An Empirical Study of Constraint Usage in Graphical Applications

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

Scott A. Venckus
Computer Science Department
University of Tennessee
Knoxville, TN 37996
svenckus@cs.utk.edu

ABSTRACT
One-way constraints have been widely incorporated in research toolkits for constructing graphical applications. However, although a number of studies have examined the performance of these toolkits' constraint satisfaction algorithms, there have not been any empirical studies that have examined how programmers use constraints in actual applications. This paper reports the results of a study intended to address these matters. Seven graphical applications were chosen for their diversity and profiling information was gathered about their use of constraints. The data reveal 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. This finding suggests that constraint satisfaction algorithms should be able to resatisfy constraints rapidly after a change to one or more variables. It also suggests that debugging constraints should not be unduly burdensome on a programmer since the number of constraints that must be examined to find the source of an error is not terribly large. Overall, the results of this study should provide a repository of data that will be useful in directing future research on optimizing constraint solvers and developing effective debugging techniques.

KEYWORDS: one-way constraints, graphical applications, optimization, debugging, toolkits, profiling

1 INTRODUCTION

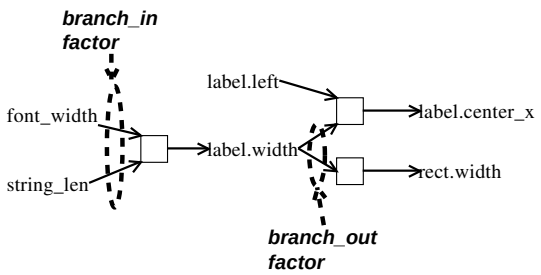
A one-way constraint states a relationship among one or more variables that is automatically maintained by a constraint solver. For example, the relationship `right = left + width` specifies that the right side of a rectangle is equal to the left side of the rectangle plus the rectangle's width. A one-way constraint can be solved in only one direction, typically for the variable on the left side of the equation.

One-way constraint systems are increasingly being employed in research graphical user interface toolkits and applications, including Amulet [10], Garnet [9], Eval/vite [4], Penguims [5], and Rendezvous [1]. However, there have not as yet been empirical studies that examine the characteristics of constraint networks in actual applications or how constraints are used in actual applications. The characteristics of a constraint network include such factors as its width and its depth. How constraints are used include such factors as the frequency with which constraints are evaluated and the manner in which they are used by different operations. Information about these facets of a constraint system could be useful for optimizing constraint solvers and developing effective debugging techniques.

This paper undertakes an empirical investigation of seven graphical applications in an effort to begin building a repository of information about actual constraint systems. The selected applications are representative of the diversity of applications created by toolkits.

Perhaps the most interesting result revealed by the empirical data is that the constraint networks constructed by typical applications are generally modular. In other words, there are many small, independent sets of constraints rather than one large, interrelated set of constraints. This finding has implications for both performance and debugging. The implication for performance is that relatively few constraints must be reevaluated when a variable changes value. Consequently, the constraint solver should be able to update a constraint system rapidly, even when the constraint system is very large, because only a small portion of the constraint system will actually be affected by a change. The implication for debugging is that in practice 1) dependencies are fairly well structured rather than tangled, and 2) programmers do not have to traverse many constraints in order to identify a problem. In other words, while large connected constraint networks may in theory pose debugging problems, in practice they typically do not arise. Hence, in practice, programmers should not encounter significant difficulties debugging constraint systems.

The remainder of this paper discusses these results in greater detail and discusses how they were obtained. Section 2 describes the types of data that were collected by this study. Section 3 describes the methodology employed in profiling the applications. Section 4 presents the profiling results and analyzes their implications. Finally Sections 5 and 6 describe ideas for future work based on some limitations of the study and our conclusions.



label.width = font_width * string_len
label.center_x = label.left + label.width / 2
rect.width = label.width + 9

Figure 1: The constraint network that would be constructed for the above three constraints. The *branch-in* factor is the number of inputs that a constraint depends on. The *branch-out* factor is the number of constraints that depend on a variable.

2 DATA TYPOLOGY

In undertaking the study, we identified the following types of information as being necessary to address the issues raised in the introduction.

2.1 The Structure of Constraint Networks

A one-way constraint network is a directed, bipartite graph G = (V, C, E) where 1) V is a set of nodes denoting variables, 2) C is a set of nodes denoting constraints, and 3) E is a set of directed edges (see Figure 1). There is a directed edge `(v,c)` from node `v` to node `c` if variable `v` is used as an input by constraint `c`, and a directed edge `(c,w)` from node `c` to node `w` if variable `w` is an output of constraint `c`. Two nodes, `x` and `y`, belong to the same *connected component* of the constraint network if there is a directed path from either `x` to `y`, or vice versa. The study described in this paper examined only acyclic constraint networks. Consequently, if there is a path from `x` to `y`, there will not be a path from `y` to `x`. The study also examined only single-output constraints, that is, constraints that have exactly one output variable.

Three factors that contribute to the complexity of a connected component in the constraint network and the amount of time required to satisfy this component are: 1) the *variable depth* which, for each variable, is the number of formulas along the longest path from a variable to a sink node (i.e., a node with no edges leaving the node), 2) the *branch-in factor*, which is the number of edges entering a constraint node, and 3) the *branch-out factor*, which is the number of edges leaving a variable node (Figure 1). The *branch-in factor* is the number of variables that a constraint depends on and the *branch-out* factor is the number of constraints that depend on a variable.

In general, the smaller these factors are, the smaller are the number of constraints in a connected component. Hence, when these factors are small, it is more likely that fewer constraints will have to be resatisfied if a variable changes value, thus improving performance. Of course, other factors also affect performance, such as the time to execute the code associated with a constraint or the overhead of calling the code associated with a constraint. However, in analyzing competing constraint satisfaction algorithms, researchers typically treat these other factors as constants, and instead focus on the total number of formulas that are evaluated. Consequently, we chose to examine the factors that influence the total number of formulas evaluated, such as variable depth, branch-in factors, and branch-out factors.

2.2 The Frequency of Constraint Evaluation

The frequency of constraint evaluation refers to how often a constraint is evaluated. Our study employed two measures of frequency of constraint evaluation:

- The number of times a constraint is evaluated.
- The average number of times a variable is read between constraint evaluations. In other words, how many times is a variable's value accessed before its constraint is reevaluated?

2.3 Patterns of Constraint Usage in Operations

The Amulet toolkit which was used in our study (see Section 3.3) divides operations into a small number of categories, such as move/grow operations and text editing operations [10]. It further subdivides each operation into 1) a start action that executes when the operation begins, 2) a running action that repeatedly executes when the operation is in progress, and 3) a stop action that executes when the operation ends. Our study collected data on the average number of constraints that were used by each category of operation, and the average number of constraints that were used by each subpart of an operation. Although the separation of interactions into very distinct startup, run, and shutdown components is particular to Amulet, these general categories of action are likely to be found in all interfaces, even if they are not explicitly segregated.

3 METHODOLOGY

Our profiling information was obtained from seven graphical applications written using the Amulet toolkit [10].

3.1 Selection of Graphical Applications

The selection of the graphical applications was governed by criteria of diversity, size, and varying levels of programmer experience, and is detailed in Table 1.

3.2 How the Study was Performed

A task-to-be-performed script was prepared for each application and then executed using an instrumented version of Amulet. Since performance measurements were not part of the study, the overhead introduced by the instrumentation did not affect the results.

3.3 Amulet Overview

Amulet is a toolkit written in C++ that provides support for building highly interactive graphical applications[10].[1] It currently runs on Unix, Macintosh, and Windows platforms. A number of features of Amulet are relevant to the study:

- A prototype-instance object model [12, 9]. A prototype-instance model allows any object to be used as a prototype

[1] Amulet is free and in the public domain. Information about obtaining Amulet can be found at http://www.cs.cmu.edu/~amulet.

Name	*Description*	*Programmer Experience*	*Number of Constraints*
Checkers	Game of checkers.	Amulet developer	3571
Constraint Animator	Graphical animation of a constraint satisfaction algorithm	Graduate student	3998
SillyDraw	Drawing editor	Undergraduate	2297
VCR	Widget-based, VCR interface	Undergraduate	2551
Space	Star-Trek like game interface	Amulet developer	2055
Mathnet	Visual programming editor that allows creation of arithmetic expressions using boxes and arrows	Graduate student	3646
Whiteboard	Example-based, visual programming environment for creating array-based programs	Graduate student	3293

Table 1: Summary of the graphical applications selected for the study.

at run-time for creating instances of objects. Amulet provides a set of graphical prototype objects, such as lines, rectangles, and text, that may be used by an application.

- A one-way constraint model. A programmer can write a formula containing arbitrary C++ code and then attach an instance of this formula to any variable in an object. For example, the formula can compute the position of an object, its size, its color, its line-style, or its visibility. Formulas may be inherited, so that when an instance of an object is created, the variables in the instance will inherit formulas from their corresponding variables in the prototype.
- A behavior model. Amulet provides *interactors* to simplify the handling of events [8]. Five generic interactors are provided, representing abstract behaviors exhibited by programs. These generic interactors serve as prototype objects, from which new objects can be created, customized, and attached to graphical objects. The five generic interactors are:

  ChoiceInteractor – The choice interactor is used to select from a collection of one or more screen objects. It can provide both interim feedback to show interim selections and final feedback to show the final selection.

  OneShotInteractor – The one shot interactor is used to provide an immediate response to events like a keystroke or mouse press. It is similar to a choice interactor, except that it does not provide interim feedback.

  Move/GrowInteractor – The move/grow interactor is used to move or resize an object.

  NewPointsInteractor – The new points interactor is used to enter the coordinates of a new object.

  TextEditInteractor – The text edit interactor is used to edit a string of text.

4 RESULTS

Perhaps the most general result of our empirical study was that despite the diversity of chosen applications and programmer experience, the profile data for all the applications was remarkably similar. This finding suggests that optimization and debugging strategies which are based on the data presented in this paper will be broadly applicable to diverse graphical applications written by programmers of all experience levels.

The results in this section are described according to the typology of Section 2: 1) the structure of constraint networks, 2) the frequency of constraint evaluation, and 3) the patterns of constraint usage in operations.

4.1 The Structure of Constraint Networks

Results. Figures 2-4 show the variable depth, branch-in, and branch-out factors for the seven applications. Aggregate graphs that combine the data for all seven applications are also shown for the branch-in and branch-out factors.[2] Figure 2 shows that the variable depth is quite small, typically six or less (and no greater than 12 in our applications). Since the depth of a constraint network cannot be greater than the maximum depth of any variable in that network, the shallowness of the variable depths means that constraint networks are shallow. We found that the reason for this shallowness derived from the fact that constraint networks tended to be modular. As applications grew larger, they added new "modules" to the constraint system, rather than significantly expanding old networks.

Figure 5 provides another measure of the size of constraint networks. It shows the distribution of formulas that are downstream from each root variable. The figure reveals that the majority of root variables, 60-80%, have fewer than 10 constraints downstream of them. This finding means that if these variables are changed, fewer than 10 constraints will have to be evaluated in order to bring the constraint system up-to-date. It also means that very few constraints will have to be examined in order to determine the cause of an error.

Taken together, the figures in this section suggest that constraint networks tend to be narrow, as well as shallow. Figures 3-4 suggest that constraint networks tend to be narrow, since the branch-in and branch-out factors tend to be small. Of course, it is certainly possible to get bushy constraint net-

[2] Aggregate graphs are not shown for the other graphs in the paper, because, although the shapes across applications are similar, the magnitudes of the numbers are dissimilar.

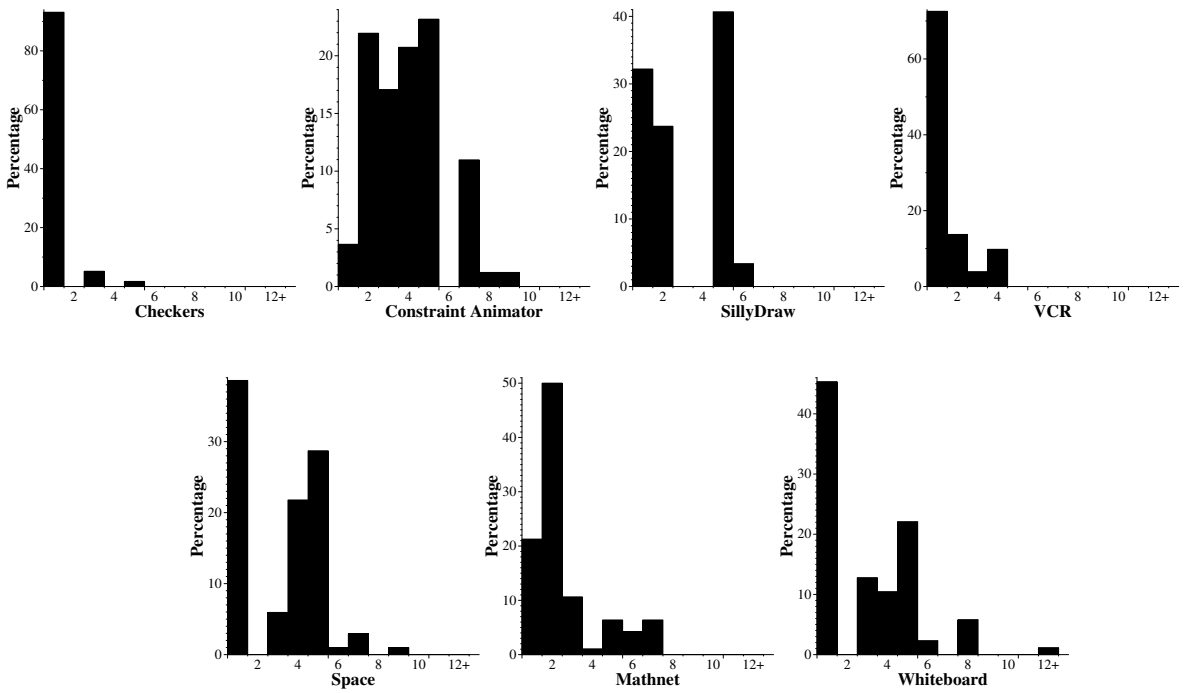


Figure 2: Depth of variables in constraint networks.

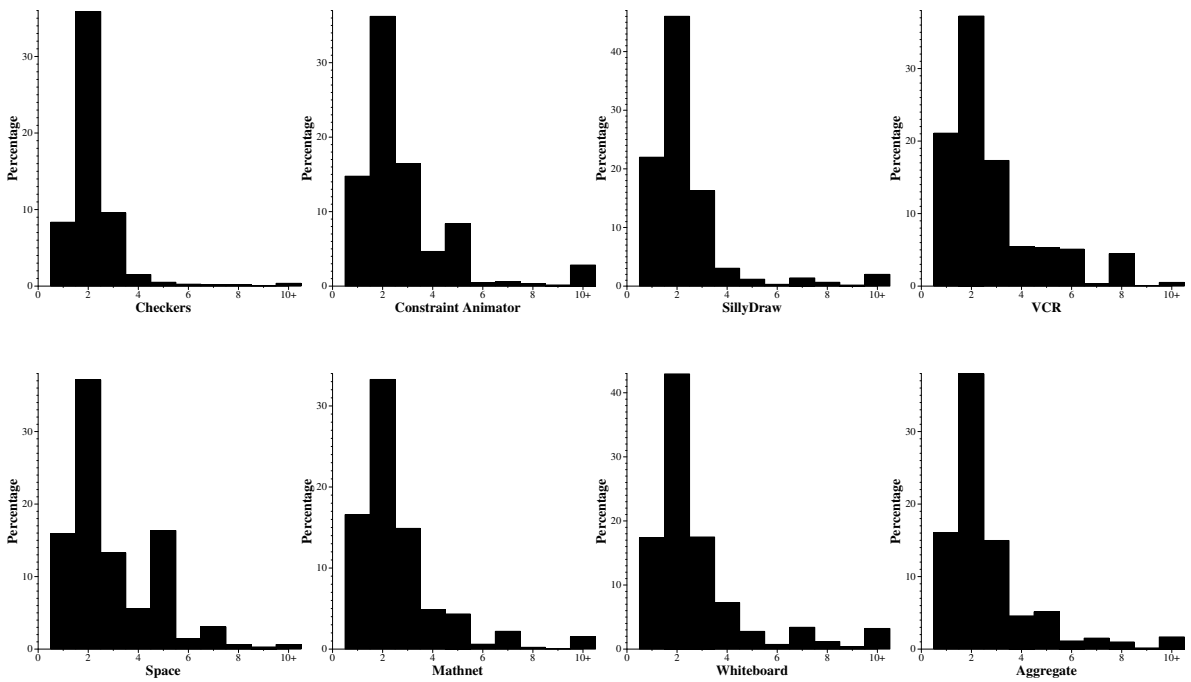


Figure 3: Branch_in factors for constraints. “Aggregate” is a combination of data from the seven applications.

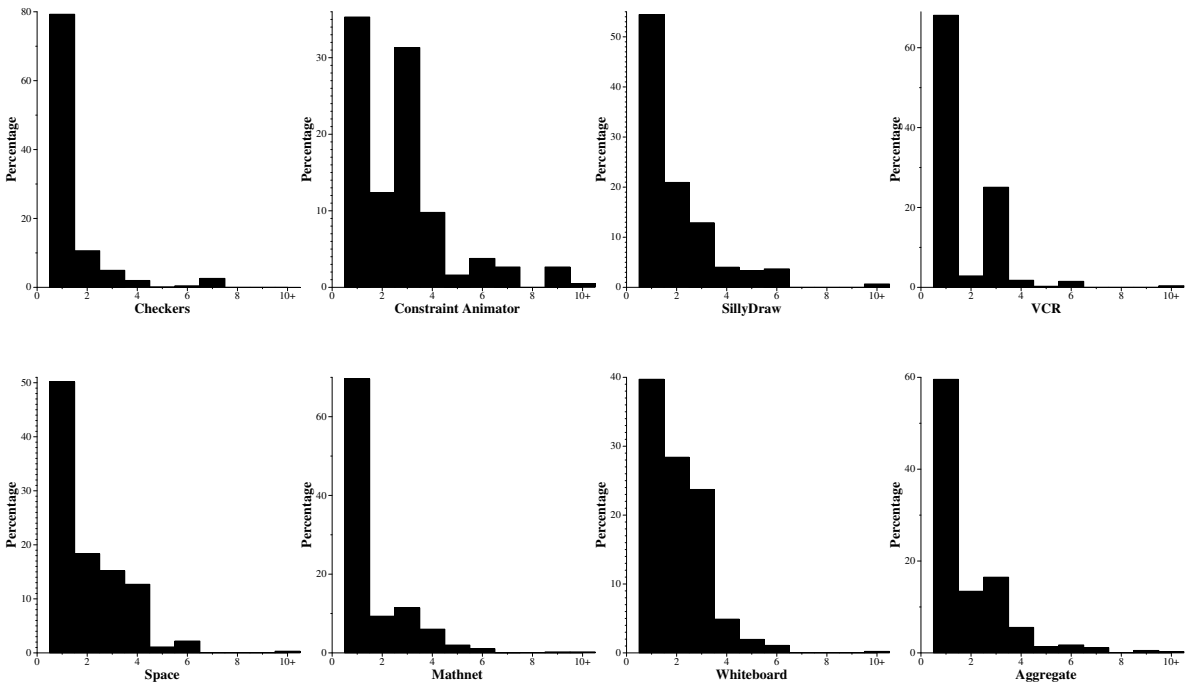


Figure 4: Branch_out factors for variables. "Aggregate" is a combination of data from the seven applications.

works with small branch-in and branch-out factors. For example, a constraint network that looks like a binary tree will have a branch-in factor of two at every constraint node and a branch-out factor of one at every variable node. However, Figure 5 virtually rules out these types of networks since the number of formulas downstream from root variables is typically too small to support bushy networks. Consequently, the data suggests that most constraint networks are narrow.

Analysis. The structure of a constraint network has implications for both debugging and for the efficiency and optimization of constraint satisfaction algorithms. As noted in the introduction, the fact that constraint networks tend to be shallow and narrow implies that debugging constraint systems will not be problematic for programmers. Shallow and narrow constraint networks also imply that visual browsers will be an effective debugging tool for programmers, because there will not be a proliferation of dependencies on the screen nor a large number of constraints to traverse. This finding should allay concerns that debuggers based on visual browsers may be ineffective because large constraint systems could create tangled webs of dependencies and cause the programmer to visually navigate hundreds of constraints.

The findings about the structure of constraint networks in graphical applications also have the following implications for efficiency and optimization:

- Running times of one-way constraint satisfaction algorithms. Studies that mathematically evaluate the efficiency of one-way constraint satisfaction algorithms typically assume that the *branch-in* and *branch-out* factors are, in most cases, bounded by a constant [13, 3, 1]. However, this assumption has never been empirically verified. Our data appear to validate this assumption and hence provide substantial support for the efficiency claims of many one-way constraint satisfaction algorithms.
- Storage of constraint dependencies. The data structures used to represent constraint dependencies can account for a significant proportion of the storage consumed by formula data structures. For example, an informal survey of Garnet applications showed that dependencies consumed roughly 50% of the storage consumed by formula data structures (our study of Amulet applications did not measure storage). Additionally, an examination of the data structures representing constraint dependencies in the Amulet [10], Garnet [9], Rendezvous [1], and Eval/vite [4] systems reveals that they consume 16-24 bytes of memory (although Garnet and Rendezvous were implemented in Lisp, the bytes consumed by their data structures were counted as if the implementations were in C, so that the numbers would be comparable with Amulet and Eval/vite). Roughly 8 bytes of this total is accounted for by pointers to the next dependency (one pointer to the next dependency edge leaving a node and one pointer to the next dependency edge entering a node).
  Figures 3-4 show that the majority of constraints have a branch-in factor of two or less and that the majority of variables have a branch-out factor of two or less. This finding suggests that the storage required to represent constraint dependencies could be reduced by modifying the dependency data structures so that they contain a fixed amount of memory for four dependencies (two incoming dependencies and two outgoing dependencies). Overflow dependencies can be handled as a linked list.
- Probabilistic constraint satisfaction algorithms. Most constraint satisfaction algorithms attempt to avoid evaluating a constraint before all of its arguments have been brought up to date. Waiting until all of a constraint's arguments are brought up to date ensures that the constraint will not be evaluated more than once. However, a certain amount of

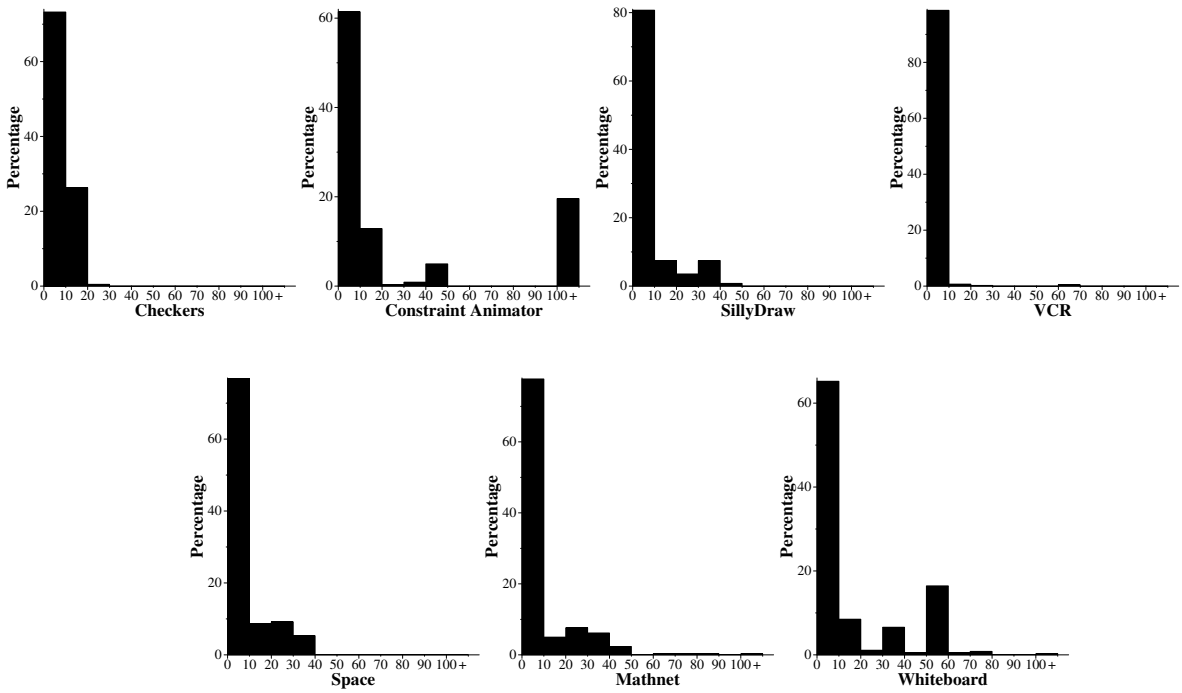


Figure 5: Number of constraints that depend on a root variable (i.e., are downstream of a root variable). The y-axis shows the percentage of root variables with a particular range of downstream constraints (e.g., 0-10, 10-20, etc).

overhead must be incurred in order to avoid these unnecessary reevaluations.

This overhead can be avoided if one is willing to evaluate a constraint before all of its arguments have been brought up to date. Naive propagation schemes that evaluate a constraint as soon as one of its arguments changes employ this strategy. So does a scheme called approximate topological ordering, which evaluates constraints in approximately the correct order, but not the guaranteed correct order [2]. These algorithms are "probabilistic" in the sense that when they evaluate a constraint, there is some probability they are evaluating the constraint out of topological order and will have to evaluate it again.

These probabilistic algorithms work well in "serial" networks in which there is at most one path from one node to another node. They do not work as well in "parallel" networks where there are multiple paths from one node to another node. Indeed, certain pathological constraint networks can cause these algorithms to evaluate a constraint an exponential number of times [3].

The data in Figure 4 suggest that networks are indeed somewhat serial, since 1) variables that have a branch-out factor of one constitute roughly 60% of all variables, and 2) variables that have a branch-out factor of one cannot create parallel networks. Consequently, in practice, probabilistic algorithms may perform better than researchers have anticipated. In addition, since constraint networks are shallow, probabilistic algorithms are not likely to perform too poorly, even if they do evaluate some constraints more than once. Consequently, our data suggest that further evaluation of probabilistic algorithms may be warranted.

4.2 The Frequency of Constraint Evaluation

Results. Figure 6 shows the number of times constraints were evaluated in the seven applications and Figure 7 shows the average number of times a variable's value was read between constraint evaluations. Two results are prominently highlighted by the graphs in these figures: 1) a significant percentage of constraints are evaluated yet their values are never requested, and 2) the great majority of constraints are evaluated relatively infrequently. Both of these results, plus a couple other facets of the data are analyzed below.

Analysis. Both of the above measures of constraint evaluation have implications for constraint optimization:

1. Eager versus lazy evaluation: The data for average reads between evaluations reveal that a large percentage of variables were *never* demanded. This finding implies that the constraints associated with these variables were unnecessarily evaluated (Amulet employs eager evaluation). Consequently, the start-up times for applications were longer than necessary, and formula data structures were unnecessarily allocated. This finding suggests that proponents of lazy evaluation make a strong point when they suggest that the unnecessary constraint evaluations performed by eager evaluation can degrade the performance of an application.

2. Lightweight formula data structures: Figure 6 shows that a significant majority of constraints are evaluated infrequently in an application. Additionally, Figure 7 shows that a significant majority of constraints have at least several average reads per constraint evaluation. Together, these two findings suggest how an infrequently evaluated for-

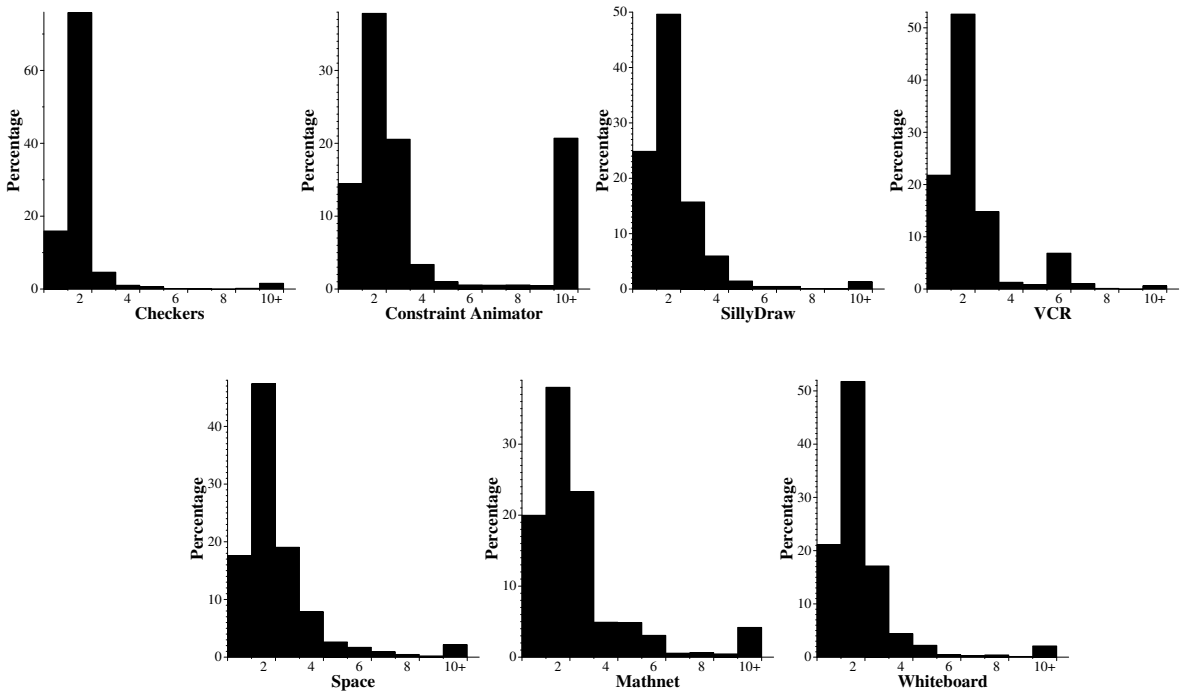


Figure 6: Number of times constraints were evaluated.

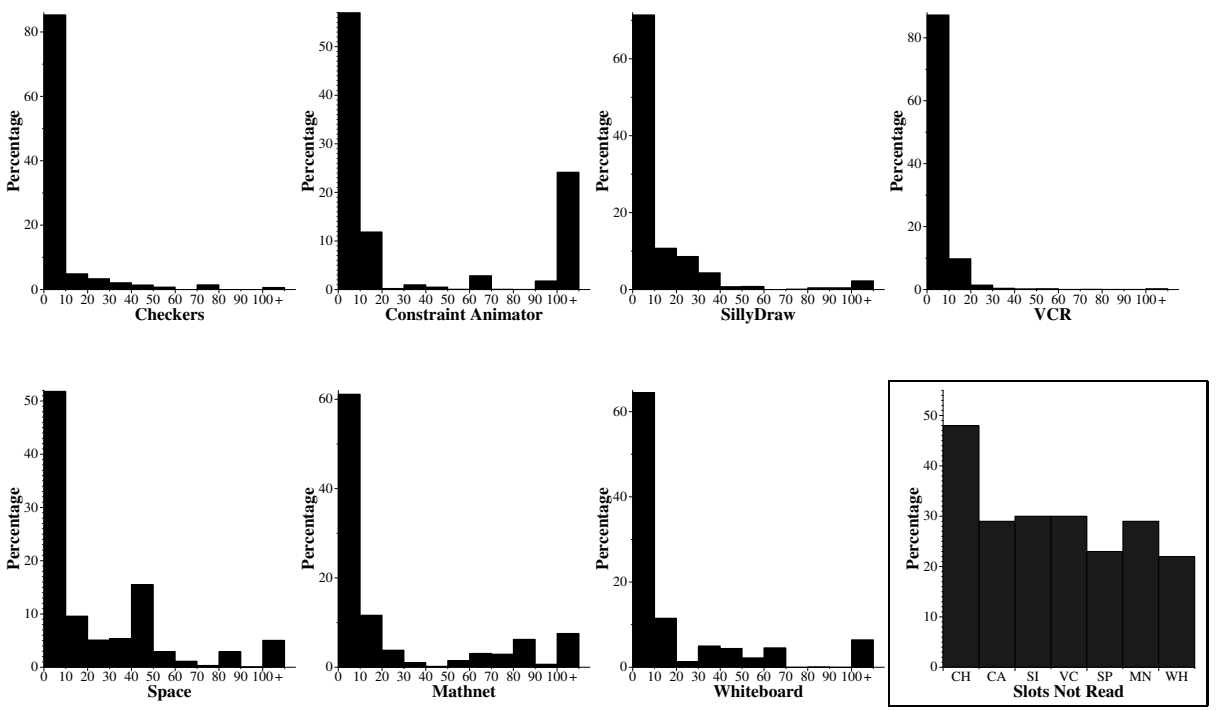


Figure 7: Number of times a variables was read between constraint evaluations. The last graph shows the percentage of variables in each application that were computed by a constraint but which were never read.

mula's data structure might be optimized to reduce storage, without incurring a large performance penalty. Since a constraint's value is often read between evaluations, the constraint's value and whether the constraint's value is up to date should be readily accessable. These fields should not be eliminated from a formula's data structure.

However, information about a constraint's dependencies does not have to be as readily accessable, since this information is accessed only when a constraint is going to be marked out of date or when it is evaluated. Hence, it may be feasible to find a more economical means of storing a constraint's dependencies that involves a somewhat longer access time (e.g., packing a constraint's dependencies into a dynamic array which might have to be reallocated if the number of dependencies change). Similarly, the pointer to a constraint's function does not have to be as readily accessable. For example, in a prototype instance system, it may be possible to store a pointer to the constraint's function in the prototype's formula data structure rather than in each instance's data structure. The function can then be located via inheritance when it is needed.

3. Constant propagation: Constant propagation is an optimization technique whereby constraints with constant arguments are eliminated and replaced with the value they compute [6, 7]. The data in Figure 6 suggests that roughly 15-25% of the constraints in an application can be eliminated via constant propagation. The significance of this number is that constant propagation can be expected to achieve good, but not terrific, storage savings. It should also be noted that the data represent an upper bound on constraints that depend on constants, since our study does not guarantee that a constraint which was evaluated only once depends only on constant inputs. However, every effort was made to thoroughly exercise the applications, so the data should provide a good approximation of the percentage of constraints that will be evaluated only once in an application.

4.3 Patterns of Constraint Usage in Operations

Results. Figure 8 shows the patterns of constraint usage in each of the five types of operations that Amulet supports. These numbers were obtained by averaging the results from all seven applications. The figure separately categorizes the results for starting, running, and stop actions. Consequently, an interactor that executes no constraints in the start action but numerous constraints in the running action will be grouped in the 25th percentile for start actions but a higher percentile for running actions.

The results show that start and running actions make fairly sparing use of constraints whereas stop actions are fairly heavy users of constraints. These results are understandable in light of the fact that start and running actions typically set up and provide only interim feedback whereas the stop actions typically perform a significant activity, such as creating a new object or rearranging objects on the screen.

Interim feedback is usually provided by displaying some type of object, such as a dashed line box or a highlighting box. During the start and running actions, generally only constraints in the interim feedback objects are reevaluated and these constraints are typically positioning constraints (e.g., position a highlighting box over a selected item). Since only a few constraints are required to position these objects, relatively few constraints are evaluated during the start and running actions. The only exceptions to this general rule occur when the actual processing of the operation occurs both during the running action and the stop action. For example, sometimes either a text editing or move/grow operation will rearrange objects on the screen as the user enters text or moves/resizes an object (when entering text, a frame around the box may resize itself to accommodate the text or arrows attached to the frame may reposition themselves to remain attached to the frame). In these cases, significant numbers of constraints may be evaluated both during the running and stop actions.

As noted above, stop actions typically performed the majority of processing in an operation and hence were the largest user of constraints. The activities performed by stop actions included 1) providing final feedback, such as selection handles, 2) rearranging objects on the screen, and 3) altering properties of objects, such as their color, size, and line style, based on changes that were made to application data. All of these activities were typically performed by constraints. The direct actions of a stop procedure were generally limited to modifying a few variables or creating a new object. Constraints propagated these changes to the appropriate places in the application and in the graphical interface.

Overall, the choice, move/grow, and new point operations were the largest users of constraints. Choice operations used constraints to position selection handles, or, if an operation was selected, to modify objects' graphical properties based on changes to application data. Move/grow operations used constraints to reposition objects on the screen. New point operations used constraints to position newly created objects on the screen and to calculate these objects' graphical properties based on the application data they were displaying. Text operations used fewer constraints because they caused fewer objects to be repositioned than move/grow operations, and because they typically did not communicate the new text string to the application (often the text strings were simply labels). There were too few one-shot interactors to draw any definitive conclusions about how they were being used.

Analysis. One use of the above information is in deciding whether it is advantageous to construct a constraint plan for an operation. A constraint plan is a set of constraints that are executed by an operation. A straight forward technique for constructing a plan is to collect constraints onto a list or some other suitable data structure the first time an operation is executed. If this operation is then repeated on the same set of objects, the constraints can then be reexecuted, without having to incur the expense of recollecting them. The drawback of this approach is that the plans are limited to the same set of objects. Consequently, they are useful if the operation is repeatedly executed, as for example, when an interim feedback object is moved about the screen, and when a large number of constraints are involved. However, since they re-

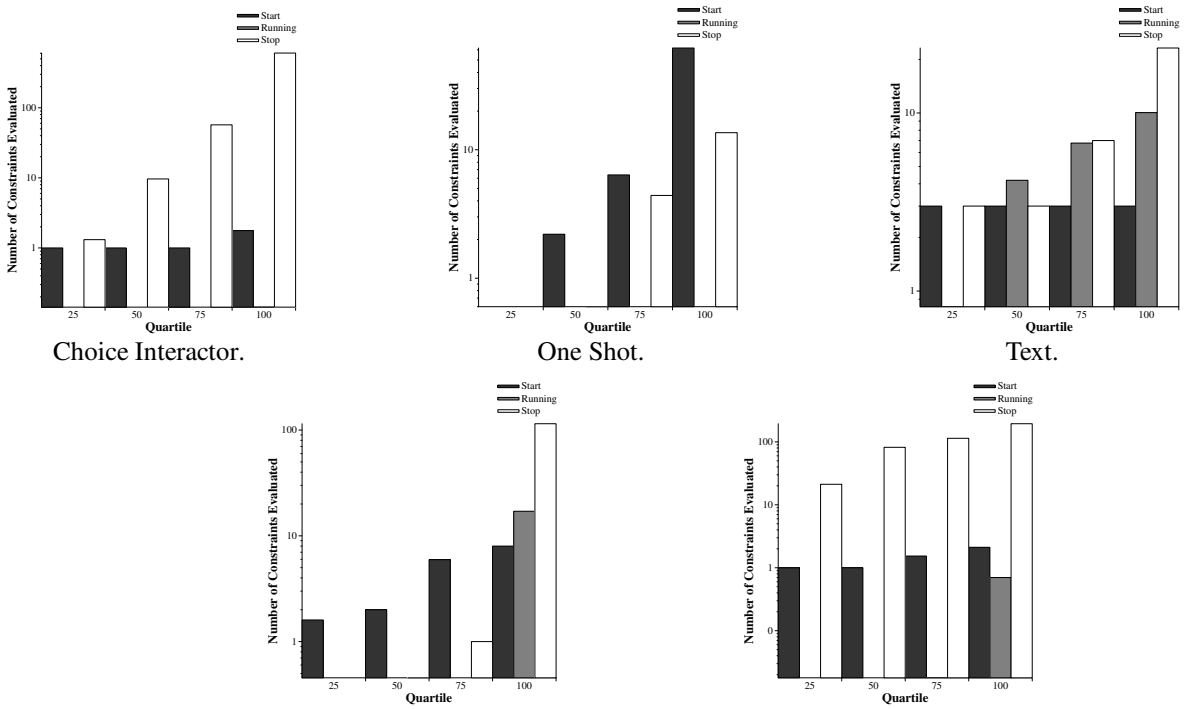


Figure 8: Constraint evaluation by interactor type.

quire some overhead to construct, it is not feasible to create them unless the operation will be repeatedly reexecuted. In general, the starting and stopping actions of an operation are not repeated, and thus plans are less useful for these actions. On the other hand, running actions tend to be repeatedly executed. In addition, the user is more likely to expect good response time from running actions. Consequently, if a running action uses a great many constraints, constructing a plan would prove useful. On the basis of this analysis, the data in Figure 8 suggest that the running actions for move/grow and text editing operations may be particularly fruitful parts of a graphical application for the construction of straightforward plans. However, the data also suggests that since relatively few running actions use significant numbers of constraints, plans that are limited to a fixed set of objects are unlikely to result in significant performance improvements.

A more promising approach is to use a parameterized plan in which the constraints can contain references to the parameterized objects. These plans can be repeatedly reused with different sets of objects. Consequently, they would be useful for any type of action—start, running, or stop—that used a significant number of constraints. The data in Figure 8 suggest that a fairly large proportion of actions do involve a significant number of constraints. Hence, it appears that parameterized plans could lead to significant performance improvements.

5 Limitations and Future Work

One of the limitations of this study is that it is based on only one toolkit, Amulet. This limitation was necessary because Amulet is currently the only widely distributed and used graphical toolkit that includes a one-way constraint system (actually Garnet [9] is a second such system, but Amulet is based on Garnet, so including Garnet applications in the study would not have provided an independent set of results).

It is conceivable that the results could vary for other toolkits since the manner in which a toolkit is structured can affect the constraint networks that are constructed by applications. For example, most toolkits, including Amulet, will automatically generate a certain number of constraints when objects, such as widgets are created. The set of constraints that are generated will differ from toolkit to toolkit, and hence will result in different constraint networks. Similarly, other toolkits will potentially structure their input operations (e.g., selection, text editing, moving objects) differently than Amulet.

Despite these potential differences, it is our belief that as additional toolkits become available, the constraint networks constructed by these toolkits will roughly meet the profile of the data presented in this paper. Our optimism is based on the following two reasons:

1. Despite toolkit-generated constraints, a substantial number of constraints are still created by the programmer. Surveys of other papers about one-way constraint systems (e.g., Rendezvous [1] and Eval/vite [4]) suggest that the style of constraints written in these systems mirrors those written in Amulet.

2. Although toolkits may structure their input operations differently than Amulet, they will still support a comparable

set of operations (e.g., selection, text editing, and moving or resizing an object). These operations will change a similar set of variables in an application (e.g., an object's coordinates or its string variable). Consequently, one would expect to see roughly the same distribution of constraint usage across operations and the same frequency of constraint evaluation.

It is less evident how toolkit-generated constraints might alter the results. However, a wide range of programmer generated constraints were examined in this study and invariably the programmers created small sets of independent constraints. This finding at least suggests that developers of future toolkits may well create code that generates small sets of independent constraints.

Of course, as additional toolkits that include one-way constraints become available in the future, it will be useful to expand this study to include applications written using these toolkits.

6 CONCLUSIONS

Previous studies have evaluated the performance of various one-way and multi-way constraint satisfaction algorithms [3, 11, 1, 13, 15]. However, this paper is the first to describe the results of an empirical investigation of how constraints are used in actual graphical applications. The study found that the constraint systems constructed by applications tend to consist of small, independent sets of constraints rather than one large, interrelated set of constraints. In addition, the number of formulas that depend either directly or indirectly on any given variable typically quite low and often less than ten. Since most one-way constraint solvers can rapidly satisfy small sets of constraints, these findings suggest that constraint satisfaction is unlikely to consume a significant portion of an application's runtime. These findings are also consistent with previous studies that have shown that the time to update the display, rather than the time to update constraints, is the dominant factor in most graphical interfaces [1, 14].

This study also provides a repository of data for researchers attempting to develop more effective optimization and debugging strategies for constraint systems. Hence, the results in this paper should provide an impetus to researchers to develop more effective optimization and debugging strategies for constraints.

REFERENCES

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

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

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

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

5. Scott E. Hudson. User interface specification using an enhanced spreadsheet model. *ACM TOGS*, 13(3):209–239, July 1994.

6. John Maloney, Alan Borning, and Bjorn Freeman-Benson. Constraint technology for user-interface construction in ThingLabII. *Sigplan Notices*, 24(10):381–388, October 1989. ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'89.

7. Brad Myers, Dario A. Giuse, Andrew Mickish, and David Kosbie. Making structured graphics and constraints practical for large-scale applications. Technical Report CMU-CS-94-150, Carnegie Mellon University Computer Science Department, May 1994.

8. Brad A. Myers. A new model for handling input. *ACM TOIS*, 8(3):289–320, Jul 1990.

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

10. Brad A. Myers, Rich McDaniel, Alan Ferrency, Andy Mickish, Alex Klimovitski, and Amy McGovern. The Amulet reference manuals. Technical Report CMU-CS-95-166, Carnegie Mellon University Computer Science Department, June 1995.

11. Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experiences with the DeltaBlue algorithm. *Software Practice and Experience*, 23(5):529–566, 1993.

12. David Ungar and Randall B. Smith. SELF: The power of simplicity. *Sigplan Notices*, 22(12):227–241, Dec 1987. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications; OOPSLA'87.

13. Brad Vander Zanden, Brad A. Myers, Dario Giuse, and Pedro Szekely. Integrating pointer variables into one-way constraint models. *ACM TOCHI*, 1:161–213, June 1994.

14. Brad Vander Zanden. Optimizing toolkit-generated graphical interfaces. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 157–166, Marina del Rey, CA, Nov 1994. Proceedings UIST'94.

15. Brad Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way, dataflow constraints. *ACM TOPLAS*, 18(1):30–72, January 1996.