply typing a slot name and value into a blank row.

¹ All slot names start with a colon.

MY-RECTANGLE		STRING1		ARROW	
Left	10	String	"C32"	X1	
Top	10	Font	OPAL-DEFAULT-F...	Y1	
Width	50	Left	25	X2	
Height	50	Top	28	Y2	
Visible	T	Width	21	Left	
Line-Style	OPAL-DEFAULT-L...	Height	14	Top	
Fill-Style	NIL	Visible	T	Width	
Draw-Function	COPY	Line-Style	OPAL-DEFAULT-L...	Height	
Window	W	Fill-Background	NIL	Visible	
Parent	A	Actual-Height	NIL	Line-Style	
Is-A	OPAL-RECTANGLE	Draw-Function	COPY	Fill-Style	
		Window	W	Draw-Function	

(a)



(b)

Figure 1: (a) C32 viewing three objects (b). The scroll bars can be used to see more slots or columns. Changing the window's size will change the number of slots and objects displayed (the number of rows and columns). Field values are clipped if they are too long, but can be scrolled using editing commands. The  icon means that the slot value is computed with a formula. All inherited slots are shown in italics and marked with the  icon. (Inheritance is discussed in a later section.) When a formula is inherited the value is shown in a regular font since it is usually different from the prototype's. The inherited icon is also shown next to the formula icon rather than next to the value.

Formula for STRING1. Left






```
(= (GV MY-RECTANGLE :LEFT)
  (FLOOR (- (GV MY-RECTANGLE :WIDTH)
            (GV :SELF :WIDTH))
    | 2))
```

Figure 2: A formula window showing the constraint in the :left slot of the STRING1 object of Figure 1, which centers the string in the rectangle.

To view an object in the spreadsheet, the user can simply type the object's name into the title of a column. Alternatively, the user can select a column, and then point to an object in a graphics window.

As with financial spreadsheets such as Microsoft Excel, we provide a menu of the common functions used in formulas.² Another menu contains the graphical commands provided by Garnet, including functions to center objects with respect to other objects and to make their size

be proportional. The user can easily add commands to this menu, either written in a conventional way or created from formulas. When functions are inserted from the menus, C32 puts the parentheses in the correct places and leaves the cursor where the arguments to the function go. In this way, C32 can be used like a syntax-directed editor, which has the following advantages:

- the user does not have to deal with the syntax of the language so there will be fewer syntax errors,
- the system will handle the parenthesis matching, which otherwise can be annoying in Lisp, and
- this makes the system accessible for people who do not know Lisp.

GENERATING OBJECT REFERENCES

One of the most interesting aspects of C32 is the way that object references can be specified. As in a financial spreadsheet, the user can point to a slot and have a reference to that slot inserted into the formula at the current point. Figure 3 shows how this can be used.

In Garnet, there are different ways to reference an object in a formula. Unlike other systems such as Peridot and conventional spreadsheets, Garnet allows *indirect* references to objects, where the object to be referenced is stored in a slot of the object that contains the formula. One place this is

²There are so many functions in Common Lisp that only the most commonly used are provided in the menu.

```
(gv-indirect :parent :shadow :left)
```

USE OF INFERRING

The other mode ignores the current positions of the objects, but looks at the slot being filled and where the mouse is pressed in the selected object. For example, if the slot is :left, and the mouse is pressed at the right of an object, then the reference will be to the right of the object. For the :width slot, however, the same press would generate a reference to the width of the object. Unlike Peridot, C32 does not try to confirm any of the inferences, but rather simply inserts the text into the formula. If the guess is incorrect, it is easy for the user to delete the text and type the correction.

HY-RECTANGLE 1		STANDARD 1	
Left	10	SLASH	
Top	10	PLANE	
Width	50	Left	0
Height	50	Top	0
Visible	ON	Width	0
Line Style	CPM DEFAULT 1	Height	0
Fill Pattern	HT	Visible	0
Draw Function	COPY	Line Style	
Random	N	Fill Pattern	
Parent	A	Actual Width	
1-2-A	HY-RECTANGLE	Draw Function	

Formula for STANDARD 1: Top

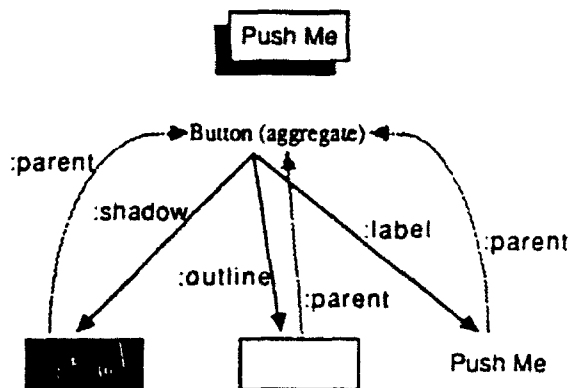
(= (qz HY-RECTANGLE Top)
 (PLANE (-)

HY-RECTANGLE 1		STANDARD 1	
Left	10	SLASH	
Top	10	PLANE	
Width	50	Left	0
Height	50	Top	0
Visible	ON	Width	0
Line Style	CPM DEFAULT 1	Height	0
Fill Pattern	HT	Visible	0
Draw Function	COPY	Line Style	
Random	N	Fill Pattern	
Parent	A	Actual Width	
1-2-A	HY-RECTANGLE	Draw Function	

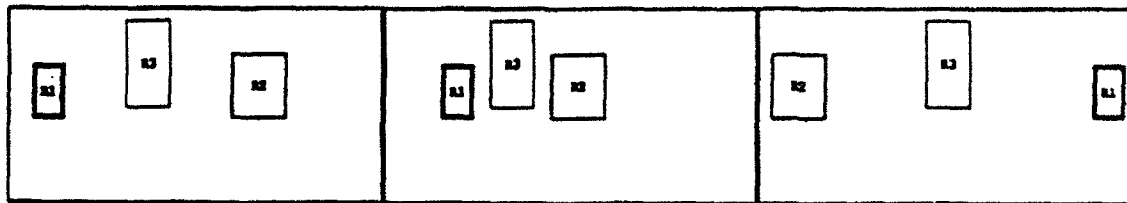
Formula for STANDARD 1: Top

(= (qz HY-RECTANGLE Top)
 (PLANE (- (qz HY-RECTANGLE Top)

Figure.3: The spreadsheet before and after the user has selected the :top cell of MY-RECTANGLE to be inserted into the formula.



246



(a)

```
(formula '(floor (+ (gv r1 :left) ;floor does not divide
                    (gv r1 :width)
                    (gv r2 :left)
                    ; get this object's width.
                    (- (gv-indirect :width)))
2))
```

(b)

Figure 5: (a) Rectangle R3 is centered horizontally between R1 and R2 using the formula shown in (b) so R3 moves automatically when R1 and/or R2 moves.

appears that slot names should be changed, the user is queried with a dialog box, and if the answer is OK, then the formula is modified automatically.³

AUTOMATIC GENERALIZATION

Another possibility is that the references in the formula should be *generalized* into variables. C32 therefore provides a command that will change the entire formula into a function that takes the objects and/or slots as parameters. This process is controlled by a dialog box. As an example, generalizing the formula of Figure 5 creates the dialog box of Figure 6-a, and the code would be changed as shown in Figure 6-b. The new Center-Between-X function can now be used in other formulas. It will also be added to the C32 graphics functions menu, so it can be easily retrieved later.

The intelligent copying and generalizing discussed here helps the user generate correct constraints *by example*. Without these aids, it is quite common to forget to change one or more of the references when formulas are copied. Generalizing also helps the programmer decrease the size of the code by promoting the reuse of existing formulas. Future research will investigate further ways to use generalization.

TRACING AND DEBUGGING

Experience with Gamet and all other constraint-based systems shows that people have a difficult time debugging their formulas. The primary problem is that constraints are not local because values in one object can affect values in many different objects. Therefore, C32 provides a set of tracing and debugging tools.

The most straightforward method is to simply use the

(a)

```
(defun Center-Between-X (obj1 obj2)
  (floor (+ (gv obj1 :left)
            (gv obj1 :width)
            (gv obj2 :left)
            (- (gv-indirect :width)))
2))
```

(b)

Figure 6: (a) The dialog box for generalizing from a formula. All the values shown are the defaults provided by C32. After the user hits OK, the formula of Figure 5-b is converted automatically into the function shown here in (b).

spreadsheet to exercise and monitor the user interface in action. Often, seeing the current values of all the slots is sufficient to determine the problem. To help relate the cells and objects, commands are provided to blink the object associated with a cell, or the cells for an object.

Other commands display arrows in the spreadsheet to show which cells are used in the computation of the current cell (see Figure 7-a) or the cells that use the value of this cell (Figure 7-b). We are exploring additional graphical constraint debugging tools.

³Since this is a more radical change than the inferred slots discussed in the previous section, it seems prudent to require confirmation.

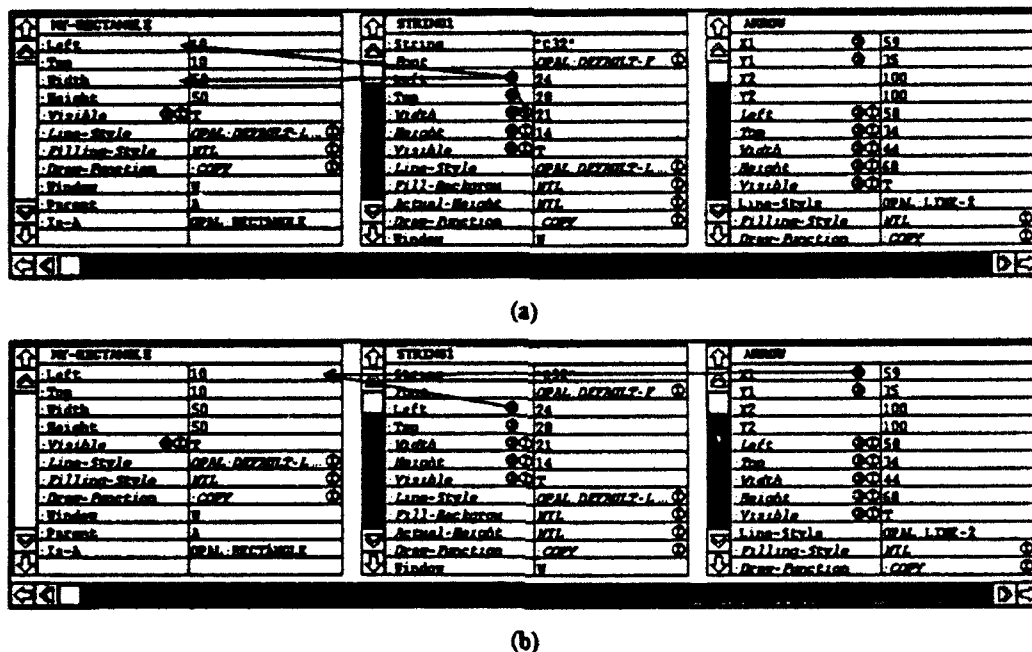


Figure 7: C32 will display arrows to show which slots the current one depends on or which slots depend on the current cell. (a) shows that the :left of STRING1 depends on the :left of the rectangle, and the :widths of the rectangle and itself (since it is centered). (b) shows that the :left of the rectangle is used by the :left of the string and :x1 of the line. In both views, the arrows point to the slot being used.

INHERITANCE

Garnet uses a prototype-instance model instead of the conventional class-instance model. This means that any object can be used as a prototype for a new set of objects; there is no distinction between classes and instances. One result of this is that each instance decides which slots to inherit and which to override. For example, many graphical objects do not have a :filling-style slot, but rather inherit this value. Inherited slots are shown in italics with the Φ icon next to their value (see Figure 1).

Inheritance in Garnet is determined dynamically. This means that setting the value of a slot which used to be inherited, changes the slot to be local with the new value. C32 shows this by removing the inherited icon after a slot is edited. When a local slot is deleted from an object in Garnet, the system looks in the prototype to see if that same slot exists there, and if so, the slot becomes inherited. Therefore, if a slot is deleted in C32 but there is a value that can be inherited, C32 will change the display to show the inherited value. This makes it clear to the user that although the local slot is removed from the instance, there is still a legal value for it.

Formulas can also be inherited. In this case, there is often a different current value for the slot, even though the formula is the same as the prototype's. For example, the :width slot of a text object usually contains a formula that computes the width based on the object's font and string value. Most text objects inherit this formula, but still have a dif-

ferent current value because their string and font values are different. Therefore, if a formula is inherited, the inherited marker is shown next to the formula icon in the spreadsheet and the value part is not shown in italics.

INTEGRATION WITH OTHER TOOLS

There are many different mechanisms and tools in the Garnet system. Therefore, unlike previous spreadsheet systems such as NoPump, C32 does not have to address every aspect of user interface design.

When the programmer wants graphics in Garnet to respond to the mouse or keyboard, an Interactor object [9, 11] is attached to the graphics to handle the input events. There is a standard protocol that the Interactors use to query and modify the graphics. Since Interactors are objects, they can also be read into C32. Unlike graphical objects, however, Interactors are not visible so they cannot be pointed at. Therefore, there are commands to display a menu of all the Interactors, of all those that affect a particular graphic object, or all those that respond to a particular input event.

C32 can also be used with the Lapidary interactive user interface design tool [8]. Lapidary allows the designer to draw pictures of what the user interface should look like and then demonstrate how it should act. Although Lapidary provides an iconic interface to many common constraints, C32 can be used when more complex or custom constraints are necessary.

STATUS AND FUTURE WORK

The spreadsheet described here is mostly working, and we expect to release it to Garnet users within the next few months. Their feedback will be valuable in deciding what features to add and modify. We expect to explore:

- Other ways to use demonstration to create formulas.
- More clever generalizations from existing formulas.
- Better connection with Interactors. There is a well-defined protocol between Interactors and graphic objects that serve as feedback objects. The spreadsheet could set the appropriate fields of the graphic object automatically if the object was placed in a slot of the Interactor, as is done by Lapidary [8].
- Ways to use C32 to create objects from scratch, so C32 can be used as an interface builder. Once objects have been created in memory, Garnet already contains a built-in mechanism that will write them to a file so they can be used by real applications.

CONCLUSION

The C32 spreadsheet contains a number of novel features, including the use of demonstrational techniques to generate object references, automatic generalization of formulas, and graphical tracing and debugging. These make it easier to use than previous spreadsheet-based graphical tools. C32 enhances the Garnet user interface development environment by providing an appropriate mechanism for specifying complex, custom constraints, which occur frequently in user interface software. C32 has demonstrated that a spreadsheet tool can be a valuable addition to an existing constraint-based system, and that it is possible to get totally carried away in acronym building.

ACKNOWLEDGEMENTS

Brad Vander Zanden contributed to the design of C32. An earlier version of C32 (then called C29) was implemented by Andrew Mickish. The Garnet system as a whole has been developed by a large number of people. For help with this paper, I want to thank Brad Vander Zanden, Bernita Myers and the referees.

This research was sponsored by the Avionics Lab, 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. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Additional support for Garnet was supplied by Siemens, Apple Computer, Inc. and General Electric.

REFERENCES

1. Allen L. Ambler. Forms: Expanding the Visualness of Sheet Languages. 1987 Workshop on Visual Languages, Visual Language '87, Linköping, Sweden, Aug., 1987, pp. 105-117.
2. Paul Barth. "An Object-Oriented Approach to Graphical Interfaces". *ACM Transactions on Graphics* 5, 2 (April 1986), 142-172.
3. Alan Borning. Thinglab--A Constraint-Oriented Simulation Laboratory. Tech. Rept. SSL-79-3, Xerox Palo Alto Research Center, July, 1979.
4. Alan Borning. Defining Constraints Graphically. Human Factors in Computing Systems, Proceedings SIGCHI '86, Boston, MA, April, 1986, pp. 137-143.
5. Tyson R. Henry and Scott E. Hudson. Using Active Data in a UIMS. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software, Banff, Alberta, Canada, Oct., 1988, pp. 167-178.
6. Scott E. Hudson. Graphical Specification of Flexible User Interface Displays. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, Williamsburg, VA, Nov., 1989, pp. 105-114.
7. Brad A. Myers. *Creating User Interfaces by Demonstration*. Academic Press, Boston, 1988.
8. Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg. Creating Graphical Interactive Application Objects by Demonstration. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, Williamsburg, VA, Nov., 1989, pp. 95-104.
9. Brad A. Myers. Encapsulating Interactive Behaviors. Human Factors in Computing Systems, Proceedings SIGCHI '89, Austin, TX, April, 1989, pp. 319-324.
10. Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. "Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment". *IEEE Computer* 23, 11 (Nov. 1990), 71-85.
11. Brad A. Myers. "A New Model for Handling Input". *ACM Transactions on Information Systems* 8 (1990), To appear.
12. Michael Spenke and Christian Beilken. A Spreadsheet Interface for Logic Programming. Human Factors in Computing Systems, Proceedings SIGCHI '89, Austin, TX, April, 1989, pp. 75-80.
13. Ivan E. Sutherland. SketchPad: A Man-Machine Graphical Communication System. AFIPS Spring Joint Computer Conference, 1963, pp. 329-346.
14. Brad T. Vander Zanden. Constraint Grammars--A New Model for Specifying Graphical Applications. Human Factors in Computing Systems, Proceedings SIGCHI '89, Austin, TX, April, 1989, pp. 325-330.
15. Nicholas Wilde and Clayton Lewis. Spreadsheet-based Interactive Graphics: from Prototype to Tool. Human Factors in Computing Systems, Proceedings SIGCHI '90, Seattle, WA, April, 1990, pp. 153-159.

Marquise: Creating Complete User Interfaces by Demonstration

Brad A. Myers

Richard G. McDaniel

David S. Kosbie

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
{bam, richm, koz}@cs.cmu.edu

ABSTRACT

Marquise is a new interactive tool that allows virtually all of the user interfaces of graphical editors to be created by demonstration without programming. A "graphical editor" allows the user to create and manipulate graphical objects with a mouse. This is a very large class of programs and includes drawing programs like MacDraw, graph layout editors like MacProject, visual language editors, and many CAD/CAM programs. The primary innovation in Marquise is to allow the designer to demonstrate the overall behavior of the interface. To implement this, the Marquise framework contains knowledge about palettes for creating and specifying properties of objects, and about operations such as selecting, moving, and deleting objects. The interactive tool uses the framework to allow the designer to demonstrate most of the end user's actions without programming, which means that Marquise can be used by non-programmers.

KEYWORDS: User Interface Software, User Interface Management Systems, Interface Builders, Demonstrational Interfaces, Garnet.

INTRODUCTION

One important goal of the Garnet project [6] is to allow user interface designers who are not programmers to design and implement the *look and feel* of user interfaces. The Marquise tool is the newest addition to the Garnet environment, and it ties together all the previous tools, while supporting, for the first time, interactive specification of the entire user interface.

In particular, Marquise allows the overall graphical appearance of the interface to be drawn, and the behaviors for object creation, selection and manipulation to be demonstrated.

Unlike many previous tools which concentrate on widgets, Marquise is aimed mostly at the main drawing window of graphical editors where the user creates and manipulates graphical objects with a mouse. For example, with Marquise you can demonstrate how the rubber banding will

appear as you move the mouse, rather than having this as a hard-wired, unchangeable component. Another important capability in Marquise is demonstrating the modes of the interface. Although "mode-free" interfaces are often touted, all modern graphical interfaces are in fact highly moded. For example, in most drawing tools such as Macintosh MacDraw, a palette controls whether the next mouse click will select an object, insert a text string, or draw a rectangle, circle, polygon, etc. Other modes include the current colors, line styles, and arrowhead styles for the objects that will be created. Marquise provides an intuitive, demonstrational method for specifying the modes that control and are affected by an operation.

With Marquise, we have concentrated on providing complete control of when and how the behaviors are initiated. The primary innovations in Marquise are: (1) the use of special icons to represent the mouse positions while demonstrating the behavior, so the designer can then demonstrate what happens at those locations, (2) sophisticated control over the locations where those events should take place to begin and end behaviors, (3) a "mode window" to make explicit the modes of the interface that control the behaviors and values, (4) the formalization of "palettes" to control modes and object properties, and (5) the ability to interactively specify the attributes for built-in layout operations and objects.

Marquise stands for Mostly Automated, Remarkably Quick User Interface Software Environment. (A "marquise" is a gem having the shape of a short, pointed oval with many facets.) Marquise is part of the Garnet system, which is a comprehensive user interface development environment written in Lisp for the X window system.¹

RELATED WORK

Previous design tools have shown that it is possible to in-

¹The Garnet system is available by anonymous FTP. Although Marquise is not yet ready for distribution as this paper is being written, you can get the toolkit, the Gilt interface builder, and Lapidary. Send mail to Garnet@cs.cmu.edu for information.

interactively specify the graphical appearance and behavior of limited parts of an application's user interface. For example, many interface builders, such as the NeXT Interface Builder, UIMX for Motif, Druid [8], and Gilt [7], allow the designer to interactively specify the placement of widgets. Peridot [3] allows new widgets to be created interactively without programming, and Lapidary [4] allows application-specific graphical objects to be demonstrated. Marquise goes beyond these tools since it supports creating, editing, and deleting of objects at run time, and allows the overall behavior to be defined. DEMO [10] used the idea of demonstrating the end-user's actions that start a behavior (called the "stimulus") and then demonstrating the response to that stimulus. DEMO II [1] added sophisticated techniques for inferring constraints to control how objects are placed or moved. Marquise uses the stimulus-response idea, here called "train" and "show," but concentrates on which high-level actions are appropriate and the context of the stimulus.

Some previous systems have provided frameworks to help code graphical editors. Unidraw [9] and many graph editors (e.g., [2]) provide a standard set of built-in operations as methods, and the designer writes code to override these methods for the specific application. However, none of these other systems allow new behaviors to be defined by demonstration.

USER INTERFACE

The basic windows for Marquise are shown in Figure 1. There is a palette of objects that can be drawn, some palettes for controlling the properties of those objects, and a set of commands in a menubar. In all conventional interface builders there are two modes: *Build* and *Run*, where in Build mode the designer constructs the interface, and in Run mode it is tested. Marquise adds two additional modes to demonstrate behaviors: *Train* and *Show*. Train mode is used to demonstrate what the end user will do, and Show mode is used to demonstrate the system's response to that action. A different mouse cursor for each mode insures that the designer always knows what the current mode is.

In Build mode, the static parts of the interface are drawn. For example, the designer might add to the window some widgets that should always be visible. Lines could be added as decorations. Many applications contain palettes that show which objects can be created, or that show various values for a property (like color, line-style, etc.). These palettes are drawn with Marquise in Build mode. In Run mode, the interface can be exercised to see how it will operate for the end user.

In Train mode, the designer operates the mouse and keyboard in the same way the end user would, and then goes into Show mode to demonstrate what the system's response should be. While in Train mode, the end-user behaviors are operational, but in addition, the keystrokes and mouse movements are saved. In Show mode, the designer can create and edit objects exactly the same as in Build mode, except that the operations are remembered so they can be attached to the events demonstrated in Train

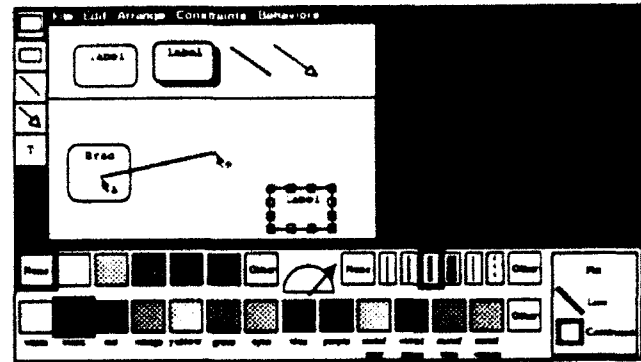


Figure 1:

The main Marquise windows: the basic object palette on the left, the main work area, and the palette for controlling the color and halftones for filling-styles and line-styles at the bottom. The designer is creating an interface with a "create palette" at the top containing two types of nodes and two types of links. The node at the lower right of the work window is selected. The Marquise commands are in the menubar at the top. The "Constraints" menu allows graphical constraints to be specified. The "Behaviors" menu allows objects to be declared as palettes, and displays the mode and feedback window:

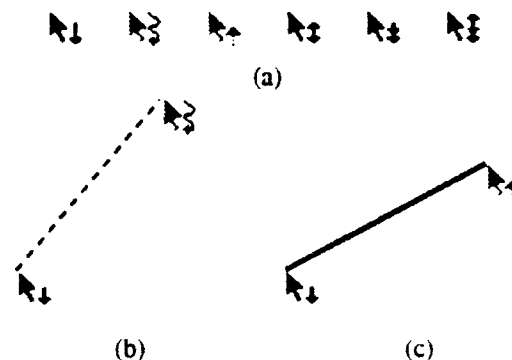


Figure 2:

(a) The icons that show where the mouse was pressed, moved to, released, clicked (pressed and released in the same place), double-clicked, and double clicked and released. (b) In Train mode, the designer pressed the mouse down and moved, and then in Show mode, drew a dotted line as the interim feedback. (c) Going back to Train mode, the designer released the mouse button, and in Show mode, deleted the dotted line and drew a solid line.

mode.

As an example, here is how the designer would demonstrate that when the mouse button goes down, a feedback dotted line should be drawn which follows the mouse, and then when the button is released, the dotted line should be erased and a real line drawn. First, the designer would go into Train mode, press down the mouse button, and move away from the initial press. Without releasing the mouse button, the designer would change to Show

Behavior Name: **Creating-A-Line**

Objects:

The object **Dashed Line** is an instance of LINE-2212 with properties:

Slot :Line-Style is **Constant** **Dashed**

Placement is **constant** End1 = Mouse Down Point

End2 = Mouse Move Point **Edit Placement**

The object **Line** is an instance of LINE-2212 with properties:

Placement is **constant** End1 = Mouse Down Point

End2 = Mouse Up Point **Edit Placement**

Events and Actions:

The relevant mode(s) are:

Create-Palette-1 has value **:Line**

When the **left** mouse button is **pressed** with **no modifier** over the **specific** object **The Work Window**

As the mouse is moved over the **specific** object **The Work Window**

Show **internal** object **Dashed Line**

When the button is lifted over the **specific** object **The Work Window**

Create **internal** object **Line**

Figure 3:

The feedback window for behaviors. At the top is a pull-down menu of commands, then the name of the behavior, then the objects that participate in the behavior, and finally the events and actions. Pushing on the buttons displays a popup window of the other possible choices. Changing the option at the beginning of a "sentence" will change the options available for the rest of the sentence. An entire section of the window can be selected and cut, copied, etc.

mode. The window will now contain two icons which show where the mouse was pressed and to where it was moved. Now in Show mode, the designer draws a dotted line between the icons (Figure 2-b). Marquise infers that a dotted line should be created on the down press and one end should follow the mouse as it moves. Then the designer presses the mouse button somewhere on the background and switches to Train mode with the mouse button still down, so the mouse *release* can be demonstrated. Because this second demonstration does not include a down-press, the original down-press icon is retained. Next, in Show mode, the designer deletes the dotted line and draws a solid line from the initial down press icon to the final button release icon (Figure 2-c). This entire behavior takes less than 30 seconds to demonstrate, and very few new concepts or commands are necessary, since the designer already knows how to draw and delete objects in the editor.

If the mouse had been pressed and released in the same place, then a click icon would be displayed instead of the down, move and up icons. Double-clicking or double-clicking followed by a move are also supported. To allow modes to be changed while mouse buttons are being held down and while the mouse is at a particular place, keyboard accelerator keys are used to change modes.

Marquise generalizes from the designer's example actions to create the user interface. Any system that tries to generalize will sometimes guess wrong. Various mechanisms have been explored in other systems to show the user what has been guessed, so that users can verify and correct the inferences. Older systems, such as Peridot [3] and Druid [8], required the user to confirm each inference, which can be disrupting and annoying. In Marquise, a feedback window (Figure 3) shows the inferred operation. The labels and buttons can be read as a sentence, and the buttons can be pressed to pop up a list of other alternatives and change the values. Since Marquise appears to guess correctly most of the time, Marquise applies the inferred property immediately, and allows the designer to verify or change it afterwards in the feedback window.

ENVIRONMENT

Marquise makes heavy use of many features of Garnet. Garnet uses a retained object model and a prototype-instance object system. This means that there is an object in memory for every object on the screen, and that any object can be used as the prototype to make a copy or instance. Since all Garnet objects are represented the same way, there is a single mechanism for copying and creating objects, whether they are simple rectangles or aggregates containing many components. Therefore, Marquise can always generate appropriate code to create items for run time, without having to know the types of the objects the designer has drawn.

Implementing the behaviors that are demonstrated is quite straightforward once they have been determined because Marquise can create instances of "interactor" objects [5] and fill in the appropriate attributes. Each interactor implements a particular kind of behavior, such as selection, creation, moving, etc., and contains attributes to support most of the popular interaction styles.

The object system supports *constraints*, which are relationships that are declared once and maintained by the system. Constraints are used to maintain the relationships among the graphical objects in Marquise. Constraints can also be used to connect application data to Marquise-generated interfaces, or else application-specific call-back procedures can be invoked when a behavior is completed.

Garnet contains a number of other high-level interactive tools, such as the Lapidary tool for creating individual widgets or objects, the Gilt tool for editing dialog boxes, the Jade tool for automatically creating dialog boxes, and the C32 spreadsheet system for specifying complex constraints. Because all the tools use the same data structures

and file format for describing objects, Marquise does not have to re-implement the functionality already provided by those tools—it can concentrate on the global behavior. The designer can have more than one tool operating at the same time, and use whichever is appropriate for the current part of the task.

FRAMEWORK

Marquise is able to construct the interface from the demonstrations because it has built-in knowledge of the kinds of operations that are common in graphical editors. This knowledge is part of the underlying Marquise framework that supports the interactive front end. The operations include: creating an object of the type in a palette, selecting objects, directly manipulating the size and shape with the mouse, specifying properties of objects (color, font, etc.) with a palette or property sheet, miscellaneous editing operations (deleting, duplicating, etc.), and application-specific commands.

Modes

It is very common in user interfaces for different behaviors to result from the same action, determined either by the location of the initiating event or by the value of a global mode variable. As an example of the first case, in MacProject II for the Macintosh, pressing the mouse button down will start text editing (if inside a box), select a box (if at its edge), create a new box (if in the background), draw a link between two boxes (if pressed in one box and released in another), grow a box (if pressed on a selection handle), or draw a link and create a box (if pressed in one box and released outside). Designers specify these differences in Marquise by demonstrating the different operations. Marquise notices what objects are underneath the demonstrated events (including the mouse release), so it can distinguish the correct times to use the different behaviors. The object being used is shown in the feedback window, and the buttons there can be used to edit the choice.

In other cases, the selection of the behavior is determined by the value of a global mode variable, which is set by a palette or an external application program. To make these modes explicit and visible, Marquise provides a *mode window*, shown in Figure 4, which lists each mode variable and its current value. The values displayed will change as the interface is operated, and the designer can directly edit the values for user modes. When the value has a fixed list of choices, these are available in a pull-down menu. To make an interaction dependent on whether a mode has the current value, it is only necessary to click on the check box next to the mode name before demonstrating the behavior. When a user action causes a mode to change, this can be demonstrated by simply editing the value in the mode window while in Show mode.

The combination of the icons and mode window make the control of the behaviors explicit and direct. In contrast, DEMO II [1] uses multiple examples to determine in which situations the operations should occur, which we feel will be more prone to errors.

Palettes

One of the important innovations in the Marquise framework is the formalization of a *palette*, which is a list of options, usually presented graphically. Each palette controls a single value or mode. Since palettes are conventional interaction techniques internally (i.e., they are usually a list of buttons), their internal behavior (how the user changes the current selection and what feedback shows the selected objects) can be easily specified using Lapidary or Marquise. The innovation here is the automatic connection of the palette to the rest of the interface.

Marquise identifies two main classes of palettes: *create* palettes and *property* palettes. A create palette contains the different kinds of objects that can be created. For example, the create palette for MacDraw II contains a selection arrow, strings, lines, rectangles, rounded-rectangles, ovals, arcs, curves, polygons, and text fields. A create palette for a CAD/CAM program for circuit boards might have a long list of different IC types, plus wires, pads, etc.

Add Edit Display		
System Modes:		
☐	The Created Object	DASHED-LINE-4:02
☐	The Selected Object	NIL
User Modes:		
☒	Create Palette 1	:LINE
☐	Line Style Palette	Solid
☐	Color Palette	Blue
☐	User Mode	FRIENDLY

Figure 4:

The mode window showing the defined modes and their current values. The designer can click on the check box at the left of a row to indicate that the next action depends on the mode having the specified value. Modes based on palettes change as the palette is operated. Applications can also directly set a mode, and one of the actions resulting from a behavior might be a mode change.

A property palette contains the different values for a single property. MacDraw II has a property palette for the filling style at the top of the window, and property palettes for the line style, font size, font style, etc. in pull-down menus. Marquise supports palettes which are not always visible, and a palette can even be a subset of the items in a different widget (e.g., a section of a pull-down menu). In addition, the list of choices can be dynamically changing, for example, if the application has commands to read new libraries or to edit the palette itself.

There are two important distinctions between the two types of palettes. First, the create palette usually enables dif-

ferent interaction techniques. For example, the selection arrow enables selection, the rectangle enables dragging out new rectangles, and the text string enables clicking to start entering text. A property palette is assumed to only set values of properties and not to control interaction techniques.² Note that a create palette can enable various kinds of behaviors, such as selecting and deleting, and not just creating. The second difference between the two types of palettes is that Marquise assumes that objects cannot change type, so that selections in the create palette cannot affect the selected objects. However, clicking in property palettes usually changes the value of that property for the selected objects.

Create Palettes. To make a create palette, the designer only needs to draw the set of objects using Marquise or Lapidary, select them, and declare them to be a create palette using a menu command. Marquise will then add a row to the mode window (Figure 4) showing the palette and its current value. The designer would select the new row in the mode window, click on each item in the palette to put the system into the appropriate mode, and demonstrate the desired behavior.

The create palette has some additional attributes which control common features found in graphical editors. Some of these can be demonstrated, and the rest are specified in dialog boxes.

- In some palettes, when the user clicks on an item, that sets up a mode so that the next operation will create the kind of object represented by that item. This was shown in the example above, and is the way that MacDraw works. In other cases, clicking in the palette causes the object to be created immediately (e.g., at the current mouse position for a popup menu of choices, or at a computed place if the objects are laid out automatically by the system). Other times, objects are dragged off the palette.
- After an object is created, some applications select the newly-created object, some leave the selection unchanged, others add the new object to the selection set (if objects were selected before the create), and yet others clear the selection.
- Sometimes, after the object is created, the mode of the create palette will change. For example, in MacDraw II, after creating a rectangle, the mode changes back to selection (the arrow). However, if you double-click on the palette, the mode does not change after creation. In the original MacDraw, all object creation modes changed back to selection except for text strings.

Property Palettes. Property palettes allow the user to control the value of properties of objects. Typically, the same palette is used for specifying the global default value used for newly-created objects, and for changing the property of selected objects. The same palette might also be used to

show the value for the selected object.

To create a property palette, the designer only needs to draw a set of objects representing the different values (for example, the line-style items of Figure 5), and declare them to be a property palette. Marquise then checks whether a single property seems to change in each element (as it does in Figure 5 and in most graphical property palettes), and if so, proposes this as the property to use. Alternatively, the user can specify the name of the property and the value for each item of the palette.



Figure 5:

After drawing this picture, the designer would select the lines, and declare them to be a property palette. Marquise would notice that the line-style changes, and would create an appropriate palette description.

Each primitive Garnet object describes which properties are relevant to it, and the designer can add additional properties for application-specific objects. Therefore, Marquise can automatically guess which properties are probably relevant to each type of object that is created. These guesses are reflected in the feedback and mode windows (Figures 3 and 4), and if Marquise guesses wrong, then the designer can adjust the values.

Some attributes for property palettes provided by Marquise are:

- Whether setting the property of a particular (selected) object also changes the global default used when a new object is created.
- If an object is selected that does not have the property represented by the palette (e.g., if the palette is for the font property and a line is selected), whether the palette goes inactive (greyed out) or not. When there are multiple objects selected, whether the palette is valid if at least one of the objects has this property, or only when all of the objects have this property.
- When an object is selected, whether the palette shows the value of that object. If more than one object is selected, then the palette might show the value only if all objects have the same value, pick the value of one of the objects to show, show the global default value, or just be cleared to show no value.
- If the palette does not echo the value of the property when the selection changes, then the newly selected object might get the current value of the property (as opposed to requiring another click in the palette after changing the selection).

Positions of new objects

Once Marquise knows which object to create, there is then the question of where and how to create it. There are two

²This restriction could be lifted in the future if it becomes onerous, but it is consistent with the behavior of all editors we have studied.

possibilities: the position is computed automatically, or is specified by the user with the mouse.

Automatic Layout. Garnet has built-in routines for list, table, tree, and graph layout. These automatically place the nodes, rather than requiring the user to specify a location. Each type of layout has a set of methods for creating and deleting nodes, and Marquise allows the designer to demonstrate how these methods are invoked and how their attributes are specified. Many previous systems have allowed a designer to build custom graph layout applications by writing code, but Marquise is the first to allow the look of the nodes to be drawn and the editing behaviors (creating, deleting, editing labels, etc.) to be demonstrated interactively.

First, the designer specifies which kind of layout is desired.³ Next, the designer draws pictures to show the graphics for the nodes (and the graphics for the arcs for trees and graphs). If these have complex internal structure, then the Lapidary tool will be useful for drawing them. The built-in layout algorithms have many attributes that control the display, and some of these can be demonstrated (e.g., the spacing and direction). The rest are specified in a dialog box.

Next, the designer demonstrates the creation behavior. Using knowledge of the type of layout in use, Marquise tries to determine if the new object should be placed in some relation to a selected object, or globally with respect to all objects. For example, in a directed-graph editor, there might be commands for "Add new child" and "Add new parent." Marquise does not try to understand the words in the command names. Instead, the designer would go into Run mode and select a node, and then in Train mode the designer would select the command. Finally, in Show mode, the new object would be created with the correct relationship to the selected node.

In some cases, the new object's position will not depend on the selection, but rather on global properties. For example, the new object might always go at the end of a list. In this case, the designer would make sure that no objects are selected before demonstrating the position of the new object, and Marquise would try to determine the appropriate place for the object. Alternatively, the position might depend on some global mode, so the appropriate row of the mode window would be selected before the demonstration. Usually, the position will be obvious (e.g., first or last), but if Marquise cannot guess it, then currently the designer will have to write a Lisp function to compute the position, possibly based on values in the mode window.

User Layout. Most graphical editors, however, require the user to explicitly specify the position of new objects. The example of Figure 2 shows how the simple case of a new

line can be demonstrated in Marquise.

It is very common for the objects to be constrained in their placement. Marquise has built-in knowledge about grid-ding, so this can be easily used in an application. A more interesting problem is attachment. For example, an arrow connecting the boxes in Figure 6 might always be attached to the centers of the boxes. In an earlier article [4], we discussed how Lapidary allows the arrow prototype to be defined interactively with parameters that refer to the objects to which it should be attached. Lapidary creates constraints that keep the arrows attached as the objects move. Marquise allows the designer to interactively show how those parameters are filled in based on the designer's actions. For example, look back at Figure 1 where a creation palette is being drawn. To demonstrate the arrow creation mode, the designer would select the arrow in the create palette while in Run mode. This will change the value shown in the mode window for the create palette mode (Figure 4). The designer would then click on the check box next to this mode, which tells Marquise that the mode is significant for the next operation.

Assume that the arrow was defined so that setting the *from* and *to* parameters with objects would cause the line to be attached to those objects. In Train mode, the designer would press down inside a rounded-rectangle, and drag outside. Then, in Show mode, the designer would create an instance of the arrow with the shaft end inside the rounded-rectangle and the arrow end at the  icon. Then, the designer specifies that the rounded-rectangle corresponds to the *from* parameter, and Marquise infers that it should determine the parameter value based on where the mouse is first depressed, and that the other end should follow the mouse. Next, the designer demonstrates the mouse button-up response by deleting the feedback line and creating a new arrow between the two nodes. These nodes are declared as the *from* and *to* parameters. In the future, we will provide facilities for *gravity* so the designer could specify that while the mouse is moving, the feedback should jump to the attachment points of objects if they are close enough.

Selection

One of the most important operations in a graphical editor is selecting objects. Typically, the selected object will be shown by changing its appearance (e.g., to reverse video) or by showing "selection handles" around it (Figure 6). Marquise supports virtually any graphical response to show the selection. The designer simply draws an example of the selection graphics (or if the object itself changes, the designer draws the object first in its normal and then in its selected state). If the standard Garnet selection widget is desired, then it is only necessary to go into Show mode and select an object. A special line of the mode window shows which objects are selected, and this value can be edited to show whether the interaction being demonstrated adds to the selection set, removes from it, clears it, etc. This provides a uniform, intuitive mechanism for specifying almost any selection behavior. The designer can also specify whether a different form of feedback is used when there are

³Marquise cannot infer a new layout algorithm. For example, if a new kind of graph layout is required, the designer has to program it in Lisp, but it can then be used by Marquise-generated programs.

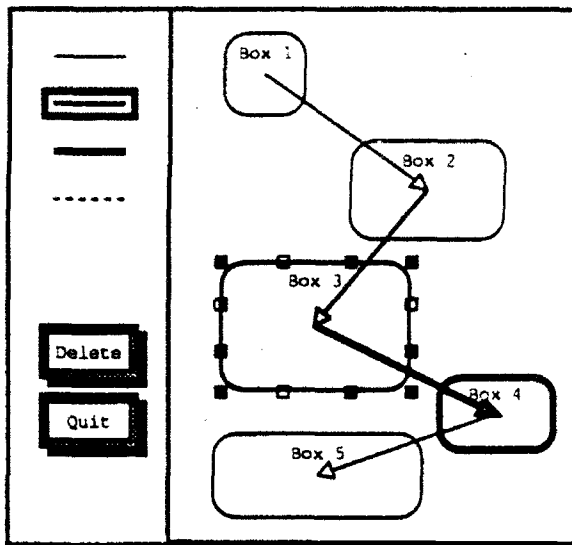


Figure 6:

The arrows are constrained to be in the centers of the boxes. Box 3 has "selection handles" around it, which show that it is selected, and the user can click on white handles to move it or black handles to grow it. The formula that computes the labels was hand-coded using C32.

multiple selections (as in Macintosh PowerPoint and MacProject II).

Moving and Growing Objects

Demonstrating what commands cause objects to be moved and grown works similarly to demonstrating how they are created: first, the designer demonstrates in Train mode what user action causes the interaction to start, and then in Show mode, moves or grows the appropriate object. Since the standard editing actions work in Show mode, the designer would just use the Marquise move-grow selection handles to demonstrate the behavior. Of course, if other objects are attached to the moved object with constraints, they will also move.

One complication is that often the object that the mouse is over is not the object that should be modified. For example, with selection handles, the user clicks on a handle, but wants to grow the object *underneath*. Marquise knows about this special case, and if the object the designer moves is attached by a constraint to the object clicked on, then this is reflected in the generated behavior.

Other Properties of Objects

Many properties of objects are controlled by palettes, but some are not. In some graphical editors, a menu command or double-clicking on an object opens a property sheet or dialog box with other properties. Marquise provides hooks to pop up a property sheet or a dialog box created automatically by Jade or interactively using Gilt. Of course, the designer can specify which fields are presented.

Miscellaneous Editing Commands

Because Garnet uses a retained object model, there is a standard format for all Garnet objects. Therefore, common editing commands such as bringing objects to the top (uncovered), sending to the bottom, cutting, copying, pasting, deleting (clear), duplicating, and printing in PostScript, are all provided. The designer simply demonstrates what action causes it to occur, and then which operation is desired. Note that unlike other frameworks that provide messages that must be overridden by each application, the code provided by Marquise for these operations can often be used without change.

Semantic Actions

Naturally, many of the commands in a graphical editor will invoke application-specific functions (sometimes called "semantic actions"). Since these may involve arbitrary computation, it is impossible for Marquise to infer these from a demonstration. However, techniques like those previously reported for Gilt [7] are used to allow the application procedures to be independent of the way they are invoked (from a button, menu, double-click, etc.) and somewhat independent of the graphics. However, most functions will want to walk through the graphical objects computing values, so they will clearly have to look at the graphical objects in the window.

If the result of the function is a change to the graphic appearance of nodes, then this can be specified demonstratorially. For example, a "critical-path" command in a graph editor might want all the nodes on the critical path to turn red. The designer can bring up a property sheet on the nodes, add an on-critical-path property,⁴ and demonstrate that the nodes are black when it is NIL and red when it is T. Then, the critical-path function would only be responsible for setting the on-critical-path value in each node. This makes the application function more independent of the graphical response to its actions.

Semantic feedback can often be provided in the same way. For example, Marquise supports highlighting of only those objects that an object is being dragged can legally be dropped into, as in the Macintosh Finder. Here, a function could be called to set a particular property of each object to T or NIL. Then, the designer would demonstrate the appropriate color change when the node is over an object which has the value T for that property.

Similarly, if the application wants to control which mode is in effect, it can simply change the value of one of the mode variables, and the designer can demonstrate interactively what this controls.

EDITING

An important aspect of an interactive builder is how to edit the interfaces after they have been created. It is easy to edit the graphics, since they can be directly manipulated in Build mode. For the behaviors, the feedback window of

⁴The Garnet object system allows properties to be added to objects at any time.

Figure 3 shows the properties. When in Train mode, the feedback window continually shows the name and properties of the behaviors being executed, so the designer can determine which behaviors are associated with which events. There are also commands to list all the behaviors, or all those affecting a particular object.

CONCLUSION

One of the important questions for an interactive tool is what is the range of interfaces that it can create. Unfortunately, this is very difficult to quantify, except by example. Using the Lapidary, Gilt and Marquise tools in Garnet, it is possible without programming to create complete user interfaces like those in Macintosh MacDraw, MacDraw II, PowerPoint, and MacProject II (which are surprisingly different), as well as applications with various kinds of automatic layout for nodes. Later, we hope to expand the range of Marquise to handle gestural interfaces (the Garnet toolkit already supports gesture recognition), and those with 3-D graphics. We also plan to add support for animations, which will probably make possible the demonstration of various visualizations and video games. Another addition will be to support defining constraints among objects directly in Marquise, probably using demonstrational techniques similar to Peridot [3] or Druid [8].

Marquise is still under development. When it is more robust, we will perform user-testing to see if the demonstrations and feedback are understandable to both non-programmers and programmers. After that, we will release it for general use as part of the Garnet system. All this will help show what kinds of behaviors it can capture, and we will continually work to expand the range.

We believe that interactive, demonstrational creation of user interfaces is easier, faster, and more fun than programming. Many interactive builders have already shown that dialog boxes and forms can be created interactively. Marquise shows that direct manipulation techniques can be used to generate the user interfaces of a much wider class of graphical applications as well.

ACKNOWLEDGEMENTS

For help with this paper, we would like to thank Dario Giuse, Brad Vander Zanden, Andrew Werth, and Bernita Myers.

This research was sponsored by the 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.

The views and conclusions contained in this document are

those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

1. Gene L. Fisher, Dale E. Busse, and David A. Wolber. Adding Rule-Based Reasoning to a Demonstrational Interface Builder. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'92, Monterey, CA, Nov., 1992, pp. 89-97.
2. Anthony Karrer and Walt Scacchi. Requirements for an Extensible Object-Oriented Tree/Graph Editor. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'90, Snowbird, Utah, Oct., 1990, pp. 84-91.
3. Brad A. Myers. *Creating User Interfaces by Demonstration*. Academic Press, Boston, 1988.
4. 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.
5. Brad A. Myers. Encapsulating Interactive Behaviors. Human Factors in Computing Systems, Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 319-324.
6. Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces". *IEEE Computer* 23, 11 (Nov. 1990), 71-85.
7. 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., 1991, pp. 211-220.
8. Gurinder Singh, Chun Hong Kok, and Teng Ye Ngan. Druid: A System for Demonstrational Rapid User Interface Development. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'90, Snowbird, Utah, Oct., 1990, pp. 167-177.
9. John M. Vlissides and Mark A. Linton. Unidraw: A Framework for Building Domain-Specific Editors. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'89, Williamsburg, VA, Nov., 1989, pp. 158-167.
10. David Wolber and Gene Fisher. A Demonstrational Technique for Developing Interfaces with Dynamically Created Objects. ACM SIGGRAPH Symposium on User Interface Software and Technology, Proceedings UIST'91, Hilton Head, SC, Nov., 1991, pp. 221-230.

Screen Shots from Selected Garnet Applications

ted by
Brad A. Myers

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

As of the winter of 1993, Garnet has been used by over 40 projects from all over the world. The following pages show pictures of a few of those applications. None of the applications shown in the pictures were developed by the Garnet group—they are all the work of other projects, and the pictures and text were provided by the developers. Most were generated by Garnet's code that produces Postscript files directly from the graphics on the screen. If you are using Garnet and have interesting screen shots, please send them along with a description of your project, to garnet@cs.cmu.edu.

Many of these pictures were originally in color. If you have a color printer and would like to see the original pictures, they can be retrieved by anonymous FTP from [a.gp.cs.cmu.edu](ftp://a.gp.cs.cmu.edu) in directory `/usr/garnet/garnet/doc/usegarnet/`.

Companies

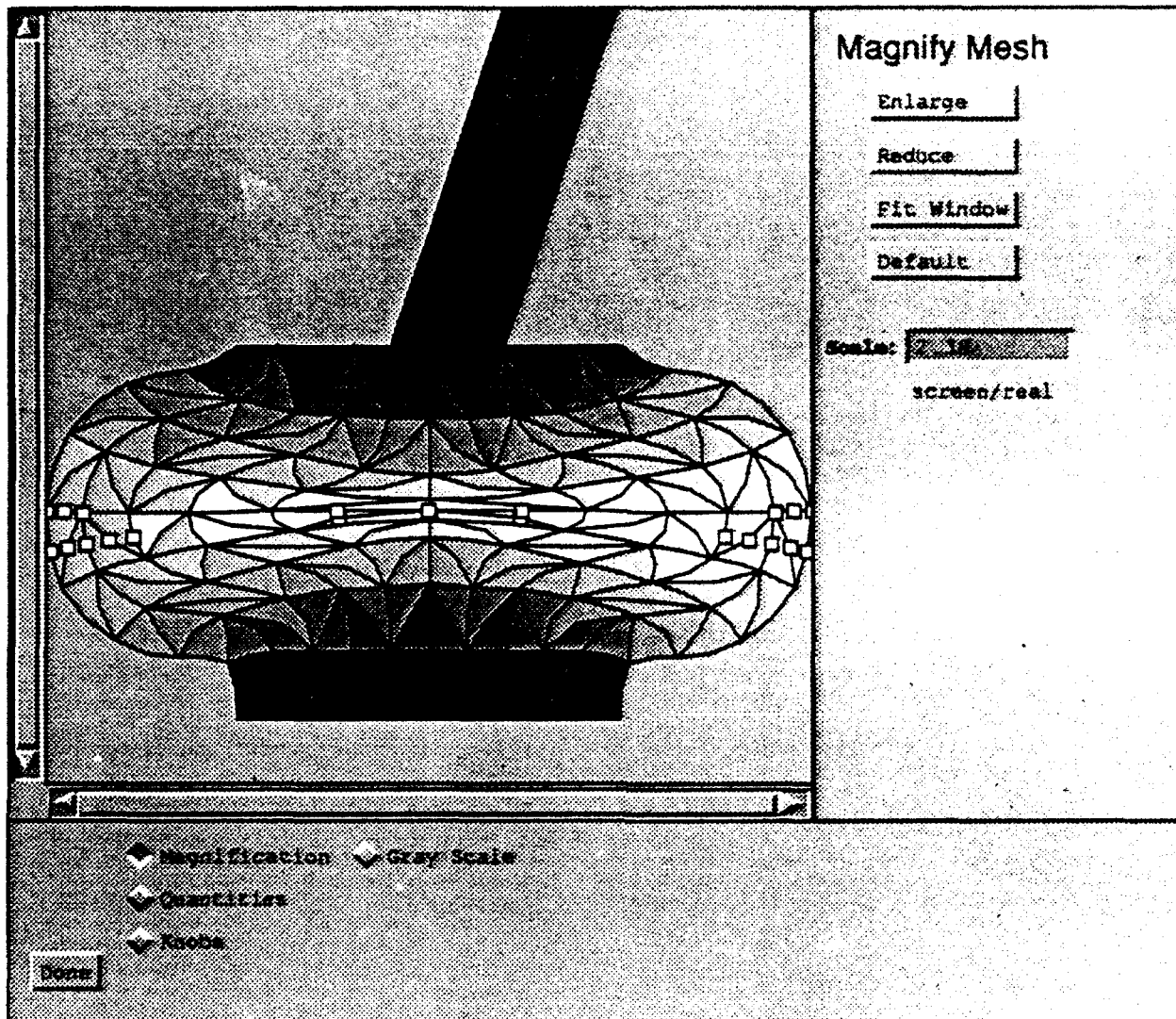
1. Corporation For Open Systems
Automated Protocol Analysis/Reference Tool (APART)
Frank J. Wroblewski
2. Design Research Institute (Cornell & Xerox)
(various)
Jim Davis
3. Deutsches Forschungszentrum fuer Kuenstliche Intelligenz GmbH
COSMA
Stephen P. Spackman
4. GE Research and Development Center
Metallurgical Expert Systems for Manufacturing
K. J. Meltner
5. GE Research and Development Center
SKETCHER
K. J. Meltner
6. Hughes AI Center
NLP Project
Seth Goldman or Charles Dolan
7. Lawrence Livermore National Lab
PLANET (Pump Layout AND Evaluation Tool)
Tom Canales
8. Microelectronics and Computer Technology Corp. (MCC)
Scan: Intelligent Text Retrieval
Elaine Rich
9. The MITRE Corporation
AIMI (An Intelligent Multimedia Interface)
John D. Burger
10. National Science Center Foundation
Learning Logic
Lawrence Freil
11. StatSci
GRAPHICAL-BELIEF
Russell Almond
12. The MITRE Corporation
AIMI (An Intelligent Multimedia Interface)
John D. Burger
13. Transarc Corporation
Lisp GUI for Encina
Mark Sherman
14. U S WEST Advanced Technologies
KBNL natural language system
Randall Sparks
15. USC/ISI
Humanoid
Pedro Szekely
16. USC/ISI
SHELTER knowledge-based development environment
Pedro Szekely

Universities

17. Carnegie Mellon University, CS
Miro
J. D. Tygar and J. M. Wing
18. Carnegie Mellon University, CS
Learning Calendar System
Conrad Poelman
19. Carnegie Mellon University, CS
Interactive Fiction Editor
Merrick Furst
20. Carnegie Mellon University, CS
Redstone
Jeff Schlimmer
21. Carnegie Mellon University, CS
Architectural Design
Mikako Harada

22. Carnegie Mellon University, CS
PURSUIT
Francesmary Modugno
23. Carnegie Mellon University, ERDC
LOOS
Ulrich Flemming or Robert Coyne
24. Carnegie Mellon University, Math
Educational Theorem Proving System
Peter Andrews
25. Carnegie Mellon University, Psych
Soar Graphics Interface
Frank Ritter
26. Carnegie Mellon University, Robotics
MICRO-BOSS
Norman Sadeh
27. Carnegie Mellon University, Robotics
SAGE Data Visualization
Steve Roth
28. MIT, Dept. of Brain and Cognitive Sciences
SURF-HIPPO Neuron Simulator
Lyle J. Borg-Graham
29. MIT, LCS, Computation Structures Group
Debugging tools for the Id Language
Steve Glim and R. Paul Johnson
30. State University of New York at Buffalo, CS
Air Battle Simulation
Henry Hexmoor
31. State University of New York at Buffalo, CS
SNePS Graphical UI
John S. Lewocz
32. Tulane University, CS
Natural Language Processing
Robert Goldman
33. Tulane University, CS
THESEUS
Raymond Lang
34. University College London
The Cognitive Browser
Gordon Joly
35. University of Leeds
Graphical Multi-User Domain Designer
Roderick J. Williams
36. University of Leeds
CLARE
Nikos Drakos
37. University of Leeds
ADVISOR
Andrew J. Cole
38. University of Leeds
PORSCHE
Colin Tattersall
39. University of Saskatchewan
DISCUS (Distributed Computing at the U of S)
Beth Protsko (User Interface only)
40. University of Southern California
Dynamic Aggregation in Qualitative Simulation
Nicolas Rouquette
41. University of Washington, CS
Multi-Garnet
Michael Sannella
42. University of Washington, CS
Electronic Encyclopedia Exploratorium
Mike Salisbury

A list of some of the projects that have used or are using the Garnet user interface development environment, as of February, 1993. If you are using Garnet and your project is not here, please send us mail!

**Kenneth Meltsner**

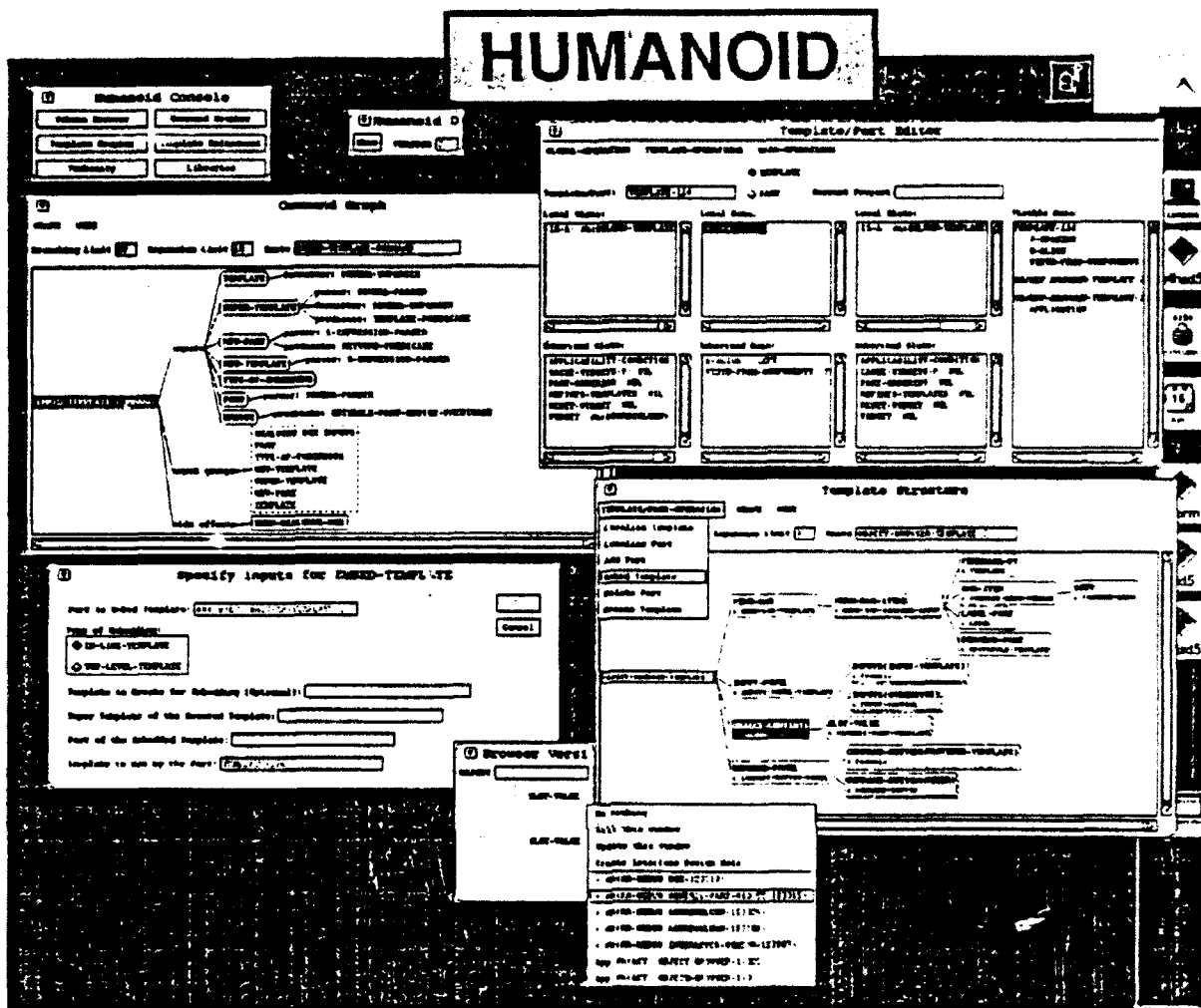
General Electric Company, Corporate Research and Development

Metallurgical Expert System

meltsner@crd.ge.com

A mesh created using a virtual aggregate for the polygons and another virtual aggregate for the square knobs. For the polygons, the virtual aggregate is passed a prototype for a polygon, and an array containing the list of points and the color for each polygon. The virtual aggregate then pretends to allocate an object for each element of the array, but actually just draws the prototype object repeatedly.

Kenneth J. Meltsner. "A Metallurgical Expert System for Interpreting FEA," *Journal of Metals*, Oct. 1991, vol. 43, no. 10, pp. 15-17.

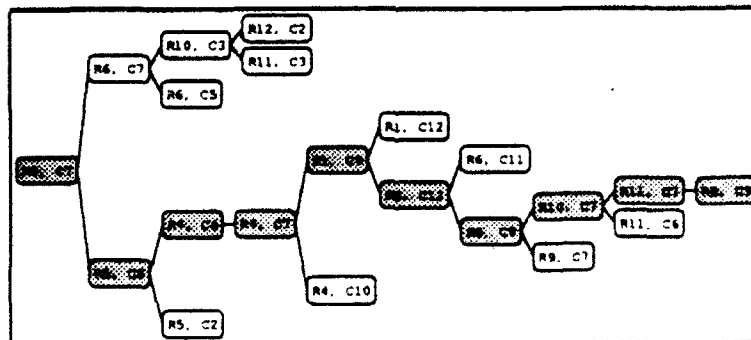
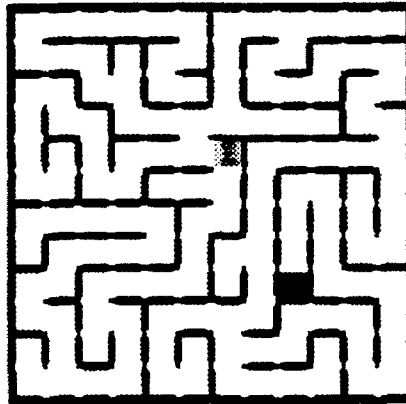


Pedro Szekely
 USC/ISI
 Humanoid
 szekely@isi.edu

Humanoid is a user interface design environment. The goal of Humanoid is to allow interface designers to incrementally construct interfaces by composing building blocks (garnet gadgets and interactors). Humanoid allows designers to specify the conditions when gadgets and interactors are appropriate for displaying/interacting with information. Given an application data structure, Humanoid constructs, at run-time, a display appropriate for interacting with the given data structure. Humanoid keeps track of how the display depends on the input data, so that if the data changes at run-time, Humanoid can automatically update/reconstruct the display.

Pedro Szekely, Ping Luo, and Robert Neches, "Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design," *Proceedings SIGCHI'92: Human Factors in Computing Systems*, Monterey, CA, May, 1992, pp. 597-515.

12	Number of Rows	☒ Leave a Trail
12	Number of Cols	☐ Pause at Choice Points
25	Room Width	☐ Turn Toward Exit
25	Room Height	☐ Depth First Search
6	Wall Thickness	☐ Breadth First Search
New Maze	☐ Allow Loops	☐ Best First Search
QUIT THESEUS	Graph Each Room	Start Search



Raymond Lang

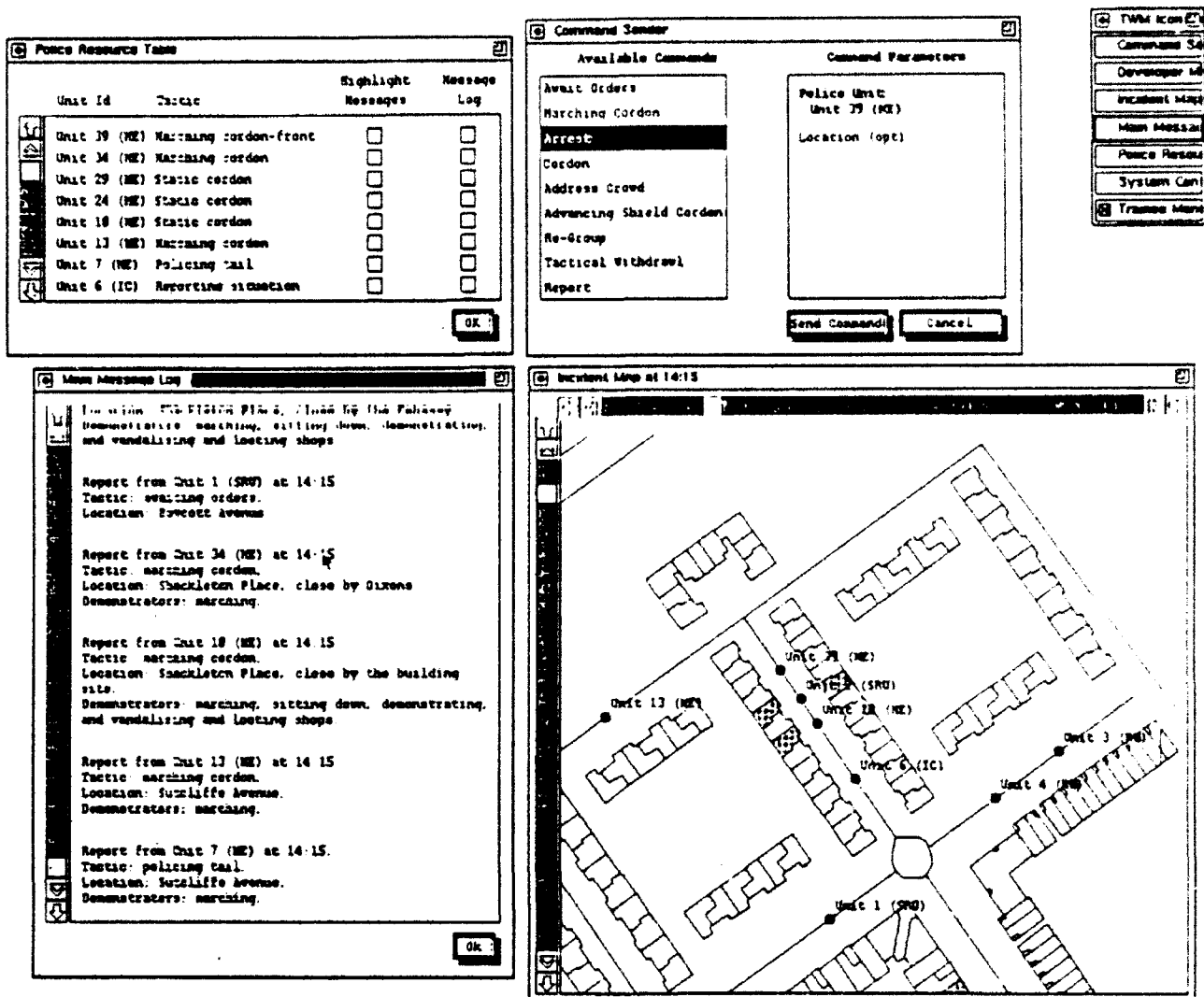
Tulane University, Computer Science Department

THESEUS

lang@rex.cs.tulane.edu

These are images of windows from the *THESEUS* application used by the Tulane University Computer Science Department on guided tours of the department given to visiting high school seniors and other interested parties. *THESEUS* is intended to be used as part of a presentation on what the study of computer science entails. It does this by showing graphically the progress and results of common search methods applied to the problem of finding the exit of a randomly created maze. *THESEUS* was developed in CMU Common Lisp version 16d and the Garnet X-Windows toolkit version 2.01.

R. Raymond Lang, *THESEUS: Using Maze Search to Introduce Computer Science*. Technical Report ****. Computer Science Department, Tulane University. ****, 1992.



Roderick J. Williams

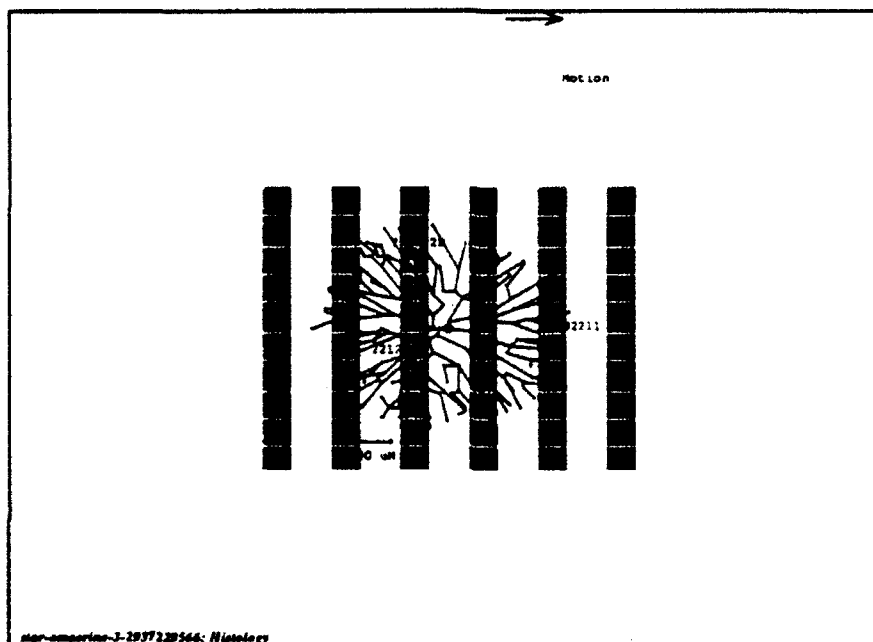
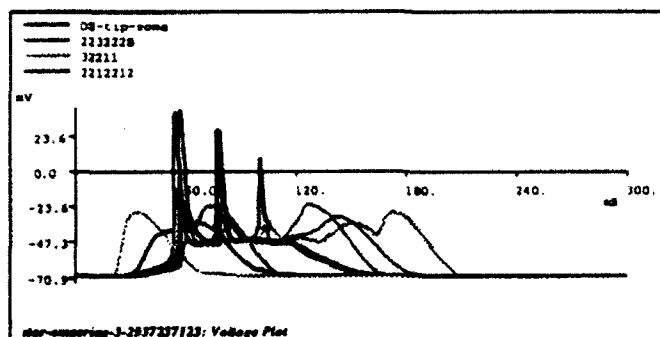
The University of Leeds, Leeds, LS2 9JT, UK.

Cactus

rodw@cbl.leeds.ac.uk

A system has been developed to train senior police officers to manage public order incidents, such as marches. The system enables pre-demonstration planning, the management of (simulated) events requiring meta- and contingency-planning, and post-event debriefing. The training incidents are generated by interactions between autonomous agents, and take place in a simulated world derived from digital map data. In addition to the training environment there are facilities to graphically specify the agents' behaviours.

Hartley, R.J., Ravenscroft, A. and Williams, R.J. "Cactus: Command and Control Training Using Knowledge-based Simulations." *Interactive Learning International*, Vol. 8, no. 2, 1992. pp. 127-136.



Lyle J. Borg-Graham

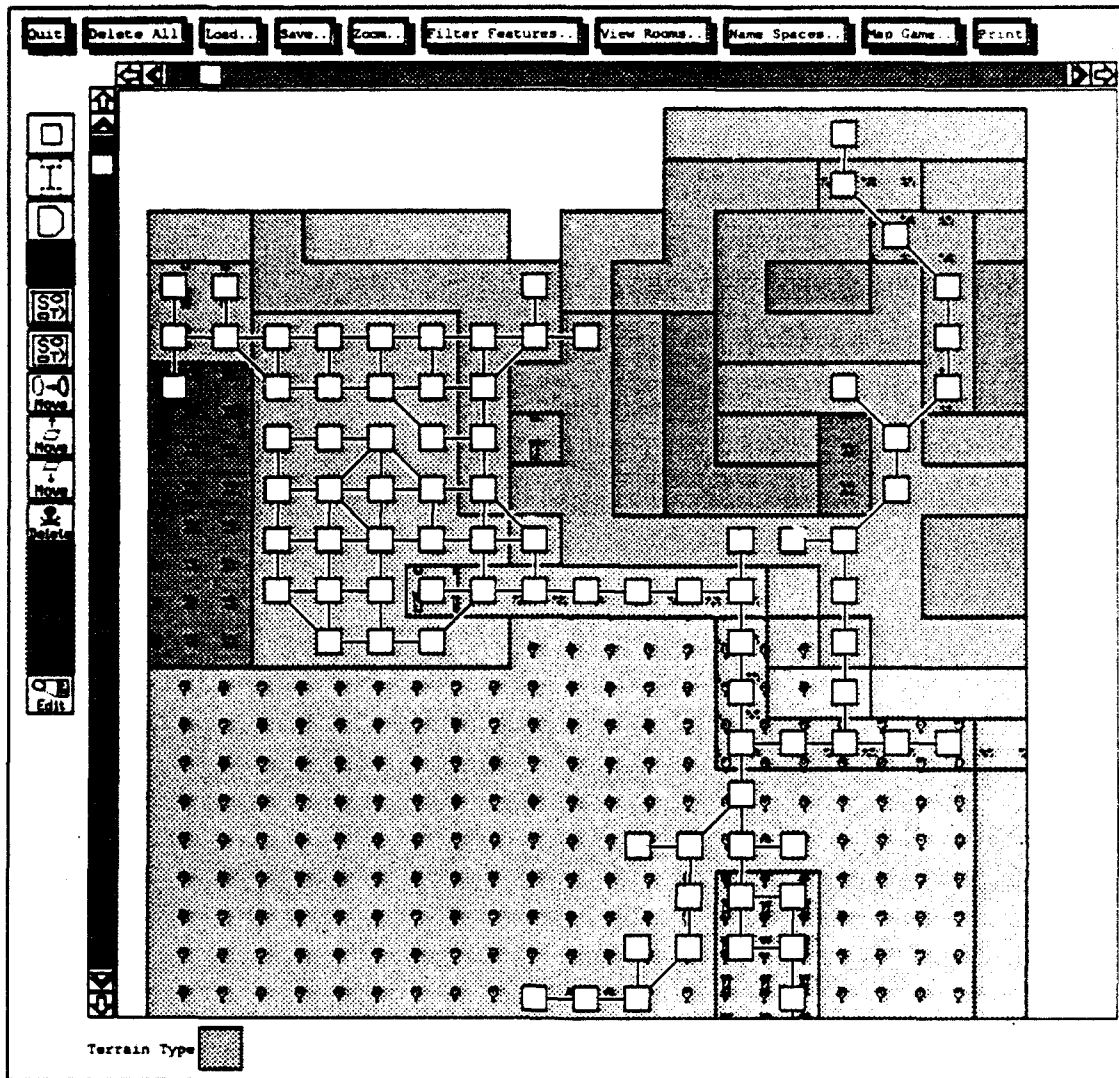
MIT Dept. of Brain and Cognitive Sciences

Surf-Hippo

lyle@ai.mit.edu

The SURF-HIPPO Neuron Simulator is a circuit simulation package for investigating morphometrically and biophysically detailed models of single neurons and small networks of neurons. SURF-HIPPO allows ready construction of multiple cells from various file formats, which can describe complicated dendritic trees in 3-space with distributed non-linearities and synaptic contacts between cells. Cell geometries may also be traced from the histology directly on the screen, using the mouse. An extensive user interface is provided, including menus, 3D graphics of dendritic trees, and data plotting. Data files may also be saved for analysis with external tools. A research version of SURF-HIPPO (available by anonymous ftp from ftp.ai.mit.edu [pub/surf-hippo]) is written in LISP, and is configured to run using the public domain CMU Common Lisp and Garnet packages. Our version is compiled for SPARC workstations, and should be easily ported to other UNIX machines running X. LISP is a useful simulator language because it has the benefits of a powerful interpreted script language, but it may also be compiled. Thus it is convenient to integrate custom code into SURF-HIPPO. The simulator may also be used with a minimum of programming expertise, if desired.

Borg-Graham, L. and Grzywacz, N. M. "A Model of the Direction Selectivity Circuit in Retina: Transformations by Neurons Singly and in Concert," in *Single Neuron Computation*, edited by T. McKenna, J. Davis, and S. F. Zornetzer. Academic Press, 1992.



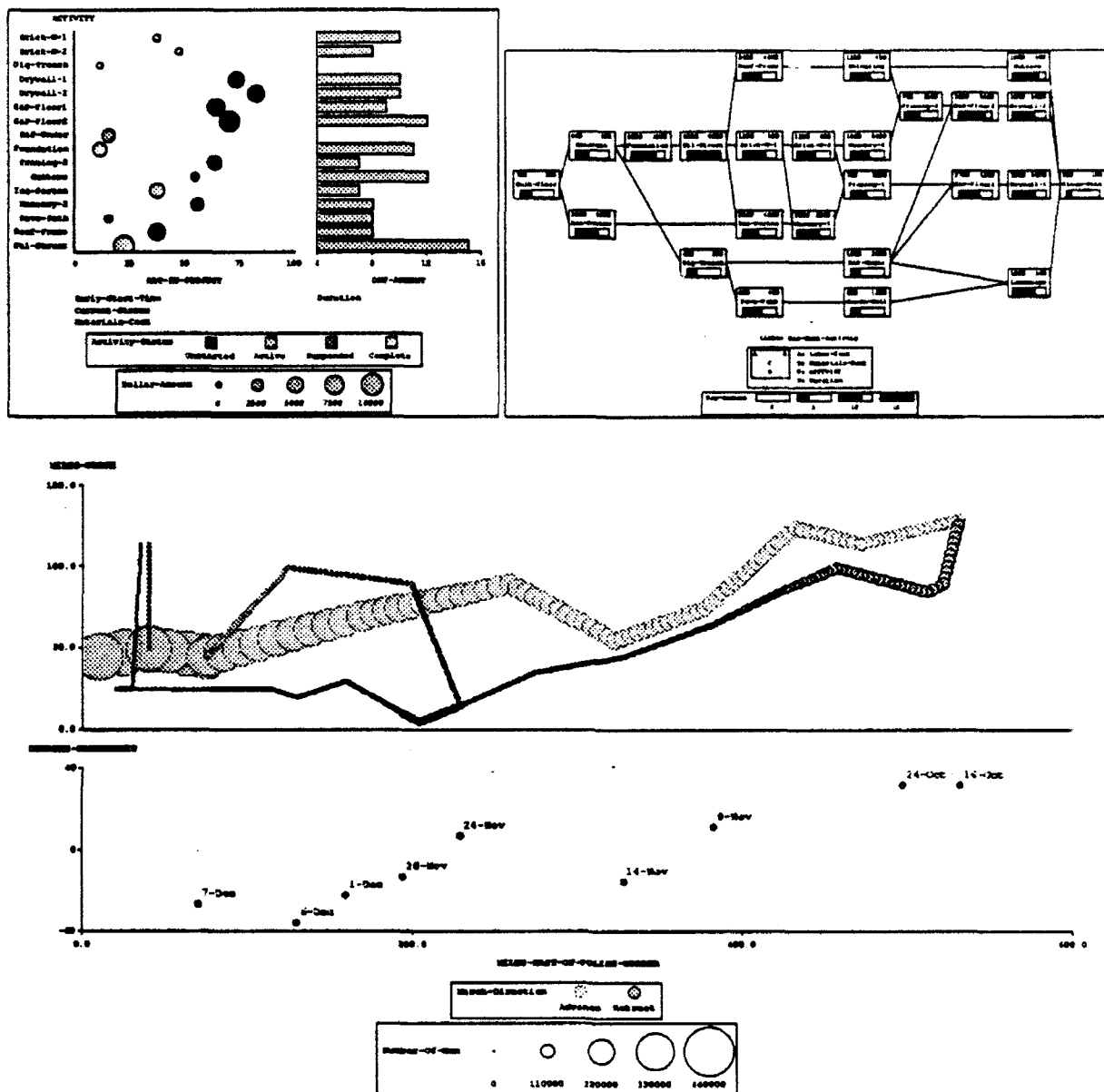
Roderick J. Williams

The University of Leeds, Leeds, LS2 9JT, UK.

GMD (Graphical Mud (Multi-User Domain) Designer)

rodw@cbl.leeds.ac.uk

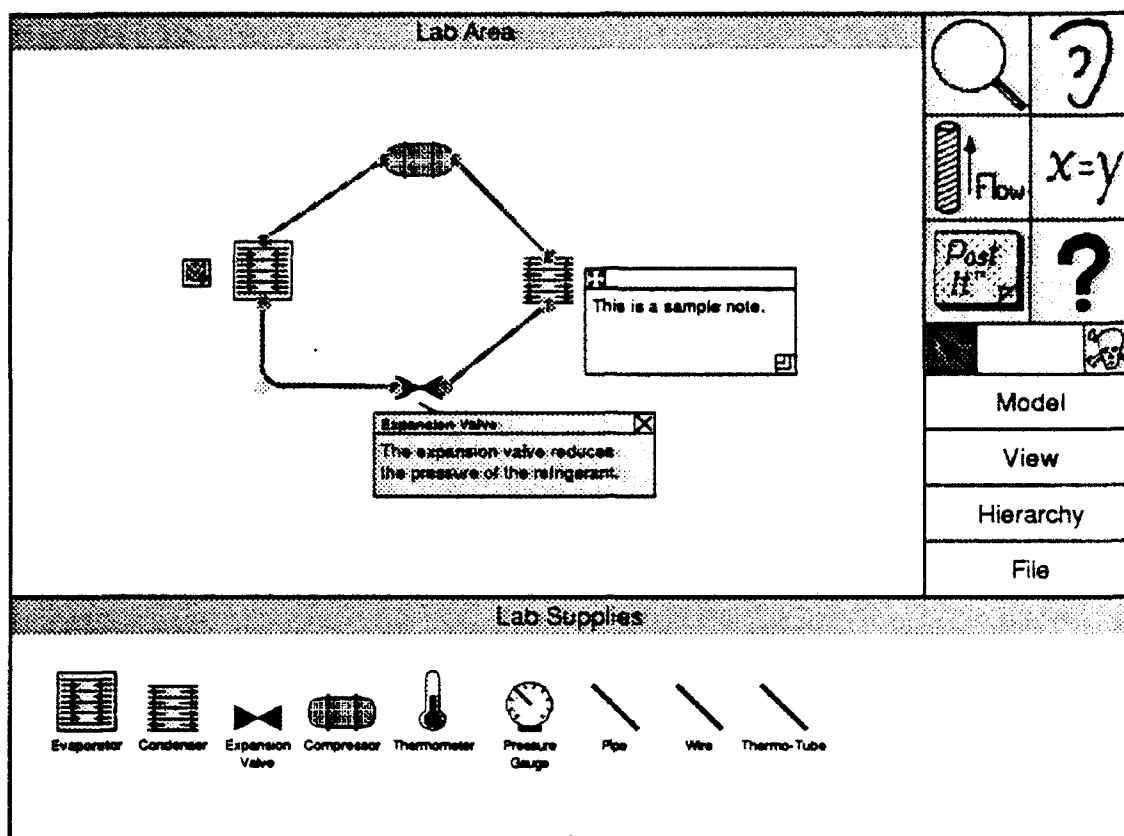
This application is aimed at supporting the creation of text-based multi-user domains. Current techniques use text-based tools to create these environments, but these tools have very little computer support so complexity and consistency are sacrificed. Our new application supports the graphical creation of MUD areas and enforces topological constraints together with hierarchical grouping of features. The graphical tool can be used in a number of modes which allow the information to be filtered, zoomed and viewed in 2.5 D. Areas created can be printed and additionally they can be saved as native code that can be executed.



Steven F. Roth
 Carnegie Mellon University, Robotics Institute
 SAGE
 roth@is11.ri.cmu.edu

The SAGE project is developing systems which automate the process of designing presentations of information. An automatic presentation system is an intelligent interface component which receives information from a user or application program and designs a combination of graphics and text that effectively conveys it. Its purpose is to assume as much responsibility for designing displays as required by a user, from layout and color decisions to broader decisions about the types of charts, tables and networks that can be composed within a display. The SAGE project is developing an interactive data exploration environment which contains automatic display design capabilities integrated with data navigation, manipulation and modification tools. These tools are being used to explore large amounts of diverse data from marketing, logistical, real estate, census and other databases.

Roth, S.F. & Mattis, J.A. "Data Characterization for Intelligent Graphics Presentation", In *CHI'90: Proceedings of the ACM SIGCHI Conference on Computer Human Interaction*, Seattle, April, 1990, pages 193-200.



Mike Salisbury

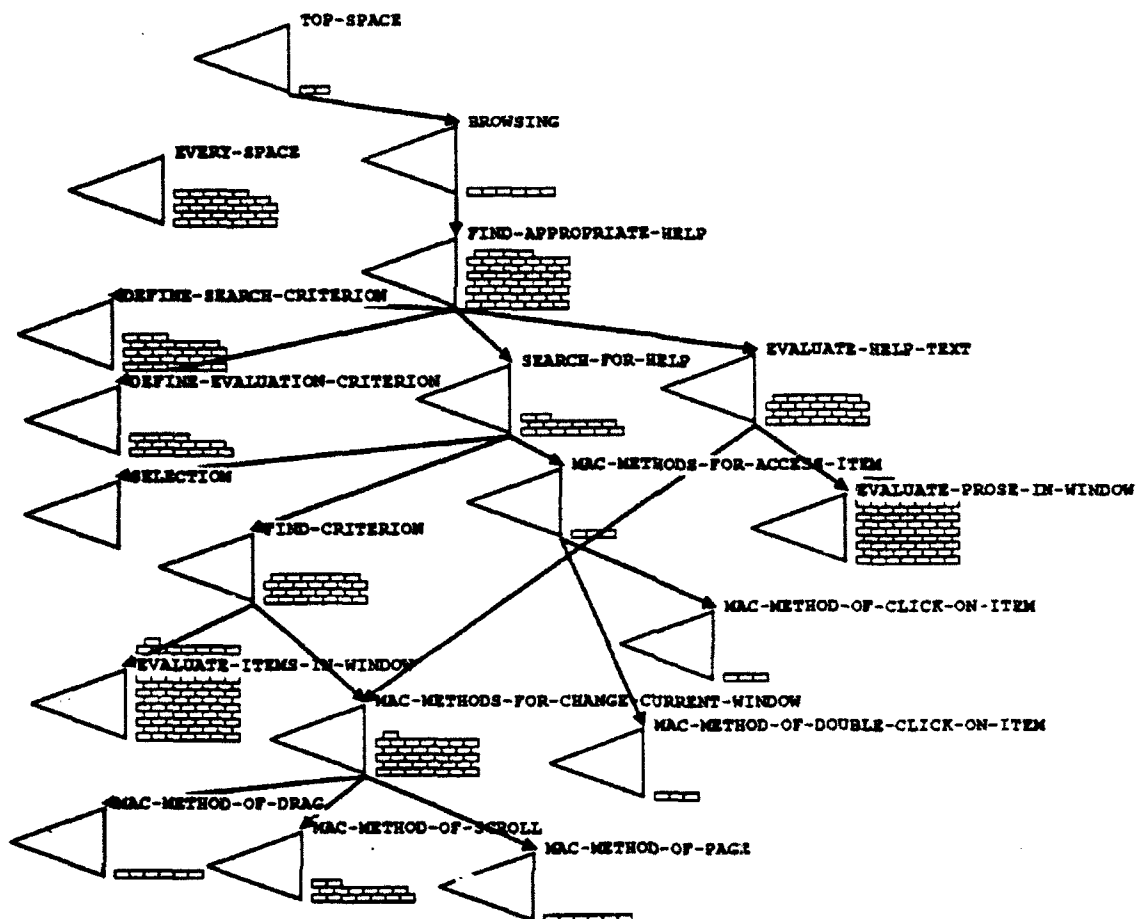
University of Washington, Department of Computer Science

Electronic Encyclopedia Exploratorium

salisbur@cs.washington.edu

The Electronic Encyclopedia Exploratorium is an electronic how-things-work book. It allows the user to learn about devices by experimenting with the components of those devices in a lab simulation setting. A causal model simulator lies beneath the user interface which simulates the current device and can provide causal explanations of the results of that simulation. Other high-level tools are planned for future enhancement.

F. G. Amador, D. Berman, A. Borning, T. DeRose, A. Finkelstein, D. Neville, Norge, D. Notkin, D. Salesin, M. Salisbury, J. Sherman, Y. Sun, D. S. Weld, and G. Winkenbach. *Electronic "How Things Work" Articles: A Preliminary Report*. University of Washington, Department of Computer Science and Engineering Technical Report 92-04-08. June, 1992.



Frank E. Ritter

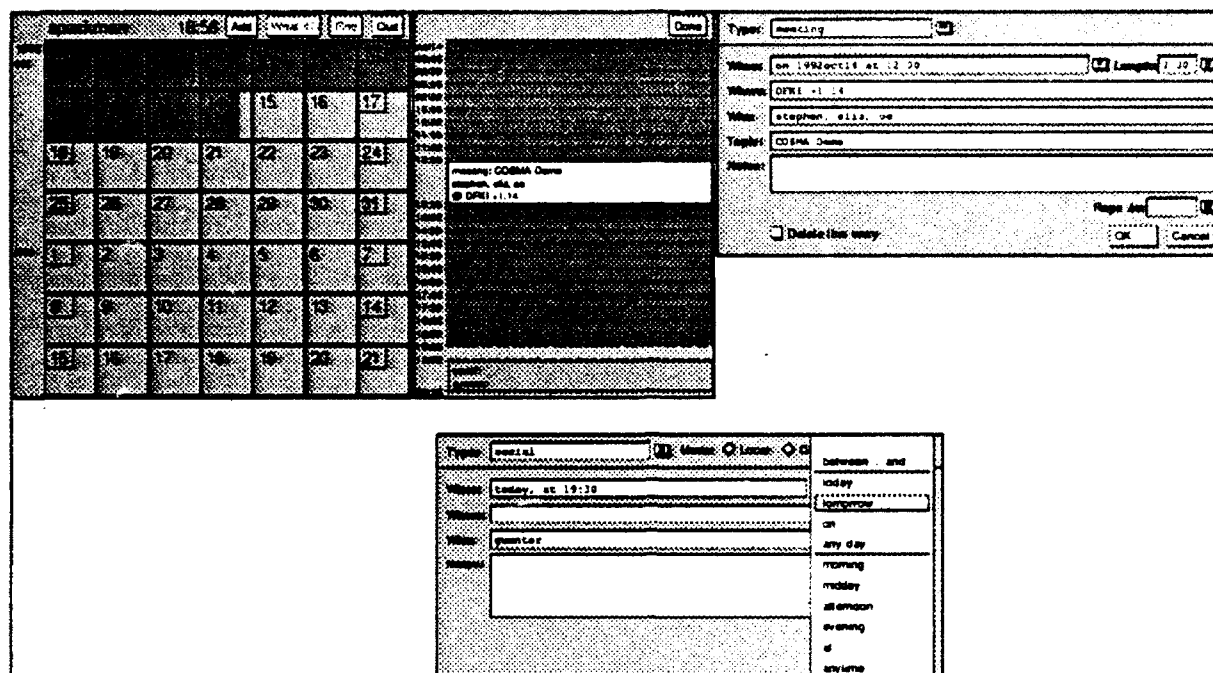
Department of Psychology, U. of Nottingham

The Developmental Soar Interface

Ritter@psyc.nott.ac.uk

The Developmental Soar Interface provides a graphical and textual interface to observe and modify models (programs) for Soar, an AI programming language that also realizes a unified theory of cognition. Garnet is used to graphically represent Soar's goal stack and internal state, and to help users modify and observe structures in Soar.

Ritter, F. E. (1993) *TBPA: A methodology and software environment for testing process models' sequential predictions with protocols*, PhD thesis, Department of Psychology, Carnegie-Mellon University. Reprinted as techreport CMU-CS-93-101, Carnegie-Mellon University.



Stephen P. Spackman

Projekt DISCO

Deutsches Forschungszentrum fuer Kuenstliche Intelligenz GmbH

COSMA, the CoOperative Scheduling Management Agent

spackman@dfki.uni-sb.de or stephen@acm.org

The calendar window shows the dark bar of the past sweeping, one pixel each half hour of the day and night, across a horizontal line [not visible in this Postscript image] summarising by its width and height the user's working hours and appointments, tentative and firm. The marginal time tags can be dragged up and down, and it will eventually be possible to type over the top of them to jump to a given time. The datebook window presents an expanded view of time as an infinite tape from which appointment forms can be popped up by pointing or by sweeping out free areas. Most importantly, when arrangements involve several people the system communicates with its peers and with meeting participants by reading and writing email in German; the displays are updated in real time.

The fields of the appointment form are semi-structured: they can be filled in with the help of menus - such as that visible on the lower window - that drop down from the small icons on the right; numeric, date and time values within them can be incremented and decremented directly with mouse buttons; and experienced users can type structured values straight in. Unconstrained German text (the graphic interface will soon be English/French/German trilingual, but the natural language parser and generator speak only German) can also be entered. It is routed to the natural language system for analysis; planned improvements to the pragmatics module will allow you to give up on the structured form completely and type informal questions and instructions into the notes field, as you might for a human secretary who had stepped out of the room.

The work underlying this picture was supported by a research grant, FKZ ITW 9002 0, from the German Bundesministerium fuer Forschung und Technologie to the DFKI project DISCO.

Elizabeth A. Hinkelman and Stephen P. Spackman, "Abductive Speech Act Recognition, Corporate Agents and the COSMA System," in *Abduction, Beliefs and Context: Proceedings of the second ESPRIT PLUS workshop in computational pragmatics*. W. J. Black and G. Sabah and T. J. Wachtel, eds. Academic Press, 1992.