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, 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 are commonly used in graphical interface toolkits, programming environments, and circuit applications. Previous papers on dataflow constraints have focused on the design and implementation of individual algorithms. In contrast, this paper focuses on the lessons we have learned from a decade of implementing competing algorithms in the Garnet and Amulet graphical interface toolkits. These lessons reveal the design and implementation trade-offs for different one-way, constraint satisfaction algorithms. The most important lessons we have learned are that, (1) mark-sweep algorithms are more efficient than topological ordering algorithms, (2) lazy and eager evaluators deliver roughly comparable performance for most applications, and (3) constraint satisfaction algorithms have more than adequate speed except that the storage required by these algorithms can be problematic.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Tech-

---

This research was supported in part by the National Science Foundation under grants CCR-9633624 and CCR-9970958.
Authors' Addresses: B. Vander Zanden, Computer Science Department, University of Tennessee, Knoxville, TN 37920; email: bvz@cs.utk.edu; R. Halterman, School of Computing, Southern Adventist University, P.O. Box 370, Collegedale, TN 37315; email: haltermn@cs.southern.edu. B. Myers, 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; D. Kosbie, Sewickley Academy, 315 Academy Avenue, Sewickley, PA 15143; email: dkosbie@yahoo.com.

niques—*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 10 pixels below the bottom of `rect1`.

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 1990a; Myers et al. 1990; 1997; Hill 1993; Hill et al. 1994; Hudson and King 1988; Hudson 1993; 1994; Henry and Hudson 1988; 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 algorithms, nothing has been published that describes the long-term experiences that have been gained from using these algorithms or the algorithmic trade-offs that have been discovered as a result of these experiences.

This paper describes the insights we have gained from 10 years of experience with implementing and adapting these algorithms in the Garnet and Amulet toolkits [Myers et al. 1990; 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 the following:

(1) Arbitrary code: A constraint can contain any code that is legal in the underlying toolkit language. In particular, a constraint can contain arbitrary loops, conditionals, function calls, and recursion [Myers et al. 1990; 1997].

(2) Pointer variables: A constraint can reference variables indirectly via pointers [Szekely and Myers 1988; Vander Zanden et al. 1991]. 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

left = 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]

(3) Automatic Parameter Detection: A constraint's parameters are automatically deduced as the constraint executes, so the programmer does not have to declare a constraint's parameters [Vander Zanden et al. 1994].

(4) Support for Cycles: A constraint is evaluated at most once if it is in a cycle. If the constraint is asked to evaluate itself a second time, it simply returns its original value.

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 1980], and algorithms for performing efficient, incremental constraint satisfaction [Reps et al. 1983; Hoover 1987; Alpern et al. 1990; Hudson 1991]. Both Garnet and Amulet support all the features listed above.

The rest of this paper focuses on the lessons we learned in adapting and extending incremental, one-way constraint satisfaction algorithms to work with the innovations developed for the Garnet and Amulet toolkits. Section 2 provides background about one-way constraints. Section 3 describes various approaches to one-way constraint satisfaction. Section 4 provides an overview of the Garnet and Amulet toolkits. Section 5 describes our experiences with different constraint satisfaction algorithms, including mark-sweep algorithms and topological-ordering algorithms. Section 6 describes the speed and storage efficiency of the constraint systems. Finally, Section 7 describes directions for future work and sums up the lessons we learned.

2. TERMINOLOGY

A *one-way dataflow constraint* can be formally written as an equation of the form

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

where each p_i is a parameter to the function F. The function F is called a *formula*. If the value of any p_i is changed during the program's execution, F is automatically recomputed, and the result is assigned to v. If v is changed by the application or the user, the constraint is left temporarily unsatisfied. Hence, the constraint is *one-way*.

2.1 Dataflow Graphs

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},

[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.

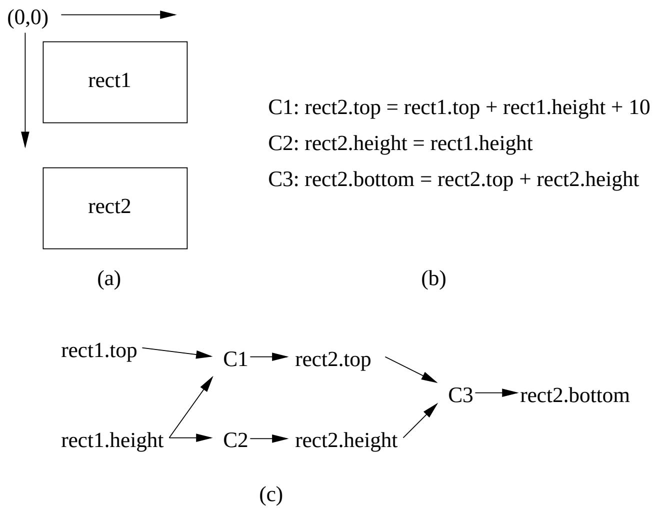


Fig. 1. The dataflow graph (c) generated by the three constraints, C_1, C_2, and C_3 (b) which position the boxes in (a). C_1 positions `rect2` below `rect1`. C_2 makes `rect2` the same height as `rect1`, and C_3 computes `rect2's` bottom. The constraints assume that (0,0) is at the top left, as is in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

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.

2.2 Constraint Satisfaction

Constraint satisfaction refers to the process of bringing constraints up-to-date by evaluating their formulas. 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:

(1) A mark phase that starts at a set of changed variables, performs a depth-first search of the dataflow graph, and marks as out-of-date any constraints it visits.

(2) A sweep phase that evaluates out-of-date constraints. The sweep phase can either evaluate only those constraints whose values are requested, or it can evaluate all out-of-date constraints. The former strategy corresponds to a *lazy* evaluator and the latter strategy to an *eager* evaluator.

A topological-ordering algorithm also has two phases:

(1) A numbering phase that assigns numbers to constraints that indicate the constraints' position in topological order. For example, in Figure 1, C_1 might be assigned 1, C_2 2, and C_3 3.

(2) A sweep phase that evaluates the constraints. The sweep phase uses a priority queue to evaluate the constraints in order using their topological numbers.

Two metrics are often used to evaluate the performance of constraint satisfaction algorithms [Reps et al. 1983; Alpern et al. 1990]:

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

3. APPROACHES TO ONE-WAY CONSTRAINT SATISFACTION

The first algorithms developed for one-way constraint satisfaction were in the area of attribute grammars. 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; Reps et al. 1986].

Later research focused on more general one-way, constraint systems that do not have the restrictions imposed by attribute grammars and hence required the evolution of new 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 (the algorithm often evaluated 70–100% more constraints than necessary—in other words, each constraint was evaluated an average of 1.7–2 times). 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]. A variation of this scheme that accommodated pointer variables and arbitrary code in the constraints was devised for Garnet [Vander Zanden et al. 1994].

Mark-sweep algorithms also received attention from researchers. Hudson devised a lazy mark-sweep algorithm that 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 5.2.1), mark-sweep algorithms gained widespread usage in graphical interfaces. Mark-sweep algorithms that accommodated pointer variables and arbitrary code in the constraints were devised for both Garnet and Amulet [Vander Zanden et al. 1994; Myers et al. 1997].

Other techniques for performing incremental computation, such as function caching and partial evaluation, have also been examined [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. For example, function caching may be able to avert the execution of formula functions, or, if the computation in a function is structured in a certain way, to reduce the amount of required computation. For example, if a computation involves a large data structure, like a symbol table or a list, a change to the data structure may require only an incremental recomputation. If the computation and the data structures are organized properly, then it may be possible for the function cacher to use results from the unchanged portions of the data structure and only perform computations on the changed portions of the data structure [Pugh and Teitelbaum 1989]. These techniques were not used in Garnet or Amulet because constraint satisfaction performance was acceptable without these techniques.

4. GARNET AND AMULET OVERVIEW

Amulet and Garnet are toolkits that make it easier to create graphical interfaces by providing predefined sets of graphical and behavioral objects that can be extended and adapted by programmers [Myers et al. 1990; 1997]. The graphical objects include primitive objects, such as rectangles, text, and lines, and a composite object that allows more complicated objects to be composed from these primitive objects. The behavioral objects include objects that map low-level events, such as mouse clicks, mouse drags, and key presses, into high-level behaviors, such as the selection and movement of graphical objects [Myers 1990b; Myers et al. 1997].

Both graphical and behavioral objects have property/value pairs that can be set by the user to control the objects' appearance or behavior (the properties are called *slots* in Garnet and Amulet). For example, each behavioral object has properties that a programmer can set which indicate the set of graphical objects that the behavior should cover and the set of events that should start and terminate the behavior.

Constraints provide a way for the programmer to specify relationships among these properties. For example, a popular use of constraints is to specify graphical layout, such as centering a text label in a rectangle or attaching the endpoints of lines to the centers of labeled circles.

5. ALGORITHMIC EXPERIENCE

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

(1) it can be difficult to get constraints to execute when the user expects them to execute,

(2) mark-sweep algorithms work better in graphical interfaces than topological-ordering algorithms,

(3) it is not important to avoid unnecessary evaluations in graphical interfaces, and

(4) lazy evaluation performs better than eager evaluation, but generally not by much.

5.1 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 reevaluated. 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.

5.1.1 *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 uninitialized 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. 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 6.1), this problem is less significant than the premature evaluation problem.

5.1.2 *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 then 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 reevaluated 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 reevaluated 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 reevaluated was not reevaluated. 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 reevaluated 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.

5.2 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 surprised us because the programming languages community has assumed that topological-ordering algorithms are faster and hence has focused its investigations on these algorithms [Reps et al. 1983; Alpern et al. 1990].

5.2.1 *Versatility.* The mark-sweep strategy supports both lazy and eager evaluation whereas the topological-ordering strategy supports only eager evaluation. 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 in the dataflow graph. The topological-ordering algorithms do not have this ability, since they must start at the leaf nodes in the dataflow graph, 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.

5.2.2 *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.

5.2.2.1 *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. In this case both algorithms are conceptually simple to implement. However, in practice keeping the topological order numbers properly updated in the topological-ordering algorithm requires either the use of sophisticated algorithms [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.

5.2.2.2 *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.

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]. 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.

5.2.2.3 *Pointer Variables.* Both pointer variables and conditionals may cause the dataflow graph to change dynamically during constraint satisfaction (see Figure 2). Since constraints may contain arbitrary code and since arbitrary code often cannot be statically analyzed, we cannot assume that the constraint solver can determine a priori that the dataflow graph will change during constraint evaluation.

This dynamicism is easily accommodated by the mark-sweep algorithm. For example, in Figure 2, 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. It is also not problematic for the mark-sweep algorithm to dynamically add or remove edges from the dataflow graph.

In contrast, as shown in Figure 2, the topological algorithm is adversely affected in three ways by dynamic changes to the dataflow graph:

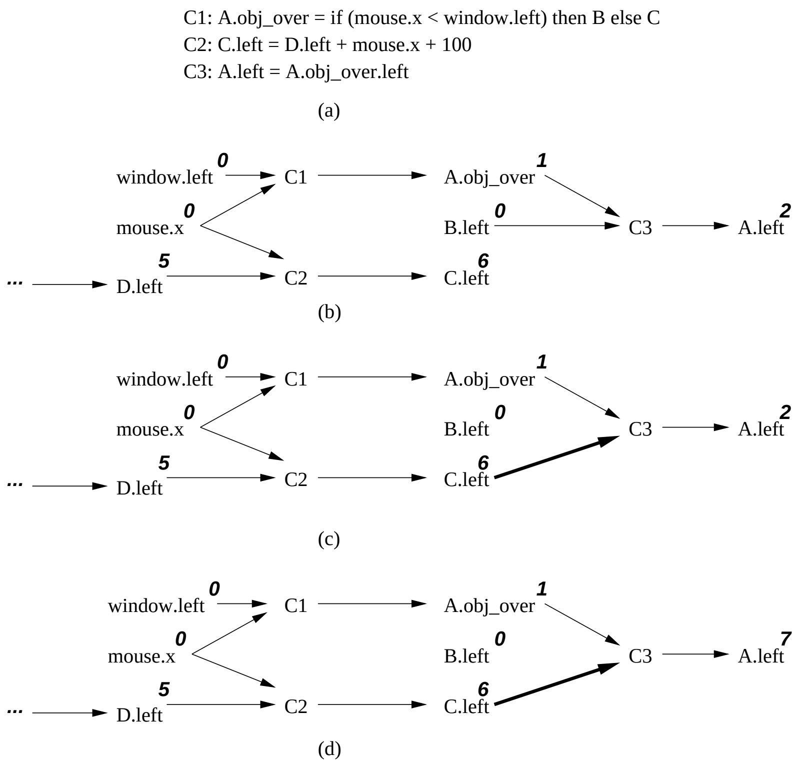


Fig. 2. (a) Three constraints that illustrate how a dataflow graph can change during the course of constraint evaluation. The initial dataflow graph is shown in (b). The numbers to the upper right of each variable in (b) denote the variable's position in topological order. The `...` denotes elided parts 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.

(1) It must suspend the evaluation phase and enter the numbering phase in order to renumber the dataflow graph.

(2) The renumbering can cause constraints that have already been placed on the priority queue to become out of order, so the priority queue may have to be reordered.

(3) The evaluation of the current constraint may have to be aborted because the current constraint may no longer have the lowest topological number.

Each of these tasks requires that additional and often tricky code be added to the constraint satisfaction algorithm. Indeed, the dynamacism of pointer variables is sufficiently problematic that no algorithm for multiway, dataflow constraints has yet been devised that handles the unrestricted use of pointer variables.

5.2.3 *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 2–2.5 times slower than the mark-sweep Garnet algorithm [Vander Zanden et al. 1994].

We were surprised to find that the mark-sweep algorithm was faster because proponents of topological-ordering algorithms have made the following persuasive arguments in their favor [Reps et al. 1983; Alpern et al. 1990]:

(1) The numbering phase of the topological-ordering algorithm should visit fewer nodes of the dataflow graph than the mark phase of the mark-sweep algorithm. The reason is that the numbering phase may only have to renumber a subset of the nodes reachable from a changed node whereas the mark phase may have to visit all the nodes reachable from a changed node.

(2) Although both algorithms evaluate only $O|\mathrm{AFFECTED}|$ 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. It should be noted that a mark-sweep algorithm does not have to evaluate a constraint if none of its parameters has changed, but it does need to check to see whether any parameters have changed.

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. All the mark phase must do is set a flag in each affected node indicating that it is out-of-date. In contrast the numbering phase must find the set of nodes that have inconsistent order numbers and then compute a set of new, consistent order numbers for these nodes. The problem is that finding this set and then creating or adapting order numbers to regain consistency is an expensive computational process.

If the set of nodes examined by the numbering process was considerably smaller than the set examined by the mark process, the large computational expense of renumbering the nodes might be justified by the considerably smaller set size. However, an empirical study of Amulet applications revealed that 60–80% of variables have fewer than 10 constraints that depend on them, either directly or indirectly [Vander Zanden and Venckus 1996]. Further, 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 reevaluated 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.

(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.

Table I. The Benchmark Amulet Applications that Were Used to Obtain the Empirical Results

Application	*Description*
Checkers	Game of checkers
Tree Debugger	Program for visualizing 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

An added disadvantage for the topological-ordering scheme is that in graphical applications, almost all the constraints that depend on a changed variable compute a new value and hence must be reevaluated (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.

5.3 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 occurs when a constraint is reevaluated because one of its inputs is marked as potentially changed, but the input's value has actually not changed.[2] For example, suppose the input is computed by the constraint `if (mouse.x < window.width) then mouse.x else window.width`. When the mouse is moved, the input's value potentially changes. However, unless the mouse moves outside the window, the input's value does not actually change.

A mark-sweek algorithm, regardless of whether it is a lazy or an eager evaluator, must perform special bookkeeping if it is to avoid these types of unnecessary evaluations. In particular, a flag needs to be added to each variable that indicates whether or not it actually changed, and a list of potentially changed inputs must be maintained for each variable. When a constraint's value is requested, the evaluator must first bring all out-of-date inputs up-to-date and then consult their changed flags before evaluating the constraint. Because the additional storage overhead can be somewhat significant (possibly eight bytes per variable for a flag and a pointer to a list), some mark-sweep algorithms simply reevaluate a constraint if it is marked out-of-date.

[2]Some advocates of lazy evaluation describe an unnecessary evaluation as an evaluation whose result is never used (i.e., the constraint is reevaluated a second time before the result of the first evaluation is ever used by the application), but throughout this paper we define an unnecessary evaluation as one in which the constraint's inputs have not changed.

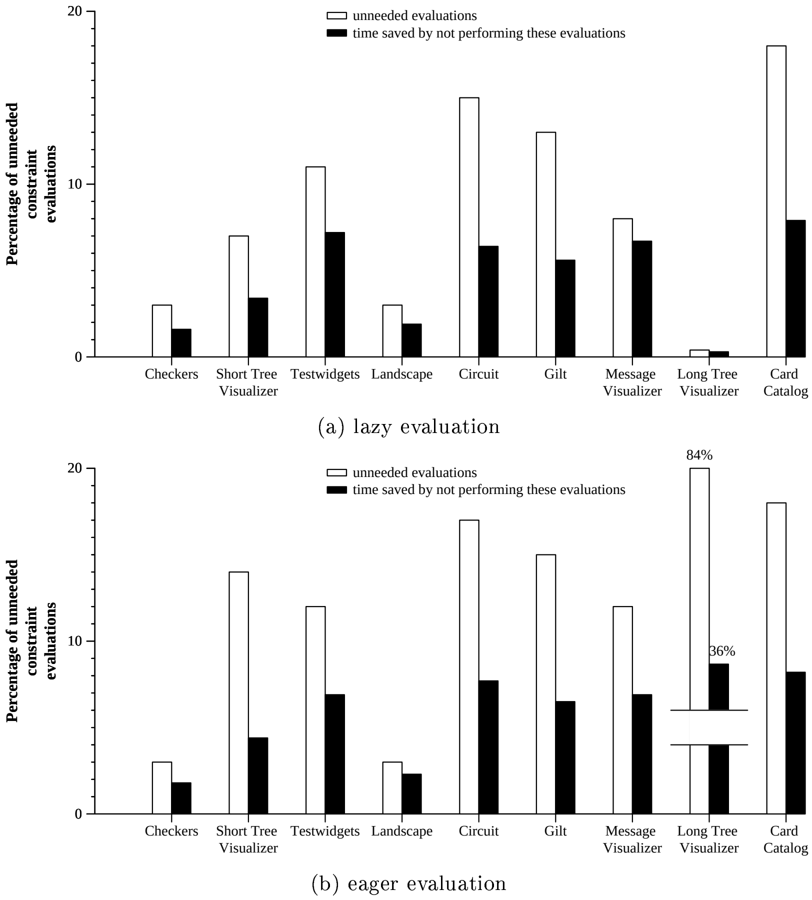


Fig. 3. The first bar shows the percentage of constraints unnecessarily evaluated by a lazy and an eager mark-sweep algorithm that evaluates out-of-date constraints whose inputs have not changed. The second bar shows the amount of constraint evaluation time that could be saved if these unnecessary evaluations were avoided.

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 reevaluation queue unless one of its inputs has changed.

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.[3] The applications are summarized in Table I. The results for both lazy and eager evaluation are shown in Figure 3. The released version of Amulet actually performs these unnecessary evaluations. To perform

[3]The benchmarks that are presented in the following two sections are always Amulet benchmarks. We did not systematically record our measurements for Garnet applications the way we did for Amulet applications. However, the measurements we made during the Garnet development process agree with the measurements presented for the Amulet benchmarks

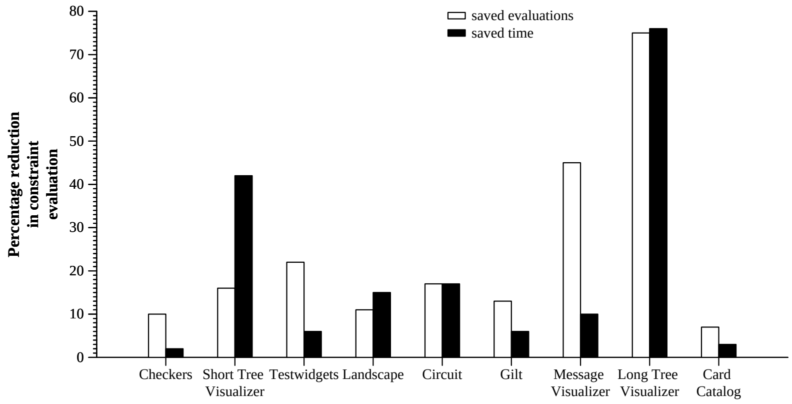


Fig. 4. The percentage reduction in the number of constraint evaluations and the percentage reduction in constraint evaluation time that was achieved by using lazy evaluation rather than eager evaluation in the benchmark applications.

the measurements, we modified Amulet in a manner suggested by Hudson [1991] so that it detected and avoided unnecessary constraint evaluations. 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 are savings in *constraint evaluation* time, not overall application time. Section 6 shows that constraint evaluation time typically represents less than 10% of an application's overall time. Consequently the savings in overall application time are insignificant. Indeed the savings were 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.

5.3.1 *Lessons Learned.* Saving unnecessary evaluations does not result in noticeable speedup in most applications. However, the code that must be written to avoid unnecessary computations is so simple and the run-time cost of doing the checking is so minimal that the optimization is worthwhile.

5.4 Lazy Versus Eager Evaluation

We have previously discussed the tradeoffs of lazy versus eager evaluation from an ease-of-use standpoint (Section 5.1). However, they can also be compared with 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. Figure 4 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 are less than 20%, both in terms of number of constraints evaluated and constraint satisfaction time. Lazy evaluation does not secure greater gains, 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 reevaluated.

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 results suggest that lazy evaluation actually avoids the evaluation of inexpensive constraints. For example, lazy evaluation reduced the number of constraint evaluations in the message-passing application by 46%, but this reduction only decreased constraint evaluation time by 11%.

The one exception to this result 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 reevaluates 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 reevaluate these constraints, because the display manager has not yet been asked to display the tree. Despite the significant savings in constraint evaluation time, Figure 5 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.

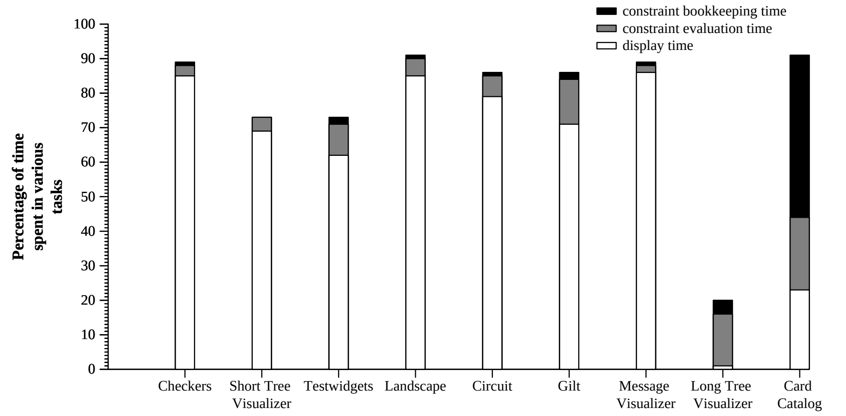
5.4.1 *Lessons Learned.* Given that constraint satisfaction already accounts for a rather small percentage of an application's time, lazy evaluation typically provides almost no speedup in most applications because (1) it does not actually avoid very many unnecessary evaluations and (2) those that it does avoid tend to be inexpensive evaluations.

6. PERFORMANCE EXPERIENCE

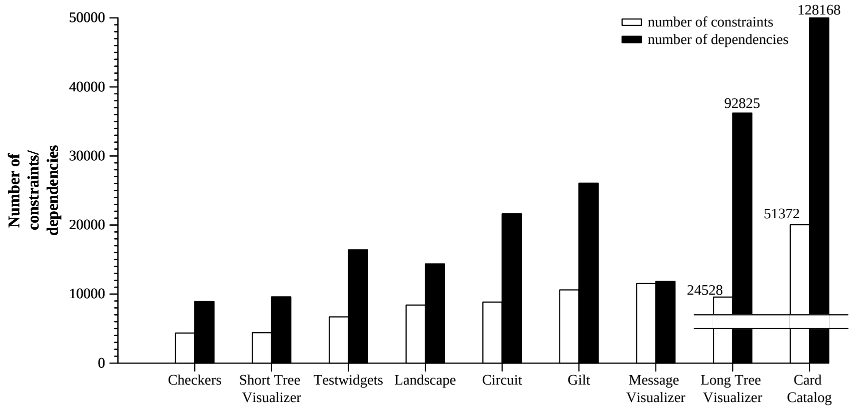
In this section we examine the time and storage efficiency of the Amulet and Garnet constraint systems.

6.1 Time Efficiency

Both the Garnet and Amulet constraint systems were able to solve constraints quickly enough to support interactive behavior. For example, feedback objects



(a)



(b)

Fig. 5. (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.

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 5(a) shows the percentage of several Amulet applications' time spent updating the display, executing formula functions, and performing the overhead required to maintain the constraint 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 were run under X Windows version 6 (several applications crashed under the -O3 option).

The percentages indicate that the constraint overhead is a small fraction of the

Table II. Size of Formula Objects and Dependency Objects in Garnet and Amulet

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

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:

(1) Display time absolutely dominates any other activity that the application performs. All the constraints are satisfied before the display manager is called, so the time shown for the display manager is purely devoted to updating the display. The card catalog application is the only exception to the display time dominance, and the times shown are skewed by the fact that 52% of the application's time was spent in shutting down the application after the exit button had been pressed. If one counts only the time the user spent interacting with the application, then the display time climbs to around 50%, and the bookkeeping overhead for constraints falls to under 6% as well. The high bookkeeping overhead is almost exclusively accounted for by the destruction of the constraint dataflow graph in the clean-up procedure.

(2) 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 5(a), bookkeeping overhead 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.

(3) 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 reevaluated on any given interactive transaction, no matter how big the application.

6.2 Storage Efficiency

Both the Garnet and Amulet constraint systems consume a significant amount of storage. Table II 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 5(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 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, a couple of users have reported that too much memory usage by constraints has been problematic for their applications.

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.

6.3 Lessons Learned

There is a time-versus-storage tradeoff in performing constraint satisfaction. The designers of the early interface toolkits, such as Garnet, ThingLab [Borning 1981], Penguims [Hudson 1994], and Rendezvous [Hill 1993], were concerned that constraint solving could seriously degrade the performance of an interactive application. Therefore, a considerable amount of effort went into devising constraint algorithms that minimized constraint satisfaction time. These algorithms use costly bookkeeping data structures, such as fine-grained dataflow graphs, to speed up performance.

Figure 5 shows that constraint solving time is no longer an issue. However, storage may become an issue as the size of interactive applications continues to increase. Consequently, researchers need to look into ways to trade speed for storage. Microconstraints [Hudson and Smith 1996] and model dependency graphs [Halterman and Vander Zanden 1998] represent two initial efforts to attack this problem.

7. CONCLUSIONS AND FUTURE WORK

Researchers in the user interface community have expended a considerable amount of effort on constraints over the past decade. The Garnet and Amulet projects represent two of these efforts. Our experiences developing constraint satisfaction algorithms for these two toolkits yielded a number of important lessons:

(1) mark-sweep algorithms are preferable to topological-ordering algorithms,

(2) the overhead of constraint satisfaction is insignificant compared with the redisplay times of applications, and

(3) lazy and eager evaluation typically deliver roughly comparable performance, meaning that the choice of an algorithm should often depend on considerations other than performance.

We also discovered a number of areas that need further work by researchers, including:

(1) the development of more storage-efficient constraint satisfaction algorithms, even if it means trading speed for storage, and
(2) the development of constraint satisfaction algorithms and perhaps programmer annotations that ensure that constraints are evaluated when users expect them to be evaluated.

The results in this paper should help guide future developers of constraint-based systems to select appropriate algorithms for constraint satisfaction. They should also help guide researchers to some of the remaining outstanding problems in the area of constraint satisfaction.

ACKNOWLEDGMENTS

We would like to thank John Reppy and the anonymous reviewers for their helpful comments on earlier versions of this paper.

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.

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.

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.

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.

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.

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

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.

Myers, B. A. 1990a. 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. 1990b. A new model for handling input. *ACM Transactions on Computer Human Interaction 8,* 3 (July), 289–320.

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., 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.

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., 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.

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

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

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.

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.

Received January 2000; revised Febuary 2001; accepted June 2001