To the Graduate Council:

I am submitting herewith a thesis written by Scott Alan Venckus entitled "An Empirical Study of Constraint Usage in Graphical Applications." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Brad Vander Zanden

Dr. Bradley T. Vander Zanden, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Associate Vice Chancellor and
Dean of the Graduate School

An Empirical Study of Constraint Usage in Graphical Applications

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Scott Alan Venckus

August 1996

Dedication

To Mary, Bonnie, Cosa, Otis, and Ripley.

Acknowledgments

There are several people who I need to thank for the help they provided in the completion of this thesis.

I thank my advisor, Brad Vander Zanden. He has given his financial support, time, and infinite patience (especially with my grammatical and spelling errors) during the last year.

I thank Jim Plank, both for serving on my committee and for writing jgraph, the ultimate postscript plotting tool.

I thank all of those people close to me who generously gave their support. This includes my parents, who have encouraged me for the last three years, and Bonnie, Cosa, Otis, and Riply.

Lastly, I thank Mary Eberlein for always being there.

Abstract

The spreadsheet model of computation has simplified the task of constructing business software. This success has led researchers to apply this model to other areas of programming, especially graphical interfaces. The spreadsheet model of computation is called constraint programming in the context of general purpose programming languages.

Although the technique of using constraints to specify graphical relationships among objects has enjoyed wide acceptance in the research community, commercial developers have been reluctant to incorporate this technique into commercial toolkits. This reluctance is based on the industry's concern about the complexity of constraints resulting in debugging problems and an overall concern about the efficiency of constraints.

To explore the merits of these concerns, we undertook a study that attempted to quantify data about the use of constraints, the nature of constraint networks and draw conclusions about constraint debugging and optimization from this data. The Amulet graphical library was instrumented to gather data on the structure of constraint networks as well as overall data access patterns. Seven applications, gathered from a variety of sources were profiled using this modified library.

The data collected in this study led us to several conclusions. First, it was found that constraint networks tended to be narrow and short. As a result, concerns about debugging constraint based applications should be lessened. Second, because of the modular nature of

constraint networks, constraints should scale well in larger applications. Lastly, the study provides a repository of data that can be used in future studies of debugging and optimizing constraints.

Contents

1 Introduction **1**

2 Data Typology **8**

2.1 Data Access Patterns 8

2.2 The Frequency of Constraint Evaluation 9

2.3 Patterns of Constraint Usage in Operations 9

2.4 The Structure of Constraint Networks 10

3 Related Work **13**

3.1 Debugging Constraint Networks 14

3.2 Constraint Plans 15

3.3 Data Access Issues 16

4 Amulet Architecture and Terminology **18**

4.1 The Amulet Object System 19

4.2 The Constraint System 20

4.3 Hierarchical Composition of Graphical Objects 21

4.4 Input Handling 21

5 Methods of Data Gathering **25**

5.1 Selection of Graphical Applications 25

5.2 Data Gathering Procedure 29

6 Results **30**

6.1 Slot Access Patterns 31

6.1.1 Data Gathered 31

6.1.2 Analysis of Slot Access Patters 37

6.2 Patterns of Constraint Evaluation 39

6.2.1 Data Gathered 39

6.2.2 Analysis of Constraint Evaluations 42

6.3 The Structure of Constraint Networks 44

6.3.1 Data Gathered 44

6.3.2 Analysis of Constraint Networks 48

6.4 Constraint Evaluation by Interactor Type 51

6.4.1 Data Gathered 52

6.4.2 Analysis of Interactor Operations 55

7 Conclusions and Future Work **56**

7.1 Conclusions 56

7.2 Future Work 58

Bibliography **59**

Appendix **64**

A.1 Data Gathering for *Checkers* 65

A.2 Data Gathering for *Constraint Animator* 66

A.3 Data Gathering for *Sillydraw* 67

A.4 Data Gathering for *VCR* 67

A.5 Data Gathering for *Space* 69

A.6 Data Gathering for *Mathnet* 70

A.7 Data Gathering for *Whiteboard* 71

Vita **72**

List of Figures

1.1 Geometric Relationships 3

2.1 Constraint Network 10

2.2 Width of Constraint Networks 12

4.1 Object Hierarchy 22

4.2 Interactor States. 23

5.1 The Applications 26

6.1 Slot Writes 33

6.2 Average Slot Writes 34

6.3 Slot Reads 35

6.4 Average Slot Reads 37

6.5 Constraint Evaluations 40

6.6 Read/Evaluate Ratio 41

6.7 Branch-in Factor 44

6.8 Branch-out Factor 45

6.9 Variable Depth 46

6.10 Maximum Level Width 47

6.11 Level Width 47

6.12 Interactor Data 52

Chapter 1

Introduction

Spreadsheets have greatly simplified the programming of applications in which the relationships between data items can be specified as an equation. In most spreadsheets, the potential workspace is divided up into a number of cells. Each cell can be assigned a value explicitly by the programmer or have its value computed by an equation. An equation uses values from other cells to compute the value of its associated cell. For example, a spreadsheet could define the following cells:

```
tax_rate = 0.07

subtotal = prices of the items added together

tax = subtotal * tax_rate

total = subtotal + tax
```

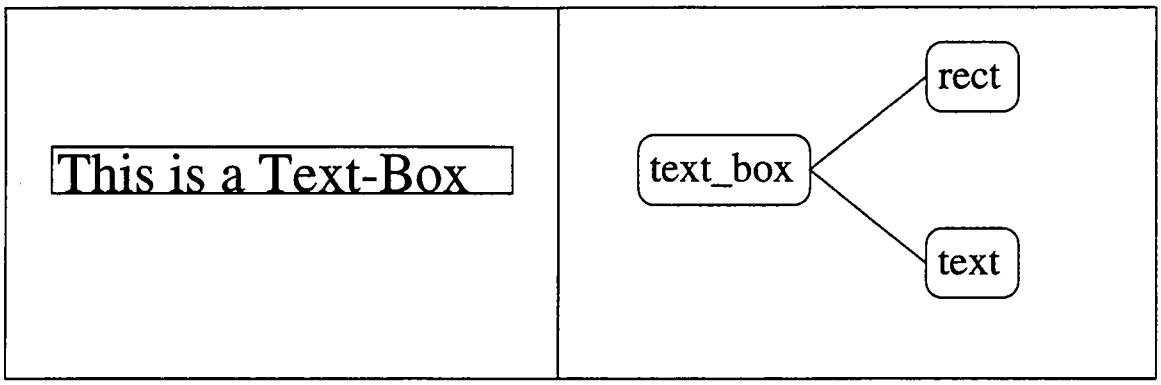
When a value used by an equation changes, the value of that equation's cell is recomputed. For

example, value of `subtotal` would be recomputed if any of the `prices` changed, `tax` would be recomputed if either the `subtotal` or the `tax_rate` changed, and `total` would be recomputed if either `subtotal` or `tax` changed. Thus, changing the `tax_rate` would cause the value of `tax` to be recomputed, which would then cause the value of `total` to be recomputed. Spreadsheets have simplified the task of maintaining relationships to such a degree that many spreadsheet users do not consider themselves programmers.

Spreadsheet Programming in Graphical Interfaces

The spreadsheet model of programming has been adapted for general purpose programming languages and in this context is called constraint programming. Constraint programming allows the programmer to specify a relationship, called a constraint, between data items; this constraint is automatically maintained by a constraint solver. Constraint programming has proven especially popular in the creation of graphical interfaces because graphical interfaces maintain many non-trivial relationships among data items that are amenable to being expressed by constraints.

Constraints are particularly appropriate for expressing two types of relationships commonly found in graphical interfaces: 1) geometric relationships, and 2) application data/graphical property consistency relationships. A geometric relationship relates to the position or sizes of objects on the screen. For instance, an object on the screen may appear as a rectangle surrounding text. What appears to the user as a single object is a composite object, composed of a rectangle and text, as shown in Figure 1.1. In order to keep the rectangle together with the text, the programmer could define the following constraints:



(a) (b)

Figure 1.1: Part (a) shows the object as it appears to a user of the program. Part (b) shows the composition of the same object as a programmer might view it.

```
rect.left = text.left

rect.top = text.top
```

Thus, as the text moves, so does the surrounding box, since the constraint solver automatically updates `rect.left` and `rect.top`.

A second use of constraints in graphical interfaces is to express relationships between the properties of an object in the interface and the application's data. For example, in a nuclear power plant, a constraint could be used to alter the color property of an indicator. The following constraint might apply:

```
indicator.color =   GREEN if (reactor.temp == normal)
                    YELLOW if (reactor.temp == hot)
                    RED if (reactor.temp == meltdown)
```

The variable `indicator.color` will automatically change to reflect the temperature of the reactor. Anecdotal comments from users of constraints suggest that constraints do simplify the maintenance of these relationships in graphical interfaces.

Like spreadsheets, the most common type of constraint solver in graphical interfaces uses the right side of an equation to calculate the value of the left side of the equation. Because the data always flows from the right side to the left side of the equation, this type of constraint solver is called a *one-way* constraint solver. In a one-way system, the constraint `right = width + left` would specify that the value of `right` depends on the value of both `width` and `left`. The constraint solver maintains a graph of data dependencies, so that, in our example, if either `width` or `left` change, the value of `right` is automatically updated.

The Research Problem

One-way constraints have been incorporated into several research graphical user interface toolkits and applications, including Amulet [14], Garnet [13], Eval/vite [8], Penguims [9], and Rendezvous [5].

However, while constraints have gained acceptance in research environments, they have not been widely used in commercial environments, despite their apparent advantage. This reluctance to use constraints can be attributed to two major concerns. First, commercial developers are concerned that constraint based applications will be difficult to debug. In particular, they are worried that the networks of constraints will be large and complicated, resulting in a labyrinth of dependencies. As

a result, programmers will have difficulty navigating these networks in order to locate erroneous constraints. Though this concern has not been borne out by the experience of researchers, there is no hard data which would allay this concern.

Another area of concern for commercial developers regarding the use of constraints has been the overall performance of constraint-based graphical interface tools. This concern is based on a number of factors. First, for the most part, constraints have been used in relatively small applications, and it is not known how constraints will scale in larger applications. Second, most of the graphical interface tools that make use of constraints have been written in languages such as Lisp and Smalltalk, which have lower performance than those used in industry. Third, constraints require significant amounts of additional storage for the dependency information. Forth, maintaining the data structures associated with the data dependency graphs reduces performance of the application. In our previous example of one-way constraints, `right = width + left`, if `width` or `left` changes, the value of `right` must be invalidated before the constraint solver will reevaluate the right side of the equation. Applications with data relationships that have been hand coded would not require this additional invalidation step. This problem is exacerbated by the fact that this overhead is often wasted, since the invalidated data may never actually be needed again, or a data value may be invalidated several times before it is needed again.

The Research Solution

To address the concerns of commercial developers, our study gathered empirical data on the data access patterns, constraint network topology, and usage patterns of constraints in applications. To

do this, we instrumented seven constraint based applications which represented several types of interfaces, including widget based applications (widgets are controls, such as buttons, scrollbars, and menus), games, visual programming environments, and visualization systems.

The most interesting conclusion from our study is that constraint networks are not, as commercial developers feared, a labyrinth of dependencies that are difficult or impossible to debug. Instead most constraint networks appear to be relatively short and narrow, as the experience of researchers has indicated.

A second interesting conclusion is that constraint programming should scale well to larger applications. In general, programmers tend to create constraint networks that are very modular. As applications grow, the number of constraints increases, but this increase comes in the form of added modules, so that the short, narrow, nature of the constraint networks is preserved. While there may be more modules in an application, input operations still interact with constraint networks that are the same size. As a result, it is unlikely that performance will degrade as applications grow.

The data collected in our study should also provide important guidance in directing future research that seeks 1) to optimize the performance of constraint satisfaction algorithms and improve the performance of graphical interface libraries, and 2) to develop more effective debugging strategies. Our data provides information about the frequency with which various constraints are reevaluated, and the manner in which constraints are utilized by operations, such as moving or resizing graphical objects or editing text.

In sum, the findings should (1) provide an impetus to the full scale use of one-way constraints in commercial applications, and (2) provide a repository of data for initiating such work.

The remainder of the thesis is organized as follows. Chapter 2 describes in more detail the types of data that were gathered in this study. Chapter 3 discusses some related optimization and debugging work. Chapter 4 gives provides a brief overview of Amulet, the graphical interface library used in our study. Chapter 5 describes the methodology employed in profiling the applications. Chapter 6 presents the profiling results and analyzes their implications. Finally, Chapter 7 presents our conclusions and some ideas for future work.

Chapter 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 Data Access Patterns

We tracked what variables were being read and written in an effort to understand what data was being used. Our study presents two measures of data access:

1. The aggregate number of times a variable in a grouping of objects with the same prototype was read or written.

2. The average number of times a particular variable in a grouping of objects with the same prototype object was read or written. A prototype object is an object from which other instance objects are created. Informally, a prototype object may be thought of as a type. Prototypes are discussed in greater detail in Chapter 4.

2.2 The Frequency of Constraint Evaluation

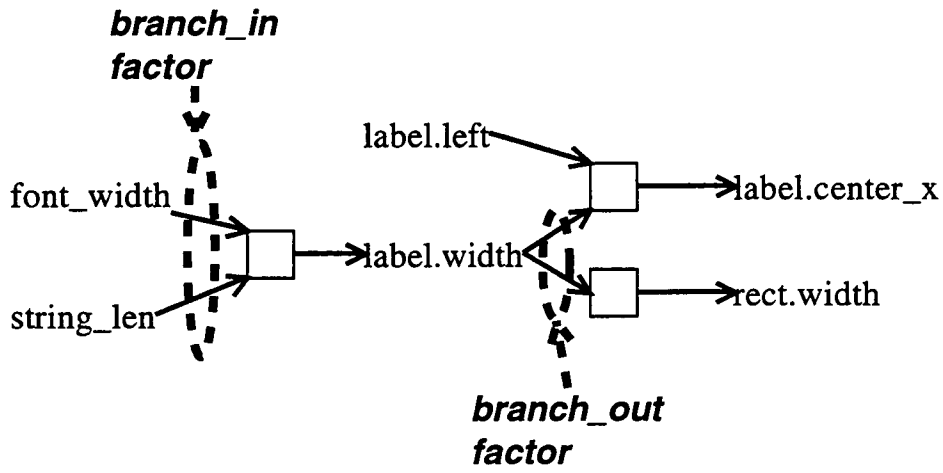
The frequency of constraint evaluation refers to how often a constraint is evaluated. Our study employed two measures of frequency of constraint evaluation:

1. The number of times a constraint is evaluated.

2. 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 Chapter 4) divides operations into a small number of categories, such as move/grow operations and text editing operations [14]. 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



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

Figure 2.1: 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.

used by each category of operation, and the average number of constraints that were used by each subpart of an operation.

2.4 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 2.1). 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. The complexity of the constraint network and the amount of time required to satisfy the constraint network depends on a number of factors: 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, 3) the *branch-out factor*, which is the number of edges leaving a variable node (Figure 2.1) and 4) the *network width*, which, informally, is the number of variables in a cross-section of the network (see Figures 2.1 and 2.2). 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. A variable with a branch-out factor of 0 is a *sink* node.

The *network width* can be more formally defined as follows. First, choose a variable v which is a sink node. Let N represent the sub-network of variables and constraints that v directly or indirectly depends on. This sub-network consists of all variables and constraints that have a directed path from themselves to v (see Figure 2.2). We would like to partition this network into cross-sections so we can measure the width of each cross-section. To partition the network into cross-sections, we use the notion of a *level*. We say that a variable w is at level i if there are i formulas on the shortest path from w to v. Each level represents a cross-section (see Figure 2.2). The width of a level, (i.e., the width of a cross-section), is the number of variables in that level.

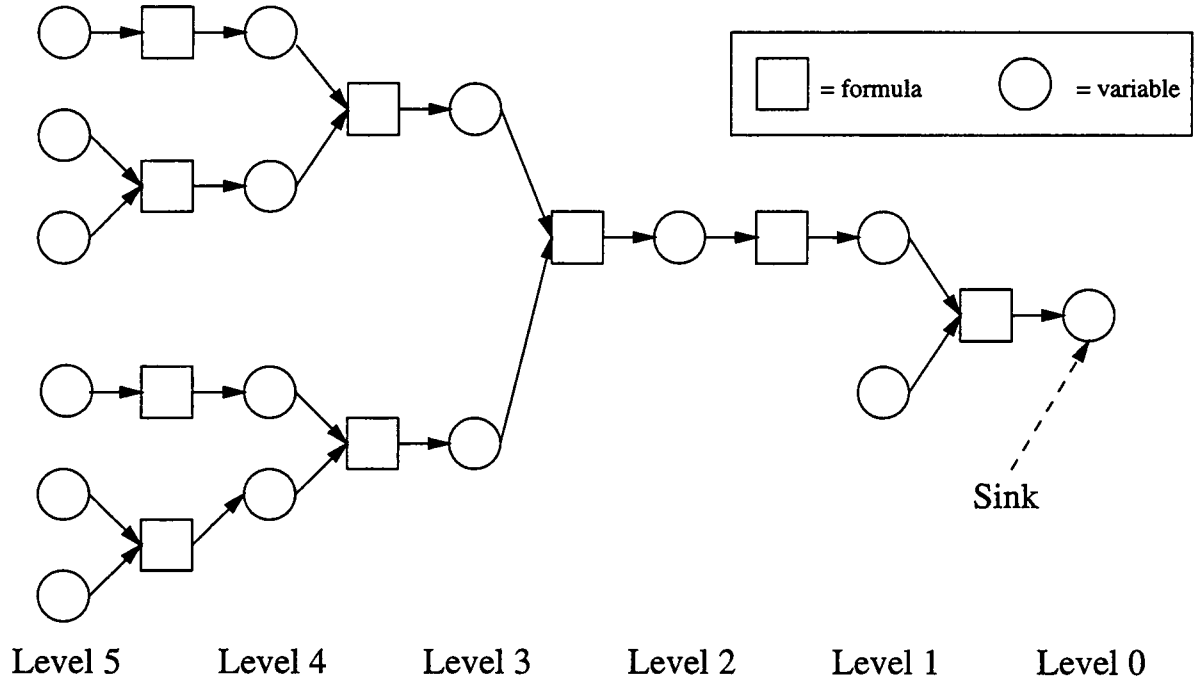


Figure 2.2: The width of constraints networks. At Level 2 the width is 1, at Level 5, the width is 6.

Chapter 3

Related Work

This thesis is the first work that we know of which presents empirical data on the composition of constraint networks and the overall use of constraints. There has, however, been work done on various optimization and debugging techniques for constraint networks, as well as research on improving data access in prototype-instance object systems such as the one included in the Amulet Graphics Toolkit (Chapter 4 describes prototype-instance systems). The remainder of this chapter gives a brief overview of some work related to the optimization of constraint networks, constraint debugging, and improved data access methods.

3.1 Debugging Constraint Networks

Commercial developers have repeatedly expressed concern about debugging applications with hundreds or thousands of constraints. This concern is understandable, as constraints are not supported by imperative language debugging tools. In addition, graph browsers, with a jumble of constraint objects on the screen, may be little or no help. However, as [16] points out, at one time imperative programming also suffered from a lack of debugging tools and methods. It took many years to develop the debugging resources for imperative languages that are currently available. Constraints are relatively new and methods and tools will need to be honed for debugging.

There have been several attempts to construct constraint debuggers which represent the constraint networks as graphs. The CVN debugger is an advanced debugger for complex constraint networks [16]. The system presents information with a graph browser that also allows developers to locate disjoint subgraphs, directed cycles or directed paths. Additionally, the debugger provides a query browser to manage and reuse the previously generated debugging information. The authors of the CVN debugger recognize that it is an open question whether the techniques they used are suitable for large constraint networks, as they were not able to study the debugger's effectiveness in large networks and they did not apply all the known techniques for graph browsing [16].

One suggestion made by [16] to improve constraint graph representation for large graphs is to use improved graph formatting tools. One tool that does make use of different graph formatting tools is described in [4]. This tool provided a browser that allows the user to navigate constraint networks

interactively. The user can then divide graphs into manageable parts by applying different graph layout algorithms to different sections of the graph [4].

3.2 Constraint Plans

Constraint plans are a performance optimization technique that recognizes that sets of constraints are often repeatedly executed. For example, a move operation may repeatedly execute the same set of constraints while repositioning a composite object on the screen. When a constraint solver executes a set of constraints, it must first sort them into topological order, and then execute them. If it repeatedly executes the same set of constraints, then it only has to do the topological ordering once. The ordered set of constraints can be saved in an appropriate data structure and then reexecuted when the need arises. The saved set of constraints is called a plan. Plans can be either computed at runtime or compile time.

Plans can be computed at runtime by recording what constraints are executed as the result of a particular input operation. The recorded constraints are automatically reevaluated as the operation continues. The plan is destroyed when the operation ends [10].

Using plans computed at runtime has not always resulted in better performance. It is necessary to understand some of the characteristics of plans to understand why this is the case. Plans have an initial *latency*, which the user may see as a slight delay, as the appropriate data structures for the plan are allocated and the plan's constraints are stored in the data structure. Additionally, some systems,

such as ThingLab [1], dynamically compile the plan by placing the code for each constraint into a straight-line procedure and then compiling the procedure (i.e., they in-line the constraint code). However, once the plan is created, it results in increased *feedback bandwidth*, or number of input events that can be completed in a given amount of time. The high initial latency caused by dynamic compiling in ThingLab was too great a burden on an application since the increased feedback bandwidth before the plan was invalidated rarely made up for the initial latency of plan compilation. Additionally, the high initial latency made the system's responsiveness seem erratic. Consequently, ThingLab II, ThingLab's successor, used interpreted plans, which reduced latency at the cost of some feedback bandwidth. The result were interfaces that had a more consistent "feel" [10].

ThingLab II also introduced the idea of pre-compiled plans, called modules, in an effort to avoid dynamic constraint scheduling and the function overhead associated with calling the code associated with each constraint [10]. The Module Compiler took a set of variables, a set of constraints, and a list of variables that needed to be visible outside the module. Using these inputs, the compiler produced optimized methods which the planner used to determine how to satisfy a module's constraints. Compilation of plans, since it is time consuming, was only done after the component to be compiled was well tested, was not likely to change and was to be frequently used [3].

3.3 Data Access Issues

Amulet shares many properties with the SELF language. SELF was originally implemented at the Xerox Parc laboratory and was later revised by researchers at Stanford University [2].

SELF was built around the concept of a prototype-instance object system (see Section 4.1 for more information on the prototype-instance paradigm). Objects in SELF use slots for data storage. These slot can be dynamically added to and removed from the object, are dynamically typed, and are indirectly accessed through the object by sending messages [2].

In the research done on SELF at Stanford, a great deal of focus was placed on optimizing data storage, with attention given to both reducing memory requirements and improving data access time. The researchers at Stanford were able to greatly increase performance based on these techniques with no modification to the prototype-instance, slot based paradigm that formed the basis of the language [2]. These results are significant in that they proved that languages based on the prototype-instance paradigm can benefit greatly from optimizations, with improvement by a factor of 5 possible over Smalltalk [17]. One of the goals of this thesis is to identify data access patterns in graphical applications that could suggest optimizations which are appropriate for graphical libraries implemented using a prototype-instance system.

Chapter 4

Amulet Architecture and Terminology

Amulet is a graphical user interface toolkit developed at Carnegie Mellon University and made available in the public domain[14]. Amulet, written in C++, is a descendent of the Lisp based Garnet toolkit [13] and incorporates many features from Garnet, as well as many lessons that were learned. Amulet currently runs under Unix (through X11), Microsoft Windows, and MacOS. Additional information about Amulet can be found at http://www.cs.cmu.edu/~amulet[11].

Amulet was designed to be used by students, researchers, and commercial developers. As a result, it offers a comprehensive set of features that were well suited to our needs. In addition to a fairly standard low-level graphical object drawing package and predefined widgets (widgets are controls such as scrollbars, menus and buttons), Amulet includes an object system, a constraint system, hierarchical composition of graphical objects, and an input handling mechanism [14].

4.1 The Amulet Object System

Amulet, while written in C++, differs greatly from most C++ code. C++ is based on the concept of a *class-instance* object system. In class-instance object systems, a class is the abstract definition of the object. It contains the code (methods) and data layout (variables) but a class is never actually executed. Instead, an instance of an object defined by a class can be created [15].

For Amulet's purposes, a class-instance system is too static. Once the programmer has defined a class, the objects based on that class will never change. In other words, the methods they execute are fixed and the set of instance variables can be neither increased nor decreased. This is not well suited to the creation of dynamic graphical objects.

The designers of Amulet took a different approach to the object-oriented paradigm by using a *prototype-instance* object system. In Amulet, there is no object abstraction. Instead, each instance can be used to create other instances using an object's `Create` method. Thus, every object is considered a prototype from which the program can create additional objects [11].

Additionally, Amulet does away with the traditional concept of instance variables and methods. Instead, Amulet has implemented a concept of *slots* for data storage. Slots can be dynamically added to or removed from an object at runtime. They can be explicitly written to by passing an unique integer key and a value to store, called a slot key, to an object's *set* method, or read by passing a slot key to an object's *get* method, which returns the value of the slot with that key in the object (the programmer actually refers to slots using symbolic names; these names are compiled

into unique integer keys). Slots are dynamically typed, so arbitrary values may be placed in a slot.

The slot mechanism also provides methods to control data inheritance. Slots can be inherited (changes to a slot in a prototype object will be reflected in the instances of that object unless the object has overridden the slot), local (the slot is not passed from prototype to instance), copied (the instance object gets a copy of the slot and changes in the value of the prototype's slot will not affect the instance), and static (all instances of a object will share the same value for a slot).

Lastly, because pointers to functions can be stored in slots, Amulet allows the programmer to use the slot mechanism to dynamically add and change methods in an object. To add or change an object's methods, the programmer simply stores a pointer to a C++ function in a slot. To invoke the method, the programmer retrieves the value of the slot, then casts that value to the appropriate function type and finally calls the function to execute the method [11].

4.2 The Constraint System

Another feature of Amulet is its support for one-way constraints. A programmer can write a function containing arbitrary C++ code. The programmer can then bind the address of the function to a formula object and attach an instance of this formula to any slot 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 slots in the instance will inherit formulas from their corresponding slots in the prototype. Amulet provides a second

type of constraint, called a *web* constraint, which functions much like a multi-way constraint. Web constraints were excluded from our study, since 1) there are relatively few web constraints, 2) they are handled through a different mechanism, and 3) the addition of web constraints would have not contributed to our goal of understanding one-way constraint systems [14].

4.3 Hierarchical Composition of Graphical Objects

Amulet also provides support for hierarchies of graphical objects, or *composite objects*. In Figure 4.1, the scroll bar is composed of several graphical parts. The parts would share the same coordinate system and could reference each other through their parent, the composite object. As with other objects in Amulet's object system, calling a prototype composite object's `Create` method results in a new instance of the object, including new instances of the prototype composite object's parts.

4.4 Input Handling

Lastly, Amulet provides an input handling mechanism. In more traditional graphics libraries, events, usually consisting of keystrokes and mouse data, are gathered by the program. The programmer provides a mechanism to handle these events depending on the context of the event. Amulet provides *interactors* to simplify the handling of these events. Five generic interactor operations are provided, representing abstract behaviors exhibited by programs. These generic operations serve as prototype behaviors, from which new behaviors can be created, customized, and attached to graphical objects.

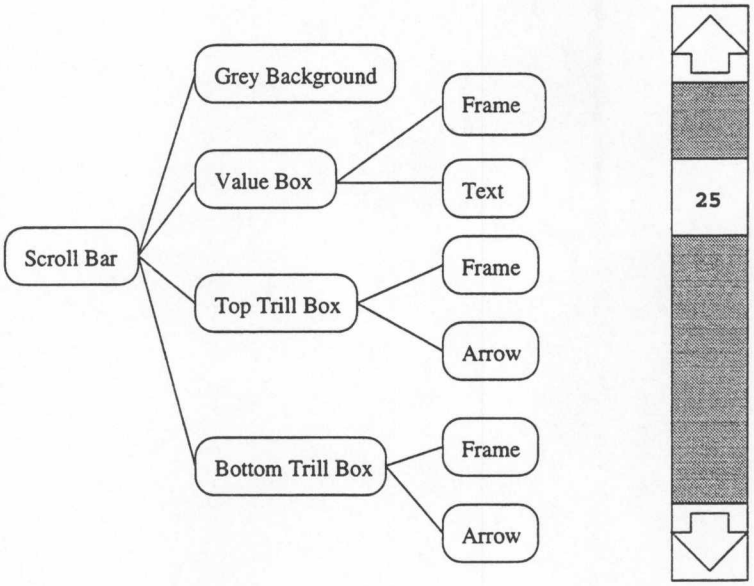


Figure 4.1: One possible hierarchy for objects that make up a scroll bar.

Amulet's event handling mechanism is able to decide what interactor the event is intended for, activate the interactor, and pass the event to the interactor. Multiple interactors can be active at the same time [14].

The interactors operate in three stages (see Figure 4.2). Initially, interactors are inactive, waiting, until they receive a start event. Once an interactor receives a start event, the interactor executes its start method, and enters its running state. In the running state, the interactor receives running events, which trigger a running action. Typically the running action generates interim feedback. Eventually the interactor receives a stop event, which triggers a stop action, and deactivates the interactor, and returns the interactor to its start state. Events outside the interactor's running area are not processed, but do cause the interactor to temporarily suspend its operation by moving the interactor to an "outside" state [14].

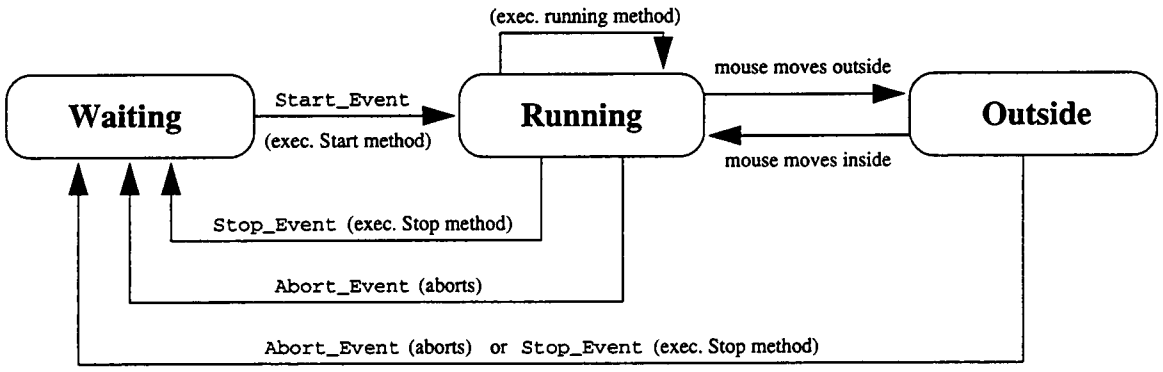


Figure 4.2: Interactor States.

The five generic interactor operations are:

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

`OneShotInteractor` – The one shot interactor is used when an immediate response is needed. Two examples of this would be a keystroke or a mouse event over a particular object. It is similar to a choice interactor, but does not provide interim feedback.

`Move/GrowInteractor` – The move/grow interactor is useful when a graphical object needs to be resized or moved on the screen.

`NewPointsInteractor` – The new points interactor is used to indicate points on the screen. This is useful in defining new objects on the screen with the mouse.

`TextEditInteractor` – The text edit interactor provides support for editing the text string of text objects.

Little additional work is required to implement most interactions with the program using these generic operations. The programmer has great flexibility in changing the behavior of these generic interactors by modifying the values of parameter slots in the interactor objects or by replacing the default start, running, or stop methods [14].

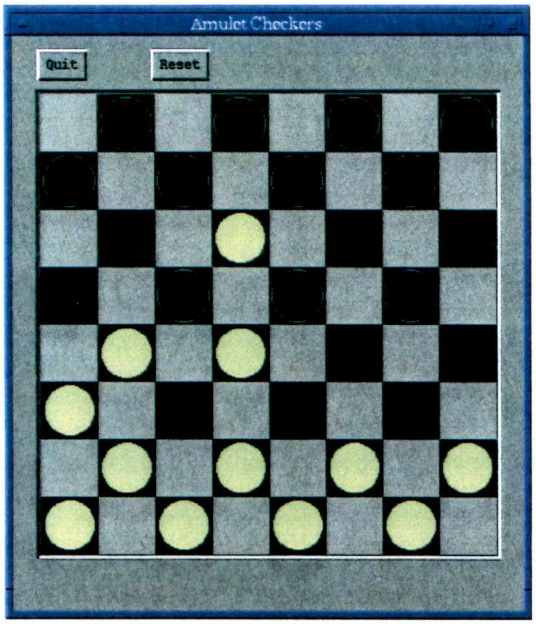
Chapter 5

Methods of Data Gathering

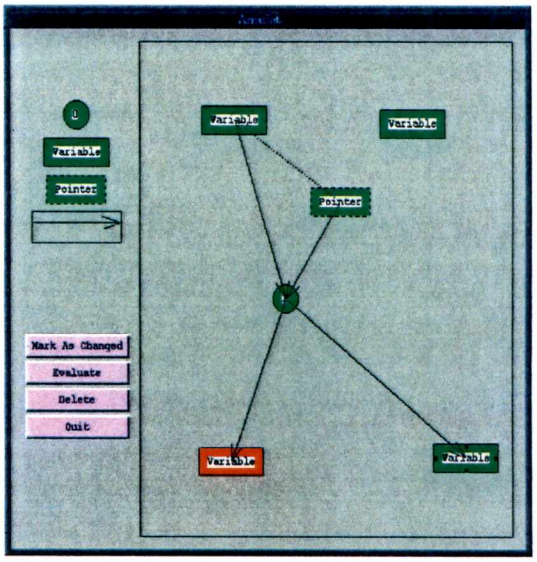
5.1 Selection of Graphical Applications

The selection of the graphical applications was governed by criteria of diversity, size, and the absence of programming bias. Pictures of the application are shown in Figure 5.1.

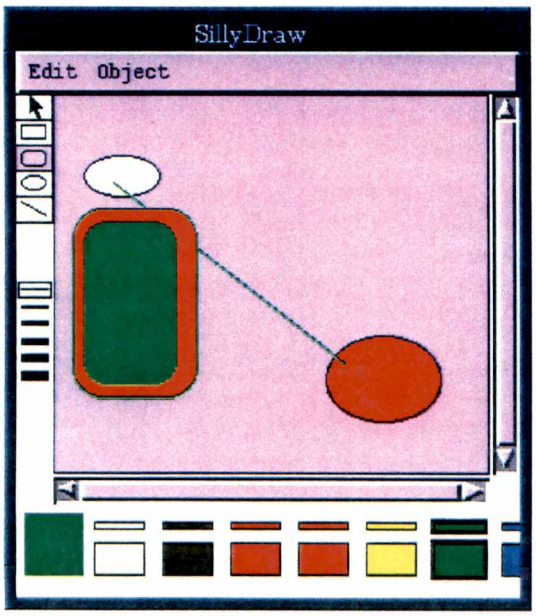
Diversity of application types was the first criteria used to select applications. Prior to this study it was not known how constraint usage might vary in differing types of applications. In order to avoid the bias of a particular type of application it was necessary to locate programs with different characteristics. The programs selected were widget based applications, games, visual programming



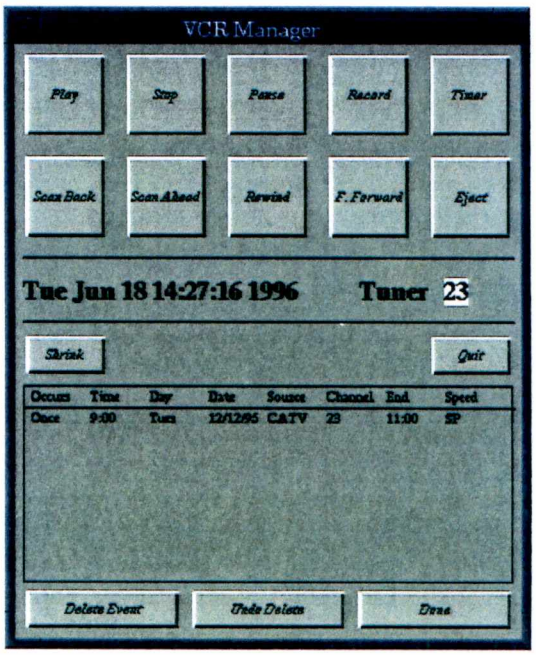
Checkers (One and Two Display).



Csdemo.

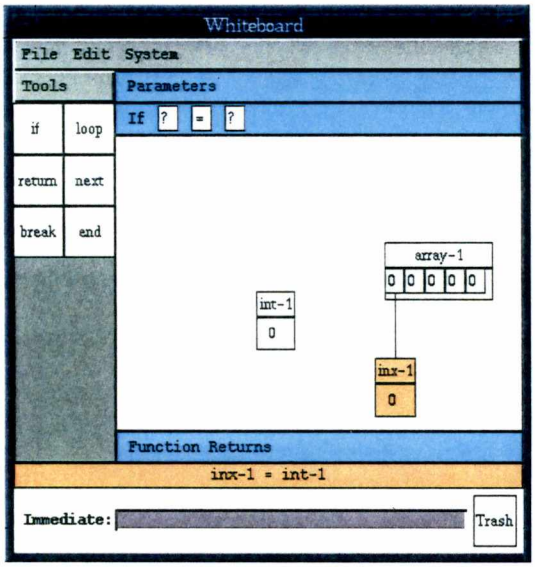


SillyDraw.

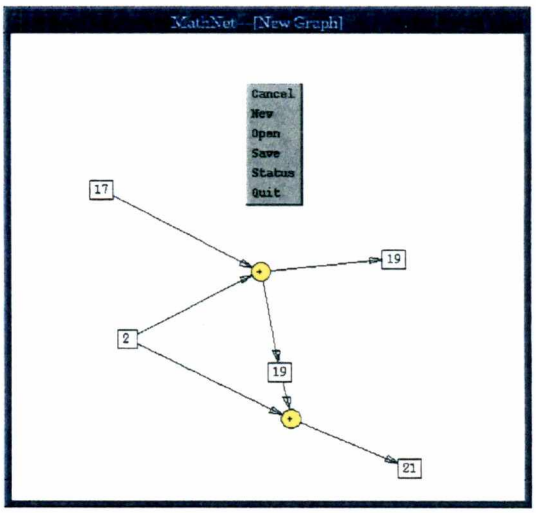


VCR Interface.

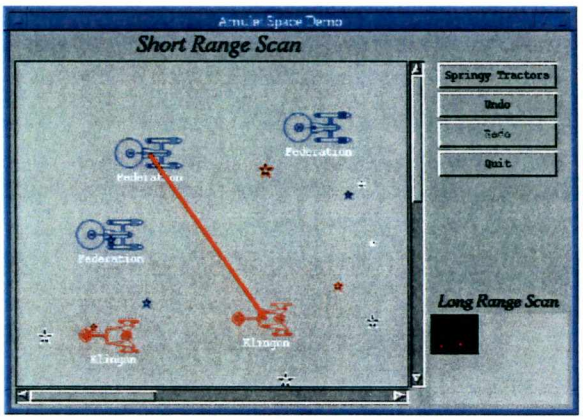
Figure 5.1: Screen snapshots of the graphical applications selected for the study.



Whiteboard.



Mathnet.



Space.

Figure 5.1(continued):Screen snapshots of the graphical applications selected for the study.

Table 5.1: Summary of the graphical applications selected for the study.

Name	*Description*	*Programmer Experience*	*Number of Constraints*
Checkers	Game of checkers (1 or 2 display).	Amulet developer	3571
Csdemo	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

environments, and visualization systems. Table 5.1 details the selection criteria, including application type, for our seven graphical applications.

Second, the programs needed to be from multiple sources in an effort to get data that was unbiased by a particular programming technique. The techniques the programmer used could be influenced by the experience level, comprehension of constraints, and the accepted coding practices in his or her environment. Therefore the code that was profiled was obtained from the University of Tennessee, Carnegie Mellon University, and Southern College and represents the work of the Amulet developers, graduate students with previous exposure to structured graphics libraries, and undergraduate students in an an introductory graphical user interface class.

Lastly, the size of the application was important. All the applications we studied used between 2,000

and 4,000 constraints. We did not try to include applications with small numbers of constraints since 1) the concerns of commercial developers deal with large, not small, constraint systems, and 2) optimizations potentially have a significant effect on large constraint systems, but a much smaller effect on small constraint systems.

5.2 Data Gathering Procedure

It was necessary to establish the conditions for the actual gathering of profiler data. The first question was how the programs should actually be run — whether to use scripted tests or to allow the user to "play" with the program. The second major issue was who would actually be doing the data generation — the actual developer, a random user, or myself.

It was decided to evaluate each program using a script of actions the user would perform. This procedure had some major advantages. By writing scripts, it was possible to make sure that all possible interactions with the program were actually performed, hopefully in a realistic way. Also, it made it possible to recreate the data and gave a reference that would allow explanation of any unusual results. Random user interaction would provide none of these advantages.

Because there were scripted actions for each program, I executed the scripts. There was no actual advantage to having the developer or a random user gather data when it was scripted. Because I gathered the data myself, this technique gave a better idea of everything that happened during execution, including adherence to the script, and made it easier to investigate and explain results.

Chapter 6

Results

This chapter presents the data gathered by running each of the eight applications. The data is categorized into four categories:

- Slot Access Patterns
- Patterns of Constraint Evaluation
- The Structure of Constraint Networks
- Constraint Evaluations by Interactor Type

For each category, we discuss how the data was gathered and tabulated, then we present the data, and finally, we analyze the data's implications for efficiency, storage, and debugging of code.

6.1 Slot Access Patterns

Amulet uses an array to store slot-value pairs, rather than instead of using C++ instance variables. To track data accesses for C++ instance variables, it would have been necessary to modify a C++ compiler. Because Amulet uses an array, this was not necessary. Instead, all that was required to track data access in Amulet was some comparatively trivial modification to the Amulet library code.

6.1.1 Data Gathered

The data that was gathered was aggregated at the prototype level. In other words, the data for each object was combined with other objects that had the same prototype. For example, if "rect_01", "rect_01_01", and "rect_03" had the same prototype, then the data for these objects was aggregated together. The profiler decided what objects were being used as prototypes by using Amulet's object naming convention. When a name was not explicitly assigned to a newly created object, Amulet concatenates a "_" and a number to the prototype object's name. When determining what group an object was part of, the profiler ignored the ending Amulet added. For instance, data from, "Am_Object", "Am_Object_111" and "Am_Object_111_193" would be stored together as a grouping of objects. Thus, by assigning a name to an object, the programmer determined the grouping of object data. The number of instances in each grouping of objects was recorded.

The reason for aggregating data rather than reporting it for individual objects is that, for data access optimizations, the decision to optimize must be made at a prototype level rather than the instance

level. In other words, an optimization will apply to all instances of a prototype or to none of them.

Explicit Slot Writes

We tracked the number of times each Amulet slot was explicitly written. Changes to the value of a slot as a result of a constraint evaluation were not included. We made this decision because changes due to constraint evaluations went through a very different mechanism than calls to a slot's write method. Changes due to constraint evaluations are investigated in Section 6.2. In addition, while there are techniques that allow writes to constrained slots, the use of these techniques is very rare compared to the default behavior, in which the constraint is written over.

In this section, the data presented is the aggregate number of writes for a slot in a grouping of objects. As seen in Figure 6.1, slots in the Amulet applications tested typically were not being written to that often. Most slots were being written less that 10 times in a typical application. In seven of the eight data sets, only about 5% of the slots were being written more frequently than 10 times.

The *VCR Interface* had the least number of slot writes and defies the pattern set by the other applications. The 95th percentile was at 6 explicit slot writes during its execution, as opposed to tens or hundreds of slot writes in other applications. This pattern was caused by the nature of the interface. It was constructed around predefined widgets. There was little interaction with the user and the view the interface presented changed little during the execution. Consequently, there were very few slot writes.

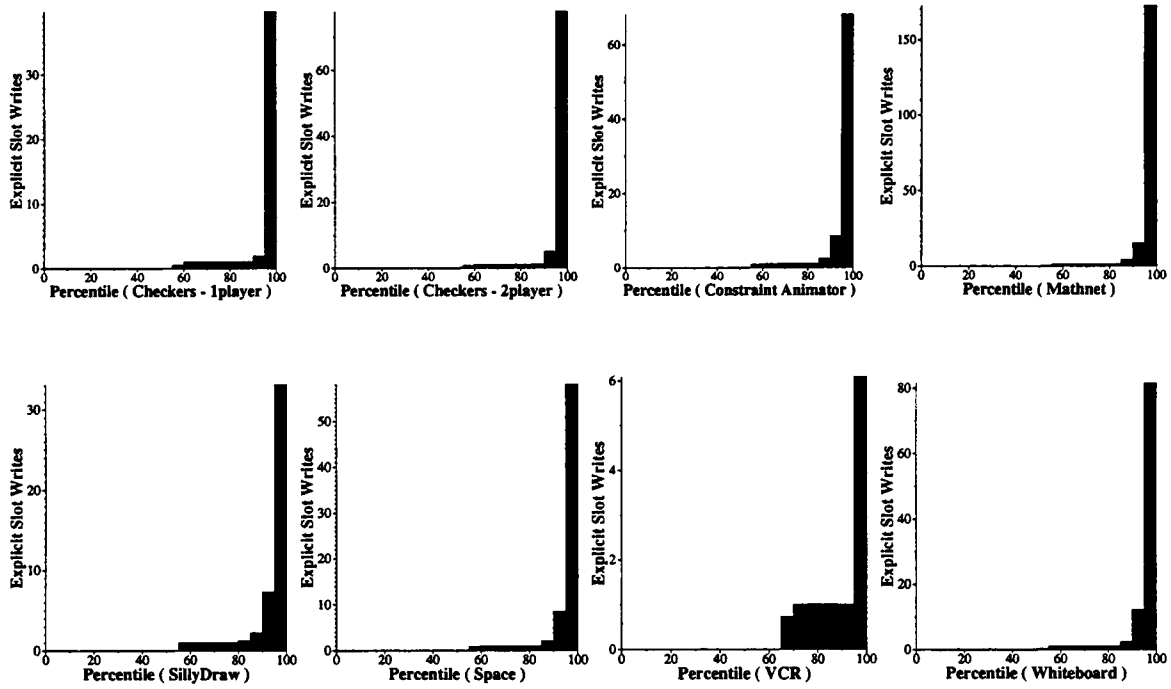


Figure 6.1: Number of times each slot was written.

Average Explicit Slot Writes

We also computed the average number of slot writes, using the aggregate number of writes for a slot in a grouping of objects and the number of instances of objects in that group. The average number of slot writes provided insight about whether the ranking of certain slots in the aggregate slot write count was being biased by the total number of objects in a grouping, as opposed to the degree to which a particular slot was accessed.

This data is presented in Figure 6.2 and indicate that most slots did not have many explicit slot writes. As in the aggregate case, only 5% had a significant number of slot writes. Once again, the *VCR Interface* did not really follow this trend. Also, other applications with more dynamic behavior

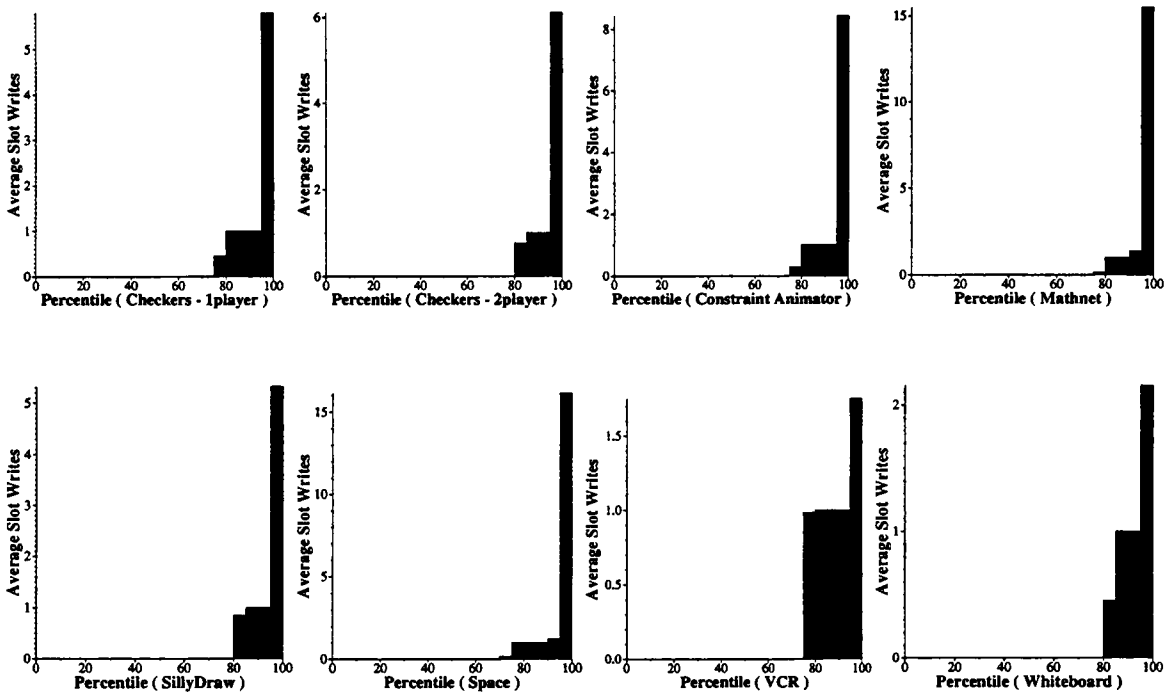


Figure 6.2: Average number of slot writes.

had lower averages than expected. For instance, the *Whiteboard* environment also had an average of less than 2 explicit slot writes.

Overall, Figure 6.2 indicates the slot write pattern observed with aggregate slot writes was representative of the actual behavior. While the averages were, in a few cases, lower than expected, the same slots with high aggregate accesses were also the slots with high average accesses. For example, the coordinate slots that determined the position of an object were still the most frequently written. This indicates that slots were not being misranked in the aggregate ranking due to fewer object instances.

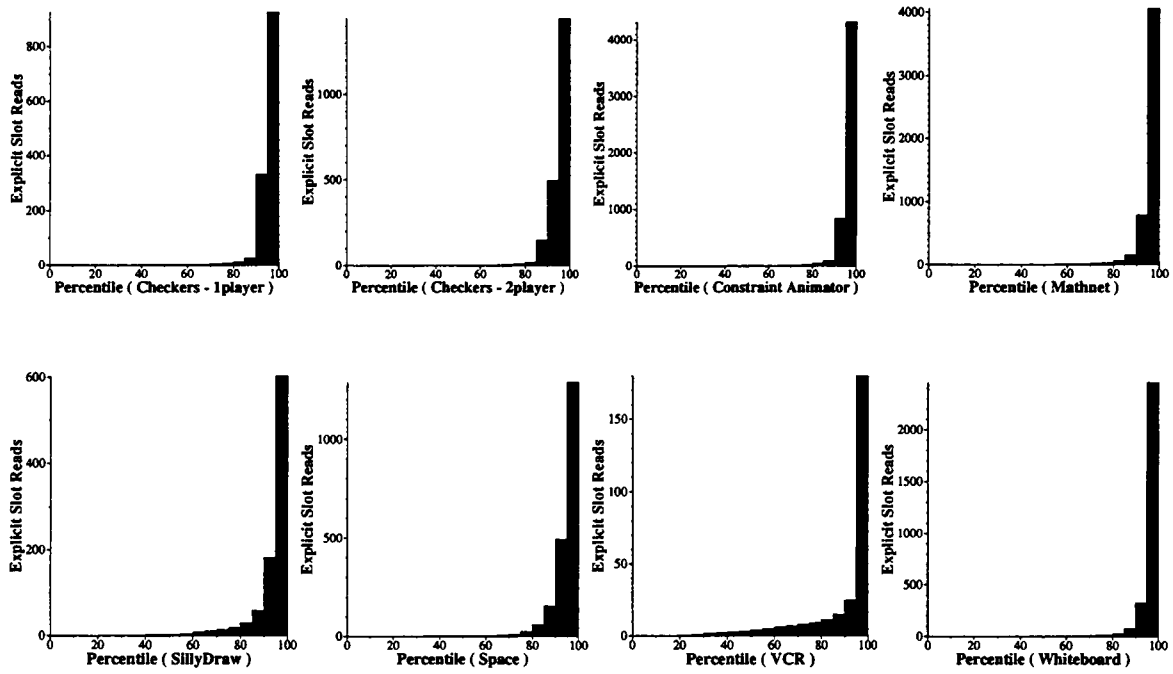


Figure 6.3: Number of times each slot was read.

Slot Read Access

There are two ways to read the value of a slot in Amulet; either a formula can read a slot's value as the formula evaluates itself, or an application can directly read the slots value. For optimization purposes, it is unimportant whether a read occurred as the result of a formula request or an application request. Consequently, the two types of reads were aggregated.

The aggregate slot read access (Figure 6.3) in Amulet had a similar pattern to slot write access. Most slots are infrequently read, with the number of reads in the tens of slot accesses per execution. However, slots in the 90th percentile are being read hundreds of times, versus tens of times. By the 95th percentile, slots are being read several hundred or even thousands of times.

Once again, the *VCR Interface* tended to defy this trend. The 95th percentile for the interface was at 150 reads, while the 95th percentile for the next closest application was around 600 reads per slot. The nature of the *VCR Interface* explains these results. The interface was really just a static collection of windows created with predefined widgets. The only real interactions the user had with the application was clicking on buttons or typing in entries.

At the other extreme, *Mathnet* and *Csdemo* had over 4000 reads per slot in the 95th percentile. The objects in the editor area of these applications were composite objects and were often operated on by interactors. The internal parts and appearance of the composite objects, as well as many arrows linking the objects, were constrained to the coordinates and state of the composite object. Consequently, a change in coordinates or state (usually triggered by a `Move/Grow` or `Choice` interactor) would result in dozens of reads by constraints that depended on the modified slots.

Between these extremes, the number of slot reads depended on the application's use of structured graphics and constraints. *Whiteboard* had composite objects but didn't have external objects (e.g., arrows) constrained to them as *Mathnet* and *Csdemo* did. *Space* had fairly simple composite objects that were occasionally connected with constrained objects (i.e., connected via lines). *Checkers* and *Sillydraw* used simple objects with few constraints.

Average Slot Read Access

The average slot read access for each slot in a group of objects was computed by taking the total read accesses for a slot in a group of objects and dividing that by the number of instances of objects

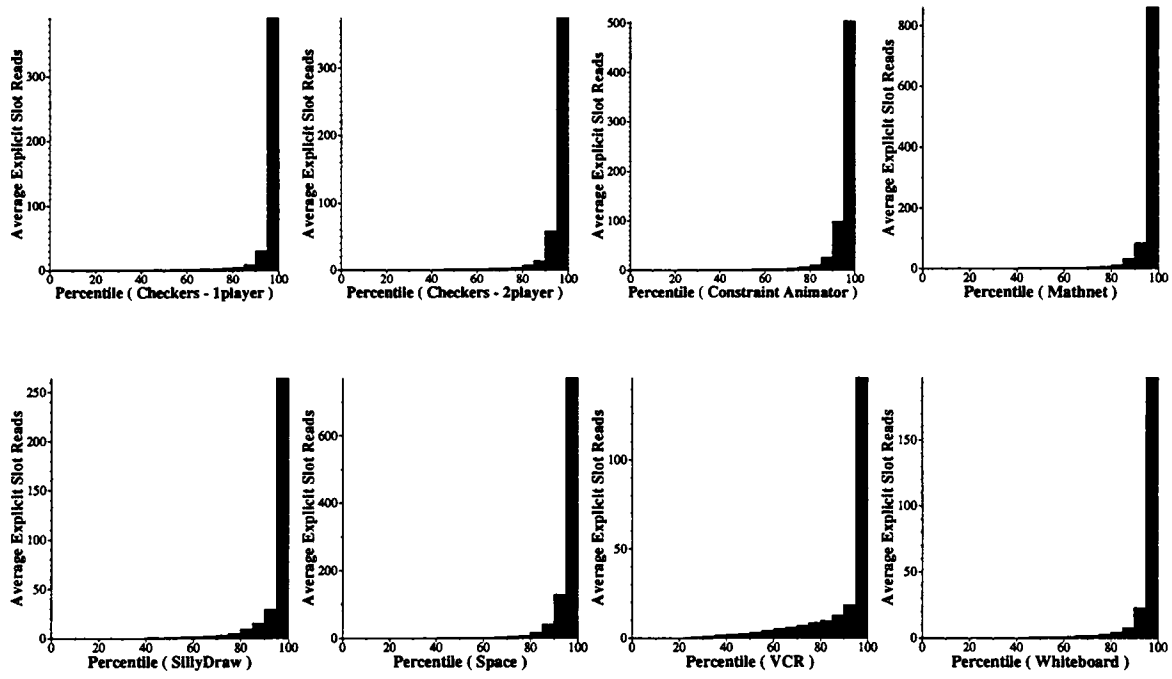


Figure 6.4: Average number of slot reads.

in that grouping.

The average slot read access had a similar pattern to the aggregate slot read access pattern (see Figure 6.4). Once again, coordinate slots were heavily read. Most other slots in each instance were read infrequently.

6.1.2 Analysis of Slot Access Patters

The slot access patterns of Amulet applications give us insight into possible performance enhancements to improve overall performance of applications.

Amulet's slot access method has a significant impact on the performance of applications written with

Amulet. Instance variables in C++ are laid out in static, predictable structures, so that references to them can be optimized by the compiler. Amulet determines the location of a slot's data at runtime by performing a linear search on an array of slots that belongs to an object If there is a match, it returns the slot. Not only is this access method inefficient, but the C++ compiler can do nothing to optimize slot access.

Caching heavily accessed slots would be an excellent start in improving performance. As seen in Figures 6.1 to 6.4 Amulet seems to be reading and writing the same slots often, and not accessing other slots at all; consequently, a cache of the most frequently accessed slots would help speed slot resolution greatly.

It would also make sense to automatically cache coordinate slots, especially in composite objects, which are reading and writing these slots a great deal due to the use of the `Move/Grow` interactor.

It could also be a worthwhile strategy to dynamically reorder the slot data so that slot references might benefit from the linear search strategy Amulet uses. By putting the most frequently accessed slots first in the ordering, the linear search would be far more efficient with no addition storage.

In general, a more efficient slot searching mechanism would be worthwhile. Minimally, Amulet could maintain a mapping between slot keys and data locations, ordered by slot key value. This would at least allow a binary search to locate a slot, as opposed to the current linear search. Since the current search method involves dereferencing a pointer on every iteration of the search, a slow operation, especially when the desired data is on a memory page which has been stored on disk,

this could result in a significant performance improvement.

Lastly, the data raises the possibility that widget based applications access slots far less frequently than dynamic applications with custom graphics. Given that dynamic applications tend to move objects around, while widget based applications do not, this finding makes sense. Additionally, this conclusion is supported by the data, which shows that coordinate slots tend to be the most heavily accessed slots in an application. Because we tested only one widget-based application, we can not definitively conclude that all widget-based applications will exhibit fewer data accesses relative to dynamic applications. However, it does suggest that the access patterns in widget-based applications may differ from access patterns in dynamic applications, which could affect the kinds of optimizations that have the greatest effect on each type of application.

6.2 Patterns of Constraint Evaluation

6.2.1 Data Gathered

Constraint Evaluations

In Amulet, constraints are attached to slots. When an instance of an object is created, instances of the constraints that are attached to the object's slots are created and placed in the new instance object. In our study, we tracked each instance of a constraint and made no effort to aggregate data (unlike slot data). For constraints, it makes sense to track individual instances, because many

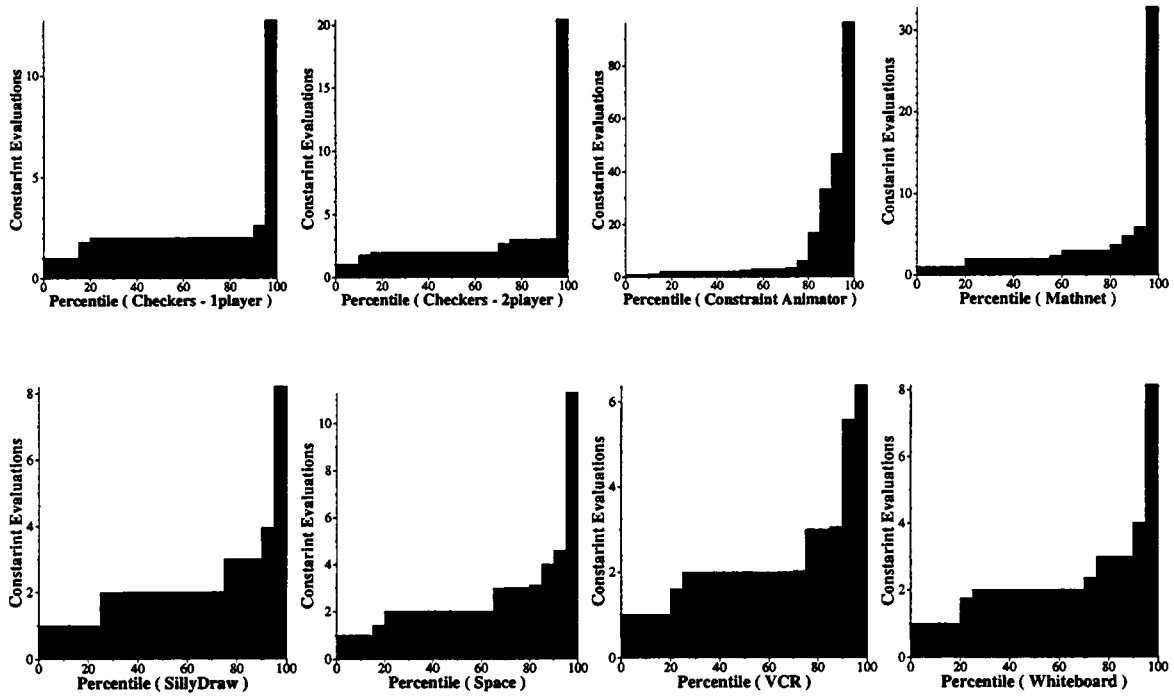


Figure 6.5: Number of times each constraint was evaluated.

constraint optimizations, such as constant propagation (discussed in Section 6.2.2), can be applied to individual constraints. For each instance of a constraint, we stored the number of times it was evaluated. This data gives us an idea of overall use of each constraint in the application.

Figure 6.5 shows that individual constraints were not evaluated that often in our applications. Around 60% to 80% of the constraints were evaluated five times or less. Around 5% of the constraints were evaluated heavily.

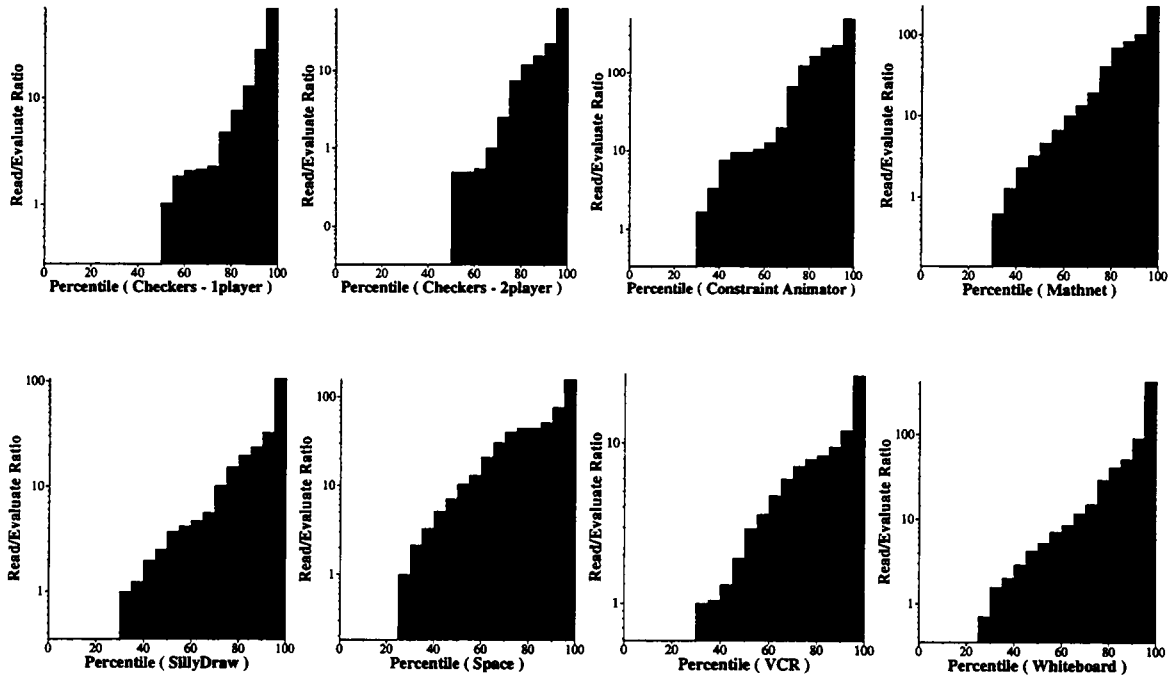


Figure 6.6: Number of times a variable was read between constraint evaluations.

Read/Evaluate Ratio

The aggregate number of reads for each slot in a group of related objects (i.e., objects with the same prototype) was tabulated, as was the aggregate number of constraint evaluations for that slot. From this, it was possible to calculate a ratio between the number of reads and the number of constraint evaluations. This led us to conclusions about how many times the result of a single constraint evaluation is being reused.

Figure 6.6 shows the ratios for our applications. Overall, a significant number of constrained slots are **never** read. These are mostly slots that are defined by Amulet in prototype objects that are never used. The majority of constrained slots which do get accessed are read several times between each

constraint evaluation. The top 5% of the slots had read/evaluate ratios in the tens or hundreds of reads per evaluation.

6.2.2 Analysis of Constraint Evaluations

The data in Figures 6.5 to 6.6 suggests several possible optimizations in constraint storage and evaluation.

In Amulet, all constraints contain 1)a cached data value, 2)an indicator of whether a constraint is up to date, 3)a list of dependencies, and 4)a reference to a formula function. All of this information is stored in order to make it rapidly accessible. The data in Figures 6.5 and 6.6 shows that many constraints are being evaluated infrequently but their cached values are being read often. As a result, efficient access to the cached value and the up to date indicator is worthwhile. However, the storage for the reference to a formula function and dependencies can be optimized to save space at a cost to efficiency. For example, the dependencies could be stored in a dynamic array which might need to be resized at runtime and the reference to a formula function could be stored in a prototype constraint, instead of in each instance. Because many of these formulas are only occasionally being reevaluated, the loss of efficiency will not be significant in terms of the overall performance of the application.

The significant number of slots in Figure 6.6 that were never read after evaluation indicate that it is not useful to automatically evaluate these constraints. By default, Amulet uses eager evaluation of constraints, where a constraint is reevaluated as soon as possible when it is out of date. Figure 6.6

supports the contention that eager evaluation degrades performance. However, Amulet allows the user to change this to lazy evaluation on a slot by slot basis, which evaluates an attached constraint only when a read is performed on the slot. Because most of the unneeded evaluations occurred in prototype objects that were never displayed, marking the constraints that are attached to unread slots as lazy would save a great deal of unneeded computation at initialization time.

Our data also suggests that constant propagation can be an effective optimization. Constant propagation eliminates constraints with constant arguments and replaces them with the value they compute [10, 12]. The data in Figure 6.5 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.

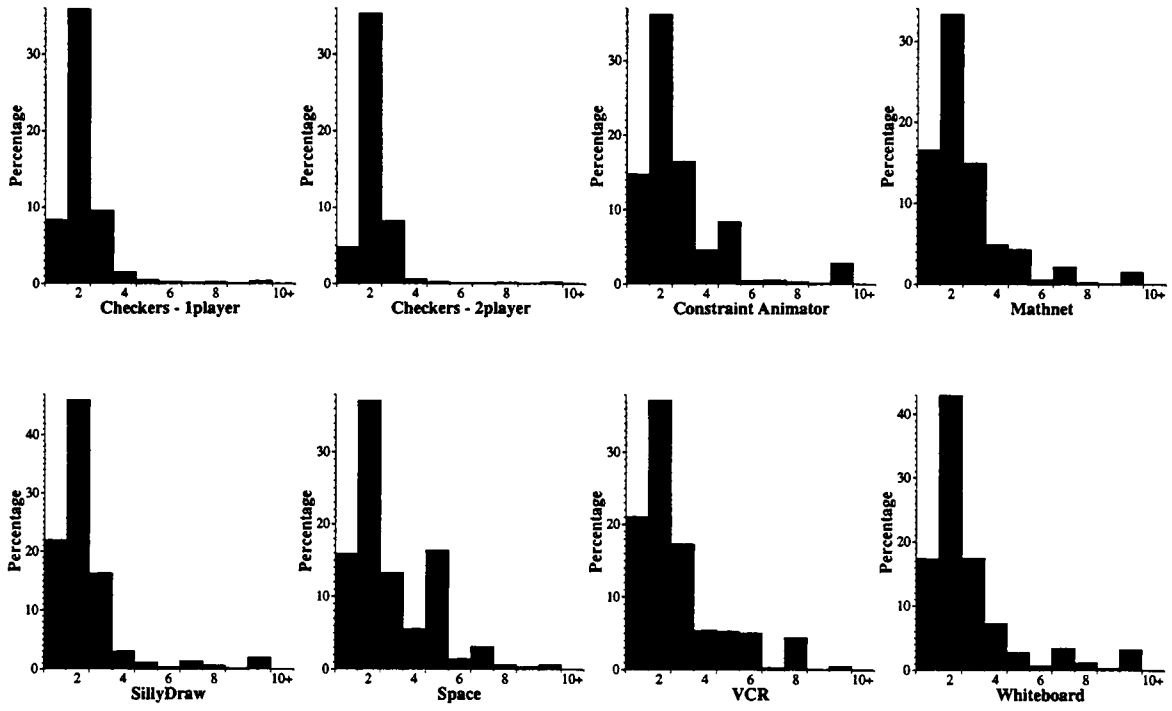


Figure 6.7: Branch-in factors for constraints.

6.3 The Structure of Constraint Networks

6.3.1 Data Gathered

Branch-in and Branch-out Factors

The branch-in and branch-out factor for each constraint was monitored and is recorded in Figures 6.7 and 6.8.

The data in Figure 6.7 and 6.8 indicate that the branch-in and branch-out factors in an application is independent of the number of constraints in an application and have the same frequency distribution,

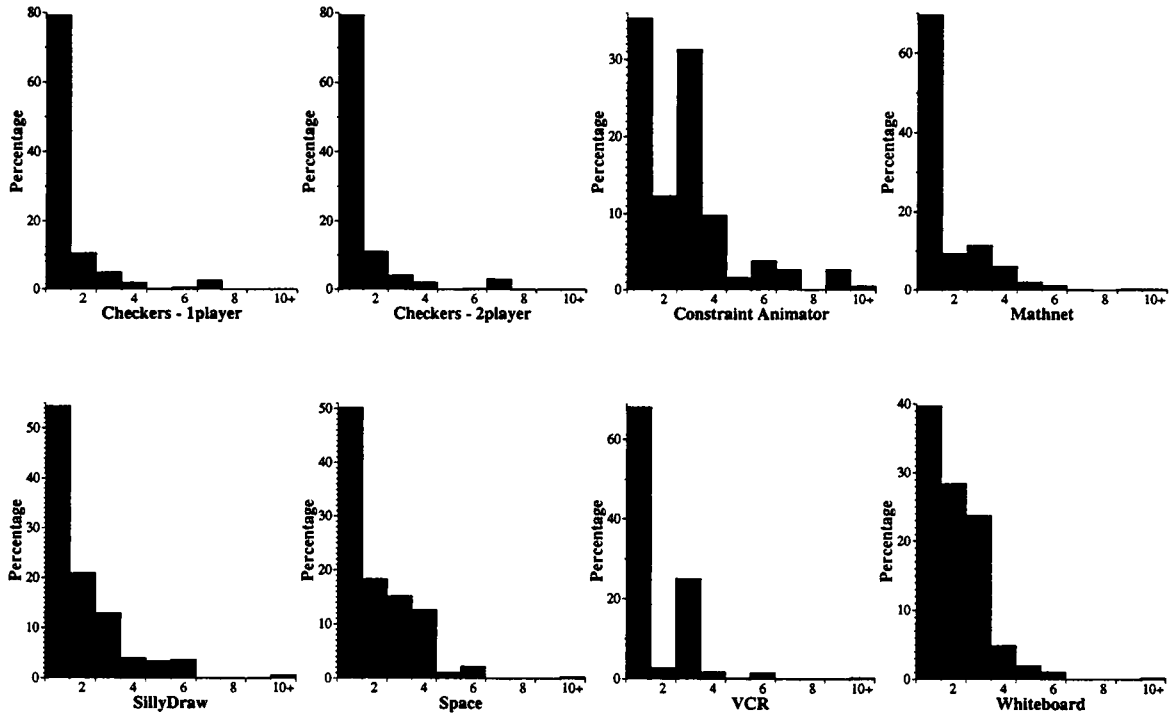


Figure 6.8: Branch-out factors for constraints.

regardless of the type of the application.

Depth of Variables in Constraint Networks

Figure 6.9 shows the variable depth in our eight data sets. The variable depth is the number of formulas on the longest path from a variable to a sink node.

The figure shows that constraint networks tend to be short. In the data we examined, depth tended to be six or less. No application had variable depth greater than 12. The short networks were caused by the modular nature of constraint networks. When new constraints were added, the additions tended to be "modules" which did not expand the existing constraint networks.

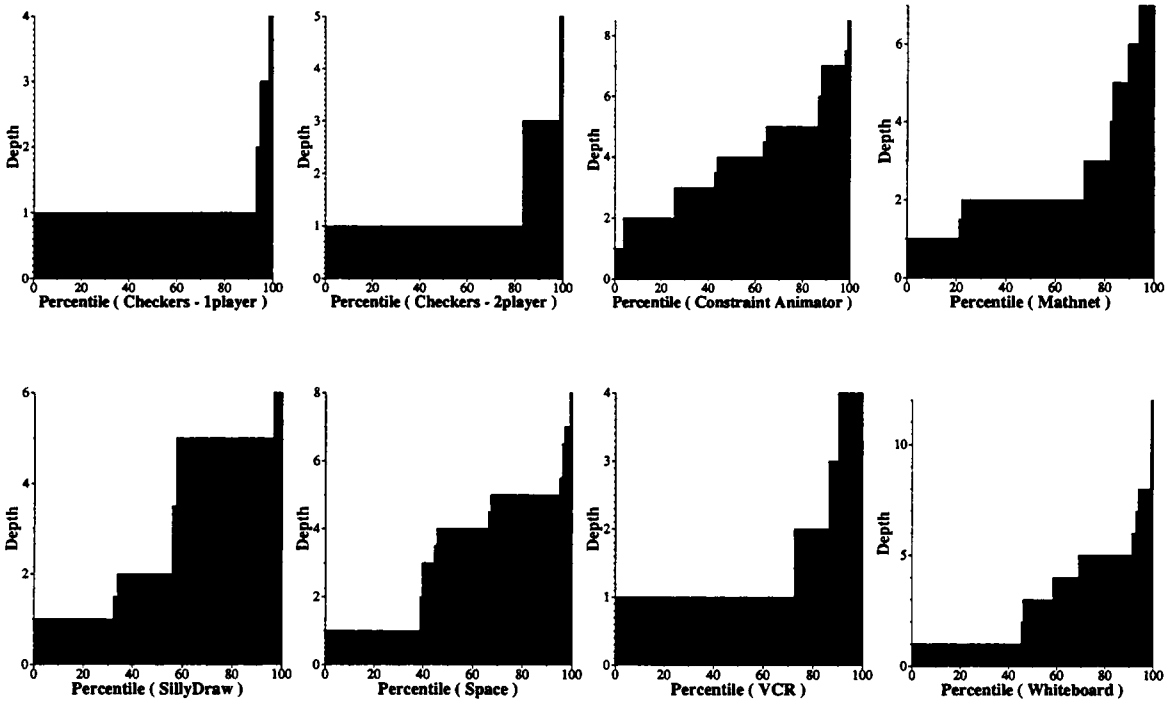


Figure 6.9: Depth of variables in constraint networks.

Network Width

Figures 6.10 and 6.11 show the maximum network width and the width of individual levels within networks for each application. Rather than reporting a single maximum network width for the entire application, Figure 6.10 shows the distribution of maximum network widths for an application's sink variables (i.e., each variable that has no outgoing edges in a constraint network). The maximum network width for a sink variable is the widest level in its sub-network. As defined in Section 2.4, a variable's sub-network is the sub-network of variables and constraints on which the sink variable directly or indirectly depends.

Whereas Figure 6.10 shows only the width of the widest level for each sink variable's sub-network,

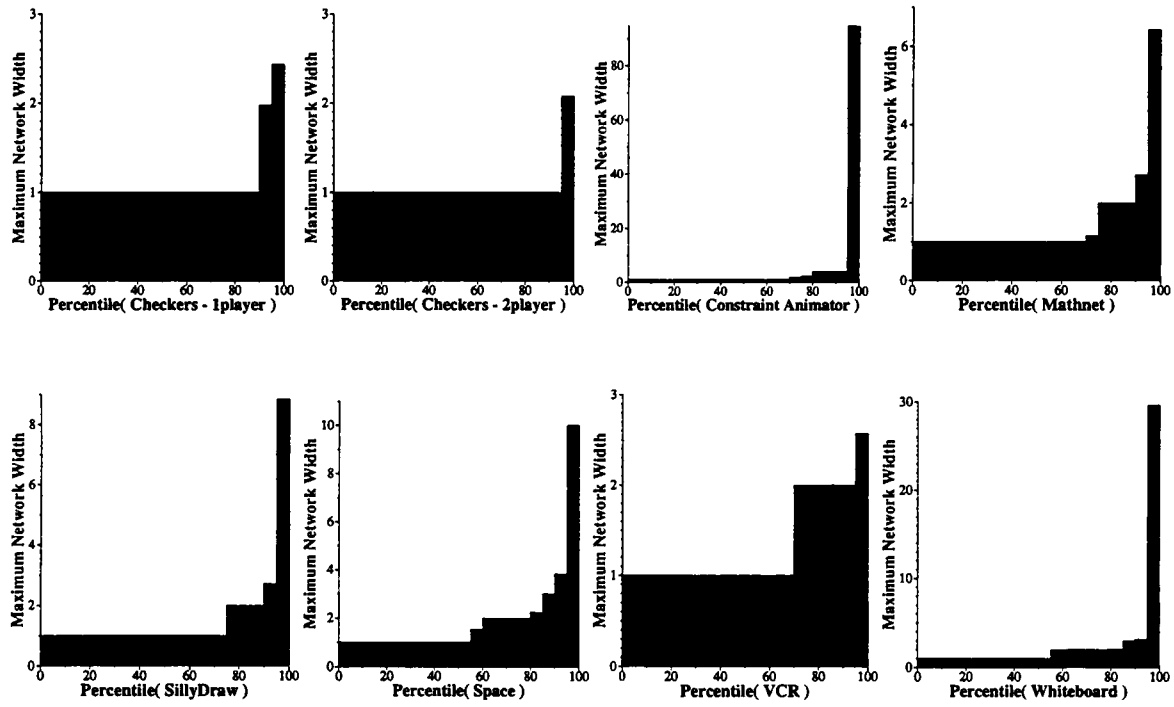


Figure 6.10: Maximum sub-network width for each sink node.

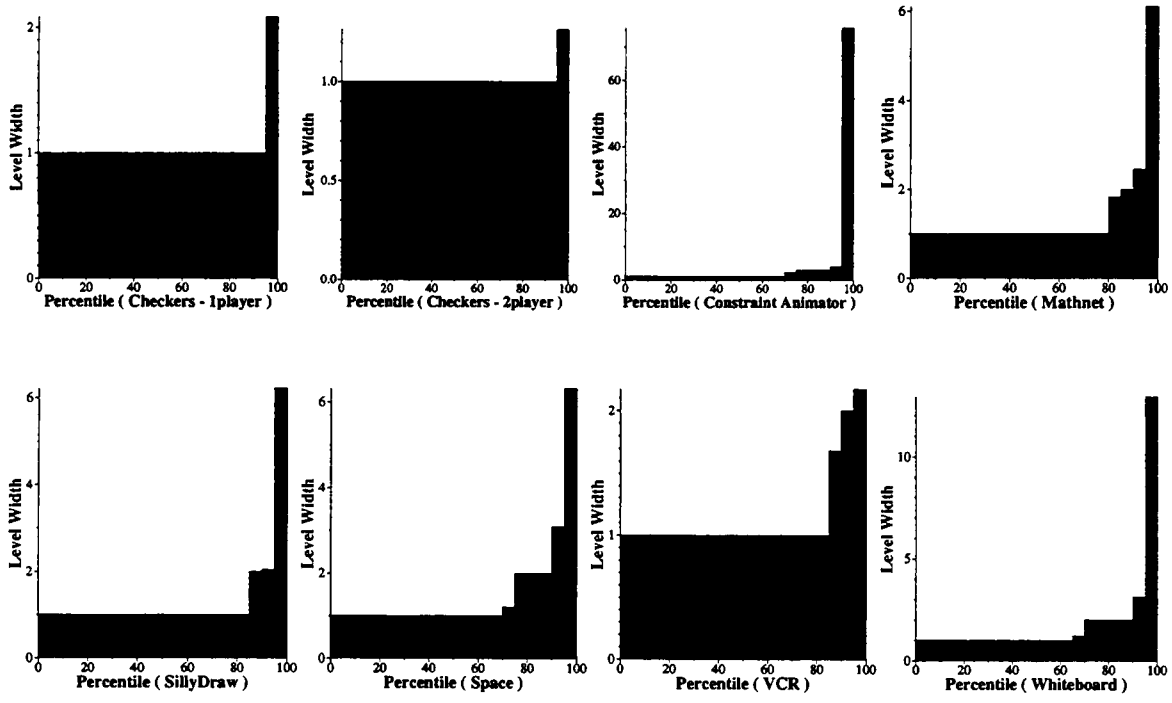


Figure 6.11: Width of each level in the sub-network for each sink node.

Figure 6.11 includes data for all the levels of a sink variables sub-network. Figure 6.11 shows the percentage of levels that had a width of 1, 2, 3, etc. For example, the sub-network in Figure 2.2 has a distribution of (1,2,0,1,0,1), which means that it has one level of width 1, two levels of width 2, 0 levels of width 3, and so on. The distributions of all the sink variables' sub-networks in an application were aggregated to create Figure 6.11.

Figures 6.10 and 6.11 show that most constraint networks are not that wide. Applications that made moderate use of constraints usually had level widths around one, with maximum widths that were less than ten. The majority of networks in applications that relied heavily on constraints (*Csdemo* and *Whiteboard*) still had widths less than ten. In our data sets, there were some extremely wide sub-networks for 5% of the sink nodes. Looking at *Csdemo*, the wide networks were used to constrain the coordinates of parts in the arrow objects. It is notable that these arrows were poorly written, with the forgotten intention of correcting them at a later time.

6.3.2 Analysis of Constraint Networks

The structure of a constraint network has implications for both debugging and for the efficiency and optimization of constraint satisfaction algorithms. The fact that constraint networks tend to be short and narrow implies that debugging constraint systems will not be problematic for programmers. Short 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. While there were a small number of constraint networks that were very wide (see Figures 6.11 and 6.10), these were often representative of poorly written code, which is also difficult to debug in more traditional environments.

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 [18, 7, 5]. 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 [14], Garnet [13], Rendezvous [5], and Eval/vite [8] systems reveals that they consume 16-24 bytes of memory. 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 6.7-6.8 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 overhead must be incurred in order to avoid these unnecessary reevaluations.

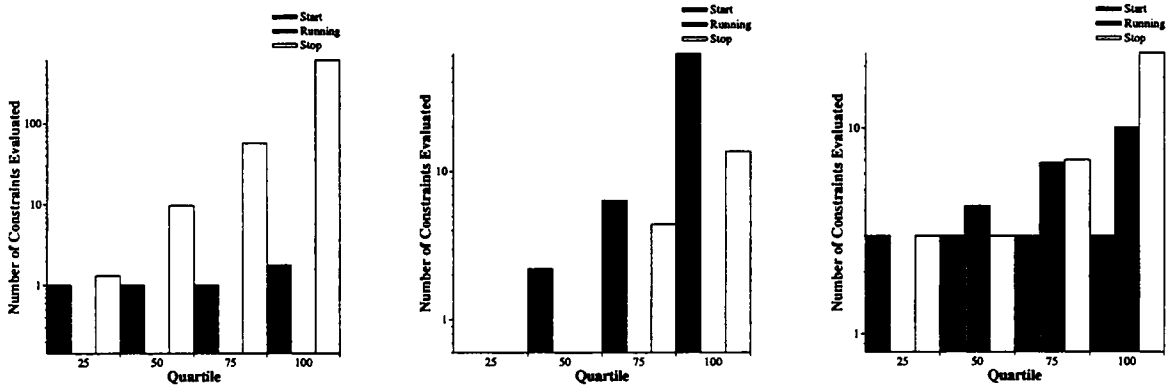
  This overhead can be avoided if one is willing to evaluate a constraint before all of its arguments have been brought up to date. Naive propagation schemes that evaluate a constraint as soon as one of its arguments changes employ this strategy. So does a scheme called approximate topological ordering, which evaluates constraints in approximately the correct order, but not the guaranteed correct order [6]. 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 [7].

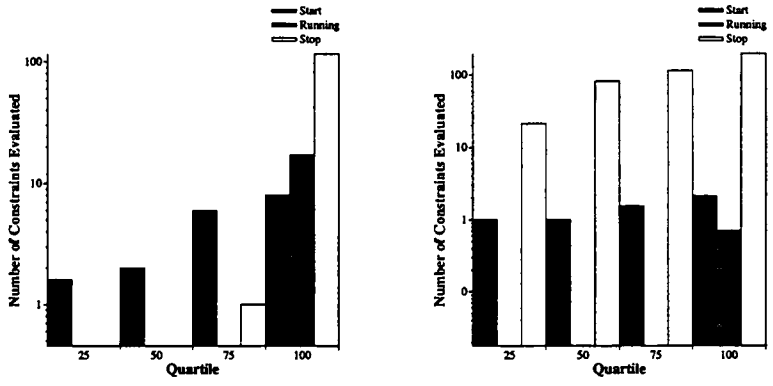
The data in Figures 6.10 and 6.11 indicates that networks are indeed somewhat serial, since most networks have a maximum width of one, and the majority of level widths are one. Consequently, in practice, probabilistic algorithms can be expected to work rather well. In addition, since constraint networks are short, 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.

6.4 Constraint Evaluation by Interactor Type

For every instance of an interactor, the number of start, running, and stop events was recorded, as well as the number of constraint evaluations that were attributed to the interactor in the start, running, and stop states of the interactor. From this, we were able to calculate for each instance of an interactor the average number of constraint evaluations for the start, running, and stop states. Using this data and a classification of each instance of the interactor as one of the five basic interactor types, the interactor data for each operation was combined for all the data sets we examined.



Choice Interactor. One Shot. Text.



Move/Grow. NewPoint.

Figure 6.12: Constraint evaluation by interactor type.

6.4.1 Data Gathered

Figure 6.12 depicts the combined data from all eight data sets by operation. Data for each state was ordered independent of interactor. That is, if an interactor's start state was in the top quartile, but its running state was in the lowest quartile, the start state would be represented in the highest quartile and the running state in the lowest quartile.

Text Interactors

`Text` interactors caused few constraint evaluations. The `Text` interactor was usually used to edit labels. While changes in a label could trigger some reevaluation of constraints (*Mathnet* would propagate a change in a label to several other objects) very little computation was initiated other than resizing as a result of the new text size. There was some appreciable processing caused by the running actions in the highest quartile as objects were adjusted around the entered text. For example, in *Csdemo*, the width of a variable object depended on the name of the variable; as the user changed the name of a variable, it resized itself around the changing name.

One Shot Interactors

There were not enough `One Shot` interactors to draw conclusions about constraint usage. Of the few instances, several were essentially "misused" when `New Point` or `Choice` interactors would have been the "correct" choice. In these instances, the `One Shot` interactors required far more customization than `New Point` or `Choice` interactors would have.

New Point Interactors

The stop actions of `New Point` interactors caused many constraint evaluations. The stop actions accounted for between 20 and 150 evaluations per activation. Usually, this activity was triggered by the creation of a new screen object.

Choice Interactors

The `Choice` interactor's stop action also triggered a great number of constraint evaluations, while its running action caused a few constraint evaluations. In Choice interactors, the running action will occasionally be called on to provide some interim feedback. On the other hand, since Choice interactors are often used in palletes, menus, and grouping of buttons, the stop action usually provided final feedback– such as selection handles that may be constrained to other objects– in addition to somewhat significant processing in a callback function. For example, in *Csdemo*, a `Choice` interactor can be used to mark an animated constraint network as invalid, and in *Mathnet*, a `Choice` interactor is used to select elements in a graph to delete.

Move/Grow Interactors

The `Move/Grow` interactor also caused a significant amount of constraint evaluation. It is interesting to note that 90% of the `Move/Grow` interactors were used to move, not grow, objects. In the top quartile, there were two types of "typical" behavior. In the first type, the `Move/Grow` interactors in the *Csdemo* and *Mathnet* applications did not use interim feedback objects. Instead, they moved composite objects across the screen. As a result, these interactors had more constraint evaluations in the running state. The other "typical" behavior, in applications like *Whiteboard*, was that an interactor used an interim feedback object, so that a significant amount of the constraint evaluations occurred in the stop action when the object was moved to its new location and objects whose position was constrained to the moved object had their positions updated.

6.4.2 Analysis of Interactor Operations

The data in Figure 6.12 gives us some insight on the construction of plans. As was mentioned in Section 3.2, gathering plan information requires additional overhead and the gathered information is specific to a set of objects. As a result, plans are impractical unless they will be repeatedly executed on the same set of objects. It is not practical to implement plans for the stop and start states, which are only executed once. Figure 6.12 indicates that `Move/Grow` and `Text` interactors sometimes have significant constraint evaluation in the running state and thus offer the opportunity for significant performance improvements. It would not be worthwhile to implement plans for the other three operations.

Another possibility is the use of parameterized plans, which contain references to parameterized objects. An example of this would be a `Move/Grow` Operation. Previous implementations of plans would require recomputation of the plan every time the interactor is activated. However, the `Move/Grow` interactor internally stores a reference to the object that it is operating on and to the feedback object. A parameterized plan could be computed using these references instead of the actual 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 6.12 suggests that parameterized plans could lead to significant performance improvements.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Our study of constraint networks leads to conclusions about the structure and debugging of constraint networks, the scalability of constraint programming, optimization of storage in constraints, the feasibility of plans, and ways to optimize slots in a prototype-instance object system.

Debugging. As mentioned before, the most interesting conclusion of this study was that constraint networks are relatively short and narrow. As a result, visual debugging tools and graph browsing techniques should be effective as aids for debugging constraint networks.

Scalability. A second major conclusion of our study was that constraint programming should scale well to larger applications. Programmers tend to create short, narrow constraint networks that are

very modular. Applications with more constraints have more of these modular constraint networks, instead of huge, monolithic networks. As a result, input operations, which almost always interact with only a few of these modules, will not involve large numbers of constraint evaluations, even as the size of the application grows.

Storage Optimization. Our study also suggested some possible optimizations in the storage of constraints:

- Constant propagation can yield some storage savings, but could, at most, be applied to 15% to 20% of the slots in our study.
- Some storage savings can be made by storing dependency information and formula function information in structures that will require more computation to access the data, but will yield a savings in space.
- Since most constraints have few dependencies (most constraints have branch-in and branch-out factors of two or less), use of a fixed array of dependency information, with linked lists used for additional dependency information, should result in a savings of space when compared to the current method of dependency storage.

Plan Feasibility. The use of plans as they have been implemented in some graphical interface tools was found by our study to be an ineffective technique for improving the performance of most operations. However, computation of parameterized plans, or plans which are not specific to a particular set of objects, appears to be an area that deserves additional research.

Slot Optimization. Lastly, the speed of data storage access in the Amulet prototype-instance object system could be improved by recognizing that the same slots, namely the coordinate slots, are the most often accessed. Caching of these slots or allocating storage for these slots as C ++ instance variables would be a simple way to improve the performance of the object system.

7.2 Future Work

Our study shows that current debugging techniques are suitable for the constraint networks we examined. However, because the graphics library we used does not have a constraint debugger, the applications we looked at were not developed with the aid of a constraint debugger. It would be useful to repeat this study once a debugger is available. Such a study would let us know 1) whether the structure of the constraint networks we observed in this study was affected by the lack of debugging tools, and 2) if the network structure was changed by the existence of suitable debuggers, would the debugging tools be suitable for the larger networks?

A second direction for future work would be to collect data on several additional widget-based applications to determine if their data access patterns differ from more dynamic interfaces. One ramification of a difference in data access patterns between widget-based and dynamic interfaces is that optimizations that worked well for widget-based applications might not work well for dynamic interfaces. Consequently, new optimizations might have to be devised.

Bibliography

Bibliography

[1] Alan Borning. The Programming Language Aspects of ThingLab; a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[2] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of Self, a Dynamicall-Typed Object-Oriented Language Based on Prototypes. In *OPSLA '89 Conference Proceedings*, pages 49–66, New Orleans, Louisiana, October 1989.

[3] Bjorn N Freeman-Benson. A Module Mechanism for Constraints in Smalltalk. In *OPSLA '89 Conference Proceedings*, pages 389–396, New Orleans, Louisiana, October 1989.

[4] Tyson Rombauer Henry. Interactive Graph Layout: The Exploration of Large Graphs. Technical Report TR92-093, The University of Arizona, 1992.

[5] Ralph D. Hill. The Rendezvous Constraint Maintenance System. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 225–234, Atlanta, GA,

November 1993. Proceedings UIST'93.

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

[7] Scott E. Hudson. Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update. *ACM TOPLAS*, 13(3):315–341, July 1991.

[8] Scott E. Hudson. A System for Efficient and Flexible One-Way Constraint Evaluation in C++. Technical Report 93-15, Graphics Visualizaton and Usability Center, College of Computing, Georgia Institute of Technology, April 1993.

[9] Scott E. Hudson. User Interface Specification Using an Enhanced Spreadsheet Model. *ACM Transaction on Graphics*, 13(3):209–239, July 1994.

[10] John H. Maloney, Alan Borning, and Bjorn N Freeman-Benson. Constraint Technology for User-Interface Construction in ThingLabII. In *OPSLA '89 Conference Proceedings*, pages 381–388, New Orleans, Louisiana, October 1989.

[11] Rich McDaniel and Brad A. Myers. Amulet's Dynamic and Flexible Prototype–Instance Object and Constraint System in C++. Technical Report CMU-CS-95-176, Carnegie Mellon University, Pittsburg, PA, July 1995.

[12] 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.

[13] 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.

[14] 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. Also available as Technical Report No CMU-HCII-95-102.

[15] Bindu R. Rao. *C++ and the OOP Paradigm.* McGraw-Hill,Inc., New York, first edition, 1993.

[16] Michael Sannella. *Constraint Satisfaction and Debugging for Interactive User Interfaces.* PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, 1994. Also available as Technical Report 94-09-10.

[17] David Ungar, Randall B. Smith, Craig Chambers, and Urs Holzle. Object, Message, and Performance: How They Coexist in Self. *IEEE Computer*, 25(10):53–64, October 1992.

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

Appendix

Data Gathering Procedure

A.1 Data Gathering for *Checkers*

Checkers is a sample program distributed with Amulet. It is based on the rules of the classic game of checkers. A unique feature of *Checkers* is that the interface supports either a single display, in which both players use the same mouse to manipulate checkers on the same screen, or a dual display mode in which both players have duplicate views of the game displayed on their own screen and use their own mouse to control play.

Because *Checkers* has one goal, the testing is very straight forward, playing the game several times to win. This was done once when the program was in single display mode, and once when it was in dual display mode.

A.2 Data Gathering for *Constraint Animator*

Constraint Animator is a program developed by a graduate student to learn about Amulet. It is an editor that allows for creation and modification of constraint networks that include pointer, variable, formula, and dependency objects.

I performed the following actions:

1. Variable, pointer, and formula objects were selected from the pallet and several of each were created in the editor screen.
2. A multilevel constraint graph was created by adding dependency arrows. There were two formulas. One formula was dependent on two variables and one pointer. The other was dependent on one pointer and one variable. Each formula had one output variable.
3. Selected variables were marked out of date in order to make both variables that were output by the two formulas out of date.
4. Formulas, variables, and pointers were repositioned on the screen.
5. One of the out of date variables that depends on a formula was selected and evaluated. The evaluation caused the creation of a new dependency due to the formula's dependence on a pointer variable.
6. Text in two variables and one pointer was edited.
7. A new variable was created.
8. One of the pointers that a formula depended on was made to point at the new variable.
9. The formula that was marked out of date by the previous step was selected and evaluated.
10. The formula that was evaluated was then marked out of date.
11. The out of date formula was then selected and deleted.
12. The window was resized.
13. A new formula was created.
14. All variables that were not dependent on anything were made outputs of the new formula.
15. The new formula was made dependent on a pointer v

16. The new formula was evaluated.
17. The program was exited.

A.3 Data Gathering for *Sillydraw*

Sillydraw is a drawing editor written for an undergraduate graphical user interface class at Carnegie Mellon University.

I performed the following actions:

1. Several different shapes were drawn on the screen. In this particular case, I used ovals, roundtangles, rectangles and lines to create a face.
2. A rectangle (an eye) was selected and the color changed to blue.
3. The rectangle's border was given a wider line width.
4. A different rectangle (the other eye) was deleted.
5. The window was resized.
6. The window was made larger.
7. An oval (a new eye) was created. The fill color of the eye was red.
8. A large rectangle was created over the entire face.
9. The large rectangle was selected and sent to the back.
10. An oval (an ear) was selected and sent to the back.

A.4 Data Gathering for *VCR*

VCR is an online VCR interface written for an undergraduate graphical user interface class at Carnegie Mellon University. The interface consists of two components – a control panel for the

basic functions (i.e., stop, start, play, pause, and channel selection), and a Timer control panel to control recording.

I performed the following actions:

1. The channel was changed to 18.
2. The control panel was expanded to show program information.
3. Play was pressed.
4. Forward was pressed.
5. Back was pressed.
6. Stop was pressed.
7. Pause was pressed.
8. The channel was changed to 6.
9. Record was pressed.
10. Timer was pressed, to set up recording.
11. Record 1 (the first event to be recored) was programmed.
12. The event was saved in the VCR's timer.
13. Record 2 was programmed.
14. The main window was resized.
15. The stop button was pressed.
16. The event we were programming was deleted.
17. The delete was undone.
18. Stop was selected.
19. The tape was ejected.
20. Quit was selected.

A.5 Data Gathering for *Space*

Space is a game interface (without the actual game play) distributed with Amulet. One interesting aspect of the interface (which made it unique in my samples) is its use of Amulet's built–in "Undo" capacity.

I performed the following actions:

1. Several Federation and Klingon ships were created.
2. Rigid tractors were created between some of the ships.
3. A Klingon ship used phasers to destroy two other ships.
4. The labels of two ships were edited.
5. The short range scan was scrolled to the right and down.
6. The long range scan was moved.
7. Two new ships were created and linked with springy tractors.
8. One of these ships was moved off the short range scan.
9. The window was resized.
10. The window was scrolled in order to see the short range scan.
11. Several ships were created.
12. One ship used its phasers to destroy all the other ships in the game.
13. Using undo, the ships were recreated.
14. Using redo, ships were re-deleted.
15. Quit.

A.6 Data Gathering for *Mathnet*

Mathnet is a program written at Southern College to learn about Amulet. It allows interactive creation of arithmetic networks.

I performed the following actions:

1. The menu was brought up.
2. New was selected.
3. Several arithmetic nodes were created.
4. Several variable nodes were created.
5. The operators on the arithmetic nodes were changed to "+" or "*".
6. Using the arithmetic and variable nodes, an arithmetic graph of depth 3 was created by linking variable and arithmetic nodes with dependencies. This network included a cycle.
7. Several nodes (not part of the cycle) were changed.
8. A node in the cycle was changed.
9. A node that was a part of the cycle was deleted.
10. Several nodes in the graph were moved.
11. The arithmetic network was saved.
12. Several new dependencies were created.
13. Parts of the graph were rearranged.
14. The arithmetic network we saved earlier was reloaded.
15. The program was exited.

A.7 Data Gathering for *Whiteboard*

Whiteboard is an environment that provides a visual programming-by-example for arrays. It was written by an Ph.D. student at the University of Tennessee.

I performed the following actions:

1. 5 integers were created on the editor.
2. An array was created.
3. The names and values of the integers were modified.
4. Two indices were added to the array.
5. One index was set to a value of 3, the other to a value of 4.
6. Item 3 of the array was changed to a value of 5.
7. Two of the integers were multiplied. The result was stored in an element of the array.
8. An if command was executed.
9. The conditional of the if was changed to ">=".
10. Two variables were added as the arguments of the ">=".
11. An integer was assigned a new value.
12. An end command was executed.
13. A while was executed.
14. Two integers were assigned to the conditional of the while loop.
15. An array element was assigned a new value.
16. An end command was executed.
17. An end command was executed.
18. The program was exited.

Vita

Scott Venckus was born in Great Lakes, Illinois, on December 31, 1969. He grew up in Northbrook, Illinois, a suburb of Chicago, and graduated from Glenbrook North High School in June of 1988. From there, he attended Trinity University in San Antonio, Texas. In May of 1992, he received a Bachelor of Science degree in Computer Science. After working for a year at TPS Systems in San Antonio as a software engineer, he entered the Computer Science Master's program at the University of Tennessee in August of 1993. He was awarded a Master's of Science degree in December of 1996.