Lessons Learned from Users' Experiences with Spreadsheet Constraints in the Garnet and Amulet Graphical Toolkits

BRADLEY T. VANDER ZANDEN and RICHARD HALTERMAN*
University of Tennessee

BRAD A. MYERS and ROB MILLER
Carnegie Mellon University

PEDRO SZEKELY
USC/Information Sciences Institute

DARIO A. GIUSE
Vanderbilt University Medical Center

DAVID KOSBIE
Sewickley Academy

RICH MCDANIEL
Siemens Technology-To-Business Center

May 31, 2002

*This research was supported in part by the National Science Foundation under grants CCR-9633624 and CCR-9970958.
Authors' Addresses: B. Vander Zanden, Computer Science Department, University of Tennessee, Knoxville, TN 37920, email: bvz@cs.utk.edu. R. Halterman, School of Computing, Southern Adventist University, P.O. Box 370, Collegedale, TN 37315, email: haltermn@cs.southern.edu. B. Myers and R. Miller, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, email: brad.myers, rcm@cs.cmu.edu. P. Szekely, USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292, email: szekely@isi.edu. D. Giuse, Vanderbilt University Medical Center, 2209 Garland Avenue, Nashville, TN 37232-8340, email: Dario.Giuse@vanderbilt.edu. David Kosbie, Sewickley Academy, 315 Academy Avenue, Sewickley, PA 15143, email: dkosbie@yahoo.com. R. McDaniel, Siemens Technology-To-Business Center, 1995 University Avenue, Suite 375, Berkeley, CA 94704, email: richm@ttb.siemens.com.

Summary

Spreadsheet-like constraints have been incorporated in many graphical user interface toolkits because they are simple to learn, easy to write, and can express many types of useful graphical relationships. The existing papers on spreadsheet constraints have focused on their design and implementation. In contrast, this paper is an evaluative paper that examines users' experience with two of these toolkits, Garnet and Amulet, over a 10 year time span. The lessons gained from this examination can help guide the design of future constraint systems. The most important lessons are that:

1. constraints should be allowed to contain arbitrary code that is a) written in the underlying toolkit language, and b) does not require any annotations, such as parameter declarations,
2. constraints are difficult to debug and better debugging tools are needed, and
3. programmers will readily use constraints to specify the graphical layout of an application but must be carefully and time consumingly trained to use them for other purposes.

keywords: One-way Constraints, Constraint Experience, Constraint Usage, Graphical Interfaces, Interface Toolkits.

Introduction

The popularity of spreadsheets and their ease of use has encouraged many researchers to include spreadsheet-like constraints in toolkits for constructing graphical user interfaces [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. In turn these implementations have spawned many articles describing algorithms for solving these constraints or evaluating the trade-offs among these algorithms [5, 11, 12, 13, 14, 15, 16, 17].

A pair of "retrospective" papers have also been published recently that report on longer-term experiences with constraints. A companion paper to this one provides an empirical comparison of the performance and design trade-offs of various constraint satisfaction algorithms based on our experiences with the toolkits described in this paper [18]. A second paper mentions briefly two of the observations described in this paper: 1) the fact that programmers have found constraints useful for graphical layout and 2) the fact that many programmers find constraints somewhat difficult to master in other settings due to their declarative, rather than imperative, nature [19]. However, it provides no documentary evidence for these observations, does not explore them in any detail, and does not report the other findings reported in this paper.

Despite this wealth of papers on spreadsheet-style constraints, none of them has provided an in-depth account of programers' experiences with these constraints after the toolkits have been released and in use for several years. This paper remedies that gap in the literature. It describes: 1) what programmers like and dislike about constraints, 2) how programmers use constraints in applications, and 3) how we learned to make constraints easier for programmers to use. The results are based on 10 years of experience gained

from working with users of the Garnet and Amulet toolkits [3, 4] and from surveys of these users. Garnet is a Lisp-based toolkit for developing interactive graphical applications that was first released in 1989 and Amulet is a C++-based successor to Garnet that was released in 1994. Garnet can be downloaded from `www.cs.cmu.edu/~garnet` and Amulet can be downloaded from either `www.cs.cmu.edu/~amulet` or `www.openip.org`. Both toolkits incorporate spreadsheet-like constraints and have been used by over 1,000 programmers. Garnet runs on the Unix and Macintosh platforms, and Amulet runs on the Unix, PC, and Macintosh platforms.

Overall programmers' experience with constraints have been quite positive. They consistently indicate that constraints are useful in constructing their applications. Their biggest praise of constraints is that they are helpful for specifying graphical layout. Their biggest complaints about constraints is that they are difficult to debug and can be evaluated in an unpredictable fashion (this latter complaint is discussed in our previous paper on constraint performance [18] and is not discussed further in this paper). The rest of this paper first presents some terminology and then describes these and other findings in greater detail.

Background

Terminology

A *spreadsheet constraint* is an equation in which the value of the variable on the left side is determined by the value of the expression on the right side. For example, the constraint `label.left = frame.left + 10` causes `label's left` to be offset 10 pixels from the left side of `frame`. More formally, a spreadsheet constraint is an equation of the form

$$v = F(p_0, p_1, p_2, \ldots, p_n)$$

where F is a function and the p_i's are its parameters. In a conventional spreadsheet, F would denote the formula and the p_i's would denote the cells referenced by the formula. If the program or the user modifies any of the p_i's, the constraint solver automatically re-evaluates F and assigns the updated value to v. In some cases, v can be modified by the program or the user and in this case the constraint is left temporarily unsatisfied.

Garnet and Amulet Overview

Amulet and Garnet are both toolkits that provide a highly integrated collection of features designed to make it significantly easier to create interactive, graphical applications. These features include a prototype-

instance model, structured graphics, a composite object mechanism, spreadsheet-like constraints, and a high-level event-handling mechanism. Understanding how the constraints are used requires understanding these other features as well.

Prototype-Instance Model

Garnet and Amulet support a prototype-instance system, in which any object can serve as a prototype for another object [20, 21, 22, 3]. Each object consists of a set of properties, such as left, top, width, height, and color. A property is stored in a named variable called a *slot* (a slot would be called an *instance variable* in a class-instance model). If a slot is not explicitly assigned a value in an object, then that slot's value is inherited from the object's prototype.

A constraint is created by assigning a formula object to a slot (both Garnet and Amulet have macros that allow a programmer to declare a function as being a formula and that create a formula object which contains a pointer to the function). Slots inherit constraints from a prototype unless the programmer provides alternative values for the slots. For example, if the `width` slot of the prototype contains a constraint, then the `width` slot of an instance will contain a clone of that constraint unless the programmer provides an alternative value.

If the programmer wants to display a shape on the screen, the programmer creates an instance of the desired shape and adds the newly created object to the appropriate window. A display manager then automatically redraws those objects whenever any of the objects is modified (e.g., an object's position is changed or its color is modified). Objects are allowed to overlap and the display manager ensures that the objects get redrawn in the appropriate order. This arrangement in which each graphic element on the screen is represented by an object in memory is often called a *structured graphics* model (it is sometimes also called a "retained object model" or "display list").

Event-handling is also implemented using objects. Both Garnet and Amulet provide a set of objects (called *interactor objects*) that implement a few fundamental behaviors, such as creating new objects, moving/resizing objects, choosing one or more objects from a collection of objects, or editing a text string [23]. Each interactor object contains a parameterized set of properties that allows the programmer to customize the behavior in numerous ways. For example, the programmer can specify the events that should start and stop the behavior, the types of feedback that should appear when the behavior is being performed, and the actions that should occur when the behavior starts, is running, and stops. The programmer creates an interactive behavior by creating an instance of the appropriate interactor prototype, providing an appropriate set

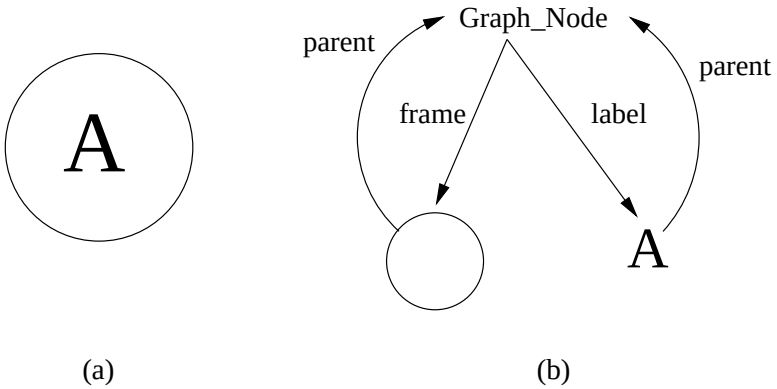


Figure 1: A graph node (a) and its structural components (b).

of property values, and attaching the object to a target set of graphical objects or to a window.

Composite Objects

A *composite object* is an object composed of other objects [24]. These other objects may either be primitive objects or themselves be composite objects. Composite objects are sometimes called "groups" or "aggregates." Figure 1 illustrates a simple composite object, a graph node consisting of a text object enclosed within a circle.

The graph node has named pointers to its children (`frame` and `label`) and the children have named pointers to their parent (`parent`) (See Figure 1b.). These pointers allow the graph node to access slots in its parts and the parts to access slots in their parent and in their siblings. The names of the pointers are derived from the names that programmers assign to the parts.

Both Garnet and Amulet support *structural inheritance*, whereby when an instance of a composite object is created, instances of all its parts are created as well. In addition, the children and parent pointers are automatically initialized for each of the parts.

Constraints simplify the creation of composite objects since constraints can be used to pass information around a composite object and to express relationships among the parts of a composite object. For example, the position and size of the graph node can be passed down to its label so that the label can center itself within the graph node.

Spreadsheet-like Constraints

Constraints provide a way for the programmer to specify relationships among the properties of graphical and behavior objects. For example, a programmer can center a text label in a rectangle by writing the constraint:

```
label.left = rectangle.left + rectangle.width/2 - label.width / 2
```

Garent and Amulet provide a number of features for spreadsheet constraints, including:

1. Full language support: A constraint is written in the underlying toolkit language and can make unrestricted use of any of the features of that language, such as pointer variables, loops, conditionals, function calls and recursion [3, 4]. Hence a Garnet constraint can contain arbitrary Lisp code and an Amulet constraint can contain arbitrary C++ code.

2. Side effects: In Amulet, a constraint may commit side effects, including creating/deleting objects or setting slots other than the slot to which the constraint is attached.

3. Transparency: The programmer is unaware of whether or not the slot's value is computed by a constraint. Garnet and Amulet both provide a *Get* method for reading the value of a property. This method takes the name of a property as an argument and returns the property's value. For example, in Amulet, the expression `label.Get(Am_LEFT) + label.Get(Am_WIDTH)` will compute the right side of a label object. The *Get* method is responsible for determining whether a property is computed by a constraint, and if so, ensuring that the constraint is up-to-date before returning the property's value.

4. Automatic Input Detection: The constraint solver automatically detects and records a constraint's inputs as the constraint is executed, so the programmer does not have to explicitly declare a constraint's inputs [12].

5. Cycle Support: Constraints are allowed to form cycles. For example, a programmer can write the constraints `a = b` and `b = a`. A constraint is evaluated at most once if it is in a cycle. If the constraint is asked to evaluate itself a second time, it simply returns its original value.

6. Path Expressions: A path expression lets formulas navigate their way through a tree of objects by specifying a series of pointers. For example, a formula that determines the `left` slot of the label object in Figure 1b could retrieve the `left` slot of the frame object by following the path `self.parent.frame.left`, where `self` is a pointer to the object containing the formula. Path expressions were first introduced in the late 1970s in two multi-way constraint systems, ThingLab [25] and Constraints [26]. Garnet and Amulet extended path expressions by allowing the tree of objects to be dynamically modified.

These features make Garnet and Amulet's constraint systems the most expressive and flexible systems available to graphical interface developers. Other systems have been more restrictive, because 1) they offer

only predefined constraints, such as the layout mechanisms in the InterViews [27] toolkit, the ultra light constraints described by Hudson [11], and the layout constraints in the Java Swing toolkit, 2) they use their own special constraint language that has restricted functionality, such as Higgens [7] or Penguims [10], or 3) they use the underlying toolkit language but do not provide full support for features such as loops, function calls, conditionals, pointer variables, or recursion [5, 8, 28].

Overall User Experience

During the 10 years that Garnet and Amulet have been in use, we have received considerable feedback from a number of sources about users' experiences with these two toolkits:

1. from hundreds of students in courses that we have taught who have used the toolkits to create various applications.

2. from email sent directly to the Garnet/Amulet developers or to the various Garnet and Amulet newsgroups by users of the toolkits.

3. from members of the Garnet and Amulet projects who themselves have written tens of thousands of lines of code using these toolkits.

4. from electronic survey forms for the Garnet and Amulet constraint systems that we posted on the Garnet and Amulet user newsgroups. These survey forms asked for users' experiences with the constraint systems and asked them to comment, if they desired to do so, about various features of these constraint systems. We received responses from 5 Garnet programmers and 12 Amulet programmers.

5. from analyzing 22 actual applications written using Amulet that were provided to us by a number of different types of users, including non-Amulet affiliated programmers, students, and Amulet project members (we have seen many more such programs but these 22 applications were formally analyzed for this paper to gather statistics about constraint usage).

The information that we have accumulated from these sources over the past decade are reported in the next two sections. This section summarizes general feedback that we have received regarding constraints while the next section summarizes how constraints are typically used in applications.

The general feedback we have received about constraints can be summarized as follows. Most programmers reported that they found constraints helpful for creating their interactive applications. Programmers

were most impressed with the flexibility they provided for defining custom graphical layouts, for computing graphical properties, and for the modularity they promoted. Programmers were most concerned with how difficult constraints could be to debug.

Graphical Layout

Programmers reported that they primarily used constraints to specify graphical layout and to a lesser extent, to specify graphical properties. In our Garnet and Amulet surveys, we asked programmers to compare using constraints for layout versus pre-defined layout managers of the type provided by Java or other toolkits such as Tcl/Tk. The responses we received indicated that for simple layouts, programmers preferred a pre-defined layout manager, such as those provided in the other toolkits. Neither Garnet nor Amulet provided such layout managers, which, given the responses, would appear to have been a useful feature (Amulet does have an extensive library of pre-defined layout constraints that partially but not fully meets these needs). However, programmers were very positive about the flexibility afforded by constraints in defining their own custom layouts and in laying out objects created dynamically. Programmers reported layouts that they were able to achieve with constraints that they felt they could not have achieved with a pre-defined layout manager included:

1. Connecting objects using arrows,
2. Tiling algebraic expressions,
3. Graph layouts,
4. Tree layouts,
5. Ring layouts, and
6. “Ribbon-chart” displays of schedules.

Pictures of several of these types of layouts are shown in Figure 2.

In general, it appears that widgets can often be laid out using pre-defined layout managers but that programmer-defined objects often require custom layouts that are best defined via constraints.

Modularity

A number of programmers reported that they liked the fact that constraints made their applications more modular. What the programmers meant by modularity is that from an application programmer’s point of

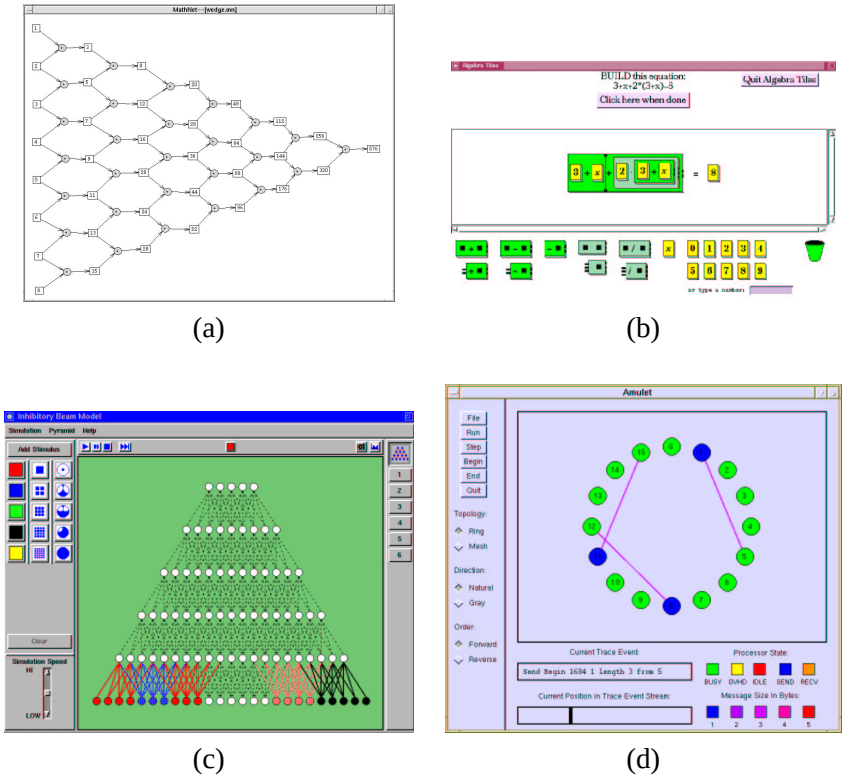


Figure 2: Some of the layouts created by Amulet programmers, including (a) connecting objects using arrows, (b) tiling algebraic expressions, (c) creating tree layouts, and (d) creating ring layouts.

view, the only object that knows about a constraint is the object to which the constraint is attached. In languages and toolkits that do not support constraints, both the object that wishes to observe a property (the observer) and the object whose property is being observed (the observee) need to know about one another's existence. The observer must notify the observee of the observer's interest in one of the observee's properties. The observee must then explicitly notify the observer when the property of interest changes. In addition, if either the observer or observee is deleted, they must notify the other object of their deletion. These interconnections between objects break down the modularity of the assignment and delete operations. The programmer must maintain data structures to keep track of the interconnections and must perform additional notification or bookkeeping work whenever an assignment or delete operation is executed.

In contrast, with constraints the programmer simply sets a property or deletes an object and the constraint solver automatically handles any side effects of the operation, such as re-evaluating constraints or deleting dependencies. Hence, from an application programmer's viewpoint, a constraint is local to an object and is not known outside the object. This means that the programmer can delete the object without worrying about notifying other objects that might depend on this object, or notifying other objects on which this object depends. Consequently, constraints improve the modularity of objects and reduce the complexity of various operations by allowing the programmer to worry only about the local effects of an operation.

Debugging

Programmers were almost unanimous in agreeing that debugging represented the greatest drawback of constraints. Some of the problems that were reported with debugging constraints included: 1) constraint cycles were easy to create but hard to find, 2) bugs often manifested themselves far from the point where they first occurred, 3) constraints started to look like spaghetti code as the number of constraints increased, 4) constraints that accessed the wrong slots could be hard to detect, and 5) constraints would not be evaluated when the programmer expected them to be evaluated (see [18] for a more detailed discussion of ways to handle this problem).

The most frequent debugging tools used were print statements and Amulet's inspector. Amulet's inspector pops up a property sheet showing the slot/value pairs associated with an object. It also allows a programmer to select a constraint and print a list of the constraint's dependencies.

What most programmers wanted, however, was a visual editor that would pictorially display the constraint network. One user suggested that it would also be helpful to tag the constraints with information about what values they have computed and when they were computed. Several programmers said that cy-

cles would be far easier to detect using such a visual editor.

Interestingly, a previous study of spreadsheets found that spreadsheets contained substantial numbers of errors [29]. Those findings, combined with our findings, suggest that the creation of effective debugging tools should be a top priority for constraint researchers.

How Constraints are Used

To determine how programmers use constraints in actual applications, we examined the source code of 22 Amulet applications. Three of the applications were written by non-Amulet members, ten of the applications were distributed as samples with the Amulet source code, and nine of the applications were written by students in a graduate course at the University of Tennessee. We also examined the source code that comprises the Amulet run-time system (every Amulet application runs on top of the Amulet run-time system). The application programs contained a total of 65,000 lines of code. The Amulet run-time environment contains 38,000 lines of code.

Since Amulet users create constraints by writing formulas and then attaching these formulas to slots, our examination focused on counting the number of formulas that were defined and the number of places in the source code where these formulas were attached to slots. Based on this examination, we divided the purpose of a formula into four categories:

1. Graphical Layout: These formulas compute an object's size and position.

2. Visibility: These formulas compute an object's visibility. They return true if the object should be visible and false if the object should be invisible.

3. Graphical Properties: These formulas compute the graphical attributes of an object, such as its color, line style, fill style, and text and font if the object is a text object. For example, a formula that computes a menu item's color based on the item's enabled status would be placed in this category.

4. Non-graphical: These formulas compute values that are used by the application to perform some non-graphical task. The values computed may be used as parameters to the event handling routines, as parameters to application callback procedures, or as error-checking values.

In a few cases a formula could logically fit in more than one category. In these cases, the formula was placed in the category with which it had the strongest connection. Formulas that simply copied values

around were categorized according to the slot to which they were attached. For example, a formula that copied the parent's width to a child would be categorized as a graphical layout formula.

Note that many of the formulas in the last three categories are not numeric and so could not be handled by many powerful, but domain-specific, numerical solvers. Hence, even if more powerful solvers were provided for graphical layout constraints, dataflow constraints would still have an important niche in graphical applications.

In studying formula usage, we also found it useful to perform two different types of counts:

1. The number of formulas written by programmers. This number refers to the number of formula functions that are defined in the source code.

2. The number of places in the source code where these formulas are used. Since a formula is declared separately from its use in Amulet, it can be used in multiple places. For example, Amulet provides a library of pre-defined formulas, such as "same width as" or "align left sides", that are declared once but used in many different places in users' code. Similarly, a programmer might define a formula that centers one object with respect to another object. This formula could be assigned to the left slot of a text object, thus centering the text object within another object, or to the left slot of a bitmap, thus centering the bitmap within another object. Hence this formula has two uses. In order to clearly differentiate a formula use from a formula definition, we will say that each time that the source code assigns a formula to a slot (i.e., each place in the source code where the formula is used) it creates a *constraint equation*.

Types of Formulas Defined

Table 1 shows the number of formula functions that were defined in the source code of the application programs and the Amulet run-time system, and the categories into which these formulas fell. Graphical layout formulas were the primary type of formula defined by both the programs and the run-time system.

The predominance of graphical layout formulas can be traced to three factors:

1. They represent the most obvious use of constraints in a graphical interface.

2. Each graphical object has four layout properties, left, top, width, and height, that would typically be constrained, whereas it would have only one visibility property and two graphical properties, line style and fill style, that would typically be constrained.

3. Our experience in teaching constraints to students shows that students immediately grasp how constraints can be used to compute graphical layout while they must be more carefully instructed on how constraints can be used for other purposes. We have repeatedly observed student programs where the student could have computed the value of a slot using a simple formula, but instead used a procedural action in several different parts of the code to set the slot. When shown how to do accomplish the same result formulaically, students will agree that the formula is the best way to accomplish the task. However, the next time their code is reviewed, they are again using procedural actions rather than formulas. Over time most students begin to define and use more non-graphical constraints, but the supervision required to obtain this result is fairly significant.

   The effect that experience and training has in using constraints can best be seen in the discrepancy between the percentage of non-graphical formulas that are defined by Amulet developers, who are well-versed in constraints, and application developers. Percentagewise, the Amulet developers defined three times as many of these types of formulas as application developers. For example, the Amulet developers defined a large number of formulas that compute many of the properties of the interactor objects.

   Unfortunately, given the considerable amount of time required to train students to use constraints in a non-graphical manner, it does not seem reasonable to expect that constraints will ever be widely used for purposes other than graphical layout. In retrospect this result should not have been surprising. Business people readily use constraints in spreadsheets because constraints match their mental model of the world. Similarly, we have found that students readily use constraints for graphical layout since constraints match their mental model of the world, both because they use constraints, such as left align or center, to align objects in drawing editors, and because they use constraints to specify the layout of objects in precision paper sketches, such as blueprints. However, in their everyday lives, students are much more accustomed to accomplishing tasks using an imperative set of actions rather than using a declarative set of actions. Hence they tend to think about most tasks procedurally, not functionally. Since constraints are simply functional programming dressed up with syntactic sugar, it should not be surprising that 1) programmers do not think of using constraints for most programming tasks and, 2) programmers require extensive training to overcome their procedural instincts so that they will use constraints.

In addition to analyzing the types of formulas that were defined, we also examined the types of properties that formulas used as inputs. The most interesting thing we found was that formulas typically depended

Category	*Formulas Defined in Application Programs*		*Formulas Defined in the Amulet Run-Time Environment*	
Graphical Layout	419	67%	111	49%
Visibility	42	7%	4	2%
Graphical Properties	87	14%	31	14%
Non-graphical	79	12%	78	35%
Total	627		224	

Table 1: The distribution of formula functions defined in the source code of application programs and the Amulet run-time system. For example, there are 224 unique formulas in the Amulet run-time source code.

on the *syntactic* properties of an application rather than on the *semantic* properties. *Syntactic* properties define an object's appearance in the user interface whereas *semantic* properties define the object's "intrinsic" meaning in the application. Examples of syntactic properties include position, size, visibility, color, and selection status (i.e., whether or not an object is currently selected by the user). Examples of semantic properties include the age of a tree in a landscaping application, the hull integrity of a ship in a ship-to-ship combat game, or the value of a variable in a visual language application.

One of the explanations for the greater frequency of syntactic properties as inputs to constraints is that Amulet formulas cannot be used to connect the instance variables of standard C++ class-instance objects with the slots of Amulet's prototype-instance objects. The reason is that the slots contain special methods and storage that allow constraints to establish dependencies from the slots to the constraints. Instance variables in standard C++ code do not have these methods or storage. Users have indicated that they would have liked to write constraints that connected application objects written in standard C++ code to Amulet objects but were unable to do so because of this restriction.

A second explanation, observed over years of experience, is that programmers just seem more comfortable with explicitly setting graphical properties that depend on application semantics. One possible reason for this behavior is that syntactic properties tend to be automatically set by the system while semantic properties tend to be set explicitly by the programmer in callback procedures. So constraints may seem like a natural way to monitor the values of these syntactic properties, since they seem to be "beyond" the control of the programmer. On the other hand, since the semantic properties are set by the programmers, the programmers may feel more comfortable with also setting the graphical properties that display these semantic properties.

Category	*Formulas Used in Application Programs*					
	Custom		*Library*		*Total Used*	
Graphical Layout	754	31%	939	38%	1693	69%
Visibility	95	3.5%	12	0.5%	107	1%
Graphical Properties	211	9%	147	6%	358	15%
Non-graphical	166	7%	122	5%	288	12%
Total	1226	50%	1220	50%	2446	100%

(a)

Category	*Formulas Used in the Amulet Run-Time Environment*					
	Custom		*Library*		*Total Used*	
Graphical Layout	125	30%	50	12%	175	42%
Visibility	4	1%	3	1%	7	2%
Graphical Properties	55	13%	43	10%	98	23%
Non-graphical	122	29%	18	4%	140	33%
Total	306	73%	114	27%	420	100%

(b)

Table 2: The number of times formulas were actually used in the source code of (a) application programs and (b) the Amulet run-time system. Custom formulas are formulas written by programmers and library formulas are formulas that are pre-defined in an Amulet library. Each percentage represents the proportion of formulas belonging to that category. For example, 69% of the formulas in the source code of application programs were graphical layout formulas.

Usage of Formulas

Once a formula is defined in Amulet, it can be used in multiple places in the source code to create constraint equations. Table 2 shows the number of uses of formulas in the source code of Amulet applications and the Amulet run-time system. The table also shows the number of uses of pre-defined formulas (called library formulas in the table) in the source code. These pre-defined formulas are counted in the Amulet run-time system in Table 1. Amulet provides 14 pre-defined formulas for 1) aligning the lefts, tops, centers, bottoms, and rights of objects, 2) computing the width or height of a composite object, 3) laying out the parts of a list, and 4) for retrieving a slot from a part, a parent, or a sibling.

The table shows that formulas are used in roughly the same proportion as they are defined. For example, 67% of the custom formulas defined by application programmers are graphical layout formulas and these graphical layout formulas comprise 62% of the custom formulas used in the source code[1]. The results show

[1]The 62% figure is derived by dividing the 31% in the custom column of Table 2 by the 50% total line in this column.

custom formulas are used just as frequently as library formulas, and are favored by a 3-1 ratio in the Amulet toolkit itself. However, the fact that a collection of 14 formulas could account for 50% of the formula usage for application programs is still indicative that if pre-defined formulas are chosen carefully, they can significantly aid an application programmer.

Figure 3 presents another way of looking at formula usage. It shows the frequency with which individual formulas were used to create constraint equations for both application programs and the Amulet runtime system. For example, it shows that 391 of the formulas defined in the application programs were used only once in the source code, 85 of the formulas were used twice, and so on. The figure shows that there was a moderate amount of re-use of formulas. The pre-defined formulas accounted for 13 of the 26 formulas that were reused 10 times or more in the application programs and accounted for the only formula that was reused 10 times or more in the Amulet runtime system.

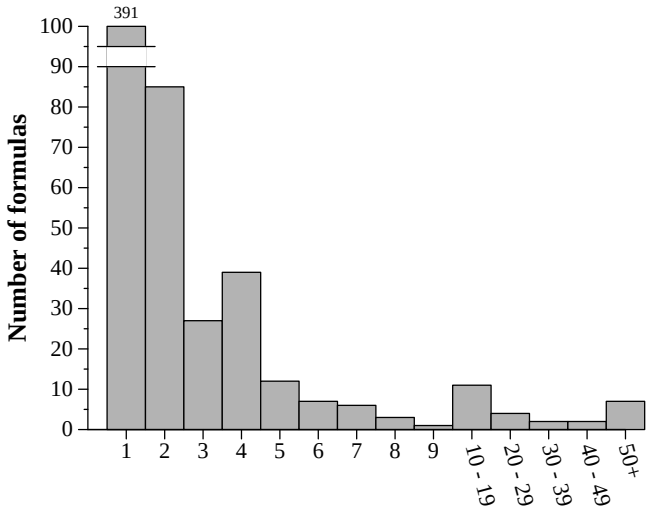
The most commonly re-used formulas were the pre-defined formulas that retrieved a slot from another object. The pre-defined formula that retrieved slots from an owner was the most commonly used formula in both application programs and the Amulet runtime system. The most commonly used of the remaining pre-defined formulas are the ones that 1) center an object with respect to another object, 2) lay out the parts of a list, and 3) compute the width or height of an object as the width or height of the bounding box of its parts.

An analysis of the formulas that were written by programmers revealed only one type of formula that was re-used consistently across multiple applications. This type of formula copied a slot from an object that was not a part of the formula's composite object. For example, an arrow object might use the formula `x2 = self.to_obj.left` to attach itself to the left side of an object[2] Consequently, it would have been helpful to have had a pre-defined formula that allowed a programmer to specify either 1) an object and a slot to be copied from that object, or 2) a slot that points to an object and a slot to be copied from the object. Overall, the lack of other candidates for pre-defined formulas is not too surprising given that the set of pre-defined formulas was based on initial usage information gathered at the beginning of the Amulet project.

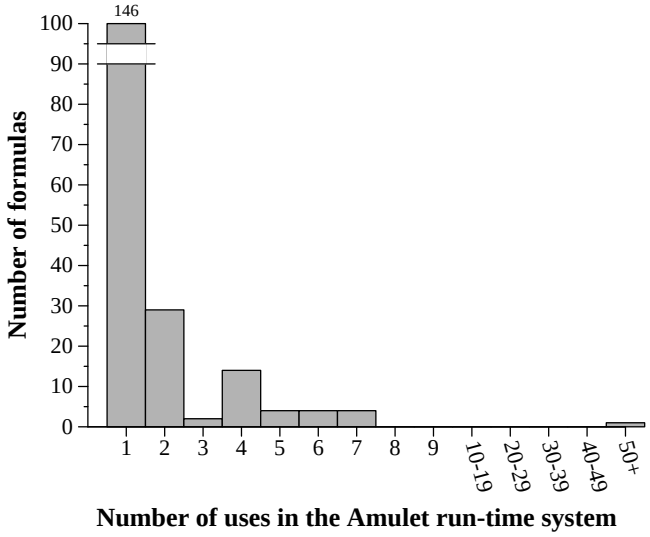
Design Guidelines for Constraint Systems

The previous two sections discussed what programmers liked and did not like about constraints and described how programmers used constraints in their applications. This section describes some of the lessons

[2]In this constraint and all remaining constraints in the paper, a variable prefixed with "self" belongs to the same object as the variable on the left side of the constraint.



(a)



(b)

Figure 3: The frequency with which individual formulas were used to create constraint equations in the source code for both (a) application programs and (b) the Amulet runtime system. For example, graph (a) shows that 391 of the formulas defined in the application programs were used only once in the source code, 85 of the formulas were used twice, and so on.

we learned about making constraints easier to use. These lessons should prove useful to future toolkit developers and can be summarized as follows:

1. Use the underlying language: A constraint should be written in the toolkit's underlying implementation language and should be able to use any of the language constructs supported by this language.

2. Avoid annotations if possible: Forcing a programmer to annotate code in ways that provide information to the constraint solver does not work well. Programmers find it burdensome to do so. If it is optional, they will avoid doing it and so optimizations based on these annotations cannot be made. If it is mandatory, they will provide the annotations but they will often make errors that lead to confusion.

3. Syntax matters: Programmers should be able to define a constraint at the point where it is assigned to a variable. Additionally, the programmer should not have to specify a great deal of excess verbiage when defining a constraint. Ideally formula creation should be as simple as specifying a keyword like `Formula` and the code that defines the constraint.

4. Path expressions are a two-edged sword: Path expressions are a powerful feature that help facilitate structural inheritance but users often write them incorrectly.

Use the Underlying Language

Prior to Garnet, constraint systems defined their own special constraint language. Such languages have two drawbacks–they force a programmer to learn a new language and they often have restricted functionality (e.g., lack of certain control structures, such as loops and procedures, and lack of many operators beyond simple arithmetic and boolean operators).

As noted earlier, Garnet and Amulet allow a constraint's formula to use any of the features in the underlying toolkit implementation language. Hence a constraint can contain arbitrary loops, conditionals, functions, and pointers to other objects. Amulet additionally supports side effects. Allowing programmers to write arbitrary code and to commit side-effects permits them to define very powerful constraints that perform more than simple graphical layout operations. For example, Garnet and Amulet programmers have used constraints to do the following, which would be impossible in most other constraint systems:

- Define complicated layout constraints. In both Amulet and Garnet a single constraint is used to lay out the items in a list based on such parameters as the orientation of the list (vertical or horizontal), the spacing between items, the length of a row or column (expressed either in pixels or as a maximum

number of items), and the amount of space to be consumed by each item (fixed or variable width). Garnet also has complete graph and tree layout algorithms implemented as constraints. A simple example of a list layout constraint might be the following:

```
layout = begin
           spacing = self.horizontal_spacing
           next_left = self.left
           for each child in self.items do
             if child.visible = true then
               child.left = next_left
               next_left = next_left + child.width + spacing
           end
```

- Control the visibility of an object, such as a feedback object, based on whether or not other objects are selected. For example, the visibility of selection handles could be controlled by whether its `obj_over` slot points to an object:

```
visible = if self.obj_over then true else false
```

- Define an object's visual properties based on the values of application data. For example, to make the color of a graphical object depend on the temperature of an application object, the programmer might write the constraint:

```
color = if self.temperature_object.temperature < 32 then blue
        else if self.temperature_object.temperature < 212 then white
        else red
```

- Compute the parameters that control the processing of an event, such as computing what type of feedback object to draw based on what object the mouse is currently over. For example:

```
feedback_object = switch TypeOf(mouse.obj_over)
                    case RECTANGLE:
                    case TEXT:
```

case ICON: self.rectangle_feedback; break;

case CIRCLE: self.circle_feedback; break;

case LINE: self.line_feedback

The slots `rectangle_feedback`, `circle_feedback`, and `line_feedback` point to appropriate feedback objects for rectangle-like objects, circles, and lines respectively.

- Create the set of items that should be displayed in a browser by reading a directory name from the appropriate widget, passing the name to the appropriate system command, and creating and then returning a list of graphical text objects that can display the result. For example:

```
items = begin
        directory_name = widget.value
        filename_list = system("list_files(directory_name)")
        graphical_items_list = ∅
        for each filename ∈ filename_list do
            text_item = new TEXT /* create a new graphical text object */
            text_item.text = filename
            graphical_items_list.append(text_item)
        return graphical_items_list
    end
```

Lessons Learned

The lessons we learned about using the underlying toolkit language as the constraint language can be divided into the lessons we learned about arbitrary code, about pointer variables, and about side effects.

Arbitrary Code. The ability to use loops and conditionals is a crucial element in many of the constraints written in Garnet and Amulet. Loops complicate constraint satisfaction because they allow a constraint to reference a dynamic, rather than a fixed, number of inputs. However, loops also considerably enhance the expressiveness of constraints because they make it possible to write constraints that handle *dynamically* changing sets of objects, such as the neighbors of a node in a graph or the set of parts in a composite object. Loops therefore present a tradeoff but it is better to design for the user of the constraint system than the developer of the constraint system, and our experience shows that allowing a user to use all aspects of a language, including loops, greatly enhances the richness of the constraints that are written.

The one drawback of arbitrary code is that it restricts some performance optimizations that might be otherwise possible since arbitrary code is hard to analyze. However, we found that constraint evaluation represents such a small portion of an application's overall execution time that performance optimization is not a crucial issue [18].

Pointers. The ramifications of pointer variables for constraints were twofold: 1) they allowed constraints to be used with data structures that are typically implemented using pointers, such as lists, trees, and graphs, and 2) they combined with loops to allow constraints to reference a variable number of objects. Consequently, the unrestricted use of pointer variables was another crucial element that allowed users to write very expressive constraints.

The problem of dangling pointers was the one potential problem associated with pointers. However, Garnet and Amulet were able to easily detect dangling pointers because both systems use garbage collection (Amulet used a reference counting scheme to garbage collect its objects). When an object is destroyed by the application, it is only deleted if there are no references remaining to it. If there are still references, the object is marked as deleted but its memory is not actually released. Every slot access in both Garnet and Amulet checks to see whether the object is marked deleted before returning a value. If the object is marked deleted, an error message about the offending constraint and the object and slot it tried to access is printed and an error interrupt is raised.

Side Effects. As noted earlier, Garnet does not have a mechanism for handling side effects in constraints whereas Amulet does[3]. The side-effect mechanism was added to Amulet because Garnet users often found that they wanted to commit side effects from within constraints. There were two types of side effects that they wanted:

1. The ability to set multiple slots with one constraint. Laying out the objects in a list and computing an object's bounding box (thus setting the left, top, width, and height slots) are two examples of where a constraint needs to set multiple slots.

   Multiple-output constraints elegantly handle this situation but they have the disadvantage that the programmer must somehow annotate the constraint to declare which slots the constraint will set. In the case of a constraint that lays out the slots of a list, the programmer must be given a way to specify not just the slots but also the objects that will be set by the constraint. Unfortunately, as discussed in the section on annotations, we have found that programmers tend to resist annotations and often

[3] A Garnet constraint can be programmed to commit side effects but since Garnet does not have a mechanism for handling constraint side effects, the side effects frequently are not executed when the programmer expects them to be executed.

get them wrong. Consequently, a decision was made not to require Amulet programmers to declare which slots they were setting in a constraint.

2. The ability to create and delete objects. Taking a list of labels and allocating a list of objects to display these labels is an example of a constraint that needs to create objects. Amulet allows a constraint to create and delete arbitrary numbers of objects.

The implementation of Amulet's side-effect mechanism has been described elsewhere [4]. The implementation can potentially lead to both non-deterministic results and to infinite cycles. So from a theoretical standpoint the algorithm is not elegant. However, from a *practical* standpoint, the algorithm works. Users have not reported difficulties with non-determinism or infinite loops. Similar positive experiences with side effects implemented using unsound algorithms have been reported in the Rendezvous system [5] and an experimental version of Garnet that was extended to include multi-output constraints and side effects [30].

An examination of some of the side-effect producing constraints written by our students helps explain why there have not been problems with such constraints. The constraints were used for precisely the reasons stated above, either to set multiple slots or to create or delete a set of items. The multiple slot constraints typically set the same slots on each evaluation, so the constraints' outputs were as predictable as single-output constraints. The more unpredictable case would seem to be the dynamic creation or deletion of objects by constraints. However, because it was not possible to predict in advance which objects would be created, constraints did not depend directly on the created or deleted objects. Instead, they would access the objects through a slot that kept track of the objects (e.g., an items slot). For example, a constraint that computed the width of a list of items would iterate through the items slot for that list and request the width of each object on that list. Invariably, it was these tracking slots that contained the object creating/deleting constraints. Thus, before a constraint could access the objects, it would first have to access the tracking slot. The tracking slot would re-evaluate its constraint, if necessary, thus making all subsequent accesses to the objects on its list safe.

In sum, the introduction of side effects into constraints has had the desired effect of allowing users to write the type of side-effect producing constraints that they would have liked to have written in Garnet, and they *have* done so. Further, users have used side-effect inducing constraints in common sense ways that do not cause unpredictable behavior.

Avoid Annotations If Possible

Annotations are statements written by a programmer that provide information about a program to a compiler or run-time environment. This information is typically difficult to obtain by other means and is either essential to the functioning of the system or allows the system to optimize the program's performance. An example of annotations in programming languages is declaring variables to have a certain type. In spreadsheet constraint systems, annotations are frequently used to specify the inputs used by a constraint. Garnet and Amulet made use of annotations in two ways. First, early versions of both Garnet and Amulet required the programmer to specify which slots were used as inputs to a constraint. Second, Garnet allowed the programmer to use annotations to specify which slots were constant so that constant propagation could be used to eliminate constraints. In both cases we found that programmers were confused by the annotations and often made incorrect specifications, and so annotations were eventually completely eliminated in Amulet.

Annotating Inputs

In order for a constraint solver to know when it needs to re-evaluate a constraint, it needs to know on which variables the constraint depends (i.e., it needs to know the inputs, or equivalently, the right-hand side variables).

Most constraint systems require that a user either declare the inputs that will be used by a constraint or else annotate the inputs in some fashion. For example, in Eval/vite the programmer annotates an input by prefixing the variable name with an at-sign (@) [8].

Early in Garnet's design we decided that declaring the inputs was unworkable because constraints could have tricky control code (e.g., loops, conditionals, and function calls) that would have to be duplicated in the declaration code. We judged that such duplication was certain to cause programming errors. As an example of the difficulties involved with declaring inputs, reconsider the constraint presented earlier for laying out the children in a list:

```
layout = begin
           spacing = self.horizontal_spacing
           next_left = self.left
           for each child ∈ self.items do
             if child.visible = true then
               child.left = next_left
               next_left = next_left + child.width + spacing
```

end

In order to *declare* the inputs for this constraint, we would have to write code that would look something like:

inputs = {self.horizontal_spacing, self.items}
for each child ∈ self.items **do**
    inputs = inputs ∪ {child.visible}
    if child.visible = **true then**
        inputs = inputs ∪ {child.left, child.width}

It is easy to write this code incorrectly and it is also easy to forget to update this code if the constraint is rewritten. If either event happens, the constraint solver may not operate correctly. As a result we abandoned the idea of making the programmer declare a constraint's inputs.

The *annotation* idea did seem workable and so both the original versions of Garnet and Amulet implemented the annotation approach. In particular, a programmer would use a `Get` method to retrieve a slot's value when the slot was being accessed outside a formula (i.e., in normal, procedural code) and a `GV` method to retrieve a slot's value when the slot was being accessed as an input to a formula.

Unfortunately, we found that users were constantly getting confused by the two different forms of the get method and would often use the wrong get method. As a result, slots that the programmers thought were inputs to a formula were not getting annotated as inputs. The programmers would then be baffled when the constraints in their programs did not get re-evaluated properly. In both Garnet and Amulet, this problem was remedied by effectively eliminating one of the get methods and using an automatic input detection scheme that is able to determine whether or not a slot is being accessed from within a formula [12].

Annotating Constant Slots

In Garnet we observed that many of the constraints were evaluated exactly once because they depended entirely on slots that were constants. These formulas could be effectively thrown away, thus allowing a savings in storage. To take advantage of this opportunity, we allowed a Garnet programmer to declare that a slot was constant. If all the slots referenced by a constraint were constants, then the slot computed by the constraint was marked as a constant, the value computed by the constraint was assigned to the slot, and the constraint was eliminated. As a result of marking the slot constant, other constraints might also become eliminable because their inputs were all constants.

Unfortunately this scheme did not work well in practice for a number of reasons:

1. Constraints were not eliminated as expected. Many of the constraints that programmers wanted to eliminate via constant propagation were Garnet-provided constraints. For example, the width of a text object was computed by a constraint that took the text object's string and font as inputs. Programmers would declare the text string to be constant and expect the constraint to be eliminated. Often they did not realize that the constraint also used the font as an input. So the constraint was not eliminated as expected and programmers grew frustrated.

2. Changes to a program made the annotations obsolete. Even after an application is released, the code is not static and changes to the code often made the annotations obsolete. For example, the code might start changing a slot that was previously declared constant. In this case, constraints did not get updated as the programmer expected. In other cases, additional slots were added and constraints were modified to include these additional slots. Programmers would forget to declare that these new slots were constant, with the result that previously eliminated constraints started mysteriously reappearing.

Both of these factors frustrated the few programmers who tried to use the constant propagation mechanism and as a result it proved to be ineffective. Ultimately, the effort required to annotate the code and to understand how to annotate it effectively proved to be too burdensome.

Pay Attention to Syntax

Garnet was implemented in Lisp whereas Amulet was implemented in C++. Although C++ is the more popular language, we found that constraints felt clumsier in C++ because of the way C++ is designed.

Syntax Issues

Users found it easier to write constraints in Garnet than in Amulet because Garnet allowed users to write a constraint's formula at the location where the constraint is assigned to a variable. In contrast, Amulet requires that the user define the constraint's formula and a name for the formula in a separate part of the program and then assign the formula's name to the variable. The following example code shows the contrasting styles in Garnet and Amulet:

Garnet	Amulet

	Formula Compute_Right {
	return (self.left + self.width);
	}
	...

A.right = Formula(self.left + self.width)	A.right = Compute_Right

In both Garnet and Amulet, `Formula` is a macro that expands into a function which contains the formula code. Amulet users complained about having to separate the definition of the formula from the use of the formula. The extra code is inconvenient to write, increases the probability of syntax errors, and makes the code less readable because someone trying to understand or maintain the code must constantly shift back and forth between two points in the program: the location where formulas are defined and the location where they are used.

The reason for the separation is that the formula code has to be wrapped inside a function. C++ does not allow a function to be defined inside another function, so the formula definitions have to be moved outside the scope of all functions (i.e., they must be global definitions). Being a research project, we did not want to spend a lot of time writing a pre-processor that would allow the formulas to be written "in-line" and then lifted outside the function. However, it is clear that syntax *does* matter and that commercial toolkit developers would be well-advised to expend the time and effort required to write such a pre-processor.

Global Variables

Amulet programmers often wanted to use constraints to connect the slots of top-level objects in a graphical editor or in a dialog box. Because the constraints' formulas are global functions, the only way to reference these objects in the formulas was to declare the objects globally as well. Hence one of the ramifications of defining formulas as global functions was a proliferation of global variables, about which a number of programmers complained.

Garnet does not suffer from this problem because the formulas used by Garnet's constraints are bound to the environment of the function in which they are defined. In particular, Lisp has two operators, *function* and *lambda*, that when used in concert, allow the inline creation of anonymous functions, and that save the bindings of the enclosing function [31]. When the enclosing function exits, the anonymous function will

still have access to the bindings in its enclosing function. For example, one can write[4].

procedure CreateDialogBox()

Button Ok;

Button Cancel;

Cancel.left = Formula(Ok.left + Ok.width + 10)

. . . Other code to add the Ok and Cancel buttons to the dialog box window . . .

end procedure

Even after the CreateDialogBox function has exited, the formula that is attached to `Cancel.left` can access the `left` and `width` properties of the `Ok` button. C++ does not permit this type of binding and hence the `Ok` and `Cancel` buttons would have to be declared as global variables.

Path Expressions: Boon and Curse

In both Garnet and Amulet, programmers can string together combinations of parent and children pointers in order to traverse their way through a composite object. For example, in the label part of the graph node in Figure 1, a constraint's formula might use the pathname `self.parent.frame.left` to retrieve the frame part's left slot. The path follows the parent pointer to the label's parent, then the frame pointer in the parent to the frame object, and finally retrieves the left slot.

Paths are an important construct that enable the inheritance of constraints since they allow instances of a constraint to refer to the appropriate properties and parts in its instance object rather than some fixed object. For example, in every instance of a labeled box, the above path will properly return the left slot in the instance's frame rather than the left slot in the prototype's frame.

Unfortunately, two common problems arose with path expressions: 1) users found it easy to write path expressions incorrectly, and 2) path expressions break when objects are moved around the composite object hierarchy or renamed. As the following two subsections make clear, we were never able to completely overcome either problem.

[4]It has been our experience that many programmers and researchers are not familiar with or are uncomfortable with LISP syntax. Consequently, we have chosen to use C-like code to illustrate what an in-line formula referencing local variables would look like. For those readers who are interested, the Formula macro would expand into the Lisp expression:

```
(function (lambda () (+ (gv Ok :left) (gv Ok :width) 10)))
```

where `gv` stands for *get-value*.

Incorrect Path Expressions

While users did not have much problem writing path expressions involving children and parents, they encountered more problems writing path expressions that went beyond either a parent or a child. For example, writing `self.parent.left` typically was not problematic but writing `self.parent.frame.left` was somewhat problematic. In general, the farther one has to traverse the composite object hierarchy to find an object, the harder it is to write the path expression correctly.

A limited solution that worked reasonably well in Amulet was the introduction of path macros such as `Get_Sibling`, `Get_Parent`, and `Get_Child`. For example, `Get_Sibling(frame, left)` would expand into `self.parent.frame.left` and return the frame part's left slot. An analysis of Amulet code shows that most path expressions do not extend beyond the grandparent or grandchild level (i.e., no more than 2 levels up or down in the hierarchy), so it is possible that a reasonably complete solution could be achieved by providing grandparent, grandchild, and nephew macros.

Another solution that several Amulet users developed was to split a path into multiple parts and put constraints that computed the multiple parts at different parts of the composite object. For example, rather than writing the constraint:

```
left = self.parent.parent.frame.left + 10
```

the programmer might write the three constraints:

in the grandparent:

```
frame_left = self.frame.left
```

in the parent:

```
frame_left = self.parent.frame_left
```

in the part:

```
left = self.parent.frame_left
```

It would have been interesting to see if a more complete set of macros could have alleviated the need to do this kind of splitting.

Broken Path Expressions

A second problem that arises with path expressions is that they break when objects are moved around the composite object hierarchy or objects are renamed. For example, in Figure 1, the path expression

`self.parent.frame.left` will break if either 1) a new object is interposed between `Graph_Node` and `label` so that `label's` new parent is the new object, or 2) `frame` is renamed `border`. Allowing the programmer to directly name an object would solve the problem of moving objects around in the composite object hierarchy. However, it would not solve the problem of renaming objects. Here again we never arrived at a satisfactory solution.

Conclusions and Future Work

A great deal of research activity in the user interface community has centered on constraints over the past decade. The Garnet and Amulet toolkits represent two of the products of these efforts. Because of their widespread distribution and use, they have provided valuable insights into both the successes and shortcomings of constraints. These insights should help the future developers of toolkits to find the best niches for constraints in their toolkits.

Overall, programmers' experience with constraints in the Amulet and Garnet projects has been quite positive. Programmers generally agree that they simplify the task of creating user interfaces and that they are a valuable and useful programming tool. However, constraints are not a panacea for writing user interfaces. Programmers primarily use them for specifying the graphical layout of objects. Our experiences suggests that most programmers will resist using constraints for specifying non-graphical behavior in an interface, because they think of these behaviors imperatively, rather than functionally.

Given the current state of debugging technology, it seems that this moderate use of constraints in interfaces is actually beneficial since it leads to more elegant, robust code, whereas the heavy usage of constraints seems to lead to spaghetti-like code that creates debugging problems.

In addition to these general findings, our experience shows that developers wishing to include constraints as part of their toolkits would be well-advised to keep the following lessons in mind:

1. Allowing programmers to write constraints using arbitrary code and using all the capabilities of the underlying language reduces the learning curve and increases the power of the constraint system. Even allowing side effects is something that a developer may wish to consider since programmers naturally do it and our experience has shown that it does not increase the difficulty of debugging constraints.

2. The programmer should not be required to provide annotations if at all possible. Our experience is that programmers often provide incorrect or incomplete annotations, which leads to errors in the initial development. Maintenance problems later arise because programmers often fail to update the

annotations when they update the code.

3. Allowing constraints to be defined at the point of use makes the relationship between the variable and the constraint much clearer to both the programmer and the maintainer.

4. Adding more complicated mechanisms, such as path mechanisms for traversing object hierarchies, should be carefully weighed. They may be necessary, as was the path mechanism in Garnet and Amulet, but they can significantly increase the learning curve of the constraint system and introduce debugging and maintainability problems. In general, these mechanisms should be added only if they are absolutely essential to the success of the constraint system, not if they marginally increase the power of the constraint system.

Finally our experiences shows that constraints still pose a number of challenging problems for researchers, including:

1. the need to develop better debugging tools that reduces the spaghetti-like feel of constraints, and

2. the need to develop theoretically sound constraint satisfaction algorithms that can tolerate side effects. A theoretically sound constraint satisfaction algorithm is one that does not enter an infinite loop or produce non-deterministic results. The development of such algorithms would help us understand what side effects are "safe" side effects and what side effects are "unsafe" side effects. Our experience with algorithms involving side effects suggests that theoretically sound algorithms may require some annotations from a programmer (e.g., which variables will be affected by the constraint). If this proves to be the case, an implementor may have to decide between an unsound algorithm that does not use annotations and a sound algorithm that uses annotations. Our positive experience with Amulet's unsound algorithm and our negative experience with annotations suggest that an implementor should not rush to use a sound algorithm if the annotation burden is too great.

References

[1] Paul Barth. An object-oriented approach to graphical interfaces. *ACM Transactions on Graphics*, 5(2):142–172, April 1986.

[2] Brad A. Myers. Creating user interfaces using programming-by-example, visual programming, and constraints. *ACM Transactions on Programming Languages and Systems*, 12(2):143–177, April 1990.

[3] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

[4] Brad A. Myers, Rich McDaniel, Alan Ferrency Robert Miller, A. Faulring, Bruce Kyle, Andy Mickish, Alex Klimovitski, and P Doane. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.

[5] Ralph D. Hill. The Rendezvous constraint maintenance system. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 225–234, Atlanta, GA, November 1993. Proceedings UIST'93.

[6] Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F. Patterson, and Wayne Wilner. The Rendezvous architecture and language for constructing multiuser applications. *ACM Transactions on Computer Human Interaction*, 1:81–125, June 1994.

[7] S. Hudson and R. King. Semantic feedback in the Higgens UIMS. *IEEE Transactions on Software Engineering*, 14(8):1188–1206, Aug 1988.

[8] Scott E. Hudson. A system for efficient and flexible one-way constraint evaluation in C++. Technical Report 93-15, Graphics Visualizaton and Usability Center, College of Computing, Georgia Institute of Technology, April 1993.

[9] Tyson R. Henry and Scott E. Hudson. Using active data in a UIMS. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 167–178, Banff, Alberta, Canada, October 1988. Proceedings UIST'88.

[10] Scott E. Hudson. User interface specification using an enhanced spreadsheet model. *ACM Transaction on Graphics*, 13(3):209–239, July 1994.

[11] Scott E. Hudson and Ian Smith. Ultra-lightweight constraints. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 147–155, Seattle, WA, Nov 1996. Proceedings UIST'96.

[12] Brad Vander Zanden, Brad A. Myers, Dario Giuse, and Pedro Szekely. Integrating pointer variables into one-way constraint models. *ACM Transactions on Computer Human Interaction*, 1:161–213, June 1994.

[13] Bowen Alpern, Roger Hoover, Barry K. Rosen, Peter F. Sweeney, and F. Kenneth Zadeck. Incremental evaluation of computational circuits. In *ACM SIGACT-SIAM'89 Conference on Discrete Algorithms*, pages 32–42, Jan. 1990.

[14] Bradley Vander Zanden and Richard Halterman. Using model dataflow graphs to reduce the storage requirements of constraints. *ACM Transactions on Computer-Human Interaction*, 8(3):223–265, September 2001.

[15] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM TOPLAS*, 5(3):449–477, July 1983.

[16] R. Hoover. *Incremental Graph Evaluation*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1987.

[17] Scott E. Hudson. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM TOPLAS*, 13(3):315–341, July 1991.

[18] Brad Vander Zanden, Brad A. Myers, Pedro Szekely, Dario Giuse, Rich McDaniel, Rob Miller, David Kosbie, and Richard Halterman. Lessons learned about one-way, dataflow constraints in the Garnet and Amulet graphical toolkits. *ACM Transactions on Programming Languages and Systems*, 23(6):776–796, Nov 2001.

[19] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present and future of user interface software tools. *ACM Transactions on Computer Human Interaction*, 7(1):3–28, Mar 2000.

[20] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. *Sigplan Notices*, 21(11):214–223, Nov 1986. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'86.

[21] A. Borning. Classes versus prototypes in object-oriented languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, Nov. 1986.

[22] David Ungar, Randall B. Smith, Craig Chambers, and Urs Holzle. Object, message, and performance: How they coexist in self. *IEEE Computer*, 25(10):53–64, Oct 1992.

[23] Brad A. Myers. A new model for handling input. *ACM Transactions on Computer Human Interaction*, 8(3):289–320, July 1990.

[24] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995. ISBN 0-201-63361-2.

[25] Alan Borning. The programming language aspects of ThingLab; a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, Oct 1981.

[26] G.J. Sussman and G.L. Steele. Constraints–a language for experessing almost-hierarchical descriptions. *Artificial Intelligence*, 14:1–39, 1980.

[27] John M. Vlissides Mark A. Linton and Paul R. Calder. Composing user interfaces with interviews. *IEEE Computer*, 22(2):8–22, Feb 1989.

[28] Scott E. Hudson and Ian Smith. The subArctic user interface toolkit. 2000.

[29] Polly S. Brown and John D. Gould. An experimental study of people creating spreadsheets. *ACM Transactions on Office Information Systems*, 5(3):258–272, 1987.

[30] William J. Rosener. *Integrating Multi-way and Structural Constraints into Spreadsheet Programming*. PhD thesis, Department of Computer Science, University of Tennessee, Knoxville, TN, 1994.

[31] Guy L. Steele, Jr. *Common Lisp: The Language, 2nd Ed.* Digital Press, Woburn , MA, 1990.