Lessons Learned About One-Way, Dataflow Constraints in the Garnet and Amulet Graphical Toolkits

BRADLEY T. VANDER ZANDEN and RICHARD HALTERMAN
University of Tennessee
BRAD A. MYERS and RICH MCDANIEL and ROB MILLER
Carnegie Mellon University
PEDRO SZEKELY
USC/Information Sciences Institute
DARIO A. GIUSE
Vanderbilt University Medical Center

and
DAVID KOSBIE
Microsoft Corporation

One-way, dataflow constraints in which an expression is evaluated whenever necessary to compute the value of a variable are commonly used in graphical interface toolkits, programming environments, and circuit applications. Previous papers on dataflow constraints have focused on their design and implementation. In contrast, this paper focuses on the lessons we have learned from 10 years of experience and user feedback with the Garnet and Amulet graphical interface toolkits. These lessons provide both design and implementation guidelines for one-way constraints. The most important lessons we have learned 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) debugging constraints is difficult and better debugging technology is needed, and
(3) constraint solvers have more than adequate speed but the storage required by constraints may continue to be problematic.

This research was supported in part by the National Science Foundation under grant CCR-9633624.
Authors' Addresses: B. Vander Zanden and R. Halterman, Computer Science Department, University of Tennessee, Knoxville, TN 37920. email: bvz@cs.utk.edu; B. Myers, R. McDaniel, and R. Miller, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. email: brad.myers, richm, 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, Microsoft Corporation. email: dkosbie@yahoo.com

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; D.2.6 [**Software Engineering**]: Programming Environments—*Graphical environments, Interactive environments*; D.3.2 [**Programming Languages**]: Language Classifications—*Data-flow languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints*; I.1.2 [**Computing Methodologies**]: Algorithms—*Nonalgebraic algorithms*; I.1.2 [**Computing Methodologies**]: Languages and Systems—*Evaluation strategies*

General Terms: Algorithms, Design, Experimentation, Human Factors, Performance

Additional Key Words and Phrases: Constraint Experience, Constraint Satisfaction, Constraint Usage, Eager Evaluation, Lazy Evaluation, One-way Dataflow Constraints

1. INTRODUCTION

A one-way, dataflow constraint is an equation in which the expression on the right side of the equation is reevaluated whenever necessary and assigned to the variable on the left side of the equation. For example, the constraint `rect2.top = rect1.bottom + 10` specifies that `rect2` should be positioned ten pixels below the bottom of `rect1`. Such a constraint is called a *dataflow* constraint because data flows from the variables on the right side of the equation to the variable on the left side of the equation. It is called a *one-way* constraint because the equation must always be solved for the left-hand side variable. For example, the above equation must always be solved for `rect2.top`. It is not permissable to invert the equation in order to solve for `rect1.bottom`. A dataflow equation is reevaluated whenever one of the variables on the right side of the equation is changed.

One-way, dataflow constraints are widely recognized as a potent programming methodology. Their initial success in spreadsheets and attribute grammars [Knuth 1968] has inspired researchers to use them as tools in a variety of applications including graphical interfaces [Barth 1986; Myers 1990; Myers et al. 1990; Myers et al. 1997; Hill 1993; Hill et al. 1994; Hudson and King 1988; Hudson 1993; Henry and Hudson 1988; Hudson 1994; Hudson and Smith 1996], programming environments [Demers et al. 1981; Reps et al. 1983; Hoover 1987; 1992], and circuit simulations [Alpern et al. 1990].

Despite the wealth of papers on the design and implementation of these tools' constraint systems, nothing has been published that describes the long-term experiences that have been gained from using these systems or the trade-offs in constraint satisfaction algorithms that have been discovered as a result of these experiences.

This paper describes the insights we have gained from 10 years of experience with the one-way dataflow constraint systems in the Garnet and Amulet toolkits [Myers et al. 1990; Myers et al. 1997]. Garnet is a Lisp-based toolkit for developing interactive graphical applications that was first released in 1989 and has been used by over 80 projects. Amulet is a C++-based successor to Garnet that was released in 1994 and has been downloaded about 50,000 times. Garnet runs on the Unix and Macintosh platforms, and Amulet runs on the Unix, PC, and Macintosh platforms.

Both toolkits have introduced a number of innovations in dataflow constraints and have incorporated innovations from other constraint systems as well. The innovations in Garnet and Amulet include:

(1) supporting pointer variables in constraints [Szekely and Myers 1988; Vander

Zanden et al. 1991],

(2) supporting arbitrary code in constraints [Myers et al. 1990; Myers et al. 1997],

(3) automatically deducing the right hand side variables in a constraint at run-time [Vander Zanden et al. 1994],

(4) supporting unrestricted multi-output constraints [Vander Zanden 1992; Rosener 1994; Myers et al. 1997],

(5) supporting unrestricted side-effects in constraints [Vander Zanden 1992; Rosener 1994; Myers et al. 1997], and

(6) developing new constraint satisfaction algorithms that implement these innovations [Vander Zanden et al. 1994].

Innovations that were incorporated from other toolkits include path expressions that allow constraints to navigate their way through a tree of objects [Borning 1981; Sussman and Steele Jr. 1980], algorithms for implementing multi-output and side-effect constraints [Hill 1993], and algorithms for performing efficient, incremental constraint satisfaction [Reps et al. 1983; Hoover 1987; Alpern et al. 1990; Hudson 1991].

Over the duration of these projects we have received considerable feedback from application developers, from students, and from former members of the two projects. This paper reports on the feedback we have received and the implementation experience we have gained by working with different constraint satisfaction algorithms. We have divided the results into three sections:

(1) **User Issues**. This section describes users' experience with the constraint systems, including how they felt about various features of the constraint systems, how they used constraints in their applications, and their debugging experiences. Most users found constraints useful in constructing their applications. Their biggest complaint was the unpredictability of constraint evaluation and their difficulty in debugging them. The biggest use of constraints was to perform graphical layout.

(2) **Algorithmic Issues**. This section describes our experiences with different constraint satisfaction algorithms with which we experimented in the two toolkits, including mark-sweep algorithms and topological ordering algorithms. The biggest finding is that mark-sweep algorithms are both easier to implement and more efficient than topological-ordering algorithms. Surprisingly, topological-ordering algorithms, which seem simple conceptually, become very complicated and inefficient when they are extended to handle cycles and pointer variables in constraints.

(3) **Performance Issues**. This section describes the speed and storage efficiency of the constraint systems. The early implementations of constraint systems were driven by speed issues and storage was sacrificed in order to gain greater efficiency. However, our experience has shown that performance is not an issue, whereas storage considerations are an issue as applications grow larger and start to get pushed into virtual storage.

Despite the fact that our experiences have been gained using constraints in graphical interfaces, we believe that the lessons we have learned will benefit the programming languages community in a number of ways:

(1) Our experiences with users will be helpful to both designers of future constraint systems and to designers of the languages that may be used to implement these systems.

(2) The tradeoffs we encountered among the various constraint algorithms will be of universal value to implementors of one-way constraint solvers, regardless of application area.

(3) The programming languages community has focused on topological-ordering algorithms for performing constraint satisfaction because of their perceived performance advantage over mark-sweep algorithms [Reps et al. 1983; Hoover 1987; Alpern et al. 1990]. However, the empirical performance results we have obtained from graphical applications reveal that in the area of graphical interfaces, mark-sweep algorithms demonstrate a marked superiority. Since at least one survey shows that roughly 50% of code in modern applications is devoted to the user interface [Myers and Rosson 1992], finding the right algorithms to efficiently implement this code is crucial. The innovations that we have added to constraint systems, especially pointer variables and loops (loops in the sense of allowing a constraint to use a for or while statement rather than in the sense of circular constraints), have also revealed serious shortcomings in the adaptability of topological-ordering algorithms.

(4) Our experiences with users and with implementing constraint systems shows areas of potential future research interest for the programming languages community. These areas include finding more efficient storage schemes for constraints, finding better debugging techniques, and finding theoretically sound constraint satisfaction algorithms that can tolerate side-effects of the type described in this paper.

The remainder of this paper is organized as follows. Section 2 briefly provides some background about one-way constraint systems. Section 3 describes the evolution of one-way constraint systems as well as related work. Section 4 provides an overview of the Garnet and Amulet toolkits. Sections 5, 6, 7, 8, and 9 discuss user experience, design guidelines, constraint usage, algorithmic experience, and performance experience respectively. Finally Section 10 describes directions for future work and sums up the lessons we learned.

2. TERMINOLOGY

Definition. The introduction informally defined a *one-way dataflow constraint* as 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. More formally, a one-way constraint is an equation of the form

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

where each p_i is a variable that serves as a parameter to the function F. The function F is called a *formula*. Arbitrary code can be associated with F that uses the values of the parameters to compute a value. This value is assigned to variable v. If the value of any p_i is changed during the program's execution, v's value is automatically recomputed. v has no reciprocal influence on any p_i as far as this constraint is concerned. If v is changed by the application or the user, the constraint

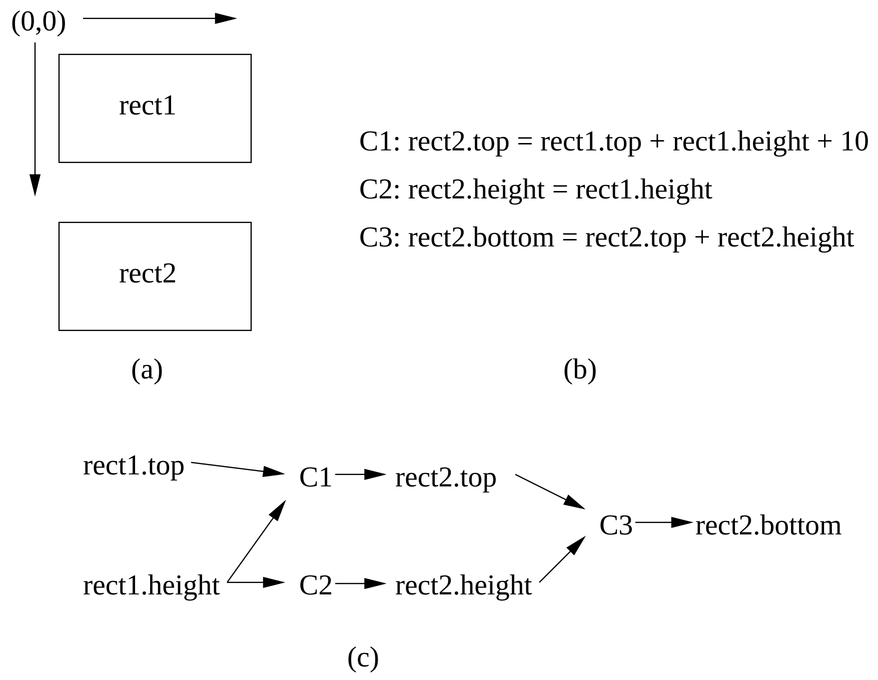

(0,0)
rect1
rect2
(a)
C1: rect2.top = rect1.top + rect1.height + 10
C2: rect2.height = rect1.height
C3: rect2.bottom = rect2.top + rect2.height
(b)
rect1.top
rect1.height
C1
C2
rect2.top
rect2.height
C3
rect2.bottom
(c)

Fig. 1. The dataflow graph (c) generated by the three constraints, C_1, C_2, and C_3 (b) which position the boxes in (a). C_1 positions `rect2` below `rect1`, C_2 makes `rect2` the same height as `rect1`, and C_3 computes `rect2's` bottom. The constraints assume that (0,0) is at the top left, as is in most windowing systems.

is left temporarily unsatisfied rather than trying to change one of the variables on the right-hand side. Hence, the constraint is *one-way.*

Dataflow Graph. A one-way constraint solver typically uses a bipartite, *dataflow graph* to keep track of dependencies among variables and constraints. Variables and constraints comprise the two sets of vertices for the graph. There is a directed edge from a variable to a constraint if the constraint's formula uses that variable as a parameter. There is a directed edge from a constraint to a variable if the constraint assigns a value to that variable. Formally, the dataflow graph can be represented as G = {V, C, E}, where V represents the set of variables, C represents the set of constraints, and E represents the set of edges. Figure 1 shows the dataflow graph for a sample set of constraints that positions one rectangle below another rectangle.

A dataflow graph can be constructed in one of three ways:

(1) Statically analyzing the syntax of the constraint's formula [Reps and Teitelbaum 1988]: If a parser can analyze a formula and automatically extract its parameters, then dependency edges from the parameters to the constraint can be established as soon as the constraint is created. A significant drawback of this approach is that it precludes the use of arbitrary code because loops, conditionals, and pointer variables can mask the true identity of parameters until run-time.

(2) Requiring the programmer to declare a formula's parameters: If the program-

mer declares the formula's parameters, then dependency edges from the parameters to the constraint can be established as soon as the constraint is created [Hill 1993; Hudson 1993; Hudson and Smith 1996]. Two significant drawbacks of this approach are that the programmer must write control code that duplicates the control code in the formula and that the programmer must remember to change the declaration if the formula is changed.

(3) Automatically deducing the formula's parameters as the formula executes [Vander Zanden et al. 1991; Hoover 1992; Vander Zanden et al. 1994]: Using a technique described in the appendix, the constraint solver can deduce a formula's parameters as the formula executes. This approach has the advantage that the programmer does not have to declare the formula's parameters. It has the disadvantage that the dataflow graph must be constructed dynamically during constraint evaluation. Whether or not this disadvantage is a hindrance depends in part on the algorithm used for constraint evaluation. The appendix shows that automatic parameter detection can be done rather simply for a mark-sweep algorithm (discussed next).

Constraint Satisfaction. *Constraint satisfaction* refers to the process of bringing constraints up-to-date by evaluating their formulas and assigning the results to the left side variables. The two schemes used for one-way constraint satisfaction are the mark-sweep strategy [Demers et al. 1981; Reps et al. 1983; Hudson 1991; Vander Zanden et al. 1994] and the topological-ordering strategy [Reps et al. 1983; Hoover 1987; Alpern et al. 1990; Vander Zanden et al. 1994]. A mark-sweep algorithm has two phases as the name suggests. The mark phase tags each constraint that depends on the changed variables as out-of-date. It does so by performing a depth-first search of the dataflow graph, beginning at the changed variables, and marking each constraint it encounters out-of-date. For example, if `rect1.top` in Figure 1 is changed, C_1 is marked out-of-date and then C_3 is marked out-of-date. C_2 is not marked out-of-date because it does not depend on `rect1.top`. In the sweep phase, out-of-date constraints whose values are requested are evaluated and the constraints are marked as up-to-date. If constraints are only evaluated when their values are requested, then the sweep phase is called a *lazy* evaluator. If all the out-of-date constraints are evaluated as soon as the mark phase is complete, then the sweep phase is called an *eager* evaluator. An example mark-sweep algorithm is presented in Appendix A.

A topological ordering algorithm is one which assigns numbers to constraints that indicate their position in topological order. In Figure 1, C_1 might be assigned 1, C_2 2, and C_3 3. The numbers may be arbitrary so long as they satisfy the property that constraints occurring later in the topological order have a larger number than constraints occuring earlier in the topological order. More formally, if a constraint C_i depends on a variable computed by constraint C_j, then the number assigned to C_i must be greater than the number assigned to C_j. Topological orders are not necessarily unique. In Figure 1, there are two equally acceptable topological orders, $\{C_1, C_2, C_3\}$ and $\{C_2, C_1, C_3\}$. Constraints that do not depend on one another, such as C_1 and C_2, may have arbitrary numbers relative to one another (i.e., C_1`'s` number may be less than, equal to, or greater than C_2`'s` number).

Like the mark-sweep strategy, the topological ordering strategy has two phases–a numbering phase that brings the topological numbers up-to-date and a sweep phase that evaluates the constraints. The numbering phase is invoked whenever an edge in the constraint dataflow graph changes. The sweep phase can either be invoked as soon as a variable changes value or it can be delayed to allow several variables to be changed. The sweep phase uses a priority queue to keep track of the next constraint to evaluate. Initially, all constraints that depend on a changed variable are added to the priority queue. The constraint solver removes the lowest numbered constraint from the queue and evaluates it. If the constraint's value changes, all constraints that depend on the variable determined by this constraint are added to the priority queue. This process continues until the priority queue is exhausted. For example, if `rect1.top` is changed, then C_1 is added to the priority queue. It is then removed from the priority queue and evaluated. If its value changes, then all direct descendents of `rect2.top`, the variable computed by C_1, are added to the priority queue. In this case the only direct descendent is C_3. It would be removed from the priority queue and evaluated. The process would then terminate because there are no direct descendents of `rect2.bottom` and the priority queue is now empty. An example topological-ordering algorithm is presented in Appendix A.

In evaluating the performance of constraint satisfaction algorithms, two metrics are often used [Reps et al. 1983; Alpern et al. 1990]:

(1) AFFECTED—the set of constraints that must be re-evaluated because one of their inputs has actually changed.
(2) INFLUENCED—the set of constraints that potentially must be re-evaluated because one of their inputs has potentially changed.

In the general case, satisfaction algorithms only have to evaluate $O(|\text{AFFECTED}|)$ constraints but must examine $O(|\text{INFLUENCED}|)$ constraints [Alpern et al. 1990]. It is difficult to say whether a mark-sweep or a topological ordering algorithm will be faster for a particular application without implementing both and running them head to head. Theoretically, a topological ordering algorithm examines fewer constraints during both of its phases than a mark-sweep algorithm but this savings is offset by considerably higher overhead incurred by priority queues and by the data structures required to keep the numbers up-to-date. These issues are explored in greater detail in Section 8.

3. RELATED WORK

This section begins by focusing on the related work for one-way constraints and then briefly considers other constraint-related research.

3.1 One-Way, Dataflow Constraints

One-way constraints were probably first used in attribute grammars [Knuth 1968]. In the 1980s, developers of syntax-directed editors developed incremental topological-ordering algorithms that allowed attribute grammars to be used as the basis for interactive programming environments [Reps et al. 1983; Reps and Teitelbaum 1988]. These algorithms exploited a restriction in attribute grammars and a restriction in the editing model that allowed them to both examine and evaluate only $O(|\text{AFFECTED}|)$ constraints. The attribute grammar restriction is that constraint equations can only reference attributes of the grammar symbols on the left and right side of a production. This restriction gives rise to limited types of dataflow graphs.

The editing model restriction was that an edit could only occur at one point in an attributed tree. These two restrictions made the dataflow graphs amenable to static analysis that could be exploited by the constraint satisfaction algorithms. The restrictions on single edits was eventually removed but the restriction on the dataflow graphs remained [Reps 1987; Repts et al. 1986].

Concomitantly with the development of syntax-directed editors, spreadsheets popularized the use of one-way constraints for non-programmers. Unlike syntax-directed editors, users could create constraint equations that could access arbitrary variables and hence the specialized satisfaction techniques developed for syntax-directed editors were not applicable. However, a simple depth-first search algorithm can be used to add constraints to a list in topological order and then to evaluate the constraints on this list [Ross 1985]. This simple satisfaction algorithm evaluates $O(|\text{INFLUENCED}|)$ constraints but has proven fast enough for interactive performance in spreadsheets.

In the middle to late 1980s, developers of applications like graphical interfaces and circuits started to use one-way constraints [Barth 1986; Myers 1990; Myers et al. 1990; Myers et al. 1997; Hill 1993; Hill et al. 1994; Hudson and King 1988; Hudson 1993; Henry and Hudson 1988; Hudson 1994; Hudson and Smith 1996; Alpern et al. 1990]. Like spreadsheets, the constraint systems that arise in these applications are not restricted in the ways that they are in attribute grammars and hence the algorithms that were evolved to perform incremental computation for attribute grammars have not been used in these applications.

The switch to unrestricted constraint systems led to the development of new, general purpose topological ordering algorithms. Hoover devised an approximate topological ordering scheme that used order numbers to keep constraints in approximate topological order [Hoover 1987]. Since constraints were only in approximate topological order, a constraint could be evaluated more than once. This algorithm worked well in the restricted world of attribute grammars but performed poorly in an experimental implementation in Garnet. In collaboration with a number of other researchers, Hoover later devised a second topological ordering scheme that kept constraints in precise topological order and evaluated each constraint at most once [Alpern et al. 1990]. In the same paper Hoover and his colleagues proved that in the general case it is not possible to examine only $O(|\text{AFFECTED}|)$ constraints, which made algorithms that examined $O(|\text{INFLUENCED}|)$ attributes much more palatable.

The switch to unrestricted constraint systems and the finding that $O(|\text{INFLUENCED}|)$ attributes must be examined in the general case also led to a resurrection of mark-sweep algorithms. These algorithms had previously been considered and rejected in the context of syntax-directed editors because they examined $O(|\text{INFLUENCED}|)$ constraints, which was more constraints than turned out to be necessary. However, in the context of unrestricted constraint systems, Hudson proved that when used as a lazy evaluator, a mark-sweep algorithm evaluates the minimum number of constraints possible [Hudson 1991]. The bound is better than an eager evaluator can achieve since a lazy evaluator can avoid constraint evaluations whose values are never needed by the application. Since topological-ordering algorithms cannot be used as lazy evaluators (see Section 8.1.1), mark-sweep algorithms gained greater acceptability.

By the early 1990s the basic constraint satisfaction strategies had been established. The programming languages community turned its focus to function caching and partial evaluation as other techniques for incremental computation [Pugh and Teitelbaum 1989; Sundaresh 1991; Sundaresh and Hudak 1991; Liu et al. 1998]. These techniques can be used in concert with incremental one-way constraint satisfaction, or, in the case of Pugh's function caching work `FunctionCachingPugh`, to replace incremental constraint satisfaction algorithms (although it appears that in general one-way constraint systems, Pugh's function caching techniques work best when combined with a mark-sweep algorithm). These techniques were not used in Garnet or Amulet because constraint satisfaction performance was acceptable without these techniques.

In the 1990s the graphical interfaces community turned its focus to developing new features for users. Garnet introduced the notion of pointer variables as first class objects in constraint systems [Vander Zanden et al. 1991; 1994]. Unrestricted pointer variables were subsequently supported by Alphonse [Hoover 1992], Rendezvous [Hill 1993], and EvalVite [Hudson 1993]. Our experiences with pointer variables is described further in Section 6.1.

Multi-output constraints were first proposed by one of the authors in [Vander Zanden 1992]. A multi-output constraint allows a formula to compute the value of more than one variable. Rendezvous appears to have been the first system to actually implement multi-output constraints [Hill 1993; Hill et al. 1994]. Amulet and an experimental version of Garnet [Rosener 1994] provided later implementations of multi-output constraints.

Side-effect constraints in which formulas were allowed to commit side-effects were independently proposed for both Garnet [Vander Zanden 1992] and Alphonse [Hoover 1992]. However the first actual implementation appears to have been in Rendezvous [Hill 1993; Hill et al. 1994], followed by an experimental implementation in Garnet [Rosener 1994] and a released implementation in Amulet [Myers et al. 1997]. A side effect allows a formula to perform useful tasks such as creating new objects or deleting objects. Our experiences with multi-output and side-effect constraints are described in Section 6.1.

Finally, path expressions, which were first introduced in the late 1970s in two multi-way constraint systems, ThingLab [Borning 1981] and Constraints [Sussman and Steele Jr. 1980], found their way into the formulas for one-way constraints. A path expression lets formulas navigate their way through a tree of objects. Garnet and Amulet extended path expressions by allowing the tree of objects to be dynamically modified. Our experiences with path expressions are described in Section 6.4.

Despite the many constraint systems that have been developed, the only previous study of which we are aware that examined how *users* employed constraints was an early study of Amulet [Vander Zanden and Venckus 1996]. That study profiled a number of Amulet applications and found that even as the size of applications grows, the number of constraints that are influenced by any particular edit does not tend to grow because the dataflow graph fractures into many smaller, independent graphs. Other papers have included brief characterizations of users' preliminary experience with constraints but the current paper is the first one that attempts to summarize user feedback about a constraint system based on a decade worth of experience.

3.2 Other Types of Constraints

One-way constraints are not the only type of constraint developed by the programming languages community. A number of graphical interface toolkits and languages have incorporated multi-way, dataflow constraints, including SketchPad [Sutherland 1963], ThingLab [Borning 1981], Constraint [Vander Zanden 1988], Multi-Garnet [Sannella and Borning 1992], and Kaleidoscope [Freeman-Benson 1990]. It is also possible to build graphical interfaces and programming languages around more powerful types of constraint solvers. A number of toolkits involving graphical interfaces use constraint solvers that can solve sets of simultaneous linear equations or linear equations and inequalities, such as SketchPad [Sutherland 1963], ThingLab [Borning 1981], IDEAL [Wyk 1982], Juno [Nelson 1985], Bertrand [Leler 1988] and Bramble [Gleicher 1993]. A number of constraint solvers have also been proposed for graphical applications or could be used with graphical applications, including linear constraint solvers [Golub and Van Loan 1989], non-linear constraint solvers [J.E. Dennis and Schnabel 1983; Witkin et al. 1990; Witkin and Welch 1990; Gleicher and Witkin 1992], and linear equality and inequality solvers [Borning et al. 1997; Borning et al. 1996; Hosobe et al. 1996; Hosobe et al. 1994; Jaffar et al. 1992; Lassez and McAloon 1992; Lassez and Lassez 1991; Helm et al. 1992; Huynh et al. 1992]. These algorithms are domain-specific algorithms that are capable of solving more expressive mathematical constraints than can be solved by a dataflow solver. However, they are restricted to the mathematical domain which excludes many types of constraints that are useful in graphical interfaces. For example, 36% of the constraints defined in the applications we examined computed non-numeric results (e.g., choosing a color based on whether or not an object is selected). Additionally, domain-specific constraints typically require a good deal of mathematical knowledge on the part of the programmer, which limits the usability of these types of constraints.

The Constraint Logic Programming community represents another significant area of constraint-based research [Jaffar et al. 1992; Dincbas et al. 1988; Borning et al. 1989; Saraswat 1989; Cohen 1990]. Constraint solving alternates with a resolution process, such as unification, in order to produce a solution to a problem. Constraint logic programming is an example of refinement-based constraint solving, in which variables are initially assigned a range of values, and then the range is gradually refined during the unification and constraint satisfaction process. The end result may be a range of values for a variable rather than one unique value. Dataflow constraints and constraint logic programming serve different niches in the programming languages community. Dataflow constraints are best adapted to perturbation-based applications, such as interactive applications, that store state information and that incrementally perturb this information by modifying one or more variables and then updating the remaining variables. Constraint logic programming is best adapted to refinement-based applications, where a set of axiomatic statements and constraints are provided, and then an answer is derived for a particular set of data. Unlike dataflow constraints, this answer may consist of a range of values for given variables, or even one or more equations that relate the values of given variables (i.e., an exact answer is not provided).

4. GARNET AND AMULET OVERVIEW

The constraint systems in Amulet and Garent are just one part of a highly integrated collection of features designed to make it significantly easier to create highly interactive, graphical applications. These features include a prototype-instance model, structured graphics, a composite object mechanism, and a high-level event-handling mechanism. Understanding how the constraints are used requires understanding these other features as well.

4.1 Prototype-Instance Model

Garnet and Amulet support a prototype-instance system, in which any object can serve as a prototype for another object [Lieberman 1986; Borning 1986; Ungar et al. 1992; Myers et al. 1990]. 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. In the parlance of the prototype-instance models, this type of inheritance is called *delegation*. Slots can be dynamically added to and deleted from an object. If a slot is deleted from an object, the slot will subsequently inherit its value from the 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). All assignments to a slot must be done via a `Set` method, so both the Garnet and Amulet run-time systems can detect a formula assignment and create a constraint object, which is what actually gets assigned to the slot. The constraint object contains a pointer to the formula object.

Constraints are inherited from a prototype like any other value. In particular, a prototype's constraint objects are cloned and stored in an instance's slots, 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.

A significant advantage of a prototype-instance model is that it supports rapid prototyping when building a graphical application. Two ways in which it supports rapid prototyping are:

(1) A programmer only has to specify values for slots whose properties should differ from the prototype. All remaining values are inherited from the prototype. In essence, the prototype contains "default" values for all properties in the instance.

(2) By changing a property in a prototype, a designer can dynamically change the look of all the objects that inherit from that prototype. In a typical class-instance system, the designer would have to shut down the application, change the code, re-compile the application, and then re-execute it.

4.2 Structured Graphics

In a *structured graphics* model, each graphic element on the screen is represented by an object in memory. "Structured graphics" is sometimes also called a "retained object model" or "display list." The advantage of this model is that the programmer is freed of all display maintenance tasks. For example, to change the color of an object, the programmer only needs to set the color parameter. The system automatically updates the display by redrawing the object and any other objects on the screen that intersect the object. Furthermore, if the window manager requests that the window be redrawn, perhaps because it was de-iconified or uncovered, the structured graphics system can handle these tasks without involving the application at all. A structured graphics model also makes it possible to provide high-level services in the toolkit, including printing, saving and reading objects from files, cut/copy/paste/duplicate and other edits, and graphical selection handles. While other frameworks require the programmer to override the standard methods for these functions, a structured graphics model usually allows the programmer to use the library functions *without change*.

The Amulet and Garnet constraint systems support the structured graphics model in three ways:

(1) Computation of Slot Values: Constraints can be attached to the slots of structured graphics objects, thus allowing the properties of an object to be automatically recomputed when the user or application changes the values of one or more other properties.

(2) Computation of Objects: In Amulet, constraints can be used to determine *which* objects should be included in a window, by creating and deleting objects in the constraints.

(3) Automatic Notification of Changes: Whenever a slot is recomputed by a constraint, the constraint solver notifies the structured graphics system so that the object containing the slot can be appropriately redrawn.

4.3 Composite Objects

A *composite object* is an object made up of other objects [Gamma et al. 1995]. These other objects may either be primitive objects or themselves be composite objects. Composite objects are sometimes called "groups" or "aggregates." Figure 2 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 2b.). 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. Garnet and Amulet both provide mechanisms that allow a programmer to define names for each of the parts [Myers et al. 1990; Myers et al. 1997]. In the above example, the programmer has assigned the names "label" and "frame" to the two parts. These names have been converted to two slots named `label` and `frame` respectively. The slots have been stored in the graph node object and each of the slots has been assigned a pointer to its respective part.

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

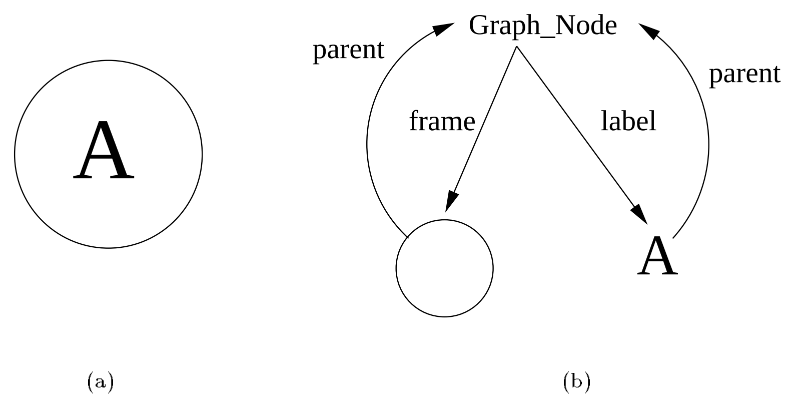


Fig. 2. A graph node (a) and its structural components (b).

of the parts.

The advantage of structural inheritance is that a programmer is able to treat a composite object like any other object in the object system, without having to worry about how the object is composed. A second advantage of composition is that the programmer is able to build up arbitrarily complicated objects from the primitive graphical objects, like rectangles and text objects, provided by the structured graphics system.

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.

4.4 Constraints

Both Garnet and Amulet support one-way constraint systems. Some of the features supported by these constraint systems include:

(1) Arbitrary code: A formula can contain any code that is legal in the underlying toolkit language. Hence a Garnet formula can contain arbitrary Lisp code and an Amulet formula can contain arbitrary C++ code. In particular, a formula can contain arbitrary loops, conditionals, function calls, and recursion.

(2) Pointer variables: A formula can reference variables indirectly via pointers. For example, an object can be made to appear 10 pixels to the right of the previous object in a list by writing the constraint:

$$\text{left} = \text{self.prev.right} + 10$$

where `self` is a pointer to the object containing `left` and `prev` is a pointer to the previous item in the list[1]. If the `prev` pointer is changed, the object will adjust itself so that it appears 10 pixels to the right of the new object pointed to by the

[1]In C++ this equation would be written as `left = self->prev->right + 10` and in Java it would be written as `left = self.prev.right + 10`. We have chosen to use the dot (.) notation in this paper.

`prev` pointer. Although pointer variables are common in programming languages, Garnet was the first system to support pointers as first-class variables in constraints [Vander Zanden et al. 1994]. Previous systems had allowed pointers in path names [Borning 1981; Sussman and Steele Jr. 1980] or as specially declared variables with special accessor methods and restrictions, such as not being able to determine the pointer variables themselves using constraints [Szekely and Myers 1988]. However, Garnet was the first system to allow the unrestricted use of pointer variables.

(3) 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.

(4) 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 slot. This method takes the name of a slot as an argument and returns the slot's value. The *Get* method is responsible for determining whether a slot is computed by a constraint, and if so, ensuring that the constraint is up-to-date before returning the slot's value.

(5) Automatic Parameter Detection: Garnet and Amulet both automatically deduce a constraint's parameters as the constraint executes so the programmer does not have to declare a constraint's parameters.

(6) Support for Cycles: Garnet and Amulet both support once-around evaluation of cycles. Once-around evaluation means that a constraint in a cycle is evaluated at most once. If the constraint is asked to evaluate itself a second time, it simply returns its original value. For example, suppose we have the cyclical constraints `a = b` and `b = a` and that both are invalid. Both Garnet and Amulet guarantee that each constraint will be evaluated at most once. In fact, both systems are somewhat more sophisticated than this. If the application sets `a` to 6 (`a = 6`), the constraint `a = b` is marked up-to-date and the constraint `b = a` is invalidated. Hence, only `b = a` gets re-evaluated and `a's` value gets properly propagated. However, in more complicated cases, where all the constraints in a cycle get invalidated, Garnet and Amulet still guarantee that each constraint will be evaluated at most once. The advantage of cycles is that one gets some of the advantages of multi-way constraints without the overhead or the usability problems (multi-way constraints often have multiple solutions and it can require careful tuning by the programmer to force the constraint solver to choose the desired solution). Indeed, one of the authors informally surveyed a number of applications written in Multi-Garnet, a version of Garnet designed to support multi-way constraints [Sannella and Borning 1992], and found that the most common type of multi-way constraint was a simple bi-directional equality constraint of the form `a = b`. Hence, for the applications surveyed, once around evaluation of cycles provides support for the single largest use of multi-way constraints.

These features make Garnet and Amulet's constraint systems the most expressive and flexible systems available to graphical interface developers. Previous systems have been more restrictive, because 1) they offer only predefined constraints, such as the layout mechanisms in the InterViews [Mark A. Linton and Calder 1989] and Java Swing toolkits, 2) they use their own special constraint language that has restricted functionality, such as Higgens [Hudson and King 1988] or Penguims [Hudson 1994], 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 [Hill 1993; Hudson 1993].

Because of the expressiveness of its constraint system, Garnet was the first toolkit to integrate constraints with the lowest-level object system and Amulet adopted this approach as well. As a result, constraints are used ubiquitously throughout the Garnet and Amulet widget sets and all Garnet and Amulet applications. A large Garnet or Amulet application might have 20,000 constraints.

4.5 Event-Handling Model

Garnet and Amulet introduced a novel, high-level, event-handling model called *interactors*. The interactors model is based on the observation that users interact with graphical interfaces in a few stereotypical ways, such as creating new objects, moving/resizing objects, choosing one or more objects from a collection of objects, or editing a text string. The interactors model encapsulates each of these stereotypical behaviors into a prototype interactor object. For example, Amulet provides the following six interactor objects:

(1) Choice Interactor: Allows the user to select one or more objects from a collection of objects.
(2) Text Edit Interactor: Allows the user to edit a text string.
(3) Move/Grow Interactor: Allows the user to move or resize an object.
(4) New Points Interactor: Allows the user to create an object by entering an arbitrary number of points.
(5) Gesture Interactor: Allows one of a pre-defined set of gestures to be recognized based on the path traced out by the mouse. A gesture can be any path traced out by a mouse, such as a circle, a line, a checkmark, the letter D, etc.
(6) One Shot Interactor: Allows an action to be executed immediately when an event happens, such as a key being pressed on the keyboard.

The programmer creates an interactive behavior by creating an instance of the appropriate interactor[2]. The programmer then customizes the behavior by modifying the instance's properties (each prototype interactor defines a rich set of properties that can be inherited or overridden).

Properties may be divided into behavior-defining properties that control an interactor's behavior and graphics-defining properties that describe which graphical objects are affected by the interactor.

The most fundamental behavior-defining properties describe the start, stop, abort, and running events for the interactor and the action procedures that should be executed when each of these events are received. Running events are events received while the interactor is running, such as mouse moved events for a move/grow interactor or keyboard events for a text editing interactor. For example, a move/grow interactor might specify a left mouse button pressed event as the start event, a left

[2]Both Garnet and Amulet refer to instances and prototypes of interactor objects as *interactors*. This paper adopts this terminology, so the term interactor means an interactor object, and this object may be either a prototype or an instance.

mouse button released event as the stop event, a mouse moved event as the running event and a Ctrl-G key as the abort event. The procedure associated with the start event for a move/grow interactor might make a dashed line feedback object appear, the procedure associated with the running event might make the feedback object track the mouse, the procedure associated with the stop event might make the feedback object disappear and move/resize the object that was under the mouse when the start event was received, and the abort procedure might make the feedback object disappear.

Each interactor also defines additional behavioral properties that are interactor-specific. For example, the behavior-defining properties for a move/grow interactor might specify whether the interactor is moving or growing an object, whether a line or non-line object is being acted upon, whether there is a minimum size for a resized object, and whether the object being acted on should snap to a grid.

The graphical properties for an interactor specify which graphical objects it operates on and what type of feedback it provides. For example, a *start-where* property lists the objects the interactor operates on and an *interim-feedback* property provides a graphical object or objects that the interactor should display as it operates so that the user can see what the interactor is doing.

The separation of the behavioral specification from the graphical specification of the interactors makes it possible to reuse the interactors in many different ways. For example, a choice interactor can be used to select objects in an editor or to implement a radio button panel. A new point interactor can be used to create a new object or to specify a selection area on the screen (e.g., it can sketch out a rectangular area which is then used to select all the objects that intersect that area).

Constraints support the interactors model in two ways:

(1) They can be used to compute the properties of an interactor. For example, a constraint can be used to select a feedback object based on the type of the object that the interactor is currently operating over.

(2) They automatically propagate changed values to the appropriate graphical objects, so the action procedures can be relatively succinct. For example, a move/grow interactor might change the `left` and `top` slots of an object. In a boxes and arrows editor, constraints would automatically ensure that any arrows that are attached to the moved object are moved as well. Similarly, if the object is a composite object with a frame and a label, constraints will ensure that both the frame and label are appropriately repositioned.

4.6 Programming Model

The programming model that a programmer uses to create an interface can be described as follows. The programmer writes an initialization procedure that creates a collection of graphical objects that comprise the interface and attaches constraints to the objects in order to compute various properties. In the initialization procedure the programmer also creates a collection of interactor objects that describe the behavior of the interface. The programmer writes a series of action procedures that are attached to each interactor which update the graphics on the display and call appropriate application procedures. Typically the action procedures update the graphics by modifying certain properties in the object that the interactor is operating on. Constraints then automatically propagate these changes to any other graphical objects that need to know about them. The application procedures perform whatever computations are needed to implement the application, and if necessary, they also change the properties of appropriate graphical objects. Again, the changes tend to be limited to a few objects since the constraints can be used to propagate the changes to the remaining objects on the display.

The programmer then writes a short main procedure that 1) calls a procedure that starts-up the Garnet or Amulet run-time system, 2) calls the initialization procedure and, 3) calls an event-handler provided by either Garnet or Amulet. The event-handler enters an event loop and handles events by dispatching them to the appropriate interactor based on the interactors' start-where property. The selected interactor then calls the appropriate action procedure, which causes the graphics on the display to be updated and the appropriate procedures in the application to be executed. These application procedures may perform certain computations that also cause parts of the graphical display to be updated.

The event loop continues indefinitely until an event finally activates an interactor that notifies the event handler that it should exit the event loop. For example, when a quit button is pressed the interactor associated with the quit button will notify the event handler that it should exit the event loop.

Once the event loop exits, the programmer written main procedure calls a procedure that shuts-down the Garnet or Amulet run-time system and exits.

Most main procedures look as follows:

```
begin main()
    InitializeAmulet();
    InitializeInterfaceObjectsAndInteractors();
    AmuletEventLoop();
    ShutdownAmulet();
end
```

5. OVERALL USER EXPERIENCE

In preparing this paper, we posted separate electronic survey forms for the Garnet and Amulet constraint systems 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 users and 12 Amulet users. The information we received in these responses has been combined with our own experience with users, students, and project members and is reported in the next two sections. This section summarizes general feedback that we have received regarding constraints. The next section describes design guidelines we have learned that we feel are applicable to constraint systems in general.

The general feedback we have received about constraints can be summarized as follows. Most users reported that they found constraints helpful for creating their interactive applications. Users were most impressed with the flexibility they provided for defining custom graphical layouts, for computing graphical properties, and for the modularity they promoted. Users were most concerned with how difficult constraints could be to debug and with how they sometimes had to worry about the evaluation order of constraints.

5.1 Graphical Layout

Users reported that they primarily used constraints to specify graphical layout and to a lesser extent, to compute graphical properties. In our Garnet and Amulet surveys, we asked users to compare using constraints for layout against 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, users 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 constriants that partially but not fully meets these needs). However, users were very positive about the flexibility afforded by constraints in defining their own custom layouts and in laying out objects created dynamically. Some of the comments we received included:

— "Constraints are the ONLY good way to lay out objects that are created on the fly, and that have specific relations to other objects... e.g., laying out hierarchical task network graphs or building 'ribbon-chart' displays of schedules. Pre-defined layout managers do not handle this situation at all well."

— "Simple parameterizable layout managers are preferable most of the time. When my layout needs are simple, I'd rather just grab a BorderLayout off the shelf than muck around with constraints. But constraints are better as a fundamental infrastructure that you can fall back on when the layout managers don't satisfy your needs. Even if your toolkit supports user-defined layout managers, defining a few constraints is far easier than designing a new layout manager."

Users 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:

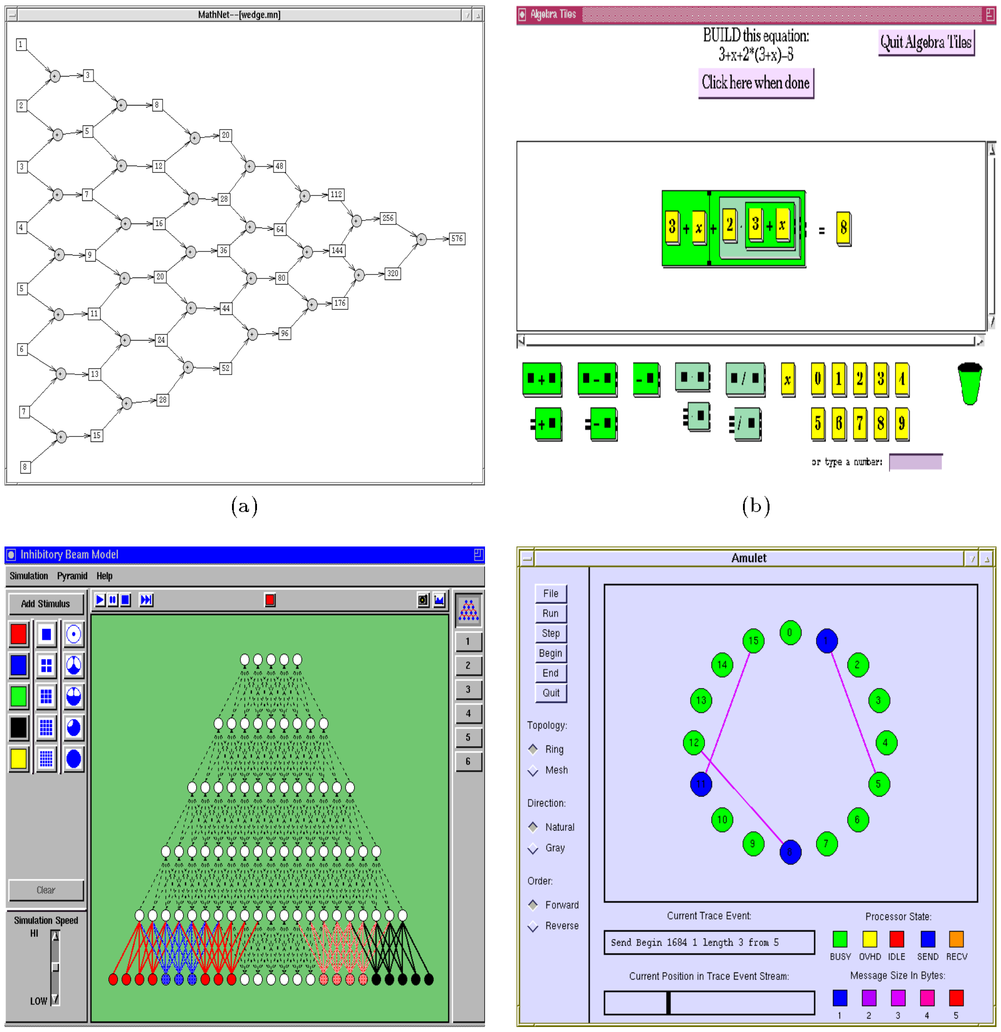
(1) Connecting objects using arrows,
(2) Tiling algebraic expressions,
(3) Graph layouts,
(4) Tree layouts,
(5) Grid layouts,
(6) Ring layouts, and
(7) "Ribbon-chart" displays of schedules.

Pictures of several of these types of layouts are shown in Figure 3.

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.

5.2 Modularity

A number of users reported that they liked the fact that constraints made their applications more modular. Two comments that we received were:



BUILD this equation:
3+x+2*(3+x)-3

Send Begin 1684 1 length 3 from 5

Fig. 3. Some of the layouts created by Amulet users, including (a) connecting objects using arrows, (b) tiling algebraic expressions, (c) creating tree layouts, and (d) creating ring layouts.

— "Constraints help to encapsulate behavior directly into the object. You could change an object without having to modify objects that were constrained to it or from it."

— "[I liked] the 'setup once, never look at it again' feel of it. Compare this with TCL/TK.... With constraints you don't have to set all these 'state variables' at interesting events. It all goes automatically. Amulet scales very well, just because of the constraint system (constraints are local to objects)."

In this case, what is meant by modularity is that from an application programmer's point of 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 set and delete operations. The programmer must maintain data structures to keep track of the interconnections and must perform additional notification or bookkeeping work whenever a set 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.

5.3 Debugging

Users 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, and 3) constraints started to look like spaghetti code as the number of constraints increased. Some of the representative comments we received regarding debugging included:

— "In general debugging's easy because of help given by the system, but when there are situations of complex cycles, one hasn't any clue in what part of the system the cycle can be."

— "They are very nice for getting something working quickly and easily. Later, as the program grows in size and complexity however, many of my constraints have become more spaghetti-like and now pose challenging debugging problems."

— "[Debugging was] incredibly, frustratingly hard (on the order of several hours spent per serious bug)...hard because of the non-linear execution of constraint programs—setting a slot can have the side effect of prematurely triggering a re-evaluation of constraints (or different order of constraint evaluation) which leads to a bug—but the trigger that set off that bug can be anywhere in the program code."

— "It is hard to set a breakpoint on the failing invocation of a constraint. Since it may run and fail many times in trivial ways as the program initializes, it can be hard to find the 'interesting' failing execution, the one that actually matters. ...Constraints may fail far from the point where the actual first failure occurred. Better ways of tracing how failures propagate through the constraint network would help."

The comments for the most part are self-explanatory. The comments about failures happening far from the point of the first actual failure reflects the fact that constraints form chains that can propagate a bad value far from the point where it is first set or computed. Hence the first constraint to break may be far from the actual point where the bug first occurred.

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 users 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 users said that cycles would be far easier to detect using such a visual editor.

A second tool that two users said would be helpful is the ability to dynamically add and remove constraints from slots using the inspector. They indicated that this capability would give them the ability to see what effect constraints were having on the system or to see if they could break a constraint cycle.

Interestingly, a previous study of spreadsheets found that users' spreadsheets contained substantial numbers of errors [Brown and Gould 1987]. Those findings, combined with our findings, suggest that the creation of effective debugging tools should be a top priority for constraint researchers.

5.4 Getting Constraints to Execute at the Right Time

One-way constraint solvers are supposed to relieve programmers of the burden of worrying about when and how constraints are re-evaluated. However, Garnet and Amulet programmers experienced problems with both the premature evaluation of constraints, and less often, with constraints not getting evaluated when they expected them to be evaluated.

5.4.1 *Premature Evaluation in Amulet.*. Amulet programmers experienced problems with premature evaluation. Programmers had to "harden" the code of the constraint's formula by introducing conditional statements that test whether the requested slot has been initialized, and if not, to return a default value. The problem was caused by the fact that Amulet uses an eager evaluator. Eager evaluation brings constraints up-to-date as soon as possible, both when they are first created and when they are later marked out-of-date. The key problem is that "as soon as possible" is ambiguous. One solution is to force the programmer to tell the constraint solver when to initiate constraint satisfaction. However, when this manual approach was experimentally tried in Garnet, we found that it was both too easy to forget to invoke the constraint solver and annoying to have to do so. So Amulet's constraint solver is invoked automatically by the system, but sometimes it is invoked before all the necessary slots have been initialized. This in turn leads to constraint crashes, debugging, and requires hardening of the constraint's formula code.

In early versions of Garnet, the premature evaluation problem also occasionally manifested itself and it was solved using the following technique:

(1) Programmers were allowed to specify a default value for a constraint. This value was returned if the constraint could not be successfully executed.

(2) The constraint solver was modified so that it checked whether a constraint's formula was accessing an uninitialized slot. When an unitialized slot was accessed, the constraint solver terminated the constraint's execution and returned the constraint's default value.

This solution was not implemented in Amulet because it requires a `try/catch` construct that can be terminated if a statement protected by the `try/catch` construct fails. At the time that Amulet was implemented, most C++ compilers either did not have such a construct, or else had a unique mechanism for handling the construct. We also found that applications that id use it were extremely slow and inefficient. Since Amulet was meant to be portable, it could not use the `try/catch` construct and hence had to forego this solution.

Even if this premature evaluation problem is fixed, there is still the problem of repeated, unnecessary evaluations of a constraint as its formula parameters become, one at a time, initialized. However, since constraint evaluation is such a small percentage of the total execution time of an application (see Section 9.1), this problem is less significant than the premature evaluation problem.

5.4.2 *Lack of Constraint Evaluation in Garnet..* Garnet programmers experienced problems with constraints not getting evaluated. This problem was caused by two different shortcomings of the constraint solver:

(1) Dependencies not getting properly established. In both Garnet and Amulet, programmers can explicitly set slots whose values are also computed using a constraint. One reason for doing this is to propagate values through a constraint cycle (sometimes users intentionally create cycles) and another is to set a slot with a temporary value. If the slot is set before the slot's constraint is evaluated, the slot is marked up-to-date and the constraint is never evaluated. Because the constraint is never evaluated, the constraint solver cannot determine on which parameters the constraint depends. Therefore, it cannot establish dependencies from these parameter slots to the constraint. As a result, the constraint is never notified of changes to these parameter slots and the constraint is not re-evaluated when the user expects it to be. In Amulet this problem was solved by placing all new constraints on a queue and evaluating them, regardless of whether or not the slots to which they are attached are up-to-date. If the slot is up-to-date, the constraint is computed but its value is temporarily discarded. This evaluation allows the dependencies to be established so that it is correctly re-evaluated in the future.

(2) Not being able to tell the constraint solver to always keep a slot up-to-date. Unlike Amulet, Garnet uses lazy evaluation. This means that a constraint will not be automatically evaluated unless the constraint's value is explicitly demanded. Programmers would therefore occasionally be surprised or bewildered when a constraint they expected to be re-evaluated was not re-evaluated. A partial solution to this problem would have been to allow a programmer to specify that a slot should always be kept up-to-date. Then the constraint associated with that slot would always be re-evaluated when one of its parameters changed. This solution would still require the programmer to know that a slot must be declared as an "always up-to-date" slot, which is why the solution is only a partial one. The lack of a satisfactory solution to this problem is one reason we switched to eager constraint solving in Amulet.

Lessons Learned. The problem with premature evaluation in Amulet and lack of evaluation in Garnet reveals that both eager and lazy evaluation still have problems that need to be resolved. In our implementations, the premature evaluation associated with eager evaluation was far more problematic for users than the occasional lack of evaluation caused by lazy evaluation. However, `try/catch` constructs that protect the premature evaluation of a constraint, in conjunction with default values, seem to provide a solution to the problem of premature evaluation. Similarly, allowing users to tag a slot as a slot to be eagerly evaluated seems to provide at least a partial solution to the problem of not evaluating a constraint in lazy evaluation. Further testing of these two solutions is required before more definitive statements can be made about the choice of either lazy or eager evaluation.

6. DESIGN GUIDELINES FOR CONSTRAINT SYSTEMS

This section describes some of the design guidelines we learned in working with users. The guidelines can be summarized as follows:

(1) Use the underlying language: A constraint should be able to contain arbitrary code. In other words, a programmer should be able to use any of the language constructs supported by the toolkit's underlying implementation language.

(2) Avoid annotations if possible: Forcing a programmer to annotate code in ways that provide information to the constraint solver did 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 confusion and errors often abound.

(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. Programmers feel most comfortable with constraints when the creation of a formula basically involves only a keyword like `Formula` and the code that defines the constraint.

(4) Path expressions are a two-edged sword: Path expressions were a powerful feature that helped facilitate structural inheritance but users often wrote them incorrectly.

6.1 Use the Underlying Language

Prior to Garnet, one-way 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 in Section 4.4, 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[3]:

```
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
```

— Perform semantic computations on trees, graphs, and lists. For example, users wrote constraints that computed the critical path in a directed graph and that kept track of the largest value of a leaf in a 2-3 tree. A simple example of a constraint that computes the longest path from a node to itself might be the following:

```
path_cost =begin
             cost = 0
             for each edge ∈ self.predecessors do
                if (edge.from_vertex.path_cost + edge.cost) > cost then
                   cost = edge.from_vertex.path_cost + edge.cost
             return cost
```

This constraint is helpful in computing the critical path through a graph.

[3]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. Any variable not prefixed by "self" is a local variable.

— 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 the 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
```

— Type check the value input by a user and revert to a saved value if the type is incorrect. For example:

```
value = begin
          if TypeOf(self.value) = INTEGER then
            if self.value >= 0 and self.value <= 100 then
              self.saved_value = self.value
              return self.value
            else
              pop up alert dialog box indicating that the value must be between 0 and 100
              return self.saved_value
          else
            pop up alert dialog box indicating that the value must be an integer
            return self.saved_value
        end
```

— 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
                    case CIRCLE: self.circle_feedback
                    line 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.

— Define monitors that invoke various application functions when a value exceeds a threshold amount. For example:

```
monitor = begin
            if self.obj_being_monitored.temperature > 212 then
              self.obj_being_monitored.excessive_temperature_handler(self.obj_being_monitored)
              return true
            else
                return false
        end
```

— 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 = $\emptyset$
            for each filename $\in$ 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
```

6.1.1 *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 write arbitrary code in constraints was one of the factors that allowed constraints to be integrated with the object system and allowed the rest of Garnet and Amulet to be built on top of the constraint system. Since the introduction of Garnet, a number of other systems, including Rendezvous [Hill 1993] and EvalVite [Hudson 1993] have allowed the limited use of the underlying toolkit language in writing constraints. However, both systems have not supported loops, which as the above examples show, 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 parameters. 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, as shown in Section 9.1, constraint evaluation represents such a small portion of an application's overall execution time that performance optimization is not a crucial issue.

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.

Pointers do have two potential drawbacks. First, dangling pointers that reference de-allocated memory are a common problem in programming languages. However, they are easily detected in Garnet and Amulet because both systems use garbage collection. 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.

A second problem with pointer variables is that they complicate the implementation of constraint satisfaction algorithms. This problem is discussed in Section 8.1.2. As with loops, however, we feel that it is better to design for the user rather than the developers. Our experience was that users obtained significant benefits from using unrestricted pointer variables. These benefits easily outweighed any problems we, as developers, encountered in developing constraint satisfaction algorithms to accommodate unrestricted pointer variables.

Side Effects. As noted in Section 4.4, Garnet does not have a mechanism for handling side effects in constraints whereas Amulet does[4]. 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. There are certain situations where a constraint performs a calculation that is useful to several slots. For example, the most natural way for a constraint to compute an object's bounding box is to compute the left, top, width and height of the bounding box simultaneously. Similarly, a constraint that lays out the elements of a list needs the ability to set multiple slots. The ability to set multiple slots can be elegantly handled using multiple-output constraints 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 Section 6.2, we have found that programmers tend to resist annotations and often get them wrong. Consequently, a decision was made not to require Amulet programmers to declare which slots they were setting in a constraint. An Amulet constraint is still expected to return a value that may be used to set a single slot. However, the constraint body itself may set an arbitrary number of slots.

(2) The ability to create and delete objects. In Garnet, a constraint could be used to compute the set of labels that should appear in a menu or a list, but a constraint was unable to allocate text objects in the menu or list elements in the

[4]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.

list. Programmers were frustrated by this restriction because their code would have to explicitly test whether the set of labels had changed. They would have preferred that the constraint automatically allocate and deallocate the menu and list items as needed, just like a constraint automatically computes other values. As a result, we decided that Amulet would allow a constraint to create and delete arbitrary numbers of objects.

The implementation of Amulet's side-effect mechanism has been described elsewhere [Myers et al. 1997]. 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 [Hill 1993] and an experimental version of Garnet that was extended to include multi-output constraints and side effects [Rosener 1994].

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.

Based on the side-effect producing constraints that we have seen, we can draw some inferences as to why users have not run into problems with infinite loops and non-determinism. The side effects are simple side effects that do not introduce any complication into the constraint system, such as feedback cycles that might cause other side-effect producing constraints to be triggered. Typically the side effects are committed on objects which are encapsulated within another object (e.g., within a menu or a list). The side effects create or delete a set of objects whose positions are then computed using non-side-effect producing constraints. These objects may cause the container object to grow or shrink, and these changes will be propagated to other objects. However, these size and positional changes do not trigger other side-effect producing constraints. In other words, the side effects are confined locally.

6.2 Avoid Annotations If Possible

Annotations are declarative statements made by a programmer that allow a system to obtain information about a program. This information is often difficult to obtain otherwise 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 one-way constraint systems, annotations are frequently used to specify the parameters 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 parameters 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 so annotations were eventually completely abolished in Amulet.

6.2.1 *Annotating Parameters..* In order for a one-way 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 parameters, or equivalently, the right-hand side variables).

Most one-way constraint systems require that a user either declare the parameters that will be used by a constraint or else annotate the parameters in some fashion. For example, in Eval/vite the programmer annotates a parameter by prefixing the variable name with an at-sign (@) [Hudson 1993].

Early in Garnet's design we decided that declaring the parameters 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 parameters, 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 parameters for this constraint, we would have to write code that would look something as follows:

```
parameters = {self.horizontal_spacing, self.items}
for each child ∈ self.items do
    parameters = parameters ∪ {child.visible}
    if child.visible = true then
      parameters = parameters ∪ {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 parameters.

The *annotation* idea did seem workable and so both the original versions of Garnet and Amulet implemented the annotation approach. In particular, a programmer was required to use a special get method in order to declare that a variable was a parameter. For example, the constraint that was shown earlier for calling a monitor when an object exceeds a certain temperature might be written as:

```
monitor = begin
            monitored_obj = self.Get(obj_being_monitored)
            if monitored_obj.GetParameter(temperature) > 212 then
              handler = monitored_obj.Get(excessive_temperature_handler)
              handler(monitored_obj)
              return true
            else
```

```
        return false
    end
```

The programmer has declared that `temperature` is the only parameter that should cause this constraint to be re-evaluated. The `obj_being_monitored` and `excessive_temperature_handler` slots are assumed to be constants.

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 parameters were not getting annotated as parameters. 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 parameter detection scheme. Both Garnet and Amulet retained a mechanism for allowing a programmer to annotate a slot as *not* being a parameter in a constraint. Although annotation is still required when a slot should not be a parameter, it is the lesser of two evils because rarely does a programmer not want a slot to be a parameter. Typically the only reason that a slot should not be a parameter is that it is a constant and some storage in the dataflow graph can be saved by not storing a dependency edge from the parameter to the constraint. However, the correctness of the constraint solver is not compromised by having these edges. Normal Amulet programmers do not worry about these storage efficiency issues. For example, in the above constraint a normal Amulet programmer would allow all three slots to be parameters—the fact that two of them are constant simply means that there are two edges in the dataflow graph that are unnecessary.

6.2.2 *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 parameters 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 parameters. 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 a parameter. 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.

6.3 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.

6.3.1 *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[5]:

```
Garnet                                      Amulet

                                            Formula Compute_Right {
                                                    return (self.left + self.width);
                                            }
                                            . . .
A.right = Formula(self.left + self.width)   A.right = Compute_Right
```

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. Unlike Lisp, 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.

[5]Although Garnet formulas are written in Lisp and Amulet's formulas are written in C++, the formulas in this paper are written using Algol-like syntax to 1) simplify the presentation, and 2) make it easier to compare the programming constructs that were used in the two systems.

6.3.2 *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 an operator that allows the inline creation of anonymous functions (the *lambda* operator) and that treats the function as a nested procedure declaration, as in Pascal. For example, one can write:

```
procedure CreateDialogBox()
    Button Ok;
    Button Cancel;

    Cancel.left = Formula(Ok.left + Ok.width + 10)
    ...
end procedure
```

`Formula` is a macro that expands into the code shown in Figure 4. Note that the formula code is placed within a lambda function. Since the lambda function is defined within the `CreateDialogBox` procedure, `Ok` is accessable within the lambda function.

6.3.3 *Lessons Learned.*. The primary lessons we learned are that 1) syntax does matter, 2) programmers prefer to define formulas at the point where they are used, and 3) programmers expect a formula to be able to reference variables in the function in which the formula is created.

Our experiences with the syntax issue in Lisp and C++ also identified several language features that C++ lacks which would have helped us write a better constraint definition facility for users (Figure 4 helps illustrate these features):

(1) A Powerful Macro Facility: Lisp's macro facility allowed us to write a Formula macro that could be placed directly at the point where the formula is assigned to a variable. This macro expanded into code that: 1) wrapped a formula definition in a function, 2) performed a number of other bookkeeping activities required to initialize the formula, and 3) returned the formula object. Lisp's macro facility also can execute Lisp code at compile time so that a certain amount of preprocessing can be done to the code. In Figure 4, the macro code has changed **`self.left`** and **`self.width`** into the executable statements **`self.get(LEFT)`** and **`self.get(WIDTH)`** where **`LEFT`** and **`WIDTH`** are appropriate constants.

(2) Inline Function Definitions: As noted earlier, Lisp's *lambda* operator allows formula functions to be defined inline and allows the formula functions to access objects declared in the same procedure as the formula functions. It is interesting to note that Java's anonymous inner class mechanism provides similar functionality. For example, in Java one can write:

```
Cancel.left = new Formula_Object () {
```

`Cancel.left = Formula(Ok.left + Ok.width + 10)` expands into:

```
Cancel.left = begin
                  ;; A formula object contains information about a
                  ;; formula including a pointer to the formula's function
                  formula_obj = new Formula_Object(compile (lambda()
                                                   { Ok.get(LEFT) + Ok.get(WIDTH) +10 }))

                  ...bookkeeping code...
                  return formula_obj;
              end
```

Fig. 4. The pseudo-code produced by Garnet's formula macro. The code 1) converts `Ok.left` and `Ok.width` into executable get statements, 2) wraps the resulting formula definition in an anonymous function, 3) compiles the anonymous function, 4) passes the function to a constructor that creates a formula object, 5) performs some bookkeeping, and 6) returns the formula object so that it can be attached to the appropriate variable.

```
    public int Formula() {
        // Ok must be declared final in the enclosing method
        return Ok.get(LEFT) + Ok.get(WIDTH);
    }
};
```

Lisp's lambda operator and Java's anonymous inner classes provide two advantages when designing a constraint language: 1) they allow programmers to define formulas at the point where they are assigned to a slot, and 2) they allow programmers to tie together the slots of objects without having to define those objects globally.

(3) Run-time Function Definitions: Lisp's lambda operator also allows a function to be defined dynamically at run-time. This functionality is nice because a programmer can create new formulas on the fly without having to shutdown an application and recompile it. Being able to create dynamic functions greatly enhances the rapid prototyping capabilities of a language. Of course, dynamic function creation requires an interpreter, which in turn requires the overhead of a run-time environment. If speed is an issue, then an interpreter can be used during the development phase of the program and a compiler can be used for the final production version.

(4) Giving Compiler Instructions in a Macro: In Garnet, the Formula macro was able to tell the compiler to create a formula function at compile time (i.e., to execute the lambda operator at compile time) and to compile the function. Thus the function for a formula could be created and compiled at compile-time, which reduces the run-time overhead of creating a formula object.

(5) Blocks that Return a Value: In Lisp a block of statements (e.g., a **begin/end** block) can be treated as an expression that returns a value. In Figure 4 the macro expands into a block that creates the formula object, does some bookkeeping with the formula object, and then returns the formula object. In Garnet we frequently encountered situations where this ability to execute several statements before com-

puting a result that was assigned to a variable was very helpful. Of course, one can write a function to accomplish the same purpose, but often that location is the only place that uses the code so writing a function is burdensome. Further, the function will have to be defined in another part of the program, forcing a person trying to understand the code to shift back and forth in the program.

Language designers would be well-advised to consider implementing these features in future languages. Most of these features pertain to preprocessing so they would not affect the run-time efficiency of the program. The current situation in C++ requires a toolkit developer to write a preprocessor to keep the syntax simple. The preprocessor essentially requires a C++ parser which can be both difficult to obtain and modify.

6.4 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 2, 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 what allow a constraint to be inherited and still reference the appropriate parts in the instance. 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.

6.4.1 *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.

The result of writing an incorrect path expression is that a constraint references the wrong objects and hence gets incorrect values. One solution would be to allow a programmer to directly name the desired object. Both the Garnet and Amulet constraint systems support such direct references (e.g., label0123.left). However, this solution is not practical because all part names would have to be unique and be global variables. In addition, constraints could not be inherited because they would refer to the prototype's part rather than the instance's part.

A second solution would be to provide a more limited name scope within an object. Unfortunately, objects can be composed from previously defined objects so either the programmer would have to ensure that part names were not duplicated in objects that become siblings or else there would have to be restrictions on part references (e.g., only parent and child references might be allowed). Unfortunately neither of these design alternatives were appealing so we were never able to develop an acceptable design [Myers et al. 1998].

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.

6.4.2 *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 2, 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.

6.4.3 *Lessons Learned.*. In summary, paths proved to be a boon from an implementation perspective. They are a very powerful construct that enable the inheritance of constraints and that allow a constraint to maneuver about a composite object hierarchy. From a user perspective, paths that extend beyond the immediate parent/child neighborhood of a constraint are a bit of a curse because they are problematic to write and to maintain, and as a result, tend to be buggy. Despite several different tries, we were never able to completely overcome this problem.

7. 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 contained 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 call-back procedures, or as error-checking values. In essence application maintenance constituted an "other" category.

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.

7.1 Terminology

In this section it is important to be especially precise about terminology. In particular, it is necessary to differentiate among three different concepts:

(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, 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

Table I. 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.

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	

a slot (i.e., each place in the source code where the formula is used) it creates a *constraint equation*.

(3) The number of constraint instances dynamically created from constraint equations at run-time. Each constraint object created by Amulet, whether it is attached to a prototype or to an instance object, is counted as a constraint instance. Hence, for counting purposes, the assignment of a formula to a slot both creates a constraint equation in the source code and creates an instance of that constraint equation at run-time.

The statistics for the definition and use of formulas are discussed in this section because they pertain to how programmers used constraints in the source code. The statistics for the actual number of constraint instances dynamically created by the various applications are deferred until Section 9.2, because the number of dynamically created instances is storage performance issue rather than a programmer usage issue.

7.2 Types of Formulas Defined.

Table I 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. First, they represent the most obvious use of constraints in a graphical interface. Second, 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 (a color is specified with a line style and a fill style). Third, 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. Consequently novice to intermediate programmers of constraints tend to write far more graphical layout constraints than any other type of constraint.

The biggest discrepancy between the application programs and the Amulet run-time environment occurs in the percentage of application maintenance formulas that are defined, with the Amulet run-time environment defining, percentagewise, twice as many of these types of formulas as application developers. In general, Amulet developers are more skilled in the use of constraints than application programmers and hence it is not surprising that they have defined a greater percentage to perform tasks other than computing various graphical aspects of an object. For example, while the Amulet developers defined a large number of formulas that laid out the pre-defined widgets, such as menus and radio buttons, they also defined a large number of formulas to compute many of the properties of the interactor objects. This result also agrees with our own empirical observations of students. Typically as they progress from novice constraint programmers to more experienced constraint programmers, they begin to define a greater percentage of non-graphical constraints.

We also examined the types of parameters that formulas used. The most interesting thing we found was that formulas typically depended 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 parameters in 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. Over the years, several 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 with programmers, 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.

7.3 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 II 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 in the source code (these pre-defined formulas are counted in the Amulet run-time system in Table I).

Table II. The number of times formulas were actually used in the source code of (a) application programs and (b) the Amulet run-time system. Each percentage represents the proportion of formulas that belonged to that category. For example, 69% of the formulas in the source code of application programs were graphical layout formulas.

Category	*Formulas Used in Application Programs*					
	Programmer-Defined		*Pre-Defined*		*Total Defined*	
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*					
	Programmer-Defined		*Pre-Defined*		*Total Defined*	
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%

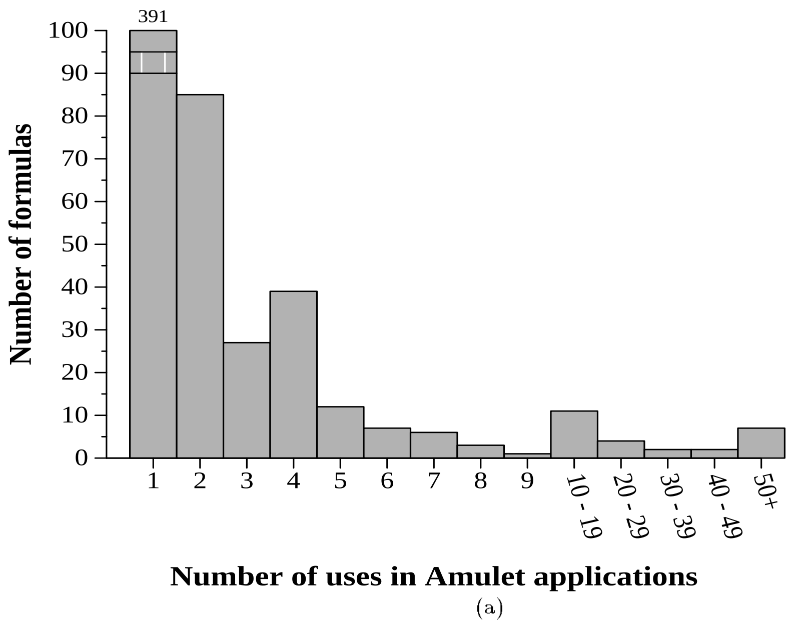
(b)

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 formula functions defined by application programmers are graphical layout formulas and these formulas account for 62% of the constraint equations that are created by programmer-defined formulas in the source code[6]. The results show programmer-defined formulas are used just as frequently as pre-defined formulas, and predominate 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 5 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

[6]The 62% figure is derived by dividing the 31% in the programmer-defined column of Table II by the 50% total line in this column.



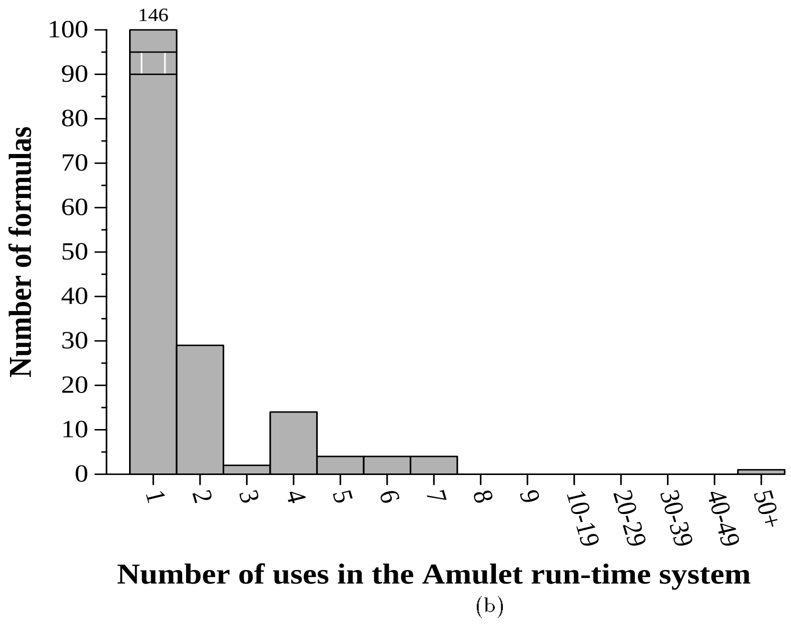


Fig. 5. The frequency with which individual formulas were used to create constraint equations in the source code for both application programs (a) and the Amulet runtime system (b). For example, graph (a) shows that 391 of the formulas defined in the application programs were used only once in the source code, 85 of the formulas were used twice, and so on.

there was a moderate of 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 other commonly re-used formulas revealed only one type of formula that was re-used consistently across multiple applications. This type was a formula that used a pointer to copy 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. A pre-defined formula that allowed a programmer to specify both a pointer slot and a slot to be copied from the object pointed to by the pointer slot would therefore have been helpful. 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.

7.4 Lessons Learned

When we first started the Garnet project we thought that programmers would readily use constraints for many programming tasks because they have been used so successfully in spreadsheets. However, as both the usage statistics and our own experiences with programmers bear out, programmers will readily use constraints for graphical layout but must be carefully and time consumingly trained to use them for other purposes. 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, using constraints for graphical layout readily matches many programmer's mental model of the world, both because they use constraint-like commands, such as left align or center, to align objects in drawing editors, and because constraint-like language is typically used to specify layout in precision paper sketches, such as blueprints. In contrast, most programmers think imperatively, not declaratively, for other programming tasks. Hence it should not be surprising that 1) programmers do not think of using constraints for these other tasks and, 2) programmers require extensive training to overcome their procedural instincts so that they will use constraints.

8. ALGORITHMIC EXPERIENCE

This section describes lessons we have learned in designing and implementing Garnet and Amulet's constraint solvers. The lessons included:

(1) Mark-sweep algorithms work better in graphical interfaces than topological-ordering algorithms

(2) It is not important to avoid unnecessary evaluations in graphical interfaces, and

(3) Lazy evaluation performs better than eager evaluation, but generally not by much.

8.1 Mark-Sweep Algorithms Work Best

During the course of the Garnet and Amulet projects, we experimented with various types of mark-sweep and topological-ordering algorithms for performing constraint satisfaction. We found that mark-sweep algorithms were the most versatile, the easiest to implement, and the most efficient. These findings were surprising to us because the programming languages community has predominantly investigated topological-ordering algorithms [Reps et al. 1983; Alpern et al. 1990] because of their supposedly greater speed.

8.1.1 *Versatility..* The mark-sweep strategy gives the implementor a constraint system the option of using either lazy or eager evaluation whereas the topological-ordering strategy supports only eager evaluation. The reason for the mark-sweep algorithm's greater versatility lies in the different directions in which the two strategies traverse the dataflow graph. The mark-sweep algorithm starts at internal nodes in the directed graph and works its way down to the leaf nodes (also called source nodes or nodes with no incoming edges). In contrast, the topological-ordering algorithms start at the leaf nodes and work their way up to the "root" nodes (also called sink nodes or nodes with no outgoing edges).

Lazy evaluation requires that an algorithm be able to start at an arbitrary node in the dataflow graph. The mark-sweep algorithms have this ability since they can start at an arbitrary internal node. The topological-ordering algorithms do not have this ability since they must start at the leaves, and they cannot determine in advance which set of leaves will need to be evaluated in order to determine the value of an arbitrary node.

8.1.2 *Implementation..* Mark-sweep algorithms proved to be simple to implement even when features like cycles and pointer variables were added to the constraint system. In contrast, the topological-ordering algorithms proved to be very brittle when cycles and pointer variables were added to the constraint system and hence their implementation proved to be very complex.

Basic Implementation. The simplest case for both a mark sweep algorithm and a topological ordering algorithm is the case where the dataflow graph has no cycles and does not change as constraints are evaluated. Even in this simplest case, a mark-sweep algorithm is much easier to implement than a topological-ordering algorithm. The mark phase of a mark-sweep algorithm is simply a depth-first search. In contrast, the numbering phase of a topological-ordering algorithm typically requires either the use of sophisticated algorithms in order to keep the topological numbers up-to-date [Vander Zanden et al. 1994; Alpern et al. 1990] or simpler algorithms that keep the topological numbers only partially up-to-date [Hoover 1987]. In the latter case, constraints may be evaluated more than once because the topological numbers are not completely up-to-date.

The evaluation phase of a mark-sweep algorithm is also simple relative to topological-ordering schemes. The mark-sweep algorithm simply checks the out-of-date flag for a constraint, sets it to false if the flag is true, and executes the constraint's code. In contrast, the topological-ordering scheme must implement a priority queue to handle constraints' evaluation.

Cycles. The mark-sweep algorithm handles cycles trivially. As long as a constraint is marked up-to-date before its evaluation starts, any cycle will halt when it reaches this constraint again. The second time the constraint's value is requested, it will simply return its original value because it has been marked up-to-date. Hence every constraint is evaluated at most once.

In contrast, a topological-ordering algorithm requires an elaborate algorithm to handle cycles. Basically it must treat all the constraints in the cycle as one big node in the dataflow graph, each of which has the same order number. In order to do this, we found that the constraint solver must use a strong connectivity algorithm in order to locate cycles [Vander Zanden et al. 1994] (a cycle is a strongly connected component). Every time the dataflow graph changes, this strong connectivity algorithm must be invoked. In addition, we found that the easiest way to guarantee that each constraint in a cycle is evaluated at most once is to use a mark-sweep algorithm. Hence one ends up implementing the mark-sweep algorithm in addition to the topological-ordering algorithm.

Pointer Variables. Both pointer variables and conditionals may cause the dataflow graph to change dynamically during constraint satisfaction. For example, consider the constraint `A.left = A.obj_over.left` in Figure 6.a. Note that `obj_over` is computed by a constraint that depends on the mouse's postion and that `obj_over` currently points to object B. Figure 6.b shows the resulting dataflow graph. Now suppose the mouse moves, and that this change causes `obj_over` to point to object C. The change will cause the dataflow graph to change, since `A.left` will now depend on `C.left` rather than `B.left`. Figure 6.d shows the new dataflow graph.

Since arbitrary code is allowed and since it is not possible to statically analyze arbitrary code, we cannot assume that the constraint solver can discover statically that the dataflow graph has changed. The change becomes known only when `C3` starts executing and requests `A.obj_over`. At this point the constraint solver discovers that `A.obj_over` now points to C rather than B and that the dataflow graph must be updated. Hence the dataflow graph changes *during* constraint satisfaction.

Other than updating the dataflow graph, the mark-sweep algorithm does not have to change in any way to handle dynamic dependency changes. When `A.left's` constraint requests the value of `A.obj_over`, it will find that it now needs to request the value of `C.left` rather than `B.left`. Since the mark-sweep algorithm can begin its evaluation at an arbitrary node, choosing to evaluate `C.left` next rather than `B.left` does not cause any problem.

C1: A.obj_over = if (mouse.x < window.left) then B else C
C2: C.left = D.left + 100
C3: A.left = A.obj_over.left

(a)

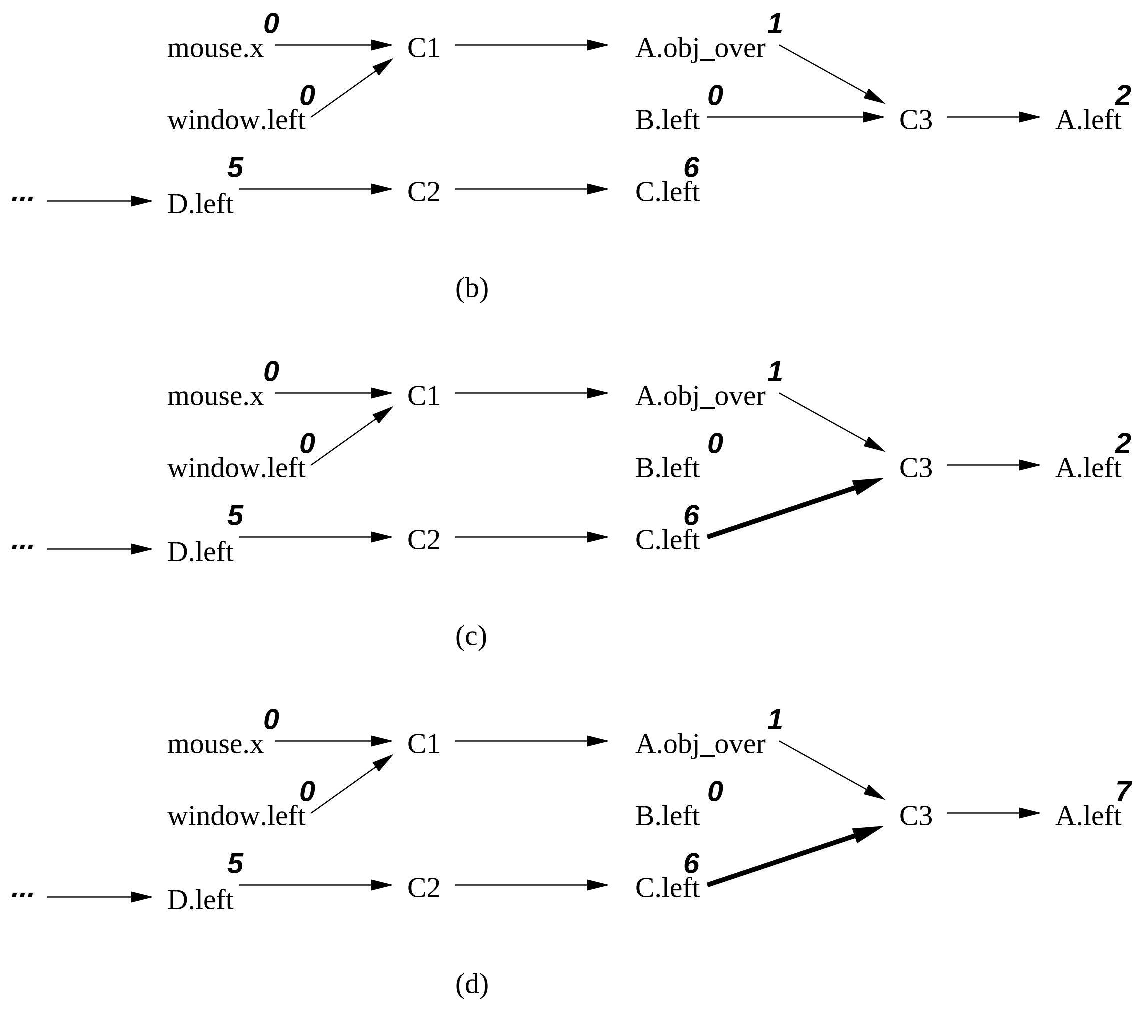


Fig. 6. (a) Three constraints that illustrate how a dataflow graph can change during the course of constraint evaluation. The initial dataflow graph is shown in (b). The numbers to the upper right of each variable in (b) denote the variable's position in topological order. The . . . denotes elided parts of the dataflow graph with edges coming into `D.left`. As the mouse moves, `A.obj_over` might switch from pointing to B to pointing to C. If this happens, the evaluation of `A.left` will dynamically change the dataflow graph from the one in (b) to the one in (c). The bold arrow denotes the added edge. The added edge results in `C.left` and `A.left` being out of order so these two variables must be renumbered (d). In addition, the evaluation of `C3` must be terminated so that `C.left` can be brought up-to-date.

In contrast, a topological-ordering scheme must renumber the dataflow graph before it can proceed. In other words, it must suspend the evaluation phase and enter the numbering phase. For example, in Figure 6.c, **`C.left`** and **`A.left`** are now out of order so **`A.left`** must be renumbered. The renumbering can cause constraints that have already been placed on the priority queue to become out of order, so the priority queue may also have to be re-ordered. Finally, the evaluation of the current constraint may have to be aborted because the current constraint may no longer have the lowest topological number. For example, the evaluation of **`A.left`** in Figure 6.c must be aborted because **`C.left`** now has a lower topological number. Each of these tasks requires that additional and often tricky code be added to the constraint satisfaction algorithm.

In sum, while the introduction of cycles and pointer variables is easily accommodated by the mark-sweep algorithms, it significantly complicates the implementation of the topological-ordering algorithms. It should be noted that despite these difficulties, we did fully implement and test a topological-ordering algorithm that handled cycles and pointer variables [Vander Zanden et al. 1994][7].

8.1.3 *Efficiency..* In Garnet we experimentally implemented a topological-ordering scheme to see if we would get a performance improvement out of the constraint solver. However, we found that the topological-ordering scheme was much slower than the mark-sweep Garnet algorithm.

We were surprised to find that the mark-sweep algorithm was faster because proponents of topological-ordering algorithms have made some persuasive arguments in their favor [Reps et al. 1983; Alpern et al. 1990]. First, the numbering phase of the topological-ordering algorithm should have to visit fewer nodes of the dataflow graph than the mark phase of the mark-sweep algorithm. Second, although both algorithms evaluate only $O|\text{AFFECTED}|$ constraints, the evaluation phase of mark-sweep algorithm examines more constraints because it looks at all out-of-date constraints whereas the evaluation phase of a topological algorithm only looks at constraints whose parameters have actually changed [8].

However, our Garnet implementation revealed that this theoretical analysis ignores important practical considerations:

(1) The depth-first search of the mark phase is so much simpler than the numbering algorithms used by the topological-ordering algorithm's numbering phase that in practice the mark phase is much faster than the numbering phase.

(2) The priority-queue handling code in the topological ordering scheme is several times slower than the simpler evaluation code in the mark-sweep scheme. Hence, even though the evaluation phase of the mark-sweep scheme might have to examine more constraints than the topological-ordering scheme, in practice, the evaluation phase of the mark-sweep scheme is much faster than the evaluation phase of the topological-ordering scheme.

(3) An empirical study of Amulet applications [Vander Zanden and Venckus

[7]The dynamism of pointer variables is sufficiently problematic that no algorithm for multi-way, dataflow constraints has yet been devised that handles the unrestricted use of pointer variables.
[8]A mark-sweep algorithm does not have to evaluate a constraint if none of its parameters have changed but it needs to check to see whether any parameters have changed.

Table III. The benchmark Amulet applications that were used to obtain the empirical results.

Application	*Description*
Checkers	Game of checkers
Tree Debugger	Program for visualizing the execution of an algorithm that inserts nodes into a binary tree. **Short Version**: User quits before binary tree completely constructed **Long Version**: Binary tree is completely constructed.
Testwidgets	Application for testing all of Amulet's widgets
Landscape	Visual editor for creating landscapes of a yard
Circuit	Visual editor for creating electrical circuits
Gilt	Interface builder
Message Sender	Editor for visualizing message sending among a number of processors
Card Catalog	Program for browsing book titles

1996] revealed that 60-80% of variables have fewer than 10 constraints that depend on them, either directly or indirectly. Further, in 6 of the 7 Amulet applications surveyed, almost no variable was depended on by more than 100 constraints. These findings indicate that very few constraints will be marked out-of-date or re-evaluated for most variable changes. As a result, the theoretically better topological ordering scheme does not get a chance to be better because its constants are so much larger than the mark-sweep scheme.

(4) In graphical applications, almost all the constraints that depend on a changed variable compute a new value and hence must be re-evaluated (this issue is discussed further in the next section). Hence the size of the AFFECTED and INFLUENCED sets are nearly identical, meaning that the mark phase wastes very little time examining unnecessary variables.

Once the algorithms are modified to handle cycles and pointer variables, the empirical performance advantage of mark-sweep algorithms over topological-ordering algorithms becomes even more pronounced, because of the greatly increased complexity of the topological-ordering algorithms.

8.2 Avoiding Unnecessary Evaluations

One of the characteristics that distinguishes some mark-sweep algorithms from others is whether or not they avoid unnecessary evaluations. An unnecessary evaluation is one in which a constraint's inputs have not changed, but the constraint is nonetheless re-evaluated [9]. If a constraint evaluation takes a significant amount of time, then avoiding an unnecessary evaluation of that constraint could significantly improve an application's response time.

Note that unnecessary evaluations are only an issue with mark-sweep algorithms. Topological-ordering algorithms only evaluate constraints whose inputs have changed because a constraint is not added to the re-evaluation queue unless one of its inputs has changed. In contrast, a mark-sweep algorithm will mark a constraint out-of-date if one of its inputs has *potentially* changed (i.e., it depends indirectly on a

[9] Some advocates of lazy evaluation describe an unnecessary evaluation as an evaluation whose result is never used but throughout this paper we define an unnecessary evaluation as one in which the constraint's inputs have not changed.

changed variable). This input may or may not actually change, depending on the outcome of its own constraint evaluation. If the input does not change, then the constraint that depends on it does not have to be re-evaluated (assuming of course that none of the constraint's other inputs have changed either). Unless the mark-sweep algorithm performs some bookkeeping to determine whether the input has actually changed, the constraint may be unnecessarily re-evaluated.

To assess the potential impact of unnecessary evaluations on graphical applications, we measured the number of required and unnecessary constraint evaluations in a number of Amulet applications. The applications are summarized in Table III and snapshots of the applications are shown in Figure 7. The results for both lazy and eager evaluation are shown in Figure 8. The released version of Amulet actually performs these unnecessary evaluations. To perform the measurements, we modified Amulet in a manner suggested by Hudson [Hudson 1991] so that it detected and avoided unnecessary constraint evaluations (in essence, when a slot is evaluated, it checks whether it actually changed, and if it did, it notifies all constraints that depend on it—additionally, before a constraint is evaluated, it forces any out-of-date inputs to be brought up-to-date so that it can determine whether these inputs have changed).

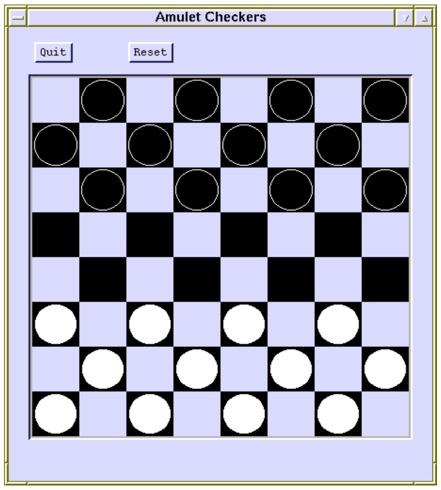
The results show that in general most evaluations are required. The reason is that when the graphical appearance of one object changes, the graphical appearance of related objects will change in a related way. For example, if a gate moves in the circuit application, then all the attached wires also move. Similarly, when the age of the trees is adjusted in the landscape application, all of the trees change graphical appearance. Hence, almost all the constraints that depend on a changed value will actually change value themselves.

We also measured the constraint evaluation time saved by not having to perform unnecessary evaluations. Interestingly, the savings in time is often less than the savings in number of constraints evaluated. Although not large, these savings still seem to be fairly good. However, the savings in time is savings in *constraint evaluation* time, not overall application time. In Section 9, we will see that constraint evaluation time typically represents less than 10% of an application's overall time. Consequently the savings in overall application time is insignificant. Indeed the savings was so insignificant that we checked to see whether the overhead of avoiding unnecessary computations was worth the savings. It was, since the overhead almost always amounted to less than 1% of the total constraint evaluation time.

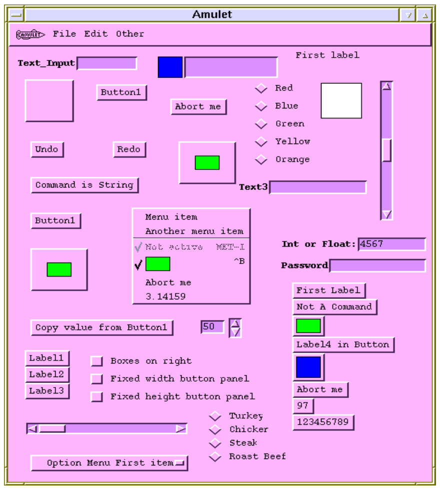
8.2.1 *Lessons Learned.*. Given the small amount of time that most graphical applications spend on constraint satisfaction (see Section 9.1), saving unnecessary evaluations does not result in noticeable speedup in most applications. However, it is relatively easy to check for unnecessary computations and the overhead is less than the time saved by avoiding the computations. Hence, it is a worthwhile optimization to add to the constraint satisfaction algorithm, but *only* because the optimization is so simple.

8.3 Lazy Versus Eager Evaluation

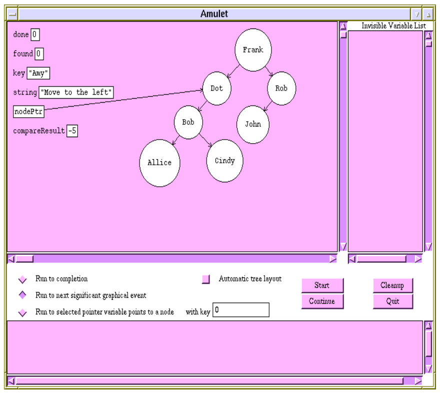
We have previously discussed the tradeoffs of lazy versus eager evaluation from an ease-of-use standpoint (Section 5.4). However, they can also be compared with



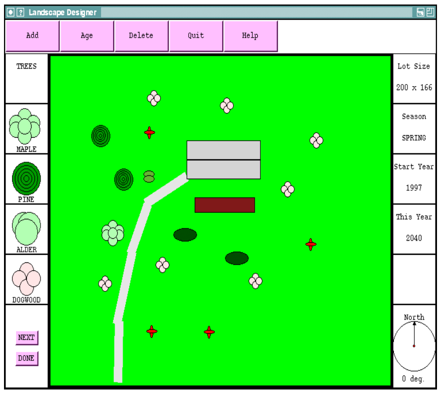
Checkers



Test Widgets

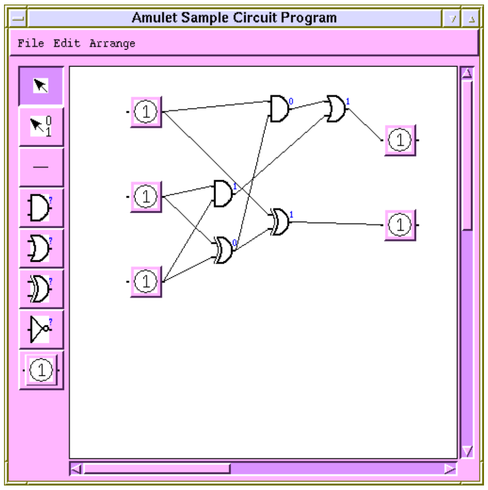


Tree Visualizer

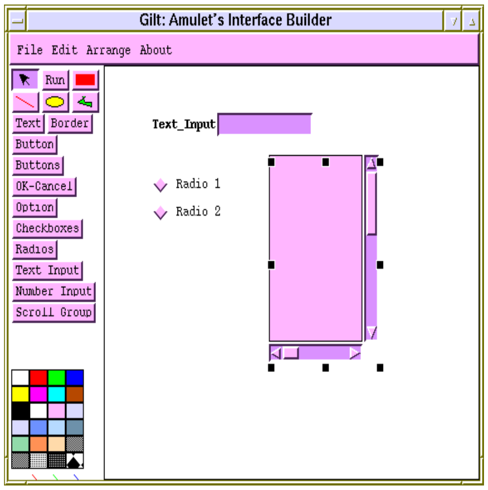


Landscape Editor

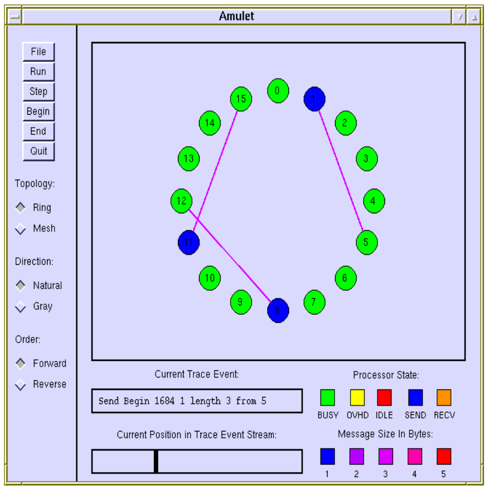
Fig. 7. Snapshots of sample applications created using Amulet. The speed and storage results reported in this paper are derived from these applications.



Logic Circuit Editor

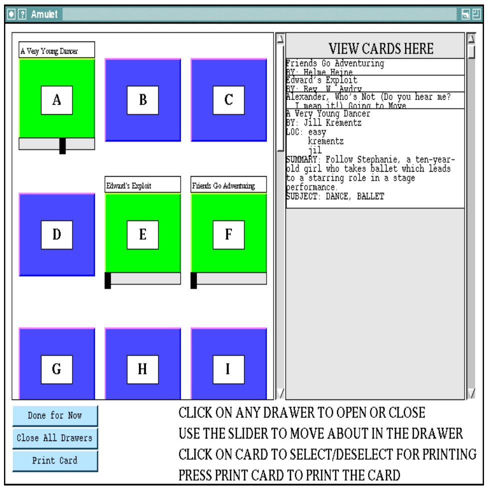


Gilt Interface Builder



Send Begin 1684 1 length 3 from 5

Message Passing Visualizer

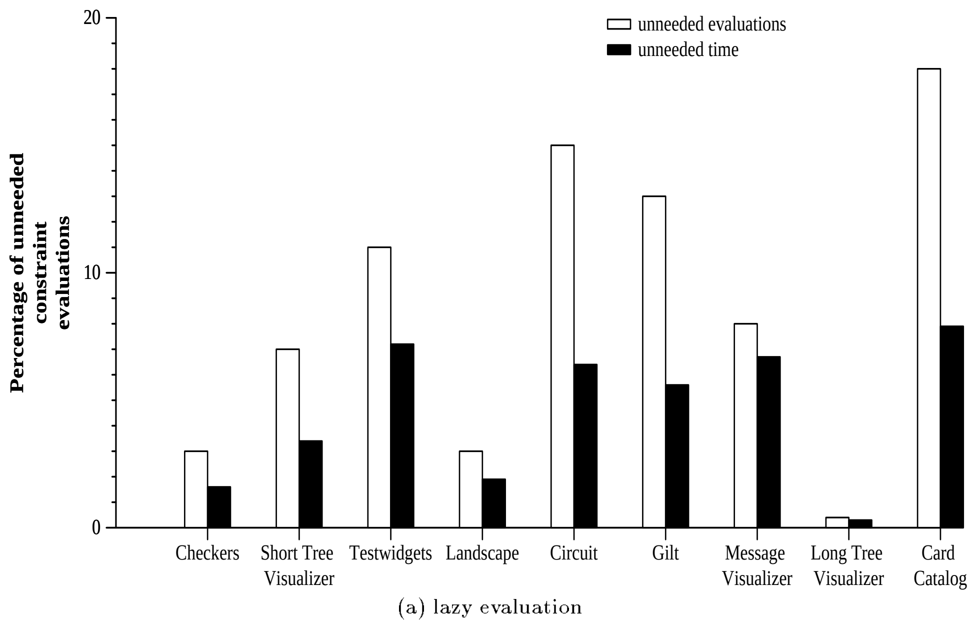


Friends Go Adventuring
BY: Helme Heine
Edward's Exploit
BY: Rev. W. Awdry
Alexander, Who's Not (Do you hear me? I mean it!) Going to Move
A Very Young Dancer
BY: Jill Krementz
LOC: easy
krementz
jil
SUMMARY: Follow Stephanie, a ten-year-old girl who takes ballet which leads to a starring role in a stage performance.
SUBJECT: DANCE, BALLET

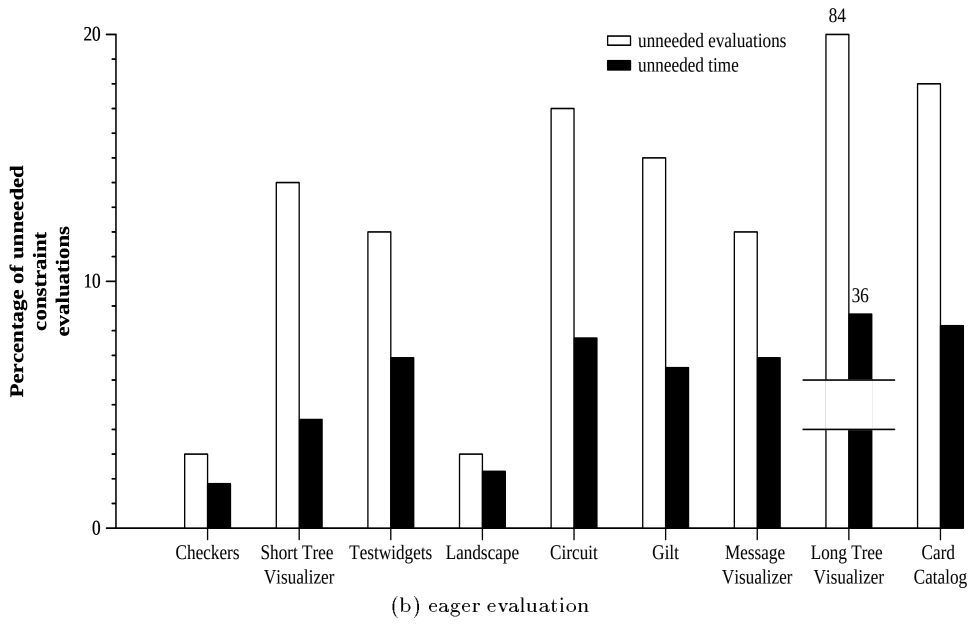
CLICK ON ANY DRAWER TO OPEN OR CLOSE
USE THE SLIDER TO MOVE ABOUT IN THE DRAWER
CLICK ON CARD TO SELECT/DESELECT FOR PRINTING
PRESS PRINT CARD TO PRINT THE CARD

Card Catalog

Figure 7 continued.



(a) lazy evaluation



(b) eager evaluation

Fig. 8. The percentage of unneeded evaluations for the benchmark applications and the percentage of constraint evaluation time saved if the unneeded evaluations are not performed.

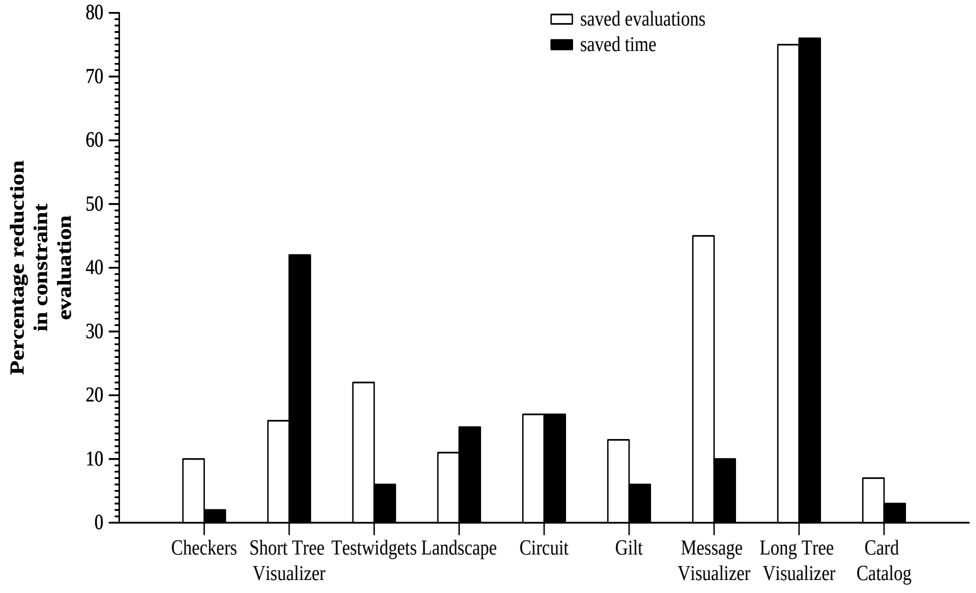


Fig. 9. The percentage of constraint evaluations and the percentage of time saved by lazy evaluation versus eager evaluation for the benchmark applications.

respect to efficiency. Proponents of lazy evaluation claim that lazy evaluation can potentially avoid a significant number of unnecessary evaluations and thus increase an application's response time. To determine what types of time savings are possible, we compared our benchmark set of applications using both eager and lazy evaluation. The released version of Amulet uses eager evaluation but a one line change in the Amulet code changes Amulet into a lazy evaluator[10]. Figure 9 shows the percentage of constraint evaluations and the percentage of time that lazy evaluation saved over eager evaluation for the various benchmark applications.

The results generally show that the expected savings for graphical applications is less than 20%, both in terms of number of constraints evaluated and constraint satisfaction time. The reason why lazy evaluation does not secure greater gains is because the display manager causes most constraints to be evaluated when it tries to determine whether or not an object should be drawn on the display. Objects whose positions place them outside the current viewing area do not have to be drawn but the only way a display manager can ascertain this fact is to demand the values of their position and size slots. Hence, regardless of whether lazy or eager evaluation is used, the constraints on these slots must be re-evaluated. The only way to avoid the evaluation of the position and size constraints is if the object's visible attribute is false, indicating that the object should not be drawn. However, almost all the graphical objects in the benchmark applications have their visibility

[10]When Amulet is changed into a lazy evaluator, it is possible to specify that a slot should be evaluated eagerly. In several of the applications, a couple of the slots had to be marked eager to make the applications work properly.

slots set to true.

The relatively small percentage reduction in evaluated constraints achieved by lazy evaluation might be more impressive if the constraint evaluations that are being avoided are expensive evaluations. However, the percentage time savings achieved by lazy evaluation is typically less than the percentage reduction in constraints evaluated. Especially striking is the message passing application where a 46% savings in constraints evaluated results in only an 11% reduction in constraint evaluation time. This suggests that rather than lazy evaluation avoiding the unnecessary evaluation of expensive constraints, it actually tends to avoid the unnecessary evaluation of inexpensive constraints.

The one exception to this rule was the tree visualization application. In this application, many nodes of the tree can be created before they are actually displayed. In this case, lazy evaluation leads to enormous savings in constraint evaluation time because the eager evaluator re-evaluates the layout constraints every time a new node is created, even though the tree is not yet visible. In contrast, the lazy evaluator does not re-evaluate these constraints because the display manager has not yet been asked to display the tree. Despite the significant savings in constraint evaluation time, Figure 10 shows that less than 20% of the tree visualizer's time is spent performing constraint evaluation. Hence, even in this case, lazy evaluation does not achieve a significant reduction in overall application execution time.

8.3.1 *Lessons Learned..* Given that constraint satisfaction already accounts for a rather small percentage of an application's time, lazy evaluation typically provides almost no speedup in most applications because 1) it does not actually avoid very many unnecessary evaluations, and 2) those that it does avoid tend to be inexpensive evaluations. Thus, we feel that the choice of lazy versus eager evaluation typically should be made based on usability issues, which as discussed in Section 5.4.1, are mixed.

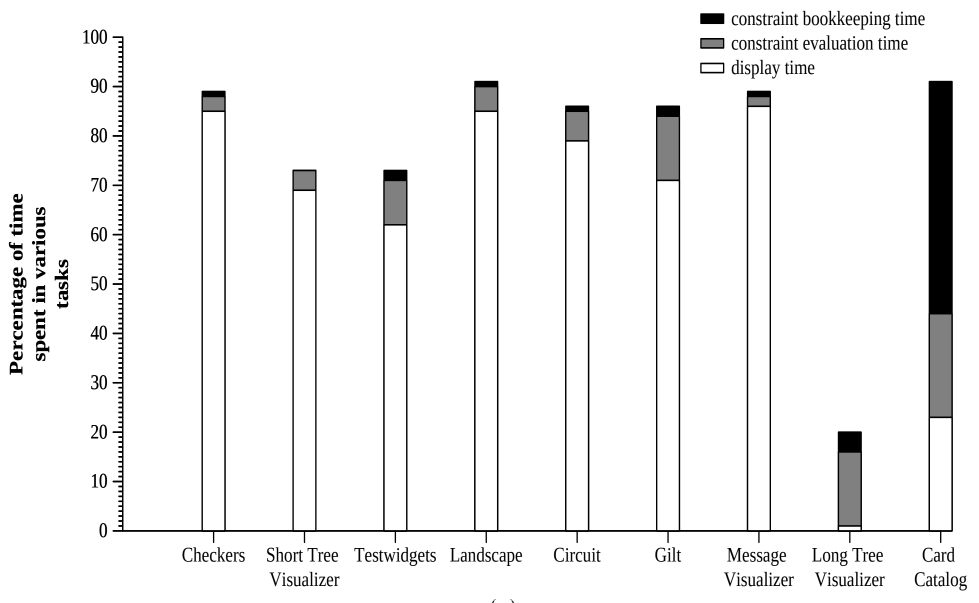
9. PERFORMANCE EXPERIENCE

In this section we examine the time and storage efficiency of the Amulet and Garnet constraint systems.

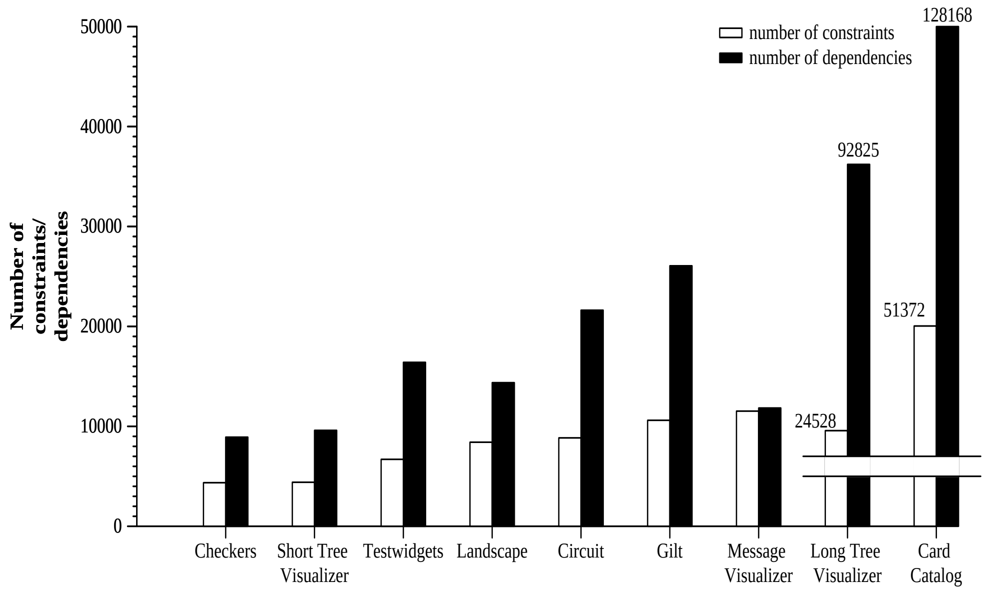
9.1 Time Efficiency

Both the Garnet and Amulet constraint systems were able to solve constraints quickly enough to support interactive behavior. For example, feedback objects can track the mouse in real time and applications can perform smooth, real-time animations, even in large, constraint-based applications.

Profiles of both Garnet and Amulet applications verify that the constraint solver is efficient. For example, Figure 10.a shows the percentage of several Amulet applications' time spent updating the display, executing formula functions, and performing the overhead required to maintain the formula and dataflow graph data structures. The percentages were obtained by running the applications on a Sparc 20 machine with 64 megabytes of RAM. The applications were compiled under g++ version 2.7.2.1 using the -O2 option and ran under X Windows version 6 (several applications crashed under the -O3 option).



(a)



(b)

Fig. 10. (a) The percentage of time spent updating the display, executing formula functions, and performing constraint bookkeeping in the benchmark applications. The remaining time went to assorted other tasks such as input handling and executing various callback routines. (b) The number of constraints and the number of dependencies created in the benchmark applications. The figures are aligned to make it clear how the percentage of time spent on the various tasks changes as the number of constraints in an application increases.

The percentages indicate that the constraint overhead is a small fraction of the time spent executing formula functions, which are in turn a small fraction of the time spent updating the display. These percentages are consistent with the numbers recorded for Garnet applications [Vander Zanden et al. 1994].

The numbers indicate a few interesting facts. First, display time absolutely dominates any other activity that the application performs. Since all constraints are satisfied before the display manager is called, the percentage shown for the display manager is purely devoted to updating the display, and includes the time spent determining which objects need to be redrawn and making calls to the appropriate window system routines. The only exception to the display time dominance is the card catalog application, and the times shown are skewed by the fact that 52% of the application's time was spent shutting down the application after the exit button had been pressed. If only the time that the user spent interacting with the application is counted, then the display time climbs to around 50% and the bookkeeping overhead for constraints falls under 6%. The high bookkeeping overhead is almost exclusively accounted for by the destruction of the constraint dataflow graph in the clean up procedure.

The second fact that emerges is that the constraint solver adds almost no time to the execution of the application. The formula functions would have to be executed whether or not there was a constraint solver, so the only real time added by the constraint solver is in its bookkeeping overhead (an eager evaluator may also unnecessarily evaluate some formulas, but as shown in the previous section, this additional evaluation is not typically significant). As shown by Figure 10.a, this number is typically under 2% for an application, and is often under 1%. Clearly if one is looking for a place to optimize an application, the constraint solver is not the first place to look.

The third fact that emerges is that the percentage of time spent in constraint satisfaction, both in overhead and executing formula functions, does not significantly increase as the number of constraints in the application increases. This result might seem somewhat anomalous since one might expect that large applications should have large chains of constraints which would consume a considerable amount of constraint satisfaction time. However, an earlier study that we conducted of Amulet applications revealed that constraint networks tend to be modular, that is, divided into a number of small, independent sets of constraints rather than one monolithic set of constraints [Vander Zanden and Venckus 1996]. Since any given interactive transaction, such as moving an object on the screen, typically only changes a small number of variables, and since constraint networks tend to be small and modular, only a few constraints will have to be re-evaluated on any given interactive transaction, no matter how big the application.

9.2 Storage Efficiency

Both the Garnet and Amulet constraint systems consume a significant amount of storage. Table 9.2 summarizes the constraint overhead imposed by both systems. In general, Garnet and Amulet applications were not large enough for the constraint system size to pose a problem. For example, Figure 10.b shows the number of constraint instances and dependencies created by each of the benchmark applications. None of them have enough constraints or dependencies to pose a serious memory

Table IV. Size of formula objects and dependency objects in Garnet and Amulet.

System	*Bytes Per Formula*	*Bytes Per Dependency*
Garnet	44	16
Amulet	48	24

problem. The largest application in terms of constraint storage is the card catalog application, and its constraints plus dependencies only require 5.5 megabytes of memory.

However, the current set of Garnet and Amulet applications is somewhat misleading. Both Garnet and Amulet use extremely heavyweight objects that limit the number of objects that can be held in RAM memory to less than roughly 5000. Beyond this amount the application is forced into virtual memory and performance significantly degrades. Hence, the size of Garnet and Amulet applications is effectively limited to a few thousand objects. Indeed, one survey respondent did report that too much memory usage made constraints a bad choice for their application, although it is unclear whether it was just the constraints, or whether the overall size of Amulet objects also contributed to the problem.

For various types of information visualization applications, it is quite conceivable that the number of objects an application would need to create would be in the hundreds of thousands, or even millions, in which case constraint storage would become problematic.

9.3 Lessons Learned

There is a time versus storage trade-off in performing constraint satisfaction. In the early interactive toolkits, such as Garnet, ThingLab [Borning 1981], Penguims [Hudson 1994], and Rendezvous [Hill 1993], there was considerable concern that the performance of the constraint solver could seriously degrade the performance of an interactive application. Since this degradation was unacceptable, a considerable amount of effort went into devising constraint algorithms that minimized the time spent in constraint satisfaction. These algorithms use costly bookkeeping data structures, such as fine-grained dataflow graphs, and maintain a variety of flags and other information in both constraints and dependencies in an effort to speed up performance. Since early interactive applications were relatively small, this trade-off did not constitute a problem.

As the speed numbers in Figure 10 illustrate, the performance battle has been won. However, the storage battle may be lost as the size of interactive applications continues to increase. Consequently, researchers have begun to look into ways to trade speed for storage. For example, Hudson and Smith have introduced the notion of microconstraints to reduce the physical storage required for constraints and dependencies in certain common graphical layout relationships [Hudson and Smith 1996]. These constraints fit in four bytes of memory, thus saving considerable storage. However, they require that the dataflow graph be dynamically inferred, thus increasing the amount of time spent in constraint satisfaction. We are currently examining ways that a dataflow graph can be stored in a prototype and then generated for an instance on demand [Halterman and Vander Zanden 1998]. This approach also saves storage by using far less memory to store a dataflow graph. Like the microconstraints approach, it requires the dataflow graph to be dynamically inferred, thus increasing the time spent in constraint satisfaction. Initial experience with this algorithm indicates that the trade-off is a good one, as might be expected given the dominance of redisplay time over constraint solving overhead.

10. 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 projects represent two of these efforts. Because of their widespread distribution and use, they have provided valuable insights into both the successes and shortcomings of constraints. Perhaps the biggest successes have been:

(1) the acceptance by programmers of constraints as a useful tool for graphical layout, and to a lesser extent, for defining the graphical properties of objects,
(2) the development of simple, efficient algorithms for performing constraint satisfaction, and
(3) the development of algorithms that allow arbitrary code to be used within a constraint without requiring any annotations from the programmer.

The areas that researchers need to focus on to gain further acceptance of constraints are:

(1) the development of better debugging tools for constraints,

(2) the development of more storage efficient algorithms, even if it means trading speed for storage,

(3) the refinement of constraint satisfaction algorithms so that constraints are evaluated when users expect them to be evaluated, and

(4) the development of 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. An unsafe side-effect would be one that would cause the constraint solver to enter an infinite loop or produce non-deterministic results. 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.

The experience of the two projects also show that developers of constraint systems would be well advised to keep constraints usable for programmers. Some of the lessons we have learned are that:

(1) Allowing programmers to write constraints 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 the system.

(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.

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. The resolution of some of the problems identified in this paper should help make constraints an even more valuable tool in the future.

APPENDIX

A. CONSTRAINT SATISFACTION ALGORITHMS

In this appendix, we have included algol-like pseudocode showing how the mark-sweep (Figure 11) and topological-ordering algorithms (Figure 12) are implemented. The variables used by the algorithms are shown in Table A. These algorithms have been adapted and simplified from the algorithms we presented in [Vander Zanden et al. 1994]. The intention is to provide the reader with a general understanding of how these two algorithms are implemented, without delving into some of the intricacies of the algorithms. Consequently, the mark-sweep algorithm shown in this appendix handles cycles, pointer variables, and the automatic detection of parameters but it does not attempt to avoid evaluating constraints if none of their parameters has changed and it does not attempt to remove a constraint from a slot's dependency list when the constraint no longer uses the slot as a parameter. The topological-ordering algorithm shown in this appendix does not handle cycles, pointer variables, or the automatic detection of parameters. We also have not shown how the order numbers get updated. All of these elements significantly complicate the code for the topological-ordering algorithm. Readers who wish to see the full implementation details for these algorithms are referred to [Vander Zanden et al. 1994].

The one implementation detail we have chosen to show in the mark-sweep algorithm is the automatic detection of parameters [Vander Zanden et al. 1991; 1994; Hoover 1992]. This detection is done by maintaining a stack of constraints, with

Table V. Definition of the variables used by the mark-sweep algorithm in Figure 11 and the topological-ordering algorithm in Figure 12. These algorithms assume that slots and constraints are objects containing the variables defined below.

Global Variables	**Definition**
constraint_queue	A queue that contains constraints which need to be re-evaluated. In the mark-sweep algorithm this queue is an ordinary queue and in the topological-ordering algorithm it is a priority queue.
dependency_stack	The stack of constraints that is used for inferring parameters.
Slot Variables	
value	The value of the slot.
owner	The object to which this slot belongs.
out_of_date	In the mark-sweep algorithm, this flag indicates whether or not the slot's value is out-of-date. In the topological-ordering algorithm, this flag indicates whether or not the slot's constraint is on the constraint queue.
dependents	A pointer to the set of constraints that depend on the slot.
constraint	A pointer to the constraint that computes the value of the slot.
Constraint Variables	
output_slot	A pointer to the slot computed by this constraint.
formula	A pointer to the function that computes this constraint's value.

the currently executing constraint being the top-most stack element. When a constraint is about to be executed, the constraint solver pushes the constraint onto the stack and when the constraint finishes executing, the constraint solver pops the constraint off the stack. The stack ensures that when a constraint requests a slot's value, the slot can locate the requesting constraint by consulting the top element of the stack.

When a slot's value is accessed by Garnet or Amulet's `get` method, the get method checks whether there is a constraint currently on the constraint stack[11]. If there is no constraint on the stack, then the value was requested by an application function and no dependency needs to be created. However, if there is a constraint on the stack, then the `get` method knows that it was the topmost constraint that requested the slot's value. It therefore adds the topmost constraint to the slot's list of dependents, if it is not already there. Thereafter, when the slot's value is changed, it can notify the set of constraints on its dependents list.

ACKNOWLEDGEMENTS

We would like to thank John Reppy for his helpful comments on an earlier draft of this paper.

[11]The get method is the only way that a slot's value can be accessed, regardless of whether the application or a constraint is requesting the value. For example, a sample constraint might look as follows: my_box.right = Formula(self.get(LEFT) + self.get(WIDTH)). `self` refers to the object containing the constraint, in this case, `my_box`.

```
;; Programmers call set to change the slot to the designated value.
procedure set(slot, value)
    slot.value = value
    mark(slot)
    slot.out_of_date = false ;; indicate that the slot is up-to-date

;; mark recursively tags as out-of-date all slots that depend on the parameter
;; slot and saves all of these slots' constraints on an ordinary queue.
procedure mark(slot)
    slot.out_of_date = true;
    for each constraint ∈ slot.dependents do
        if constraint.output_slot.out_of_date = false then
            constraint_queue.Insert(constraint)
            mark(constraint.output_slot)

;; Programmers call solve to bring the values of all slots up-to-date. solve
;; evaluates each constraint on the constraint queue and assigns the result to the constraint's output slot.
procedure solve()
    while not constraint_queue.Empty() do
        constraint = constraint_queue.Dequeue()
        slot = constraint.output_slot
        slot.out_of_date = false
        dependency_stack.Push(constraint)
        slot.value = constraint.formula(slot.owner)
        dependency_stack.Pop(constraint)

;; Programmers call get to retrieve the slot's value. If the slot's value is out-of-date, get first
;; evaluates the slot's constraint and assigns the result to the slot's value field.
procedure get(slot)
    if not dependency_stack.Empty() then
        slot.dependents = slot.dependents ∪ dependency_stack.Top()
    if slot.out_of_date = true then
        slot.out_of_date = false
        dependency_stack.Push(slot.constraint)
        slot.value = slot.constraint.formula(slot.owner)
        dependency_stack.Pop(slot.constraint)
    return slot.value
```

Fig. 11. A mark-sweep algorithm implemented as an eager evaluator.

```
;; Programmers call set to change the slot to the designated value. set adds the slot's
;; constraint to the priority queue unless it is already there.
procedure set(slot, value)
    slot.value = value
    if slot.out_of_date = false then
        slot.out_of_date = true
        priority_queue.Insert(slot.constraint)

;; Programmers call solve to bring the values of all slots up-to-date. solve
;; removes constraints in topological order from the priority queue and evaluates them. If
;; the value they compute is different then the value of their output slot, solve
;; adds any constraints that depend on the output slot to the priority queue.
procedure solve()
    while not priority_queue.Empty() = do
        constraint = priority_queue.DeleteMin()
        slot = constraint.output_slot
        slot.out_of_date = false
        new_value = constraint.formula(slot.owner)
        if slot.value ≠ new_value then
            slot.value = new_value
            for each dependent_constraint ∈ slot.dependents do
                if dependent_constraint.output_slot.out_of_date = false then
                    dependent_constraint.output_slot.out_of_date = true
                    priority_queue.Insert(dependent_constraint)

;; Programmers call get to retrieve the slot's value.
procedure get()
    return slot.value
```

Fig. 12. A topological-ordering algorithm.

REFERENCES

Alpern, B., Hoover, R., Rosen, B. K., Sweeney, P. F., and Zadeck, F. K. 1990. Incremental evaluation of computational circuits. In *ACM SIGACT-SIAM'89 Conference on Discrete Algorithms*. 32–42.

Barth, P. 1986. An object-oriented approach to graphical interfaces. *ACM Transactions on Graphics 5,* 2 (Apr.), 142–172.

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

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

Borning, A., Anderson, R., and Freeman-Benson, B. 1996. Indigo: A local propagation algorithmfor inequality constraints. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*. Proceedings UIST'96, Seattle, WA, 129–136.

Borning, A., Maher, M., Martindale, A., and Wilson, M. 1989. Constraint hierarchies and logic programming. In *Proceedings of the 6th International Logic Programming Conference*. Lisbon, Portugal, 149–164.

Borning, A., Marriott, K., Stuckey, P., and Xiao, Y. 1997. Solving linear arithmetic constraints for user interface applications. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*. Proceedings UIST'97, Banff, Alberta, Canada, 87–96.

BROWN, P. S. AND GOULD, J. D. 1987. An experimental study of people creating spreadsheets. *ACM Transactions on Office Information Systems 5,* 3, 258–272.

COHEN, J. 1990. Constraint logic programming languages. *Communications of the ACM 33,* 7 (Jul), 52–68.

DEMERS, A., REPS, T., AND TEITELBAUM, T. 1981. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the Principles of Programming Languages Conference.* Williamsburg, VA, 105–116.

DINCBAS, M., HENTENRYCK, P. V., SIMONIS, H., AGGOUN, A., GRAF, T., AND BERTHEIR, F. 1988. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988.* 249–264.

FREEMAN-BENSON, B. N. 1990. Kaleidoscope: Mixing objects, constraints, and imperative programming. In *OOPSLA/ECOOP'90 Conference Proceedings.* 77–88.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, MA. ISBN 0-201-63361-2.

GLEICHER, M. 1993. A graphics toolkit based on differential constraints. In *ACM SIGGRAPH Symposium on User Interface Software and Technology.* Proceedings UIST'93, Atlanta, GA, 109–120.

GLEICHER, M. AND WITKIN, A. 1992. Through-the-lens camera control. In *Computer Graphics.* Proceedings SIGGRAPH'92, Chicago, IL, 331–340.

GOLUB, G. H. AND VAN LOAN, C. F. 1989. *Matrix Computations, 2nd Ed.* The Johns Hopkins University Press, Baltimore, MD.

HALTERMAN, R. AND VANDER ZANDEN, B. 1998. Using model dataflow graphs to reduce the storage requirements of constraints. Tech. Rep. UT-CS-98-413, University of Tennessee. Dec.

HELM, R., HUYNH, T., LASSEZ, C., AND MARRIOTT, K. 1992. A linear constraint technology for interactive graphic systems. In *Proceedings Graphics Interface.* GI'92, Vancouver, Canada.

HENRY, T. R. AND HUDSON, S. E. 1988. Using active data in a UIMS. In *ACM SIGGRAPH Symposium on User Interface Software and Technology.* Proceedings UIST'88, Banff, Alberta, Canada, 167–178.

HILL, R. D. 1993. The Rendezvous constraint maintenance system. In *ACM SIGGRAPH Symposium on User Interface Software and Technology.* Proceedings UIST'93, Atlanta, GA, 225–234.

HILL, R. D., BRINCK, T., ROHALL, S. L., PATTERSON, J. F., AND WILNER, W. 1994. The Rendezvous architecture and language for constructing multiuser applications. *ACM Transactions on Computer Human Interaction 1*, 81–125.

HOOVER, R. 1987. Incremental graph evaluation. Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, NY.

HOOVER, R. 1992. Alphonse: Incremental computation as a programming abstraction. *Sigplan Notices 27,* 7 (July), 261–272. ACM SIGPLAN'92 Conference on Programming Language Design and Implementation.

HOSOBE, H., MATSUOKA, S., AND YONEZAWA, A. 1996. Generalized local propagation: A framework for solving constraint hierarchies. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming.* Lecture Notes in Computer Science, Vol 1118, Springer Verlag, Boston, MA, 237–251.

HOSOBE, H., MIYASHITA, K., TAKAHASHI, S., MATSUOKA, S., AND YONEZAWA, A. 1994. Locally simultaneous constraint satisfaction. In *Principles and Practice of Constraint Programming, Second International Workshop, PPCP'94.* Lecture Notes in Computer Science, Vol 874, Springer Verlag, Alan Borning, editor, Orcas Island, WA, 51–62.

HUDSON, S. AND KING, R. 1988. Semantic feedback in the Higgens UIMS. *IEEE Transactions on Software Engineering 14,* 8 (Aug), 1188–1206.

HUDSON, S. E. 1991. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM TOPLAS 13,* 3 (July), 315–341.

HUDSON, S. E. 1993. A system for efficient and flexible one-way constraint evaluation in C++. Tech. Rep. 93-15, Graphics Visualizaton and Usability Center, College of Computing, Georgia Institute of Technology. April.

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

Hudson, S. E. and Smith, I. 1996. Ultra-lightweight constraints. In *ACM SIGGRAPH Symposium on User Interface Software and Technology.* Proceedings UIST'96, Seattle, WA, 147–155.

Huynh, T., Lassez, C., and Lassez, J.-L. 1992. Practical issues on the projection of polyhedral sets. *Annals of Mathematics and Artificial Intelligence 6*, 295–316.

Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. 1992. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS 14,* 3 (July), 339–395.

J.E. Dennis, J. and Schnabel, R. 1983. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice Hall, New York, NY.

Knuth, D. 1968. Semantics of context-free languages. *Mathematical Systems Theory 2*, 127–145.

Lassez, C. and Lassez, J.-L. 1991. Quantifier elimination for conjunctions of linear constraints via a convex hull algorithm. Tech. Rep. Research Report RC 16779, IBM.

Lassez, J.-L. and McAloon, K. 1992. A canonical form for generalized linear constraints. *Journal of Symbolic Computation 13,* 1 (Jan).

Leler, W. 1988. *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley Publishing Company, New York.

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

Liu, Y. A., Stoller, S. D., and Teitelbaum, T. 1998. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems 20,* 3 (May), 546–585.

Mark A. Linton, J. M. V. and Calder, P. R. 1989. Composing user interfaces with interviews. *IEEE Computer 22,* 2 (Feb), 8–22.

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

Myers, B. A., Giuse, D. A., Dannenberg, R. B., Vander Zanden, B., Kosbie, D. S., Pervin, E., Mickish, A., and Marchal, P. 1990. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer 23,* 11 (Nov.), 71–85.

Myers, B. A., McDaniel, R., Miller, R., Brad Vander Zanden, D. G., Kosbie, D., and Mickish, A. 1998. Our experience with prototype-instance object-oriented programming in Amulet and Garnet. *Interfaces 39*, 4–9.

Myers, B. A., McDaniel, R., Robert Miller, A. F., Faulring, A., Kyle, B., Mickish, A., Klimovitski, A., and Doane, P. 1997. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering 23,* 6 (June), 347–365.

Myers, B. A. and Rosson, M. B. 1992. Survey on user interface programming. In *Human Factors in Computing Systems.* Proceedings SIGCHI'92, Monterey, CA, 195–202.

Nelson, G. 1985. Juno, a constraint-based graphics system. In *Computer Graphics.* Proceedings SIGGRAPH'85, San Francisco, CA, 235–243.

Pugh, W. and Teitelbaum, T. 1989. Incremental computation via function caching. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages.* 315–328.

Reps, T. 1987. Incremental evaluation for attribute grammars with unrestricted movement between tree modifications. *Acta Informatica 25*, 155–178.

Reps, T. and Teitelbaum, T. 1988. *The Synthesizer Generator.* Springer-Verlag, New York.

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

Repts, T., Marceau, C., and Teitlebaum, T. 1986. Remote attribute updating for language based editors. In *13th ACM Symposium on Principles of Programming Languages.* ACM, 1–13.

Rosener, W. J. 1994. Integrating multi-way and structural constraints into spreadsheet programming. Ph.D. thesis, Department of Computer Science, University of Tennessee, Knoxville, TN.

Ross, R. A. 1985. *Design of Personal Computer Software.* IEEE Press, New York, 282–300.

Sannella, M. and Borning, A. 1992. Multi-Garnet: Integrating multi-way constraints with Garnet. Tech. Rep. 92-07-01, Department of Computer Science and Engineering, University of Washington. Sept.

Saraswat, V. A. 1989. Concurrent constraint programming languages. Ph.D. thesis, School of Computer Science, CMU, Pittsburgh, PA.

Sundaresh, R. 1991. Building incremental programs using partial evaluation. In *Proceedings of the Symposium on Partial Evaluation and Semantics-based Program Manipulation.* New Haven, CT, 83–93.

Sundaresh, R. and Hudak, P. 1991. Incremental computation via partial evaluation. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages.* Orlando, FL, 1–13.

Sussman, G. and Steele Jr., G. 1980. Constraints–a language for experessing almost-hierarchical descriptions. *Artificial Intelligence 14*, 1–39.

Sutherland, I. E. 1963. Sketchpad: A man-machine graphical communication system. In *AFIPS Spring Joint Computer Conference.* Vol. 23. 329–346.

Szekely, P. A. and Myers, B. A. 1988. A user interface toolkit based on graphical objects and constraints. *Sigplan Notices 23,* 11 (Nov), 36–45. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'88.

Ungar, D., Smith, R. B., Chambers, C., and Holzle, U. 1992. Object, message, and performance: How they coexist in self. *IEEE Computer 25,* 10 (Oct), 53–64.

Vander Zanden, B., Myers, B. A., Giuse, D., and Szekely, P. 1991. The importance of pointer variables in constraint models. In *ACM SIGGRAPH Symposium on User Interface Software and Technology.* Proceedings UIST'91, Hilton Head, SC, 155–164.

Vander Zanden, B., Myers, B. A., Giuse, D., and Szekely, P. 1994. Integrating pointer variables into one-way constraint models. *ACM Transactions on Computer Human Interaction 1*, 161–213.

Vander Zanden, B. and Venckus, S. 1996. An empirical study of constraint usage in graphical applications. In *ACM SIGGRAPH Symposium on User Interface Software and Technology.* Proceedings UIST'96, Seattle, WA, 137–146.

Vander Zanden, B. T. 1988. Incremental constraint satisfaction and its application to graphical interfaces. Ph.D. thesis, Cornell University, Ithaca, NY.

Vander Zanden, B. T. 1992. *An Active-Value-Spreadsheet Model for Interactive Languages.* Jones and Bartlett Publishers, Boston, MA, 183–210.

Witkin, A., Gleicher, M., and Welch, W. 1990. Interactive dynamics. *Computer Graphics 24,* 2 (Mar), 11–22.

Witkin, A. and Welch, W. 1990. Fast animation and control of nonrigid structures. In *Computer Graphics: SIGGRAPH'90 Conference Proceedings.* 243–252.

Wyk, C. J. V. 1982. A high-level language for specifying pictures. *ACM Transactions on Graphics 1,* 2 (Apr), 163–182.