© Copyright 2000

Gregory Joseph Badros

Extending Interactive Graphical Applications with Constraints

by

Gregory Joseph Badros

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2000

Program Authorized to Offer Degree: Department of Computer Science and Engineering

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Gregory Joseph Badros

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by the final

examining committee have been made.

Chair of the Supervisory Committee:

____

Alan Borning

Reading Committee:

____

Richard Anderson

____

Alan Borning

____

David Notkin

Date: _______________________

In presenting this dissertation in partial fulfillment of the requirements for the Doctorial degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with “fair use” as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted “the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform.”

Signature________________________________

Date____________________________________

University of Washington

Abstract

Extending Interactive Graphical Applications with Constraints

by Gregory Joseph Badros

Chair of the Supervisory Committee

Professor Alan Borning
Computer Science and Engineering

A constraint is a relation that we would like to maintain. Over the last thirty years, a number of problems have prevented constraints from being widely accepted for use in interactive graphical applications. The biggest difficulty of applying constraints has been finding the right tradeoff between performance and expressiveness. To be able to satisfy systems of constraints efficiently enough for real-time use, interactive applications have restricted the problem to solving less-expressive sets of constraints. One common such restriction is to require that the constraint relationships are acyclic, thus enabling simple techniques based on local propagation. Unfortunately, these limitations in expressiveness counteract the benefits that the declarative specification of relationships with constraints is intended to provide.

A sophisticated new constraint solving toolkit, Cassowary (based on the simplex algorithm), is presented. It is designed to support especially easy embedding of efficient constraint-solving capabilities in arbitrary interactive graphical applications. To prove its usefulness and refine its capabilities, the Cassowary toolkit was used in three such applications: the Scheme Constraints Window Manager (SCWM) explores exposing the power of constraints to the end user via a graphical user interface; Constraint Cascading Style Sheets (CCSS) for the Amaya web browser illustrates how constraints can be a powerful means of understanding a complex procedural specification and can provide a unifying implementation mechanism; and Constraint Scalable Vector Graphics (CSVG) for the CSIRO SVG viewer demonstrates that constraints provide a powerful framework for de-

laying the final layout of diagrammatic illustrations to enable visualizations better adapted to the viewing environment. Each of these applications benefits from constraint features and maintains good performance by leveraging Cassowary.

Constraint solving, and in particular the Cassowary constraint solving toolkit, can provide useful capabilities that are easy to take advantage of. It, and similar technologies, are ready for more significant and ambitious use in the domain of interactive graphical applications.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY OF WEB TERMS

CSS: Cascading Style Sheets, a W3C recommendation specifying the various properties of elements on a web page that can be set. “Cascading” refers to the way conflicting preferences from multiple style-sheet specifications are resolved.

DOM: Document Object Model, a W3C recommendation describing an application programming interface for manipulating the in-memory tree representation of an XML document.

HTML: HyperText Markup Langauge, the standard markup language for textual content delivery on the World Wide Web through the 1990s.

SGML: Standard Generalized Markup Language, a meta-language for describing arbitrary tree structured-documents and data standardized in the 1980s. SGML was the inspiration for HTML, and XML is a much-simplified version of SGML.

W3C: World Wide Web Consortium, an industry consortium that issues “recommendations” on various web protocols and languages including CSS, DOM, HTML, XML, and XSL.

XML: eXtensible Markup Language, a simplified version of the SGML meta-language for describing graph-structured documents and data.

XSL: eXtensible Style Language, a declarative XML-based language for describing tree to tree transformations coupled with an XML-based language for “formatting objects” that describe the layout of a page.

XSLT: eXtensible Style Language for Transformations, just the tree-to-tree transformation language piece of XSL that can, e.g., be used to convert arbitrary XML into HTML.

ACKNOWLEDGMENTS

My years in graduate school have been some of the best of my life. A large number of people have contributed to making my time here educational and enjoyable. First, I thank my advisor, Alan Borning, for his wisdom, encouragement, and advice. Alan provided the guidance I needed to stay focused throughout this long process. I also thank the rest of my committee: David Notkin, Richard Anderson, Alon Levy, and Jack Lee. I very much appreciate their valuable input and interest in this work. David, in particular, I thank for shaping the early years of my graduate school career as my master's thesis advisor. I also owe great debts to the outstanding faculty who broadened my knowledge throughout my course work: Craig Chambers, Brian Curless, Carl Ebeling, Hank Levy, Larry Ruzzo, Larry Snyder, and Martin Tompa each influenced my thinking through their thoughtful class discussions and projects. Also, thanks to Ed Lazowska for helping to make the department such a great place to do research.

All research requires basic computing facilities and I was lucky enough to have an incredible support team keeping our machines and network running smoothly. I thank our entire support staff and especially Nancy Burr, Warren Jessup, Jan Sanislo, Varadesh Yenbut, and Eric Lundberg for their going above and beyond the call of duty in handling all of my special requests.

Beyond computing facilities, there are innumerable details that completing a doctorate requires addressing. I am extraordinarily grateful to Frankye Jones and Alicen Smith for hiding nearly all of the bureaucracy from me and enabling me to focus on my research.

Over the last four years, I've interacted with a large number of incredibly talented and engaging fellow graduate students. I thank my house-mates over the years, Todd Millstein, Craig Kaplan, Doug Zongker, Tapan Parikh, Matt Carey, Will Portnoy, and Zasha Weinberg, for their innumerable conversations about both technical and non-technical topics. Additionally, I thank the numerous office-mates I've shared space with including Miguel Figueroa, Mike Ernst, Patrick Crowley, Dennis Lee, Jeremy Baer, Aaron Davis, Kevin Hinshaw, Neil Spring, Denise Pinnel, Dawn Werner, Paul Franklin, Zack Ives, Rachel Pottinger, Andy Collins, Jong Hee Kang, and Karen Liu. I cannot thank them enough for their discussions, suggestions, encouragement, and support.

Other folks around the department have been instrumental in pushing my thinking, in improving my tennis, basketball, softball, and ultimate Frisbee abilities, and just simply being good friends. Thanks go to Jonathan Aldrich, Corin Anderson, Gretta Bartels, A.J. Bernheim-Brush, Jeremy Buhler, Brad Chamberlain, David Ely, Marc Fiuczynski, Marc Friedman, Justin Goshi, Dan Grossman, Jason Hartline, Ed Hong, David Hsu, Tessa Lau, E. Christopher Lee, Soren Lerner, Vass Litvinov, Omid Madani, Amir Michael, Alex Mohr, Bart Niswonger, Michael Noth, Kurt Partridge, Sean Sandys, Stefan Saroui, Erik Selberg, Emin Gun Sirer, Brian Tjaden, Steve Wolfman, and Ken Yasuhara.

Although much of my emotional and intellectual support came from those directly around me, numerous friends stand out as important influences on my life thus far. I thank Allison Adin Bianchi, Steve Burgos, Erin Casey, Michael Chiou, Emily Coleman Kangas, Jonathan Dattlebaum, Scott Harrington, Nicole Marcson, Kristin Reddish, Andrea Restivo, and Julie Tseng for their phone conversations and emails about anything and everything.

No research is done in isolation. I'm grateful to have had the opportunity to work with incredibly great people over the years. The Constraint Cascading Style Sheets work was largely done while I was visiting Monash University in Melbourne, Australia. Kim Marriott and Peter Stuckey were an absolute joy to work with, and I thoroughly loved my visit down under thanks to their friendship and the other great people at Monash University including Warwick Harvey, Lisa and Toby Sargeant, Tony Jansen, Richard Lin, Sitt-Sen Chok, and Maria Garcia de la Banda. Also, thanks to Bert Bos and Håkon Lie of the CSS working group of the World Web Consortium (W3C); they provided very useful expert feedback on the early stages of the CCSS project.

The Cassowary Constraint solving toolkit has evolved greatly in the past several years. That evolution's success is largely due to the feedback I received from the users of the toolkit including Anthony Beurivé, Alexandre Duret-Lutz, Michael Kaufmann, Brian Grant, Pengling He, Tessa Lau, Sorin Lerner, John MacPhail, Larry Melia, Michael Noth, Emmanual Pietriga, Stefan Saroiu, and Steve Wolfman. The implementation of the Constraint Scalable Vector Graphics prototype was done in collaboration with my fellow graduate student Will Portnoy. Additionally, the CSVG work benefited from the advice of Vincent Hardy from the W3C's SVG working committee.

SCWM involved a large engineering effort that leveraged numerous contributors as it grew in the open-source community. Most notably, Maciej Stachowiak started the Scwm project a couple weeks before I began working on the code base. Jeff Nichols did a substantial portion of the prototype implementation of the constraints interface for SCWM, and Sam Steingold, Robert Bihlmeyer, and Todd Larason each made repeated significant contributions to SCWM.

Finally, I thank my mother, Karen, my father, Joseph, and my brothers, Mark and Eric. My family's love, support, and encouragement has been instrumental to my success and happiness. I cannot possibly thank them enough for how they have shaped my life.

This work was supported by a National Science Foundation Graduate Research Fellowship, by the University of Washington Computer Science and Engineering Wilma Bradley fellowship, and by NSF Grant No. IIS-9975990.

Chapter 1

INTRODUCTION

A constraint is a relation that we would like to maintain. For over thirty years, interactive graphical applications have used constraints to relate user interface elements, components of drawings, and more. The key benefit of constraints is the separation of *what* relationships we desire from *how* we must go about maintaining those relationships.

1.1 Motivation

Two main problems have plagued constraint systems: expressiveness and performance. For constraints to be of benefit, we must be able to express interesting and useful relationships. For systems solving those constraints to be acceptable in interactive applications, the algorithms must be efficient enough to support real-time use.

Expressiveness and performance are in a delicate balance. Arbitrary constraints provide maximum expressiveness, but solving such systems is undecidable. On the other hand, many constraint systems have been developed that have excellent performance, but are subject to significant limitations in what can be expressed.

A third significant concern for constraint systems is understandability and predictability. Especially when interacting with on-screen objects, it is important that users understand why the objects are acting as they do, and that they be able to achieve the results they expect. The theory of constraint hierarchies [14] provides a declarative semantics for what constitutes correct solutions for systems of constraints specified with varying levels of preferences. Solvers that respect the hierarchy and avoid procedural side effects will most likely not surprise the user.

A fairly recent algorithm, Cassowary [18] strikes a seemingly useful design point in the tradeoff between expressiveness and performance for constraints relating real-valued numbers: any linear arithmetic relationship can be expressed, constraints can be required or at arbitrarily many levels of preference, and performance of the solver is good. Cycles, which correspond to simultaneity in the underlying problem, are no problem, and the solution respects constraint hierarchy theory (Section 3.1.1). However, non-linear relationships (such as relationships involving Euclidean distance between pairs of points) and non-numeric constraints cannot be maintained by Cassowary.

The thesis of my research is that constraint systems are now advanced enough to be treated as a black box, and that they can be easily embedded in interactive graphical applications to provide the benefits of constraints more broadly.

1.2 Contributions

The primary contributions of this dissertation are:

- The Cassowary constraint solving toolkit: an efficient, modular, reusable software component that permits easy embedding of powerful constraint features in applications, including the ability to solve simultaneous systems of linear arithmetic constraints of varying strengths (Chapter 3). The toolkit implements the Cassowary algorithm from earlier work [18]. Chapters 4, 5, and 6 each describe a use of this toolkit for the constraint support of the underlying application. Experience building those systems has resulted in improvements in the toolkit, and demonstrated the wide applicability of the algorithms and implementations.

- The Scheme Constraints Window Manager (SCWM): a powerful, highly-extensible, fully-programmable, constraint-enabled X/11 window manager. SCWM embeds Cassowary and demonstrates the applicability of sophisticated constraint solving algorithms to the domain of highly-interactive graphical applications, and it provides a highly-usable interface to its constraint features, a simple programming by demonstration system, and a simple form of constraint inference. SCWM is an excellent starting place for further research into window layout policies, voice-recognition interfaces for constraints, and other constraint capabilities (Chapter 4).

- A constraint extension to Cascading Style Sheets (CSS), called Constraint Cascading Style Sheets (CCSS), along with a prototype implementation of CCSS in the Amaya World Wide Web Browser. CCSS provides a declarative framework for understanding the complicated cascading rules of CSS and a unifying implementation technique (Chapter 5).

- A constraint extension to Scalable Vector Graphics (SVG) called Constraint Scalable Vector Graphics (CSVG), along with a prototype implementation of CSVG based on the CSIRO SVG viewer. CSVG permits delaying final layout of objects in an illustration. By treating layout of objects as a presentation attribute, we increase flexibility of the images and permit more appropriate rendering based on the user's viewing conditions and desires (Chapter 6).

Chapter 7 summarizes, discusses some of the limitations of this research, and proposes avenues for future work.

Chapter 2

BACKGROUND

2.1 Introduction

Numerous interactive graphical systems, including drawing, graph layout, visualization, and animation systems, embed a constraint solver to manage the geometric layout of on-screen objects. These interactive and geometrically-based systems used in user interfaces make stringent demands of the constraint solving technology—the solver must be powerful enough to handle geometric constraints, fast enough for real-time interaction, and predictable enough to not confuse the user.

Section 2.2 describes some general issues regarding the semantics of constraints. Section 2.3 discusses the classes of constraints occurring in various interactive graphical applications. I then describe and categorize the supporting constraint-solving technologies with respect to the kinds of constraints they support. Section 2.4 details this taxonomy, and compares the expressiveness and performance of the solvers. Section 2.5 concludes by summarizing important problems that future work must address.

2.2 Constraint semantics

The theory of constraint hierarchies provides a declarative specification of the solutions to sets of constraints, independent of the algorithms used to satisfy them [14].

Formally, a constraint is a mathematical relation: a Cartesian product over *n* domains. Constraints can be labelled with strengths. There is a single distinguished required strength, and arbitrarily many levels of preferences. Required constraints are called “hard,” while the others are deemed “soft.” Often, numeric constraints are written using ordinary relational operators. We can write:

required: $x \geq y + 100$

to specify a constraint that the scalar value assigned to the variable x must be at least 100 more than the value assigned to y.

Multi-sets of labelled constraints form a constraint hierarchy. A solution to a constraint hierarchy is an assignment of all free variables in the hierarchy to values in the domain of the relation. Each such valuation, when applied, must satisfy all required constraints, and should satisfy all soft constraints as well as possible, treating all stronger constraints as more important than any weaker ones. Various “comparators” are used to compute how closely a valuation satisfies constraints. Locally-better comparators consider each constraint individually, while globally-better ones consider sums of errors across all the constraints.

Soft constraints and comparators are a means of specifying optimization functions for choosing a solution from the feasible region described by the required constraints. Though an arbitrary objective function could be allowed, comparators attempt to enumerate useful objectives and may also yield more efficient solving algorithms. For example, locally-better comparators enable the use of greedy solving algorithms.

An important consideration when choosing among possible solutions is the stability of the figure. Objects in a layout should not move unless there is some reason for them to change positions. One way to express this desire is to weakly constrain each attribute to take its value from the previous time step. To disallow the present from changing the past, we can use a read-only annotation on the past, and express these weak “stay” constraints as: $x_t = x_{t-1}?$ where the question mark denotes a read-only variable that cannot have its value changed by the solver to satisfy a constraint. Read-only variables are also important for modeling the semantics of one-way constraint systems, which are solved efficiently by various “local propagation” algorithms (Section 2.4.2).

The most important contribution of constraint hierarchy theory is the formalization of the declarative semantics of constraint systems. By understanding from first principles what the set of solutions is and which one is best, we can then judge constraint solving algorithms with respect to its ability to find solutions, the expressiveness of the supported constraints, and its time and space performance characteristics.

A proper formal semantics for constraint systems is also important for reasoning about solving. For example, knowing the right answer is essential to verify the correctness of optimizations such as constraint compilation.

2.3 Constraints across applications

Interactive graphical applications use constraints primarily to express desires regarding geometric layout. All of these systems are similar in that they expose constraints to the end user. However, the representations of constraints and to what they are attached vary substantially. Representation can influence the expressiveness and efficiency of a constraint solver by altering what kinds of relationships can be specified within the limitations of the solver.

For example, if line segments are represented using a starting point, direction, and length, a constraint that two lines are the same length or parallel is easy to express using a simple linear equality. Those same relationships will require non-linear equations if line segments are stored as Cartesian coordinates of the start and end of the segment. Since solving general non-linear constraints is computationally difficult, systems often rely on domain-specific methods to handle the non-linearities they deem most important.

I now examine constraints in applications for drawing, graph layout, visualization and animation systems. For each application, I highlight the unique and interesting features of the system while listing the class of constraints it handles. For an overview of the kinds of constraints supported by each system, the solving technique employed, and the performance of the solver, see Table 2.1. Most notably, this section will show that applications tend to restrict their use of constraints based on limitations of the underlying solver. Section 2.4 will discuss the relative expressiveness of the various satisfaction algorithms used by the applications discussed in this section.

2.3.1 Drawing

Interactive drawing applications employing direct-manipulation techniques [129] have been very successful. For professional-looking diagrams and illustrations, however, they sometimes fall short due to the lack of precision in the diagram. Most conventional drawing systems permit alignment of objects, but such relationships are only enforced once—at the time the command is issued. If an aligned object is later moved, the other objects do not follow. Constraint-based drawing systems permit the persistence and maintenance of desired relationships to ease editing burden and ensure

Table 2.1: Overview of constraints and solvers in interactive graphical applications.

	System	**Author (Year)**	**Constraints supported**	**Solving technique**
Drawing	Sketchpad	Sutherland (1963)	geometric	local propagation, relaxation
	IDEAL	Van Wyk (1982)	geometric in complex plane	local prop. w/o planning
	Juno	Nelson (1985)	CONG, PARA, HOR, VER	iterative numeric
	Juno-2	Heydon & Nelson (1994)	CONG, PARA, HOR, VER	optimized iter. num.
	Briar	Gleicher & Witkin (1994)	points-on-object,coincident	differential methods
	Unidraw	Helm et al. (1995)	linear (in)equalities	direct numeric (QOCA)
	GCE	Kramer (1992)	geometric	DOF analysis
	Chimera	Kurlander (1991)	geometric	symbolic, numeric
	Pegasus	Igarashi et al. (1997)	geometric	CLP(R)-like
Graph	GLIDE	Ryall et al. (1997)	VOFs	spring simulation
	CGL	He et al. (1996)	linear (in)equalities	iter. numeric
Visualization	TRIP*N*, IMAGE	Takahashi et al. (1998)	linear geometric	graph-layout, direct & iterative
	ICOLA	Oster & Kusalik (1998)	linear inequalities	extreme-bound propagation
	Penguins	Chok & Marriott (1998)	linear (in)equalities	direct numeric (QOCA)
Animation	TLCC	Gleicher & Witkin (1992)	geometric (on camera image)	differential methods
	Animus	Duisberg (1987)	arbitrary acyclic	local propagation, relaxation
	JIM, Parcon	Griebel et al. (1996)	linear (in)equalities, geometric	iterative numeric

precision in the diagram. To date, drawing programs are the most common interactive graphical applications to use constraints.

Sutherland's Sketchpad, the earliest interactive constraint-based system, permitted constraints on the parts of a figure to be explicitly specified. For example, a pair of lines can be made equal in length, or an angle can be marked as a right angle [134, App. A]. Additionally, the user implicitly adds constraints through the use of the "pseudo-pen location" which locks the input pointer position onto topologically-important locations in the diagram. A similar input technique called "snap-dragging," developed years later by Bier, uses the snapped-to positions only for their absolute location [9]. Sketchpad, in contrast, stores and maintains the constraint relationships as objects are rearranged.

Sketchpad was ahead of its time; IDEAL, the next constraint-based system specifically targeting drawing, appeared almost twenty years later [138]. Unlike Sketchpad, IDEAL is strictly a textual language for specifying pictures—it is not an interactive system. IDEAL permits specifying arbitrary non-simultaneous constraints on points in the complex plane. The drawing is then created procedurally from a configuration of the points that satisfies the constraints.

Like Sketchpad, Juno and Juno-2 [112, 76] are interactive systems. Juno permits specifying constraints on points and line segments. There are only four predicates: `HOR` and `VER` express the horizontal and vertical relationship between pairs of points, while `CONG` and `PARA` are congruence and parallel relationships (both non-linear) between pairs of line segments. Juno's constraint relationships are specified at a higher level of abstraction than Sketchpad or IDEAL, though internally they are maintained as numerical mathematical relationships. Juno provides double-view editing, where both the graphical picture and the (partially declarative) program that constructed it are viewed simultaneously. Interactive direct-manipulation of the picture is reflected immediately as implicit edits of the program's text.

Briar [59] is an interactive drawing editor that permits expressing exactly two geometric constraints: *points-on-object* and *points-coincident*. Although this set of relations is limited, Briar re-gains expressive power by allowing "alignment objects." Such objects exist only as constraint-assistance artifacts and are not part of the appearance of the final drawing. For example, the constraint that point p is distance k away from point q can be expressed by placing an alignment circle centered at point q, and constraining point p to be on that circle. Constraints among both regular and alignment objects are specified implicitly through an extension of Bier's snap-dragging suitably named "augmented snap-dragging" [53]. Adding a constraint on a new object corresponds directly to creating that new object while the pointer is snapped onto a pre-existing object or point. Unlike other systems, in Briar there is never a need to manage constraints explicitly. Removing a constraint is performed by breaking constraints through "ripping apart" objects—the user specifies only the desired effects and the system chooses which constraints must be eliminated.

Unidraw [72] is an extension of an earlier direct-manipulation drawing program. It permits arbitrary simultaneous linear equalities and inequalities among attributes of its various predefined objects. This class of constraints is shared by the CDA [114] drawing application, and Penguins [28], a drawing-editor construction framework (analogous to YACC for generating language parsers).[1] Among those listed here, Unidraw and the CDA are the only drawing editors without any support for non-linear constraints. Also, Unidraw is unique among constraint-based drawing editors in its support for `undo` and `redo` operations. Although typically challenging to implement, these features are permitted by Unidraw's ability to easily enable and disable constraints and to save and restore the state of the entire constraint system.

Kramer's Geometric Constraint Engine [90] (GCE is an extension of his earlier The Linkage Assistant, or TLA) is not specifically a drawing editor, but solves the same class of geometric layout problems. GCE permits five classes of binary constraints between geometric objects, or *geoms*: distance between a point and a point, line, or plane; distance between a line and a circle; and angle between a pair of vectors. These constraints are tied to the geometric degrees-of-freedom analysis performed in Kramer's underlying solver (see Section 2.4.5).

Chimera [91, 93] not only supports drawing constrained figures, but also provides a constraint inference engine. Kurlander's system permits the constraints shown in Table 2.2. Like GCE, the constraints supported by Chimera are directly related to a solving technique characterized by reasoning about transformational groups. Chimera's inference engine works by comparing multiple snapshots and constraining the properties that are invariant (within a tolerance) across the snapshots. Instead of explicitly stating what relationships one wants to hold, the user must vary all

[1]See Section 2.3.3 for more discussion of Penguins.

of the degrees of freedom that are meant *not* to be constrained. The invariant-detection tolerance mechanism employed by Chimera for detecting invariants is similar to Pavlidis's automatic beautification in PED [119]. PED, however, only infers the constraints and makes the diagram more precise, whereas Chimera dynamically and interactively maintains the constraints.

Table 2.2: Constraints permitted by Kurlander's Chimera [91, p. 14]

Absolute Constraints	**Relative Constraints**
Fixed vertex location	Coincident vertices
Distance between two vertices	Relative distance between pairs of vertices
Distance between parallel lines	Relative distance between pairs of parallel lines
Slope between two vertices	Relative slope between two pairs of vertices
Angle between three vertices	Equal angles between two pairs of three vertices

Pegasus (Perceptually Enhanced Geometric Assistance Satisfies US) [84] is a rapid sketching tool that also interactively infers constraints. Pegasus recognizes seven kinds of constraints: connection, parallelism, perpendicularity, alignment, congruence, symmetry, and interval equality. Unlike Chimera, the constraints Pegasus infers are not maintained (i.e., they are one-shot corrections, more akin to snap-dragging).

2.3.2 Graph layout

Graph layout is a particularly challenging application for use of constraints. The aesthetic criteria by which a graph layout is judged is difficult to express using simple relationships. In general, graph layout requires minimization of a greater-than-quadratic objective for visually-pleasing results. Optimization criteria often includes eliminating node overlaps, minimizing edge crossings, and maximizing symmetries. Classical graph layout algorithms are typically expensive, batch-oriented computations [37, 38].

Wieqing He and Kim Marriott describe a non-interactive system for constrained graph layout in which the constraints are used to specify further requirements beyond a classical batch layout algorithm's aesthetic criteria [71, 100]. They use three different layout modules and augment them with a constraint solver to enforce the user-specified simultaneous linear equality and inequality constraints. Davidson and Harel use simulated annealing to layout graphs nicely, but do not permit extra general constraints on the nodes in the resulting figure [35].

GLIDE [122] is an interactive system for graph layout that uses constraints in the form of Visual Organization Features (VOFs). The VOFs it supports (inherited from earlier work by Marks) include alignment, even spacing, sequence, cluster, T-shape, zone, symmetry, and hub-shape [36, 89]. Phantom nodes (similar to Gleicher's alignment objects) are used as alignment guides. All of these constraints specify local relationships among small groups of nodes.

Other interactive graph layout systems do not include any general constraints but simply provide an interactive means of viewing and manipulating graphs laid out through conventional algorithms [74, 73].

2.3.3 Visualization

Visualization systems provide pictures for abstract data. These visual representations permit viewers to exploit their perceptual skills in exploring data. Graph layout (see Section 2.3.2) is one well-studied domain of visualization. Interactive visualization systems can use constraints to aid in producing semantically meaningful pictures.

TRIP (TRanslate Into Pictures) [88] and its successors TRIP2, TRIP2a, TRIP3D, TRIP3, and IMAGE [135] are all frameworks for visualizing abstract data. The TRIP systems provide mapping rules to translate between an Abstract Structure Representation (ASR), which is derived from an Application Representation (AR), and a Visual Structure Representation (VSR). The VSR level includes graphical objects along with geometric constraints. Some constraints span multiple objects: horizontal/vertical, spacing, and averaging; another constraint specifies the position of objects (the *at* constraint). Finally, there are graph-layout constraints for adjacency and for drawing edges to connect two nodes in the VSR. From the VSR, a picture representation (PR) is generated by solving the constraints (using the COnstraint-based Object Layout system, or COOL). Constrained editing of the resulting picture is not permitted. The TRIP systems' constraints are very similar to those provided by the GLIDE graph layout system (see Section 2.3.2).

The Wand visualization system embeds ICOLA (Incremental Constraint-based Object Layout Algorithm) [116]. Wand's architecture is similar to TRIP, though it specifically targets visualization of logic program execution. ICOLA provides only linear inequality constraints—to enforce that two object attributes are equal, one must use two non-strict inequalities. ICOLA's constraint language allows higher-level, "aliased," constraints that map into one or more of four basic constraints: `left_of`, `horizontal_distance`, `above`, and `vertical_distance`. The fifth basic constraint, `connected`, draws an arc or edge between two objects (as did the similar procedural "connects" constraint in TRIP). The DOODLE (Draw an Object-Oriented Database LanguagE) [34] system provides similar functionality and constraints but provides a visual rather than textual specification language.

Penguins [28] is an intelligent diagram editor construction toolkit. It is to drawing editors what YACC is to parsers. The Penguins system uses constraints in two separate ways. First, it uses them for visual parsing, using the theory of constraint multi-set grammars (CMGs) [99, 27]. After building an internal abstract representation of a picture, editors created using Penguins then permit direct manipulation of the picture while interactively maintaining the constraints. Penguins-generated drawing editors permit arbitrary linear equality and inequality constraints.

2.3.4 Animation

Animation, like graph layout, is an especially challenging domain for the application of constraints. Much of the work on using constraints with animation systems is for non-interactive solvers. Constraint based motion adaptation [56], space-time constraints [144], and motion interpolation methods [23] all address solving huge multi-frame animation systems over time to provide meaningful character or object movement subject to certain desires. These are batch systems whose computation expense is justified in light of the resources required for subsequently rendering the frames of the animation.

Numerous visualization systems, including TRIP (see Section 2.3.3), and widget toolkits (see Section 2.3.5) provide animations meant to provide user-feedback for global changes made to the visual state of the system. TRIP provides "transition mapping rules" which are abstractions of procedures for interpolating between visual representations of two states. Artkit [82] provides a similar "transition" abstraction for animations to be used when an object's state changes. Amulet [105] exploits the constraint solving framework's monitors guarding assignments to slots (i.e., the ability to execute code on every assignment) to provide a similar interpolated animation when a slot's value is set.

Animus [12, 40] uses the ThingLab system [10] and provides animations for its simulations using constraints on time. In Animus, time is treated as a distinguished global variable. Animus provides two time-related constraints: 1) time function constraints which act as a declarative specification of events and responses to those events (similar to the Amulet monitors mechanism, above); and 2) ordinary differential equations for describing continuous motion (similar to Briar's use of differential methods [59], see Section 2.4.5).

Griebel et al. undertook a similar use of constraints for animation within the Pictorial Janus (PJ) visual programming language [65]. Other constraints they provide include linear equalities and inequalities, product equalities and inequalities, point coincidence, and distance constraints. They also provide a circular-disjoint relationship to prevent objects from overlapping.

2.3.5 Other interactive graphical application domains

Other application domains have used geometric constraints with some success. Window layout systems employing constraints include the Constraint Window System (CWS) [43] for Smalltalk, a constraint-based tiled window manager called RTL/CRTL (Research and Technology Laboratories Constrained Rectangular Tiled Layout) [31] for the Sapphire Window System, Trestle from Digital/Compaq SRC [97], and SCWM (Scheme Constraints Window Manager) [5, 6] for the X11 window system (see Chapter 4). These systems permit specification of constraints over the windows regarding their presence, size and location, adjacency and alignment, and hierarchical organization. All systems restrict their constraints to rectangular windows, but face stiff challenges due to the highly dynamic nature of windowing environments where new objects come and go frequently. Vander Zanden et al. discuss ways to cope with these dynamic relationships [140]. Their work is a generalization of Borning's "paths" [10, p. 39–41].

A similar application is web page layout. A prototype Java-based web browser permits page layout and applet layout to be specified using linear equalities and inequalities [15, 100]. For page layout, only rectangular bounding boxes are considered. The browser interactively lays out the page again and again as the enclosing window size changes, preserving the desired constraints. A subsequent effort involved extending the World Wide Web Consortium's (W3C) Cascading Style Sheets recommendation to support constraints. CSS's complex cascading rules can be understood in the context of constraint hierarchy theory, and generalized to permit greater expressiveness (Chapter 5). Constraints can also be used to extend the Scalable Vector Graphics standard (Chapter 6).

User interface widget toolkits are second only to drawing editors in their widespread use of constraints. Numerous widget toolkits, including Amulet [102, 110], its predecessor Garnet [108], and OPUS of the Penguims[2] user-interface management system [81], all provide one-way constraint solvers for relating the components in a widget hierarchy. Bramble [54] is the toolkit with which the Briar (see Section 2.3.1) drawing editor is implemented.

Other constraint-based interactive systems have been used for graphical search and replace [92, 93], curve manipulation independent of representation [46], and color management for windowing interfaces [96].

2.3.6 Summary of application domains

As Table 2.1 shows, the kinds of constraints supported by different systems within an application domain vary widely. The similarities that do exist result more from the underlying solver than from the needs of a particular class of applications (see the following section). Applications also vary greatly in at what level of abstraction the constraints are managed.

Systems including Briar [59], GCE [90], Chimera [91, 93], Pegasus [84], and GLIDE [122] express constraints on complete objects in the system. These constrain attributes such as angles between vectors, Euclidean distances between points and lines, coincidence of a point and an object, and symmetries. Such constraints are the highest level of abstraction provided by any of the systems considered.

Several drawing systems, such as IDEAL [138], Juno [112], and Juno-2 [76], permit specifying constraints on points, and then parameterize drawings based on the locations of those points. After

[2]Not to be confused with Chok and Marriot's Penguins intelligent diagram editor toolkit.

the constraint satisfaction algorithm solves for absolute point locations, procedural code fragments connect lines, draw circles, and otherwise flesh out the drawing. These systems provide greater flexibility in final appearance, but expose a mixed declarative and procedural interface to the end user. (A similar mix of paradigms is used by Animus [40], which is built on ThingLab [10].)

A third general approach, used by Unidraw [72], CDA [114] and Penguins [28, 100], involves expressing numerical constraints on attributes (also called reference points, selectors, aspects, and landmarks) of objects that have an implicit visual representation. The modification of a constrained attribute's value (e.g., a rectangle's northwest corner, or a circle's center) is reflected by updating the position of the corresponding on-screen object. "Internal" constraints often implicitly relate attributes to each other, e.g., relating two corners of a box: box.ne.x = box.nw.x + box.width.

The level of abstraction for specifying constraint relationship is significant because it can decouple the application from the solver. Higher levels of abstractions may rely less on solver dependencies, and they may permit substituting a more efficient or powerful solver without influencing the rest of the system. Limiting the constraints to simple linear numerical constraints may provide a similar benefit, despite being at the opposite end of the abstraction level spectrum.

2.4 Interactive satisfaction algorithms

Different genres of applications do not necessarily provide the same constraint types. For example, drawing programs, though attacking the same problem, have different formulations of the relationships they provide: Gleicher's Briar [59], Kurlander's Chimera [92, 93], Nelson's Juno [112], and Unidraw [72] all provide different constraint mechanisms. Conversely, Unidraw, Penguins [28], and SCWM all expose the same constraint-solving interface, yet they belong to different application domains. The constraints that a specific application permits the user to specify are dependent not on the kind of application but on the underlying solving technology.

Juno provides an excellent example of the relation between constraints permitted and the underlying solving technology. Juno's author, Greg Nelson, explains that he had attempted to provide a fifth predicate, `CC` (counter-clockwise), to disambiguate under-constrained systems. However, because the counter-clockwise relationship translates into an inequality constraint which the Newton-Rhapson solver Juno uses could not easily handle, he discarded that approach and instead chose to exploit a “feature” of Juno’s underlying iterative numerical solver: the solution’s dependence on the initial guess. Thus, Nelson added the ability to provide hints to the solver (which in turn led to the need for `REL` construct) [112, p. 238–239].

This section discusses the various satisfaction algorithms developed for interactive geometric applications, and relates them to one another while pointing out their strengths and weaknesses. Figure 2.1 graphically depicts the relationships among the over twenty different constraint satisfaction algorithms considered here.

2.4.1 Common issues

The issues constraint solvers must address and some of the approaches for remaining efficient are similar. Constraint systems for interactive graphics must deal with under-constrained systems. An under-constrained system has remaining degrees of freedom so multiple possible solutions exist. Constraint hierarchies [14, 17, 48] provide a popular and well-studied means of removing ambiguity by over-constraining the system with constraints at decreasing levels of preference (see Section 2.2).

A related concern for solvers is to maintain spatial stability of the system. When a geometric system is under-constrained, successive solutions are more useful if they are sufficiently similar to prior configurations. A satisfaction technique that alternates between two (visually) distantly related solutions will be confusing to the end user. Supporting the “principle of least astonishment” [65] is a ubiquitous goal for constraint satisfaction algorithms. “Stay” constraints are used in solvers supporting constraint hierarchies to express the desire for things to remain where they are unless some stronger constraint forces them to move. Numeric solvers often handle under-constrained systems by providing spatial stability by minimizing change from the previous solution.

Another commonality among the solver implementations is that they exploit sharing of data structures to provide the equality or coincidence-of-points constraint. Many systems manage their data structures to alias related variables when such a constraint is added and to “explode” the variables back into their unrelated instances when such a constraint is removed. This technique is largely independent of the satisfaction algorithm itself and often provides a substantial perfor-

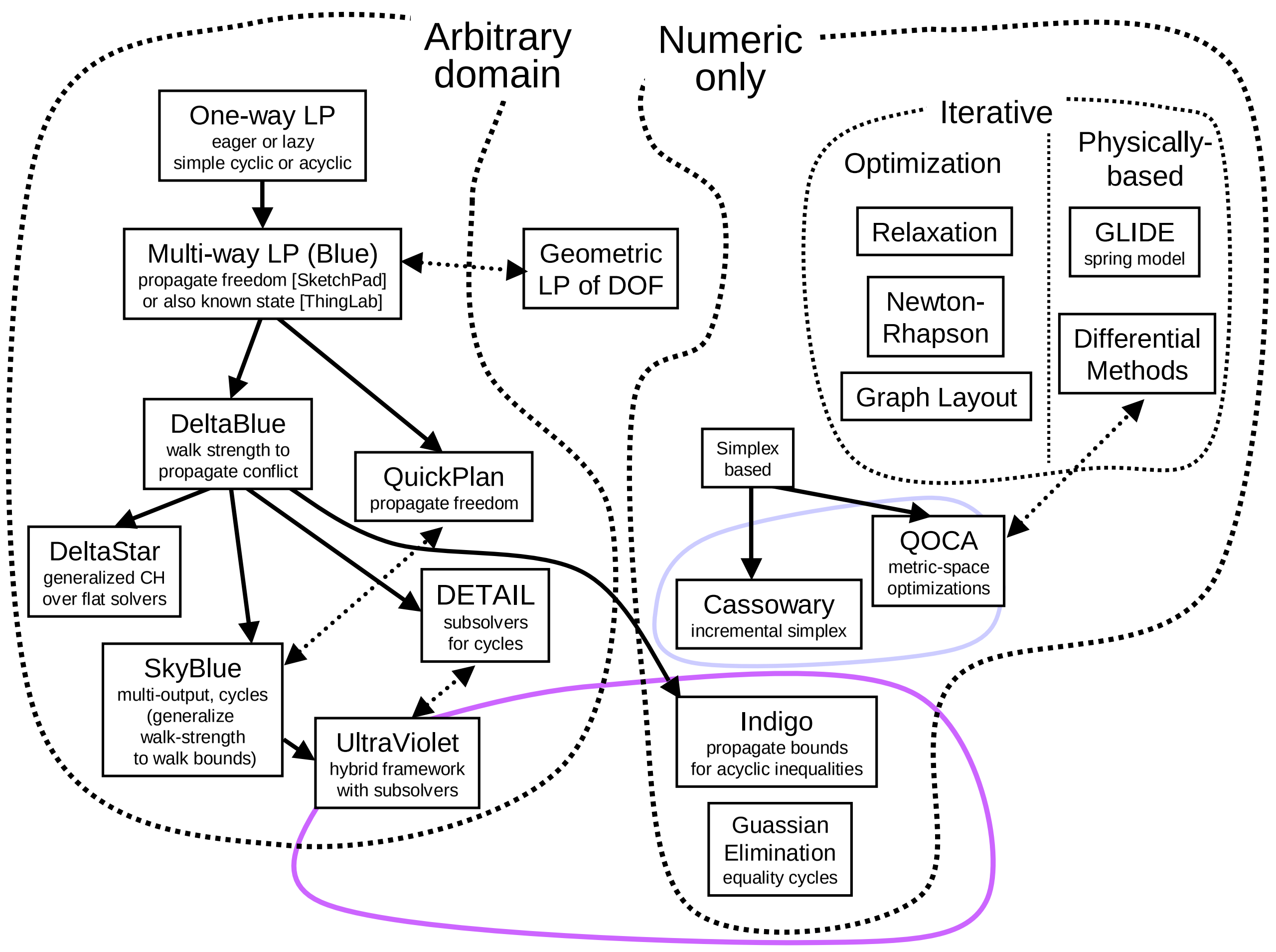


Figure 2.1: Taxonomy of interactive constraint solvers. General classes of algorithms are demarcated by dotted lines, containment of sub-solvers by light solid lines, arrows indicate evolutionary relationships, and proximity roughly correlates with relatedness. Especially closely-related but independently designed systems are connected by bi-directional dotted arrows.

mance improvement by reducing the size of the system, since equality constraints are especially common.

The variable aliasing optimization is generally beneficial because it performs work outside of the constraint solving algorithm and does not change the semantics of the system. Other techniques external to the solver are used to increase expressiveness. For example, the `connects` relationship for graph drawing and visualization systems, which states that two objects in a diagram should be connected by a line, is often not maintained by adding constraints to compute appropriate positions for the edge endpoints. Instead, `connects` is implemented by procedural code that simply draws the requested edge, performing its own computations as necessary. At first glance, this technique seems to defeat the beneficial declarative nature of constraints. However, it is important to separate the semantics of the relationship from the algorithm maintaining it. As long as the system represents the relationship clearly and adheres to its semantics, the underlying implementation need not exploit constraint solving to be of value. These special relationships, by necessity, must be disconnected from the rest of the constraint graph, and the technique can be seen as simply a very restricted domain-specific sub-solver, similar to the sub-solvers used by Ultraviolet [13] or DETAIL [79].[3]

Interactive constraint solvers are often split into a planning, or compilation, stage and an execution stage. During planning, the solver pre-computes all state that will remain fixed throughout a class of executions. These restrictions permit the system to be more efficient during the corresponding solver iterations, and must simply maintain the proper semantics. In some cases, solvers using dynamic languages actually compile the code of the inner loop. The basic idea is similar to loop-invariant code motion and dynamic compilation techniques. Some time is spent in advance, when it is less precious, to increase the performance during the tight interaction and animation loop.

The remainder of this section discusses the several constraint satisfaction algorithms and considers their performance and expressiveness.

[3]In contrast, the Juno systems [76, 112] take this approach to an extreme and permit the user to specify arbitrary code parameterized on the points. Here the benefits of declarative specification are largely lost to provide greater flexibility in drawing.

2.4.2 Local-propagation based solvers

Local propagation is one of the earliest constraint solving techniques and is conceptually very simple. Sutherland's initial formulation of local propagation, the "one-pass method" [134, p. 58–59], is a highly efficient algorithm used whenever possible before falling back to his more general (but slower) relaxation algorithm (see Section 2.4.3). The most significant limitation of propagation-based solving is its inability to consider more than one constraint at the same time. This shortcoming prevents solving simultaneous linear equations and other systems that require manipulations of multiple constraints at once.

Local propagation techniques vary along several dimensions: one-way vs. multi-way constraints; constraint hierarchies vs. flat systems; acyclic only vs. cycles allowed; single-output vs. multiple-output; and equality (functional) relationships only vs. inequalities and other non-functional relationships permitted. See Table 2.3 for an overview of the systems described in this section.

Table 2.3: Overview of local propagation algorithms.

Solver	**Multi-way?**	**C.H.?**[a]	**Cycles ok?**	**Multi-output?**	**Ineqs.?**
Sketchpad (1-pass)	yes	no	no	no	no
ThingLab	yes	no	no	no	no
ARTKit Penguims	no	no	no	no	no
Garnet and Amulet	no	no	partially	no	no
(Delta)Blue	yes	yes	no	no	no
QuickPlan	yes	yes	yes	yes	no
SkyBlue	yes	yes	yes	yes	no
DETAIL	yes	yes	yes	yes	no
Indigo	yes	yes	no	no	yes

[a]"C.H.?" abbreviates "Constraint Hierarchies?".

One-way local propagation constraint solvers

The simplest local propagation solvers are embedded in widget layout kits such as ARTKit's Penguims [81], Amulet [110] and Garnet [108]. These tools perform only one-way solving—a constraint such as $x = y + z + 10$ will be maintained only by setting x (the output variable) and never by setting y or z (the input variables). Although this example constraint is numeric, one of local propagation's strengths is that the relationships may be specified over arbitrary domains—the only restriction is that the output value is determined by a function (e.g., inequality constraints are non-functional and require a more powerful propagation algorithm).

Since one-way constraints are always maintained by evaluating the same assignment method, the satisfaction algorithm must simply decide which constraints' methods must be invoked and in what order. Consider the example in Figure 2.2. The corresponding constraint graph with variables as nodes and directed multi-edges representing constraints appears in Figure 2.3. (The nodes are directed because the constraints are one-way.) After a variable is changed, all downstream variables must be updated by enforcing the constraints in topological order.[4] The one-way local propagation solver propagates values along the constraint graph.

Although one-way constraints are often described in terms of the implementation of the underlying solver, they can be understood declaratively using read-only annotations [14]. A read-only annotation on a variable, often made using a question mark symbol after the variable in a constraint, denotes that the variable cannot be altered as a result of that constraint. Information can flow out from the read-only variable, but not back into it. A one-way constraint, then, is represented as a constraint in which only one variable is not annotated as read-only.

Simple one-way constraint solvers can maintain their relationships using a standard topological sort, based on a depth-first search of the directed constraint graph.[5] Its computational complexity is $O(V + C)$, where V is the number of variables (i.e., nodes), and C is the number of constraints (i.e., edges). Although the structure of the constraint graph only changes when constraints are

[4]Alternatively, downstream variables may be marked invalid, and the constraints can be lazily enforced when a variable's value is requested. Experience suggests that for common layout tasks the cost in maintaining the invalid bit exceeds the savings from unused evaluations [109].

[5]This algorithm only works because we restrict the constraint graph to not contain cycles—more powerful techniques are required if constraints interact (see Section 2.4.4).

$$\begin{aligned}
C_1 &: & m &= \tfrac{(x_1+x_2)}{2} \\
C_2 &: & x_1 &= \text{pointer position} \\
C_3 &: & x_2 &= x_1 + 6 \\
C_4 &: & r &= m^2
\end{aligned}$$

Figure 2.2: Simple set of constraints for local propagation examples.

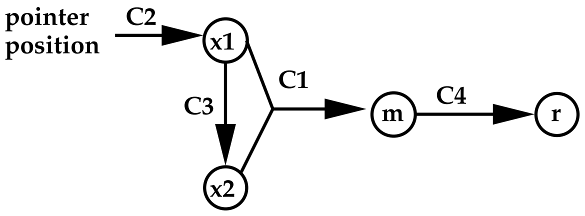


Figure 2.3: One-way (directed) constraint graph for Figure 2.2.

added or removed, the values propagated can change rapidly. For example, when the user is interacting with the system shown in Figure 2.3, x_1 will vary as the user moves the mouse pointer. Local propagation solvers can optimize for this interaction by maintaining the topologically sorted graph and simply traversing it while executing the methods for each new position. This design reflects the previously-mentioned separation of planning (sorting the graph) and executing (firing the constraint-enforcing methods) which we will see again and again. Readers fluent with linear algebra may recognize the planning stage as the ordering of rows and the execution stage as the back-substitution phase in the solving of a system of equations using Gaussian elimination. However, remember that local propagation is not limited to numeric domains—a constraint relationship can, for example, specify that a string s should always contain the printable form of the current color of a circle.

The separation of planning and execution is not essential, but is an optimization. Van Wyk's constraint satisfaction algorithm for IDEAL is a simple work-list approach that propagates state using the current constraint if enough variables are already assigned values, and otherwise delays that constraint by putting it back at the end of the work-list [138]. This worst-case $O(n^2)$ algorithm is a less efficient implementation of local propagation.

$$\begin{array}{llll} m & = & \frac{(x_1+x_2)}{2} & \\ & m & \leftarrow & \frac{(x_1+x_2)}{2} \\ & x_1 & \leftarrow & 2m - x_2 \\ & x_2 & \leftarrow & 2m - x_1 \end{array}$$

Figure 2.4: Predicate and three satisfaction methods for specification of multi-way constraint. In practice, for linear numeric constraints the satisfaction assignments can easily be inferred. For other domains where inverses are harder to compute, the methods may need to be explicitly programmed.

Multi-way constraints and solvers

One-way constraint solvers are exceptionally fast and easy to implement, but are restricted in power. Multi-way constraints permit the constraint solver more freedom in choosing how to satisfy a given constraint. Consider C_3 from Figure 2.2: $x_2 = x_1 + 6$. A one-way constraint solver may only change x_2 in response to changes in x_1, while a multi-way solver is free to set $x_1 \leftarrow x_2 - 6$ instead. Sketchpad [134] and Borning's ThingLab [10] both use multi-way local propagation solvers.

In ThingLab, constraints are specified by predicates and one or more satisfaction methods as in Figure 2.4. A multi-way local propagation algorithm not only has to choose the order by which to satisfy constraints, but also which method should be invoked for each. Figure 2.5 is the (now largely undirected) multi-way constraint graph that corresponds to Figure 2.3. Visually, the additional chore of the multi-way local propagation solver is to put arrowheads on each undirected edge. Not all edges are undirected—C_2, which constrains x_1 to the pointer position, can only be satisfied by changing x_1 so it remains represented as a directed edge.[6] The selection of edge directions corresponds to choosing a satisfaction method for each constraint. A solution to this planning stage assigns directions to all edges such that no variable node has two incoming edges. (Having two incoming edges would signify a conflict: two constraints are competing to affect the same variable's value.)

The earliest solving algorithm for multi-way constraint graphs, the aforementioned one-pass

[6]Even this restriction could be removed if the user's mouse had a motor so it could move around under program control!

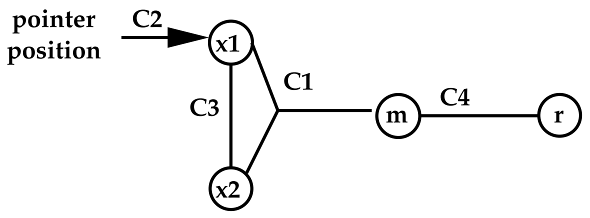


Figure 2.5: Multi-way constraint graph for Figure 2.2.

method, propagates freedom instead of values. Variables only constrained by a single relationship (i.e., those with only a single adjacent edge) are called "free" variables. These variables have enough degrees of freedom that they can be satisfied no matter what the assignments to the other variables are, so their assignment method is chosen to execute last. The edge is directed to select the method that assigns to the free variable, and that method is added to an execution list. Then the free variable node and planned-to-be-satisfied constraint edge are removed from the graph, and the process repeats. In this way, an execution plan is created in reverse order of ultimate execution [134, pp. 58–59] [12, p. 363]. The propagation of values popularized by widget toolkits (see Section 2.4.2) is an extension introduced by Borning [10, p. 67] and independently discovered by Steele and Sussman [133]. While propagation of freedom exploits nodes with *enough* degrees of freedom so they can assigned values last, propagation of known state proceeds towards a solution by finding nodes that have *no* degrees of freedom so they can be assigned values immediately.

With the extra expressiveness of multi-way constraints comes a substantial complication: multiple possible plans may exist to solve the same system. If we remove C_2 from Figure 2.2 there are two possible plans for executing as m is changed (see Figure 2.6). This ambiguity is not just an artifact of the solver, but is fundamental to the problem specification—it is under-constrained. ThingLab had an ad-hoc notion of meta-constraints to control certain aspects of the solver's behavior. For example, the user textually orders the listing of the satisfaction methods to indicate which assignment should be performed when multiple possibilities exist. This type of meta-constraint was later replaced by the now-classic notion of a constraint hierarchy [17, 48] where constraints may be specified at multiple levels of preference.[7]

[7]The DeltaStar solver shown in Figure 2.1 was designed simply to aid research in constraint hierarchies by parame-

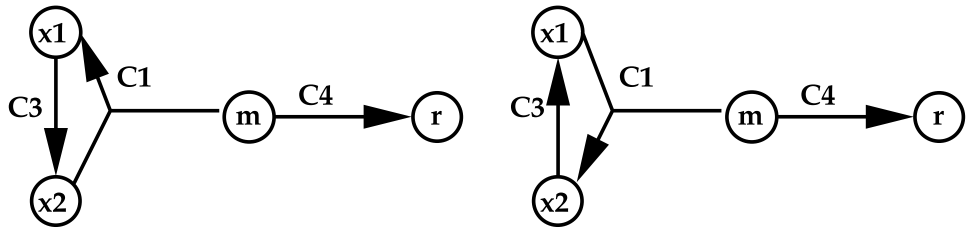


Figure 2.6: Two possible plans for executing Figure 2.5 as *m* changes.

“Blue” is a multi-way local propagation solver that respects constraint hierarchies by finding a best solution [49]. What is optimal is defined in terms of comparators. Blue uses the locally-predicate-better notion to compare two solutions and determine which is best. A locally-predicate-better solution satisfies all the required constraints and successively weaker constraints at least as well as its competing solutions through a given level in the hierarchy, and satisfies at least one more constraint at that level. For example, by the locally-predicate-better comparator, it is more desirable to have a solution that satisfies all required constraints and a single strong constraint rather than one that satisfies all the required constraints and ten (or a million) weak constraints. The comparator is “local” in that it compares solutions constraint by constraint, instead of computing some global measure of how satisfied all the constraints are; the comparator is “predicate” in that all that matters is whether the constraint was satisfied or not, without regard to how closely the constraint is satisfied (i.e., the error). The locally-predicate better solution has the advantage that it permits the use of a greedy algorithm for solving.

“DeltaBlue” is a suitably-named incremental version of the Blue algorithm. It maintains and incrementally updates a solution graph which represents a plan for recomputing variables’ values to satisfy all satiable constraints in a constraint hierarchy subject to the locally-predicate-better comparator.

The key feature of DeltaBlue is its annotating of variable nodes in the constraint graph with their “walkabout strength.” The walkabout strength of a variable is the weakest upstream constraint that could be un-enforced (i.e., removed or re-directed in the solution graph) to permit a different constraint to change the variable’s value. Figure 2.7 shows a simple example from Freeman-

terizing a constraint-hierarchy by an arbitrary flat solver [48].

Benson [49, p. 58]. In particular, variable *D*'s walkabout strength is weak because constraint *C*2 is weak, thus denoting that DeltaBlue would only need to break a weak constraint in order to permit another (stronger) constraint to assign to *D*. Variable *C*'s walkabout strength is strong despite being the output of a required constraint because its input variable *A*'s walkabout strength is only strong; weaker walkabout strengths propagate through stronger constraints.

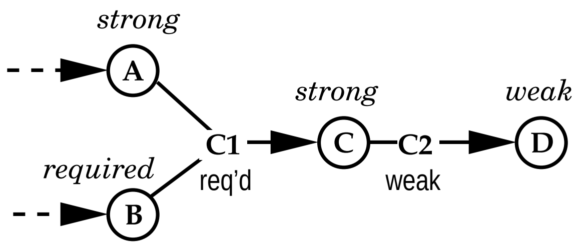


Figure 2.7: Example of walkabout strength assignments to variables. Constraint strengths are below the constraint, current variable walkabout strength assignments are in italics above the variable nodes.

Walkabout strengths encapsulate the global knowledge needed to permit incrementally modifying locally-predicate-better solution plans across constraint additions and removals. The key correlation between walkabout strengths and solutions involves the notion of a *blocked constraint*—a constraint that is unsatisfied but has a strength stronger than the walkabout strength of a potential output variable. The blocking constraint lemma states:

> If there are no blocked constraints, then the set of satisfied constraints represents a locally-predicate-better solution to the constraint hierarchy [49, p. 60]

This blocking lemma suggest the algorithm's strategy—the propagation of conflict. DeltaBlue's incremental maintenance of the constraint graph plan is straightforward [49, 128, 125]. The algorithm's complexity remains $O(V + C)$ (as was simple local propagation). As mentioned before, assigning new values given the same configuration (i.e., execution) is especially fast (only $O(C)$ since at most one method is fired per constraint).

Extensible local-propagation solvers

There are three main limitations of DeltaBlue: 1) it can handle only functional constraints which compute a single value for a variable (e.g., it cannot manage inequalities); 2) it cannot solve cyclic constraint graphs; and 3) all methods must have exactly one output variable. The “Indigo” solver [11] relaxes the first restriction by propagating bounds on value assignments instead of specific values—Indigo binds variables to intervals. This generalization requires the solver to fire multiple interval tightening methods instead of just a single method performing a value assignment. Thus, if the constraints $a \leq 20$ and $a \geq 5$ are applied in that order, Indigo will first tighten a interval to $(-\infty, 20]$ and then to $[5, 20]$. These extra method invocations increase the complexity of the Indigo algorithm to $O(MC)$, where M is the maximum number of variables related by a constraint. The second and third restrictions are relaxed by the enhanced solvers described below.

SkyBlue [126] is a multi-way, multi-output solver. Multi-output functions are useful for decomposing compound data structures and maintaining interacting constraints across multiple variables. The standard example is a two-input two-output constraint relating polar and Cartesian coordinates of a point.

As previously mentioned, cycles in the constraint graph correspond to simultaneous interactions of variables in the underlying problem. For example, the two constraints: $C_1 : x + y = 6$ and $C_2 : x - y = 2$ correspond to the bi-partite constraint graph in Figure 2.8. Because both constraints relate both variables, the graph is cyclic. The primary shortcoming of all the local propagation solvers mentioned above is that they are able to reason about individual constraints only in isolation. When cycles appear in the constraint graph, more sophisticated algorithms must handle the more complex interactions. Various possible sub-solver architectures can be used to manage those complexities.

Another potential cause of cycles is the existence of redundant constraints—although such redundancies can often be eliminated by carefully analyzing the system, forcing the constraint specifier (often the end-user for interactive graphical applications) to avoid redundancies is unacceptable. Alternate views provide another approach to avoiding problems caused by circularities [62, p. 27].

Cycles of linear numerical equality constraints correspond to systems of simultaneous linear

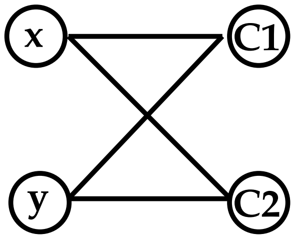


Figure 2.8: Bi-partite constraint graph showing constraints and the variables they relate.

equations, which can be solved by algorithms such as Gaussian elimination (see Section 2.4.4). The framework of a local propagation solver is one approach to handling cycles. The first challenge for the local propagation solver is in recognizing the cycles and invoking domain-specific sub-solvers on the connected subgraphs that local propagation is incapable of solving. As cycle-handling local propagation solvers find subgraphs with cycles, the solvers collapse those nodes into single meta-nodes and use the solution type of the enclosed constraints to assign a domain-specific sub-solver the task of assigning a valuation to the variables contained in the clumps. For the sub-solver to perform its task, it might need to assign values to multiple variables along the frontier where a collapsed meta-node interfaces with the full method graph. Thus, the main solver must permit multiple outputs for a single constraint (the aforementioned necessary but not sufficient condition for cycle-solving local propagation algorithms).

The SkyBlue solver's main contribution is the relaxation of the single-output restriction of DeltaBlue. In the presence of multi-output constraints, walkabout strengths are no longer powerful enough to capture the relevant global information. The SkyBlue algorithm instead computes walkbounds—any strength equal to or weaker than the walkabout strength—and maintains walkbounds incrementally as constraints are added and removed. The algorithm then computes the solution graph by building method vines using a backtracking algorithm [126]. Walkbounds and other optimization techniques help to reduce the needed backtracking substantially, but not completely. The backtracking makes SkyBlue's complexity exponential in the worst case.

QuickPlan [139] is similar to SkyBlue but uses propagation of degrees of freedom (instead of propagation of conflict), searching for free variables and selecting methods to execute in reverse order. As it encounters conflicts planning its solution, it retracts the weakest strength constraint from the graph, saving it on a priority queue (ordered by strength). After the sequence of elimination and retraction steps, QuickPlan tries to re-add the retracted constraints in decreasing order of strength. The QuickPlan algorithm has $O(C^2)$ worst case complexity, it typically runs in linear time (recall that the single-output solver, DeltaBlue, is a linear-time algorithm).

DETAIL [79] is yet another multi-output cycle-solver-capable local propagation algorithm. Its algorithm is similar to the above, and it embeds three sub-solvers: one for locally-predicate-better constraints, one for least-squares-better linear equality systems, and one that uses a physically-based spring model (similar to GLIDE [122]).

Ultraviolet, a meta-solver for invoking sub-solvers, first partitions the top-level constraint graph, and then solves the connected subgraphs independently while communicating through shared variables. Unlike SkyBlue, Ultraviolet is not a solver itself, but only coordinates the actions among its sub-solvers which include Blue (for functional local propagation), Indigo (for numeric inequalities), Purple (for simultaneous linear equalities), and Deep Purple (a partial solver for simultaneous linear equalities and inequalities; cf. QOCA and Cassowary in Section 2.4.4). One key advance of Ultraviolet was determining the order of invocation of sub-solvers to support constraint hierarchies. The outer loop for the satisfaction algorithm is over decreasing strengths of constraints: each subsolver first handles required constraints, then each deals with all of the strong constraints, and so on. Thus, each sub-solver is potentially invoked multiple times [13, p. 7].

Partitioning of the constraint graph is not only useful for increasing expressiveness but also for improving performance. The more sophisticated algorithms that support multi-output and cycles all have super-linear complexity; thus, they may benefit from being subdivided into smaller independent problems. Some evidence suggests that constraints in real applications tend to be modular, and therefore amenable to this kind of decomposition [141].

Geometric Degrees of Freedom Analysis

Kramer's Geometric Constraint Engine (GCE) [90] exploits symbolic analysis of geometric degrees of freedom which insulates the technique from the low-level representation and equations and preserves the intuitive nature of the underlying problem. GCE's solver is given the task of constructing a “metaphorical assembly plan” (MAP) to describe how to satisfy a set of geometric constraints. Although at first examination the technique seems novel and distinct from the other algorithms discussed, at its essence, it is simply a local propagation algorithm. GCE proceeds by searching for free geometric entities, and selecting transformations to assign positions to those entities. It constructs the MAP in reverse order of ultimate execution, exactly as Sutherland’s original local propagation algorithm for Sketchpad did. (In the forward direction, this can be seen as the propagation of rigidity; Brunkart calls this method contraction [24].)

Kramer’s propagation of geometric degrees of freedom is complicated by its need to infer the appropriate geometric transformation to fix (i.e., make rigid) a specific previously-free motion. In a simple local propagation system, this requires only the evaluation of a pre-specified function, perhaps with some simple inference for multi-way numerical constraints. The planning for the MAP [8] and the need to maintain a numerical model along with the symbolic geometric model distinguish GCE’s geometric degrees of freedom analysis from other forms of local propagation.

Local propagation strengths and weaknesses

Maximal efficiency and the ability to handle constraints over arbitrary domains are the primary strengths of local propagation algorithms. As previously mentioned, the key weakness of local propagation algorithms is their inability to consider multiple constraints simultaneously. These cycles must be managed by domain-specific techniques; more sophisticated local propagation solvers manage sub-solvers to provide this capability.

2.4.3 Iterative numeric solvers

Iterative numeric solvers have been used in constraint solving systems ever since Sketchpad. Their primary strength is that they are very general and thus widely applicable. In particular, numeric techniques permit solving simultaneous non-linear constraints (such as maintaining equal lengths or distances) which arise often in geometric applications. Sutherland’s Sketchpad exploits the representation of constraints directly in terms of the error, thus reducing constraint satisfaction to the well-studied problem of functional minimization. However, since iterative optimization techniques are often slow (their computational complexity is generally at least quadratic and the constant factors are relatively large), they have not been used much for interactive applications. Sutherland's relaxation technique is only used when his one-pass local propagation algorithm fails to find a solution [134, p. 57]. ThingLab also relies on relaxation as a backup technique when faster methods fail [10, p. 68–69]. This approach may gain in importance as computing hardware becomes faster and faster.

Recognizing that constraint solving via iterative numeric techniques can be viewed as classical functional optimization opens up a world of techniques [45]. Relaxation is simply an iterative hill climbing (or equivalently a gradient, or steepest descent) algorithm. These optimizers are reasonably good at finding a local minimum independent of the initial guess, but converge only linearly to the local minimum. More importantly, the technique only finds a *local* minimum, ignorant of the global search space.

Other systems' solvers, including Juno [112] and Juno-2 [76], use (multidimensional) Newton-Rhapson iteration to exploit derivative information. Some systems use automatic differentiation to relieve the user from specifying derivatives [58], while others simply limit the set of functions known to the underlying solver. Juno-2's solver performs numerous optimizations, including propagation of known state, unification of pair constraints, unpacking pair constraints to primitive constraints (thus separating numeric constraints from non-numeric constraints), and re-packing (reducing the number of constraints and unknowns before passing them along to the Newton-Rhapson solver). Newton-Rhapson converges quadratically (faster than gradient descent), but relies on a sufficiently accurate initial guess and an invertible Jacobian.[8] The Levenberg-Marquardt method [25] dynamically weights a combination of Newton-Rhapson and gradient descent, permitting solvers to exploit the faster convergence of Newton-Rhapson once in the proximity of a local minimum; this hybrid solver is used in maintaining the constraints in the Chimera editor [92, 93].

Besides being relatively inefficient, iterative numeric solvers pose other problems for constraint solvers for interactive graphical applications. Because of their iterative nature, it is sometimes difficult to tell if convergence is just slow or if the system is truly unsatisfiable. Also, because the methods are local optimizers, the solution converged upon depends on the initial solution. Slight

[8]The Jacobian is the matrix of partial derivatives.

changes in the initial conditions can result in finding radically different solutions. This behavior may not be what the end-user expects.

Difficulty of implementation is yet another hindrance to the spread of iterative solving techniques. Coding iterative numeric constraint solvers is not for the numerically-challenged. Various numerical stability problems (e.g., singular or nearly-singular matrices) crop up repeatedly. Only with an arsenal of carefully combined sophisticated algorithms (e.g., singular value decomposition can be useful for under-constrained systems in place of Gaussian elimination) can the techniques perform computations robustly. Bramble and its "Snap-Together Mathematics" package provides some of these tools in the context of Whisper—an extensible Scheme-like language [54, 58].[9]

One of the more promising uses of iterative techniques is exemplified by the GLIDE interactive graph layout system [122]. GLIDE gives up on the difficult problem of global optimization of a graph layout. Instead, it focuses on exploiting the solver's strength—local minimization—and combining that with the interactive user's strength—global layout. To make this combination most useful, the numerical solver is physically-based, using a generalized spring model. The visual organization features (see Section 2.3.2) are mapped to sets of spring-like objects[10] among nodes. The energy minimization function uses varying spring-constants to provide preferential constraint satisfaction similar to the weighting of errors in a single level of a constraint hierarchy (as in QOCA [100, 18]).

GLIDE's iterative solver then simulates its physical model, trying to minimize the energy of the system. It uses Euler's method to compute the position and momentum of each node. During solving iterations, the configuration is animated, and a kinetic energy threshold terminates the iterations once the system is stable, until the next user interaction. The animation reinforces the spring metaphor and aids the user in establishing an accurate mental model. The collaborative approach of constraint solvers augmenting user interaction through physical models and understandable metaphors seems to address many of the difficulties iterative techniques otherwise experience. Differential methods are another physically-based technique; they are discussed in Section 2.4.5.

[9] SCWM uses a similar extensible language called Guile (Chapter 4).

[10] They are not physically-precise springs (i.e., they can violate Hooke's Law) because some may have only a repulsive force.

Combinations of these physically-based approaches may prove interesting.

The constrained graph layout solver [71] takes a non-interactive approach, and attempts to perform global optimization of a spring-model energy function (a simplified aesthetic criterion) subject to arbitrary linear equality and inequality constraints. The first cost function considered, Model A, is a non-polynomial metric suggested by Kamada [71, p. 221]. Because computing partial derivatives for this function is expensive and because the second-derivatives are not continuous, Marriott and He propose Model B, a polynomial approximation to the first model. Their expectation is that the smoothness in the partial derivatives will permit better behaved solutions. The primary limitation of Model B was weakening of inter-node repulsive forces; this can result in layouts in which nodes overlap.

Marriot and He's layout algorithm is based on an active-set technique [45], which is useful for optimizations constrained by inequalities. The active set method is also used by QOCA [18], and it is related to the simplex algorithm (see Section 2.4.4). As with simplex, finding an initial feasible solution for the active set method for graph layout requires additional work. Kamada's unconstrained algorithm simply puts the n nodes onto a regular n-polygon. Marriott and He augment this to find the least-squares closest solution which is feasible—this is a quadratic (and thus convex) programming problem, so any of the numerous applicable techniques suffices.[11] Their algorithm, while only of polynomial complexity, is slow on even small problems (a twenty node graph requires 33 seconds of computation on a 486DX/2-66).

Because of their generality, iterative numeric techniques are a useful method of last resort, and some of their uses for physical simulations seems promising. Additionally, these techniques may benefit from advancements in computing power that could make their asymptotically slower algorithms more practical.

2.4.4 Direct numeric solvers

Direct numeric constraint solvers avoid the difficulties of iterative numeric solvers by attempting to find an exact solution through symbolic manipulation of the constraint equations. As with iterative

[11]Tree layout as formulated by their Model C is also only a quadratic programming problem. Again, Marriott and He use a variant of the active set method.

numeric solvers, the domain for constraints is restricted to real numbers. Additionally, to make solving manageable, direct numeric solvers further restrict the constraints they allow. The most common restriction is to permit only linear equality relationships—linear systems of equations have numerous applications, and there exist efficient algorithms for solving them.

The simplest algorithm for solving simultaneous linear systems of equalities is Gaussian elimination. In the matrix form of the equations, Gaussian elimination corresponds to computing the row-reduced form. From this triangular form, a value for a variable can be read off a row directly, then that variable's value can be substituted into the other equations, and the process repeats. This back-substitution corresponds to the local-propagation solver's behavior during the execution phase, while its planning phase corresponds to choosing the ordering of rows for the back-substitution. Computing the row-reduced form allows simultaneous systems (i.e., those involving cycles in the constraint graph) to be handled. If there are no cycles, then Gaussian elimination is unnecessary and simple propagation of known-state (as local propagation solvers do) suffices.

Gaussian elimination only finds a unique solution when a system is fully specified (i.e., the corresponding matrix is of full rank) as with systems of n independent equalities with n variables.[12] In constraint systems, however, under-constrained systems are far more common.

Simplex algorithm

As mentioned earlier, under-constrained systems require a means of disambiguating possible solutions. As we have seen, constraint hierarchies are a useful way of declaratively specifying preferred solutions, and they can be implemented by optimizing a global error metric. Dantzig's famous simplex algorithm is a simple technique for optimizing a linear function subject to linear equality and inequality constraints [101, pp. 63–72]. Although simplex works directly only on equalities, an arbitrary inequality can be handled using a non-negative slack variable. For example, $x \geq y$ becomes $x = y + s_1$, where the slack variable $s_1 \geq 0$. This last non-negativity restriction on s_1 applies to all variables in the simplex tableau (the matrix on which the algorithm operates).

The simplex algorithm is split into two phases. Phase I finds an initial solution to the con-

[12]Independence assures that rows provide useful information; rows that are linear combinations of other rows are not helpful in constraining the system.

straints, and phase II finds an optimal solution. Consider the four constraints:

$$1 \leq x \;\wedge\; x \leq 3 \;\wedge\; 0 \leq y \;\wedge\; 2y - x \leq 3$$

These inequalities correspond to the darkened region of Figure 2.9. Since the optimization function is linear, its optimal value must occur at a vertex of the enclosing polygon. In terms of the picture, phase I finds any of those vertices (called a basic feasible solution), while phase II involves pivoting the system to move between adjacent vertices, systematically and efficiently searching for the optimal solution. See Section 3.2.2 for details.

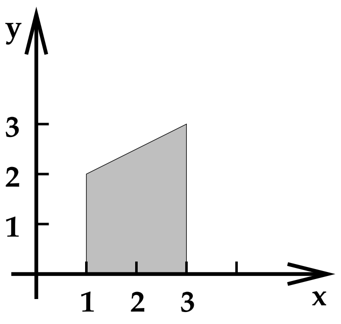


Figure 2.9: Simplex optimization problem [101, p. 64]

QOCA and Cassowary: Incremental simplex

In Borning's spectrum of solvers, a variant of the simplex algorithm is dubbed "Orange," and an incremental version, DeltaOrange, is mentioned as a research direction [49]. Cassowary and QOCA are two variants of an incremental simplex algorithm [100, 18].

As one would imagine, Cassowary and QOCA are very similar to the batch simplex algorithm. Both lift the restriction of non-negativity on all variables by using two tableaus: an unrestricted tableau and a restricted, simplex tableau. Only the variables in the simplex tableau have the non-negativity restriction.[13] Cassowary and QOCA are incremental in that they permit adding

[13]Because the optimization phase requires this restriction to find adjacent vertices, that phase of the algorithm is restricted to only the simplex tableau.

and removing constraints while maintaining basic feasible solved form. Both algorithms proceed identically until the optimization (of the original problem) phase. Adding a constraint involves re-expressing inequalities as equalities, using an artificial variable to represent the error, and minimizing that error in the added equation. If the error cannot be minimized to zero, the new constraint is inconsistent and an exception is thrown. This process is essentially an incremental version of simplex's Phase I.

Removing a constraint is a bit more complicated because the effects of a single equation are spread throughout the tableaus as they are manipulated. This difficulty is overcome by creating a distinct "marker" variable for each constraint added to the tableau.[14] A marker variable indicates the effect of a constraint on the tableau, and that constraint can be removed by pivoting to make the marker variable basic, and then removing that row. Clearly, removing a constraint cannot make the system infeasible, so the tableau remains in basic feasible solved form.

The final incremental operation the algorithms provide is the ability to change the constant of certain constraints. Often this is done for simple constraint equations which track, e.g., pointer movement. In Cassowary, these kinds of constraints are called "edit constraints." Usually changing an edit constraint's value requires only changing specific constants in the tableau. Occasionally, the change will make the system infeasible; visually, this occurs when graphical objects first bump up against or leave other objects. The bumping point corresponds to a new configuration at an optimal but infeasible solution (i.e., it is an optimal point *outside* of the shaded region in Figure 2.9). When this occurs, the dual simplex algorithm is used to restore feasibility—to move from an infeasible and optimal solution to a feasible and still optimal solution. The efficiency of this operation is essential for interactive graphical applications to maintain fluid animation while the user directly manipulates the system.

The primary difference between QOCA and Cassowary is in how they choose among possible solutions to the constraint hierarchy—which comparator they use. Cassowary uses the locally-error-better metric, while QOCA uses the globally-least-squares-better comparator.

To find locally-error-better solutions, Cassowary computes an error for each non-required con-

[14]In implementations, other variables guaranteed to appear only in a single equation (e.g., slack variables) are overloaded to serve as marker variables.

straint equation. Since the error can be either positive or negative, we need two error variables associated with each equation: δ^{+} and δ^{-}. Two variables are required because the simplex algorithm's non-negativity restriction on variables would otherwise prevent the representation of negative errors. The optimization function is then chosen to be a weighted sum of these error variables. The weighting is determined by the preferences of the constraints using a constraint hierarchy specification. To ensure we satisfy one strong constraint in preference to numerous weaker constraints, the objective function uses symbolic weights and lexicographical ordering. Generally, weak stay constraints are added to force each variable to remain where it is; these constraint values are then updated after each optimization of the system so that future optimizations will keep the variables' values the same unless they must be altered by some stronger constraint.

Instead of using preferences on constraints to control the optimization function, QOCA uses a global least-squares better comparator. QOCA's goal is to minimize the weighted sum of the squares of the error of each variable relative to its desired position. For this technique, each variable has a preferred location (analogous to the stay constraint for Cassowary) and a numerical weight of how strong the preference is. QOCA then must solve the quadratic programming problem of minimizing $\sum w_i \delta_i^2$, where w_i is the weight of the ith variable, and δ_i is the error from its effectively desired location.

Convex quadratic programming is well-studied and two algorithms have been considered for use by QOCA: the active set method (currently used), and linear complementary pivoting. Both algorithms are related to the simplex technique.

The active set method [45] is an iterative technique that maintains an active set of the equality constraints and the subset of the inequalities that are tight in the sense that their slack variables have value 0 in the current solution. At each step in the iteration, we either move as far towards an optimal solution as possible while maintaining feasibility relative to some new inequality that we add to the active set, or we move more toward optimality by removing a constraint from the active set. When the active set can no longer by modified, we are at an optimal, feasible solution [18].

Linear complementary pivoting is another approach to solving convex quadratic optimization problems. This technique works by first introducing dual slack variables and dual variables. Each of these new variables is complementary to an existing variable in the primal (original) problem:

the dual slack variables to the primal parametric variables and the dual variables to the primal basic variables. Then we augment the tableau of the primal problem with equations relating the dual slack variables to the sum of partial derivatives of the objective function with respect to the parametric variables and the dot product of rows of the primal problem with dual variables. By maintaining the property that complementary variables may not both be positive while pivoting this combined problem repeatedly, we achieve a feasible and optimal solution to the primal problem. Because the partial derivative of the quadratic objective function is linear, we can use simplex as a solution technique (this is similar to Gleicher's differential method technique—see Section 2.4.5). Borning et al. provide an illustrative example [18].

QOCA gives up the ability to express arbitrary constraints at varying preferences. It instead guarantees a variable-weighted least-squares-better solution to the under-constrained problem. This comparator is especially useful in geometric applications since it tries to place objects as close as possible to where they are desired to be. The weighting function can be used to control which objects should be placed closest to their desired positions. QOCA's least-squares comparator also comes at the price of using additional numerical techniques (computing the derivative symbolically) and further implementation complexity. Performance for both QOCA and Cassowary is good, handling re-solves (i.e., edit constraint changes) of systems of around 600 constraints and 700 variables in under 30ms on average [100].

One of the shortcomings of Cassowary and QOCA is that the performance of a re-solve can vary dramatically. In the common case, it is very fast, but a pivot of the tableau can result in a noticeable delay and an interaction that has an undesirable "jerky" quality. Another approach to solving systems can avoid this difficulty: one can instead compile constraint-free code for a specific interactive system that then runs very quickly and predictably [70]. This possibility trades generality for performance and is useful in applications where the constraint system is static and known well in advance (e.g., delivery of a geometric-demonstration applet across the web). Another benefit of compiling constraints is that the constraint solver need never be distributed, thus avoiding sharing potentially proprietary technology and simplifying distribution of the software.

2.4.5 Differential methods

Gleicher's Bramble drawing program permits quadratic constraints to be expressed and solved efficiently by using an approach he calls differential methods. The differential method technique is enabled by limiting the problem only to *maintenance* of constraints that *already* hold. All other systems discussed use a "specify-then-solve" methodology where the solver is responsible both for producing an initial solution and for maintaining that solution as the system is perturbed. Instead, Bramble requires that the user initially establish the desired relationship before adding the corresponding constraint to the solver. Via augmented snap-dragging, the user is aided in establishing a desired relationship while simultaneously adding the constraint to be maintained (see Section 2.3.1).[15]

Offloading the establishment of the initial configuration from the constraint solver simplifies the solver's task—instead of maintaining relationships regarding the absolute positions of objects, differential manipulation relates the *motion* of objects. Since the motion of an object is described by its derivative with respect to time, quadratic relationships of positions are reduced to linear relationships of derivatives. Maintenance of linear constraints is a far easier job (see Section 2.4.4). The linear systems are solved to minimize the derivative of the configuration. The one added step in Briar is to solve an ordinary differential equation after solving for the unknown time derivative; Euler's method is one simple technique for computing an absolute position from the initial conditions and the derivative.[16]

Another key benefit of differential manipulation is that it permits choosing an underlying representation of an object's state independent of the user-interface controls for that object. For Gleicher's Through-the-Lens Camera Control (TLCC), he expresses the three-dimensional location and orientation of the camera via quaternions [130] which are much better behaved numerically, but far less intuitive to the user, and thus unsuitable for exposing directly [57]. Gleicher has also applied differential manipulation techniques to character animation systems [56].

[15]Adding an arbitrary not-already-satisfied constraint to the system may be confusing to the user if enforcing the constraints requires a global rearrangement of the layout. Additionally, by knowing that the constraint is already satisfied, the solver need not worry about over-constrained systems [59].

[16]Animus also uses differential equations for the specification of continuous motion of objects being animated [12, 40].

2.5 Summary

Interactive graphical applications have explored using constraints for over thirty-five years, yet none are completely successful, and numerous challenges remain. Two important problems not yet well-addressed and not considered in-depth here include debugging constraints and reuse of solvers. Debugging constraint systems is challenging, and must be made easier for users [55, 124]. Constraint solving libraries must be developed so that the implementation effort for application programmers is minimized—software engineering research on system architectures [103] and solvers that stress simple and efficient[17] interfaces [100] can be exploited to improve this situation. The Cassowary toolkit is one attempt to fill this gap (Chapter 3).

This chapter surveys several interactive graphical application domains that use constraint systems. Table 2.1 shows that several kinds of constraints are especially relevant for geometric applications, but that the constraints provided by an application are highly influenced and restricted by its underlying solver. Thus, increasing expressiveness of constraint solvers is a primary concern. The fundamental challenge is to not sacrifice performance while expanding the class of constraints that solvers handle. Figure 2.1 relates all of the solvers surveyed here by their techniques and expressiveness. Understanding the evolution of techniques, recognizing the similarities among the approaches, and considering novel combinations of various systems exposes many areas for future work.

Extensible solvers are a useful and flexible mechanism for exploiting domain-dependent optimizations while retaining generality. In particular, Ultraviolet [13] provides a useful framework for embedding solvers, but does not have an integrated, fully general linear equality and inequality sub-solver.

Physically-based systems such as the spring-layout of GLIDE [122] and differential methods of Briar [59, 57] demonstrate advantages of using simulation-based solvers. Animating other solvers' solution processes may be beneficial to creating a seemingly more responsive system, and providing a more understandable solution due to the physical metaphor. The collaborative aspect of these

[17]The Janus In Motion (JIM) application communicates with its Parcon constraint solver via Unix named pipes. Although this design certainly decouples the constraint solver from the application, the performance cost is hard to accept in an interactive application [65].

two systems is also instructive: constraint solving technology need not do everything. Users are good at direct manipulation and interactive systems can permit leveraging those abilities.

Interactive graphical applications can benefit dramatically from fast, expressive, understandable, and reusable constraint solvers. Improving constraint satisfaction algorithms in these directions is important if we are to fully exploit the benefits that the declarative nature of constraints can provide.

Chapter 3

THE CASSOWARY CONSTRAINT SOLVING ALGORITHM AND TOOLKIT

This chapter describes a newly-developed constraint solving algorithm called "Cassowary." The algorithm was designed principally by Alan Borning and Peter Stuckey, and much of this chapter is a heavily revised, corrected, and updated version of their original description [18]. The design and implementation of the re-usable toolkit for embedding the solver into real-world applications is a contribution of this dissertation.

3.1 Introduction

Linear equality and inequality constraints arise naturally in specifying many aspects of user interfaces, especially layout and other geometric relations. Inequality constraints, in particular, are needed to express relationships such as "inside," "above," "below," "left-of," "right-of," and "overlaps." For example, if we are designing a Web document we can express the requirement that figure1 be to the left of figure2 as the constraint $\mathsf{figure1.rightSide} \leq \mathsf{figure2.leftSide}$.

In a system used for graphical layout, it is important to be able to express constraints that are just preferences as well as constraints that are hard requirements. For example, we must be able to express a preference for stability when moving parts of an image: things should stay where they were unless there is some reason for them to move. A second use for preferred constraints is to cope gracefully with invalid user inputs. For example, if the user tries to move a figure outside of its bounding window, it is reasonable for the figure just to bump up against the side of the window and stop, rather than causing an exception. A third use of non-required constraints is to balance conflicting desires, for example in laying out a graph.

Efficient techniques are available for solving such systems of linear constraints if the constraint network is acyclic. However, in trying to apply constraint solvers to real-world problems, we find that the collection of constraints is often cyclic. Cycles sometimes arose when the programmer unwittingly added redundant constraints—the cycles *could* have been avoided by careful analysis. However, the analysis is an added burden on the programmer. Further, it is clearly contrary to the spirit of the whole enterprise to require programmers to be constantly on guard to avoid cycles and redundant constraints. After all, one of the goals in providing constraints is to allow programmers to state what relations they want to hold in a declarative fashion, leaving it to the underlying system to enforce these relations. For other applications, such as complex layout problems with conflicting goals, cycles seem unavoidable. A solver that can handle cycles of both equality and inequality constraints is thus highly desirable.

3.1.1 Constraint hierarchies and comparators

Since we want to be able to express preferences as well as requirements in the constraint system, we need a specification for how conflicting preferences are to be traded off. *Constraint hierarchies* [14] provide a general theory for the semantics of constraint systems (Section 2.2). In a constraint hierarchy each constraint has a strength. The required strength is special, in that required constraints must be satisfied. The other strengths all label non-required constraints. A constraint of a given strength completely dominates any constraint with a weaker strength—the strong constraint must be satisfied as well as possible before the weaker constraint can have any effect on the solution. In the theory, a *comparator* is used to compare different possible solutions to the constraints and select among them.

Within this framework a number of variations are possible. One decision is whether we only compare solutions on a constraint-by-constraint basis (a *local* comparator), or whether we take some aggregate measure of the unsatisfied constraints of a given strength (a *global* comparator). A second choice is whether we are concerned only whether a constraint is satisfied or not (a *predicate* comparator), or whether we also want to know how nearly satisfied it is (a *metric* comparator). Constraints whose domain is a metric space such as the real numbers can have an associated error function. The error in satisfying a constraint is 0 if and only if the constraint is satisfied, and becomes larger the less nearly satisfied the constraint is.

For inequality constraints it is important to use a metric rather than a predicate comparator [11]. Thus, plausible comparators for use with linear equality and inequality constraints are *locally-error-better*, *weighted-sum-better*, and *least-squares-better*. For a given collection of constraints, Cassowary finds a locally-error-better or a weighted-sum-better solution. (The related QOCA algorithm finds a least-squares-better solution, which strongly penalizes outlying values when weighing constraints of the same strength [18].) The locally-error-better comparator is more permissive in that it admits more solutions to the constraints. Also, it is generally easier to develop efficient algorithms to find a locally-error-better solution because these can often be found using greedy algorithms.

3.1.2 Adapting the simplex algorithm

Linear programming is concerned with solving the following problem:

> Consider a collection of n real-valued variables $x_1, \ldots, x_n$, each of which is constrained to be non-negative: $x_i \geq 0$ for $1 \leq i \leq n$. Suppose there are m linear equality or inequality constraints over the x_i, each of the form:
>
> $a_1x_1 + \ldots + a_nx_n = b$,
> $a_1x_1 + \ldots + a_nx_n \leq b$, or
> $a_1x_1 + \ldots + a_nx_n \geq b$.
>
> Given these constraints, find values for the x_i that minimize (or maximize) the value of the *objective function*
>
> $c + d_1x_1 + \ldots + d_nx_n$.

This problem has been heavily studied for the past fifty years. The most commonly used technique for solving it is the simplex algorithm, developed by Dantzig in the 1940s, and there are now numerous variations of it. Unfortunately, existing implementations of the simplex algorithm are not readily usable for user interface applications.

The principal difficulty is incrementality. For interactive graphical applications, we need to solve similar problems repeatedly, rather than solving a single problem once. To achieve interactive response times, fast incremental algorithms that exploit prior computations are needed. There are two common cases that algorithmic changes should try to improve. First, when moving an object with a mouse or other input device, we typically represent this interaction as a one-way constraint relating the mouse position to the desired *x* and *y* coordinates of a part of the figure. For each screen refresh, we must re-satisfy the same collection of constraints while varying only the mouse location input. The second common need that incremental algorithms can optimize is when editing an object in a complex system. Ideally, when adding or removing a small number of constraints, we would like to avoid re-solving the entire system. Although the performance requirements for this case are less stringent than for the first case, we still wish to increase performance by reusing as much of the previous solution as possible.

Another important issue when applying simplex to user interface applications is defining a suitable objective function. We must accommodate non-required constraints of different strengths which is analogous to multi-objective linear programming problems. Also, the objective function in the standard simplex algorithm must be a linear expression; but the objective functions for the locally-error-better, weighted-sum-better, and least-squares-better comparators are all non-linear. For Cassowary, we avoid the least-squares-better comparator and use a quasi-linear objective function for the weighted-sum-better comparator (Section 3.2.3).

Finally, a minor issue is accommodating variables that may take on both positive and negative values, which is generally the case in user interface applications. (The standard simplex algorithm requires all variables to be non-negative.) Here Cassowary adopts efficient techniques developed for implementing constraint logic programming languages (Section 3.2.1).

3.1.3 Overview

I describe the Cassowary algorithm for incrementally solving linear equality and inequality constraints for the locally-error-better and weighted-sum-better comparators mentioned above. In Section 3.2, I present the algorithm's techniques for incrementally adding and deleting constraints from a system of constraints kept in *augmented simplex form*, a type of solved form. I also explain the procedures for incrementally solving hierarchies of constraints when an object is moved.

The Cassowary algorithm originally had a proof-of-concept implementation in Smalltalk. Now my Cassowary Constraint Solving Toolkit includes that code, along with C++, and Java versions [3]. The library performs very well, and a summary of results is given in Section 3.5. The algorithm is straightforward, and a re-implementation based on this chapter is reasonable, given a knowledge of the simplex algorithm. Section 3.3 contains details of my implementations of the Cassowary algorithm, and Section 3.4 discusses some subtleties of the comparators used for optimization.

3.2 Incremental simplex

I now describe Borning et al.'s incremental version of the simplex algorithm, adapted for the Cassowary algorithm for interactive graphical applications. The description will use a running example, illustrated by the diagram in Figure 3.1.

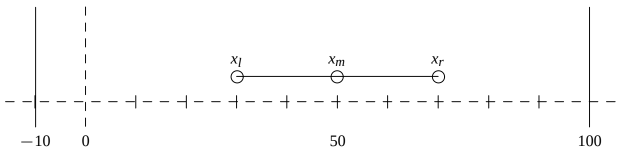


Figure 3.1: Simple constrained picture

The constraints on the variables in Figure 3.1 are as follows: x_m is constrained to be the mid-point of the line from x_l to x_r, and x_l is constrained to be at least 10 units to the left of x_r. All variables must lie in the range -10 to 100. (To keep the presentation manageable, we deal only with the x coordinates. Adding analogous constraints on the y coordinates is straightforward but would double the number of constraints in the example.) Since $x_l < x_m < x_r$ in any solution, we simplify the problem by removing the redundant bounds constraints. However, even with these simplifications the resulting constraints have a cyclic constraint graph and cannot be handled by methods such as Indigo [11].

The constraints described above are

$$\begin{aligned} 2x_m &= x_l + x_r \\ x_l + 10 &\leq x_r \\ x_l &\geq -10 \\ x_r &\leq 100 \end{aligned}$$

3.2.1 Augmented simplex form

Suppose we wish to minimize the distance between x_m and x_l, or in other words, minimize $x_m - x_l$. (This simple objective function is just used as an initial example; a realistic objective function resulting from incrementally moving a point in the diagram is described in Section 3.2.3.)

The basic simplex algorithm does not itself handle variables that may take negative values (so-called *unrestricted variables*). It instead imposes an implicit constraint $x \geq 0$ on all variables occurring in its equations. Augmented simplex form allows us to handle unrestricted variables efficiently and simply; it was developed for implementing constraint logic programming languages [101], and Cassowary adopts it. Conceptually it uses *two* tableaux rather than one. All of the unrestricted variables from the original constraints C will be placed in C_U, the unrestricted variable tableau. C_S, the simplex tableau, contains only variables constrained to be non-negative (the *restricted variables*).

Thus, an optimization problem is in *augmented simplex form* if the constraints C have the form $C_U \wedge C_S \wedge C_I$ where C_U and C_S are conjunctions of linear arithmetic equations, C_I is $\bigwedge\{x \geq 0 \mid x \in vars(C_S)\}$, and the objective function f is a linear expression over variables in C_S.

The simplex algorithm is used to determine an optimal solution for the equations in C_S, the simplex tableau, ignoring the unrestricted variable tableau (C_U) during the optimization procedure. (C_U need only be considered when finding the feasible region, prior to optimization.) The equations in the C_U are then used to determine values for its unrestricted variables.

It is not difficult to re-write an arbitrary optimization problem over linear real equations and inequalities into augmented simplex form. The first step is to convert inequalities to equations. Each inequality of the form $e \leq r$, where e is a linear real expression and r is a number, can be replaced with $e + s = r \ \wedge \ s \geq 0$ where s is a newly-introduced non-negative *slack* variable. Similarly, we replace $e \geq r$ with $e - s = r \ \wedge \ s \geq 0$.

For example, the constraints for Figure 3.1 can be rewritten as minimize $x_m - x_l$ subject to

$$\begin{aligned} 2x_m &= x_l + x_r \\ x_l + 10 + s_1 &= x_r \\ x_l - s_2 &= -10 \\ x_r + s_3 &= 100 \\ 0 &\leq s_1, s_2, s_3 \end{aligned}$$

We now separate the equalities into C_U and C_S. Initially all equations are in C_S. We move the unrestricted variables into C_U using Gauss-Jordan elimination. To do this, we select an equation in C_S containing an unrestricted variable u and remove the equation from C_S. We then solve the equation for u, yielding a new equation $u = e$ for some expression e. We then substitute e for all remaining occurrences of u in C_S, C_U, and f, and place the equation $u = e$ in C_U. The process is repeated until there are no more unrestricted variables in C_S. In the example, $x_r + s_3 = 100$ can be used to substitute $100 - s_3$ for x_r yielding:

minimize $x_m - x_l$ subject to

$$\begin{array}{rcll} x_r & = & 100 - s_3 & C_U \\ \hline 2x_m & = & x_l + 100 - s_3 & C_S \\ x_l + 10 + s_1 & = & 100 - s_3 & \\ x_l - s_2 & = & -10 & \\ & & & \\ 0 & \leq & s_1, s_2, s_3 & C_I \end{array}$$

Next, the first equation of C_S can be used to substitute $50 + \frac{1}{2}x_l - \frac{1}{2}s_3$ for x_m, giving minimize $50 - \frac{1}{2}x_l - \frac{1}{2}s_3$ subject to

$$\begin{array}{rcll}
x_m & = & 50 + \frac{1}{2}x_l - \frac{1}{2}s_3 & \\
x_r & = & 100 - s_3 & C_U \\
\hline
x_l + 10 + s_1 & = & 100 - s_3 & C_S \\
x_l - s_2 & = & -10 & \\
 & & & \\
0 & \leq & s_1, s_2, s_3 & C_I
\end{array}$$

Now we move x_l to C_U using $x_l = s_2 - 10$, giving

minimize $55 - \frac{1}{2}s_2 - \frac{1}{2}s_3$ subject to

$$\begin{array}{rcll}
x_m & = & 45 + \frac{1}{2}s_2 - \frac{1}{2}s_3 & \\
x_r & = & 100 - s_3 & \\
x_l & = & s_2 - 10 & C_U \\
\hline
s_2 + s_1 & = & 100 - s_3 & C_S \\
 & & & \\
0 & \leq & s_1, s_2, s_3 & C_I
\end{array}$$

(Hereafter, the labels for C_U and C_S will be omitted: constraints above the horizontal line are in C_U, and constraints below the line are in C_S. Also, C_I will be omitted entirely—any variable occurring below the horizontal line is implicitly constrained to be non-negative.)

The simplex method works by taking an optimization problem in "basic feasible solved form" (a type of normal form) and repeatedly applying matrix operations to obtain new basic feasible solved forms. Once we have split the equations into C_U and C_S, we can ignore C_U for purposes of optimization.

In the Cassowary implementation, all variables that may be accessed from outside the solver are unrestricted. Only error or slack variables are represented as restricted variables, and these variables occur only within the solver (Section 3.3). The primary benefit of this simplification is that the programmer using the solver always uses just the one kind of variable. A minor benefit is that only the external, unrestricted variables actually store their values as a field in the variable object; the values of restricted variables are just given by the tableau. A minor drawback is that the constraint $v \geq 0$ must be represented explicitly. (For any other constant $c \neq 0$, $v \geq c$ must be represented explicitly in any event.)

In the running example, the constraints imply that x_r is non-negative. However, since x_r is accessible from outside the solver, we represent it as unrestricted. This does not change the solutions found. Also, I show the operations as modifying C_U as well as C_S. It would be possible to modify just C_S and leave C_U unchanged, using C_U only to define values for the variables on the left hand side of its equations. This would speed up pivoting, but it would make the incremental updates of the constants in edit constraints slower (Section 3.2.4). Because the latter is a much more frequent operation, I do actually modify both C_U and C_S in the implementation.

An augmented simplex form optimization problem is in *basic feasible solved form* if the equations are of the form

$$x_0 = c + a_1 x_1 + \ldots + a_n x_n$$

where the variable x_0 does not occur in any other equation or in the objective function. If the equation is in C_S, c must be non-negative. However, there is no such restriction on the constants for the equations in C_U. In either case the variable x_0 is said to be *basic* and the other variables in the equation are *parameters*. A problem in basic feasible solved form defines a *basic feasible solution*, which is obtained by setting each parametric variable to 0 and each basic variable to the value of the constant in the right-hand side.

For instance, the following constraint is in basic feasible solved form and is equivalent to the problem above.

minimize $55 - \frac{1}{2}s_2 - \frac{1}{2}s_3$ subject to

$$\begin{array}{rcl} x_l & = & -10 + s_2 \\ x_m & = & 45 + \frac{1}{2}s_2 - \frac{1}{2}s_3 \\ x_r & = & 100 - s_3 \\ \hline s_1 & = & 100 - s_2 - s_3 \end{array}$$

The basic feasible solution corresponding to this basic feasible solved form is

$$\{x_l \mapsto -10,\ x_m \mapsto 45,\ x_r \mapsto 100,\ s_1 \mapsto 100,\ s_2 \mapsto 0,\ s_3 \mapsto 0\}.$$

The value of the objective function with this solution is 55.

3.2.2 Simplex optimization

I now describe how Cassowary finds an optimum solution to a constraint in basic feasible solved form. Except for the operations on the additional unrestricted variable tableau C_U, the material presented in this subsection is simply Phase II of the standard two-phase simplex algorithm.

The simplex algorithm finds the optimum by repeatedly looking for an "adjacent" basic feasible solved form whose basic feasible solution decreases the value of the objective function that we are minimizing. When no such adjacent basic feasible solved form can be found, we have achieved an optimum. The underlying operation is called *pivoting* and involves exchanging a basic and a parametric variable using matrix operations. Thus, "adjacent" means the new basic feasible solved form can be reached by performing a single pivot.

In the example, increasing s_2 from 0 will decrease the value of the objective function we are minimizing. We must be careful: we cannot increase the value of s_2 indefinitely as this may cause the value of some other basic non-negative variable to become negative. We must examine the equations in C_S. The equation $s_1 = 100 - s_2 - s_3$ allows s_2 to take at most a value of 100, because if s_2 becomes larger than this, then s_1 would become negative. The equations above the horizontal line do not restrict s_2, since whatever value s_2 takes the unrestricted variables x_l and x_m can take values to satisfy the equations. In general, we choose the most restrictive equation in C_S, and use it to eliminate s_2. In the case of ties we arbitrarily break the tie. In this example, the most restrictive equation (there is only one) is $s_1 = 100 - s_2 - s_3$. Writing s_2 as the subject we obtain $s_2 = 100 - s_1 - s_3$. We replace s_2 everywhere by $100 - s_1 - s_3$ and obtain minimize $5 + \frac{1}{2}s_1$ subject to

$$\begin{array}{rcl} x_l & = & 90 - s_1 - s_3 \\ x_m & = & 95 - \frac{1}{2}s_1 - s_3 \\ x_r & = & 100 - s_3 \\ \hline s_2 & = & 100 - s_1 - s_3 \end{array}$$

We have just performed a pivot, having moved s_1 out of the set of basic variables and replaced it by moving s_2 into the basis. The value of the objective function has decreased from 55 to 5.

We continue this process. Increasing the value of s_1 would increase the value of the objective function (which we are trying to minimize, so we do not want to do this). Note that decreasing s_1 would decrease the objective function's value, but as s_1 is constrained to be non-negative, it already takes its minimum value of 0 in the associated basic feasible solution. Hence we are at an optimal solution.[1] As one might expect, x_l is 10 units away from x_r in this solution, minimizing the distance between the points while still satisfying the $x_l + 10 \leq x_r$ constraint.

In general, the simplex algorithm applied to C_S is described as follows. We are given a problem in basic feasible solved form in which the variables $x_1, \ldots, x_n$ are basic and the variables $y_1, \ldots, y_m$ are parameters.

minimize $e + \sum_{j=1}^{m} d_j y_j$ subject to

$$\bigwedge_{i=1}^{n} x_i \;=\; c_i + \sum_{j=1}^{m} a_{ij} y_j \;\wedge$$

$$\bigwedge_{i=1}^{n} x_i \geq 0 \;\wedge\; \bigwedge_{j=1}^{m} y_j \geq 0.$$

Select an entry variable y_J such that $d_J < 0$. (An entry variable is one that will enter the basis, i.e., it is currently parametric and we want to make it basic.) Pivoting on such a variable can only decrease the value of the objective function. If no such variable exists, the optimum has been

[1] If we had an unrestricted variable in the objective function, the optimization would be unbounded. This possibility is not an issue for the algorithm because of the nature of the objective functions that arise from edit and stay constraints (Section 3.2.3).

```
simplex_opt($C_S$,$f$)
    repeat
        % Choose variable $y_J$ to become basic
        if for each $j \in \{1, \ldots, m\}$ $d_j \geq 0$ then
            return % an optimal solution has been found
        endif
        choose $J \in \{1, \ldots, m\}$ such that $d_J < 0$
        % Choose variable $x_I$ to become non-basic
        choose $I \in \{1, \ldots, n\}$ such that
            $-c_I/a_{IJ} = \min_{i \in \{1,\ldots,n\}}\{-c_i/a_{iJ} \mid a_{iJ} < 0\}$
        $e := (x_I - c_I - \sum_{j=1, j \neq J}^{m} a_{Ij}y_j)/a_{IJ}$
        $C_S[I] := (Y_J = e)$
        replace $Y_J$ by $e$ in $f$
        for each $i \in \{1, \ldots, n\}$
            if $i \neq I$ then replace $Y_J$ by $e$ in $C_S[I]$ endif
        endfor
    endrepeat
```

Figure 3.2: Simplex optimization

reached. Now determine the exit variable x_I. We must choose this variable so that it maintains basic feasible solved form by ensuring that the new c_i's are still positive after pivoting. That is, we must choose an x_I so that $-c_I/a_{IJ}$ is a minimum element of the set

$$\{-c_i/a_{iJ} \mid a_{iJ} < 0 \text{ and } 1 \leq i \leq n\}.$$

If there were no i for which $a_{iJ} < 0$ then we could stop since the optimization problem would be unbounded and so would not have a minimum: we could choose y_J to take an arbitrarily large value and thus make the objective function arbitrarily small. However, this potential problem is not an issue in this context since the optimization problems will always have a lower bound of 0.

We proceed to choose x_I, and pivot x_I out and replace it with y_J to obtain the new basic feasible solution. We continue this process until an optimum is reached. The algorithm is specified in Figure 3.2 and takes as inputs the simplex tableau C_S and the objective function f.

3.2.3 Handling non-required constraints

Suppose the user wishes to edit x_m in the diagram and have x_l and x_r weakly stay where they are. These desires correspond to the non-required constraints *edit* x_m, *stay* x_l, and *stay* x_r. Suppose further that we are trying to move x_m to position 50, and that x_l and x_r are currently at 30 and 60 respectively. We are thus imposing the constraints strong $x_m = 50$, weak $x_l = 30$, and weak $x_r = 60$.

As discussed in Section 3.1.1, there are various possible comparators for specifying how conflicting non-required constraints are to be traded off. Cassowary finds weighted-sum-better solutions. (Since every weighted-sum-better solution is also a locally-error-better solution [14], Cassowary finds locally-error-better solutions as well.)

The error for an equality constraint $e_1 = e_2$ is defined as $|e_1 - e_2|$, while the error for an inequality constraint $e_1 \leq e_2$ is 0 if $e_1 \leq e_2$ and otherwise $e_1 - e_2$. For example, the error for the constraint $x_m = 50$ is $|x_m - 50|$.

To form an objective function for the weighted-sum-better comparator, we can sum the errors for the each constraint, weighting the errors so that satisfying any strong constraint is always strictly more important than satisfying any combination of weaker constraints. For the example, the objective function is

$$s|x_m - 50| + w|x_l - 30| + w|x_r - 60|$$

where s and w are appropriate weights. Due to the absolute value operators, this objective function is not linear, and hence the simplex method is not applicable directly. I now show how Cassowary solves the problem using *quasi-linear optimization*.

Both the edit and the stay constraints will be represented as equations of the form

$$v = \alpha + \delta_v^+ - \delta_v^-$$

where δ_v^+ and δ_v^- are non-negative variables representing the deviation of v from the desired value α. If the constraint is satisfied both δ_v^+ and δ_v^- will be 0. Otherwise δ_v^+ will be positive and δ_v^- will be 0 if v is too big, or vice versa if v is too small.[2] Because we want δ_v^+ and δ_v^- to be 0 if possible, we make them part of the objective function, with larger coefficients for the error variables of stronger constraints. (We need to use the pair of variables to satisfy simplex's non-negativity restriction, since these variables δ_v^+ and δ_v^- will be part of the objective function.)

Translating the constraints strong $x_m = 50$, weak $x_l = 30$, and weak $x_r = 60$ which arise from the edit and stay constraints we obtain:

$$\begin{aligned} x_m &= 50 + \delta_{x_m}^+ - \delta_{x_m}^- \\ x_l &= 30 + \delta_{x_l}^+ - \delta_{x_l}^- \\ x_r &= 60 + \delta_{x_r}^+ - \delta_{x_r}^- \\ 0 &\leq \delta_{x_m}^+, \delta_{x_m}^-, \delta_{x_l}^+, \delta_{x_l}^-, \delta_{x_r}^+, \delta_{x_r}^- \end{aligned}$$

as well as the original constraints:

$$\begin{aligned} 2x_m &= x_l + x_r \\ x_l + 10 &\leq x_r \\ x_l &\geq -10 \\ x_r &\leq 100 \end{aligned}$$

To ensure that strong constraints are always satisfied in preference to weak ones, Cassowary uses symbolic weights for the coefficients in the objective function, represented as tuples and ordered lexicographically, rather than real numbers. In the presentation that follows, I will depict these symbolic weights as pairs, such as $[1,2]$, which represents the symbolic weight consisting of the unit weight for the strong strength plus twice the unit weight for the weak strength. The objective function for our example can now be restated as:

$$\text{minimize } [1,0]\delta_{x_m}^+ + [1,0]\delta_{x_m}^- + [0,1]\delta_{x_l}^+ + [0,1]\delta_{x_l}^- + [0,1]\delta_{x_r}^+ + [0,1]\delta_{x_r}^-$$

(As an aside, if we were not using symbolic weights, and instead using real numbers as coefficients in the objective function, we might use $s = 1000$ and $w = 1$ for the strong and weak

[2] Although the equation may be satisfied with both δ_v^+ and δ_v^- non-zero, the simplex optimization itself forces at least one of them to be zero (Section 3.2.4).

strengths. In that case the objective function would be

$$\text{minimize } 1000\delta^+_{x_m} + 1000\delta^-_{x_m} + \delta^+_{x_l} + \delta^-_{x_l} + \delta^+_{x_r} + \delta^-_{x_r}.$$

While simpler, this technique has the danger that in some cases the weak constraints could over-power the strong ones, contrary to the solutions allowed by the constraint hierarchy theory. Symbolic weights avoid this danger.)

Returning to our example with symbolic weights as coefficients in the objective function, an optimal solution of this problem can be found using the simplex algorithm, and results in a tableau

$$\text{minimize } [0,10] + [1,2]\delta^+_{x_m} + [1,-2]\delta^-_{x_m} + [0,2]\delta^-_{x_l} + [0,2]\delta^-_{x_r} \quad \text{subject to}$$

$$\begin{array}{rclccccc}
x_m & = & 50 & +\delta^+_{x_m} & -\delta^-_{x_m} & & & \\
x_l & = & 30 & & & +\delta^+_{x_l} & -\delta^-_{x_l} & \\
x_r & = & 70 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & -\delta^+_{x_l} & +\delta^-_{x_l} & \\
\hline
s_1 & = & 30 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & -2\delta^+_{x_l} & +2\delta^-_{x_l} & \\
s_3 & = & 30 & -2\delta^+_{x_m} & +2\delta^-_{x_m} & +\delta^+_{x_l} & -\delta^-_{x_l} & \\
\delta^+_{x_r} & = & 10 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & -\delta^+_{x_l} & +\delta^-_{x_l} & +\delta^-_{x_r} \\
s_2 & = & 40 & & & +\delta^+_{x_l} & -\delta^-_{x_l} &
\end{array}$$

This corresponds to the solution $\{x_m \mapsto 50,\ x_l \mapsto 30,\ x_r \mapsto 70\}$ illustrated in Figure 3.1. Notice that the weak stay constraint on x_r is not satisfied ($\delta^+_{x_r}$ is non-zero, read directly from the second to last line of the above tableau).

3.2.4 Incrementality: resolving the optimization problem

Now suppose the user moves the mouse (which is editing x_m) to $x = 60$. We wish to solve a new problem, with constraints strong $x_m = 60$, and weak $x_l = 30$ and weak $x_r = 70$ (so that x_l and x_r should stay where they are if possible).

There are two steps. First, we modify the tableau to reflect the new constraints we wish to solve. Second, we resolve the optimization problem for this modified tableau.

Let us first examine how to modify the tableau to reflect the new values of the stay constraints. This will not require re-optimizing the tableau, since we know that the new stay constraints are satisfied exactly. Suppose the previous stay value for variable v was α, and in the current solution v takes value β. The current tableau contains the information that

$$v = \alpha + \delta_v^+ - \delta_v^-$$

and we need to modify this so that instead

$$v = \beta + \delta_v^+ - \delta_v^-$$

There are two cases to consider: (a) both δ_v^+ and δ_v^- are parameters, or (b) one of them is basic.

In case (a) v must take the value α in the current solution since both δ_v^+ and δ_v^- take the value 0 and

$$v = \alpha + \delta_v^+ - \delta_v^-$$

Hence $\beta = \alpha$ and no changes need to be made.

In case (b) assume without loss of generality that δ_v^+ is basic. In the original equation representing the stay constraint, the coefficient for δ_v^+ is the negative of the coefficient for δ_v^-. Since these variables occur in no other constraints, this relation between the coefficients will continue to hold as we perform pivots. In other words, δ_v^+ and δ_v^- come in pairs: any equation that contains δ_v^+ will also contain δ_v^-, with one coefficient the negative of the other. Since δ_v^+ is assumed to be basic, it occurs exactly once in an equation with constant c, and further this equation also contains the only occurrence of δ_v^-. In the current solution

$$\{v \mapsto \beta,\ \delta_v^+ \mapsto c,\ \delta_v^- \mapsto 0\}$$

and since the equation

$$v = \alpha + \delta_v^+ - \delta_v^-$$

holds, $\beta = \alpha + c$. To replace the equation

$$v = \alpha + \delta_v^+ - \delta_v^-$$

by

$$v = \beta + \delta_v^+ - \delta_v^-$$

we simply need to replace the constant c in the row for δ_v^+ by 0. Since there are no other occurrences of δ_v^+ and δ_v^-, we have replaced the old equation with the new.

For our example, to update the tableau for the new values for the stay constraints on x_l and x_r, we simply set the constant of the second to last equation (the equation for $\delta_{x_r}^+$) to 0. The tableau is now:

$$\text{minimize } [0,0] + [1,2]\delta_{x_m}^+ + [1,-2]\delta_{x_m}^- + [0,2]\delta_{x_l}^- + [0,2]\delta_{x_r}^- \quad \text{subject to}$$

$$\begin{array}{rclrrrrr}
x_m & = & 50 & +\delta_{x_m}^+ & -\delta_{x_m}^- & & & \\
x_l & = & 30 & & & +\delta_{x_l}^+ & -\delta_{x_l}^- & \\
x_r & = & 70 & +2\delta_{x_m}^+ & -2\delta_{x_m}^- & -\delta_{x_l}^+ & +\delta_{x_l}^- & \\
\hline
s_1 & = & 30 & +2\delta_{x_m}^+ & -2\delta_{x_m}^- & -2\delta_{x_l}^+ & +2\delta_{x_l}^- & \\
s_3 & = & 30 & -2\delta_{x_m}^+ & +2\delta_{x_m}^- & +\delta_{x_l}^+ & -\delta_{x_l}^- & \\
\delta_{x_r}^+ & = & 0 & +2\delta_{x_m}^+ & -2\delta_{x_m}^- & -\delta_{x_l}^+ & +\delta_{x_l}^- & +\delta_{x_r}^- \\
s_2 & = & 40 & & & +\delta_{x_l}^+ & -\delta_{x_l}^- &
\end{array}$$

(For completeness, in the running example, I update the constant part of the objective function, as well as its other terms. However, the constant part of the objective function is irrelevant for the algorithm, and my implementations ignore it.)

Now let us consider the edit constraints. Suppose the previous edit value for v was α, and the new edit value for v is β. The current tableau contains the information that

$$v = \alpha + \delta_v^+ - \delta_v^-$$

and again we need to modify this so that instead

$$v = \beta + \delta_v^+ - \delta_v^-$$

To do so we must replace every occurrence of

$$\delta_v^+ - \delta_v^-$$

by

$$\beta - \alpha + \delta_v^+ - \delta_v^-$$

taking proper account of the coefficients of δ_v^+ and δ_v^-. (Again, remember that δ_v^+ and δ_v^- come in pairs.)

If either of δ_v^+ and δ_v^- is basic, this simply involves appropriately modifying the equation in which they are basic. Otherwise, if both are non-basic, then we need to change every equation of the form

$$x_i = c_i + a'_v\delta_v^+ - a'_v\delta_v^- + e$$

to

$$x_i = c_i + a'_v(\beta - \alpha) + a'_v\delta_v^+ - a'_v\delta_v^- + e$$

Hence modifying the tableau to reflect the new values of edit and stay constraints involves only changing the constant values in some equations. The modifications for stay constraints always result in a tableau in basic feasible solved form, since it never makes a constant become negative. In contrast the modifications for edit constraints may result in an infeasible tableau.

To return to our example, suppose we pick up x_m with the mouse and move it to 60. Then $\alpha = 50$ and $\beta = 60$, so we need to add 10 times the coefficient of $\delta_{x_m}^+$ to the constant part of every row. The modified tableau, after the updates for both the stays and edits, is minimize $[0,20] + [1,2]\delta^+_{x_m} + [1,-2]\delta^-_{x_m} + [0,2]\delta^-_{x_l} + [0,2]\delta^-_{x_r}$ subject to

$$
\begin{array}{rclllllll}
x_m & = & 60 & +\delta^+_{x_m} & -\delta^-_{x_m} & & & \\
x_l & = & 30 & & & +\delta^+_{x_l} & -\delta^-_{x_l} & \\
x_r & = & 90 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & -\delta^+_{x_l} & +\delta^-_{x_l} & \\
\hline
s_1 & = & 50 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & -2\delta^+_{x_l} & +2\delta^-_{x_l} & \\
s_3 & = & 10 & -2\delta^+_{x_m} & +2\delta^-_{x_m} & +\delta^+_{x_l} & -\delta^-_{x_l} & \\
\delta^+_{x_r} & = & 20 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & -\delta^+_{x_l} & +\delta^-_{x_l} & +\delta^-_{x_r} \\
s_2 & = & 40 & & & +\delta^+_{x_l} & -\delta^-_{x_l} &
\end{array}
$$

Clearly it is feasible and already in optimal form, and so we have incrementally resolved the problem by simply modifying constants in the tableaux. The new tableaux give the solution $\{x_m \mapsto 60,\ x_l \mapsto 30,\ x_r \mapsto 90\}$. So sliding the midpoint rightwards has caused the right point to slide rightwards as well, but twice as far. The resulting diagram is shown at the top of Figure 3.3.

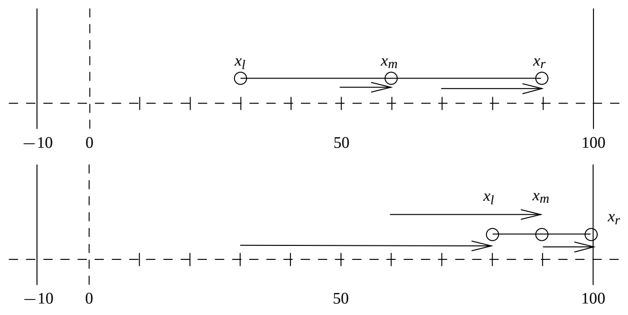


Figure 3.3: Resolving the constraints

Suppose we now move x_m from 60 to 90. The modified tableau is minimize $[0, 60] + [1, 2]\delta^+_{x_m} + [1, -2]\delta^-_{x_m} + [0, 2]\delta^-_{x_l} + [0, 2]\delta^-_{x_r}$ subject to

$$\begin{array}{rcrrrrrr}
x_m & = & 90 & +\delta^+_{x_m} & -\delta^-_{x_m} & & & \\
x_l & = & 30 & & & +\delta^+_{x_l} & -\delta^-_{x_l} & \\
x_r & = & 150 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & -\delta^+_{x_l} & +\delta^-_{x_l} & \\
\hline
s_1 & = & 110 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & -2\delta^+_{x_l} & +2\delta^-_{x_l} & \\
s_3 & = & -50 & -2\delta^+_{x_m} & +2\delta^-_{x_m} & +\delta^+_{x_l} & -\delta^-_{x_l} & \\
\delta^+_{x_r} & = & 60 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & -\delta^+_{x_l} & +\delta^-_{x_l} & +\delta^-_{x_r} \\
s_2 & = & 40 & & & +\delta^+_{x_l} & -\delta^-_{x_l} &
\end{array}$$

The tableau is no longer in basic feasible solved form, since the constant of the row for s_3 is negative, even though s_3 is supposed to be non-negative. (In this solution $x_r = 150$, so that the right endpoint has crashed through the $x_r \leq 100$ barrier.)

Thus, in general, after updating the constants for the edit constraints, the simplex tableau C_S may no longer be in basic feasible solved form, since some of the constants may be negative. However, the tableau is still in basic form, so we can still read a solution directly from it as before. Also, because no coefficient has changed (in particular the optimization function is the same), the resulting tableau reflects an optimal but not feasible solution.

We need to find a feasible and optimal solution. We could do so by adding artificial variables (as when adding a constraint—see Section 3.2.5), optimizing the sum of the artificial variables to find an initial feasible solution, and then re-optimizing the original problem.

But we can do much better. The process of moving from an optimal and *infeasible* solution to an optimal and *feasible* solution is exactly the dual of normal simplex algorithm, where we progress from a feasible and non-optimal solution to feasible and optimal solution. Hence we can use the *dual simplex algorithm* to find a feasible solution while staying optimal.

Solving the dual optimization problem starts from an infeasible optimal tableau of the form

minimize $e + \Sigma_{j=1}^{m} d_j y_j$ subject to

$$\bigwedge_{i=1}^{n} x_i = c_i + \Sigma_{j=i}^{m} a_{ij} y_j$$

where some c_i may be negative for rows with non-negative basic variables (accounting for the tableau's infeasibility) and each d_j is non-negative (so it is optimal).

The dual simplex algorithm selects an exit variable by finding a row I with non-negative basic variable x_I and negative constant c_I. The entry variable is the variable y_J such that the ratio d_J/a_{IJ} is the minimum of all d_j/a_{Ij} where a_{Ij} is positive. This selection criteria ensures that when pivoting we stay at an optimal solution. The pivot replaces y_j by

$$-1/a_{Ij}(-x_I + c_I + \Sigma_{j=1,j\neq J}^{m} a_{Ij}y_j)$$

and is performed as in the (primal) simplex algorithm. The algorithm is shown in Figure 3.4.

Continuing the example above, we select the exit variable s_3—the only non-negative basic variable for a row with negative constant. We find that $\delta^+_{x_l}$ has the minimum ratio since its coefficient in the optimization function is 0, so it will be the entry variable. Replacing $\delta^+_{x_l}$ everywhere by $50 + s_3 + 2\delta^+_{x_m} - 2\delta^-_{x_m} + \delta^-_{x_l}$ we obtain the tableau

$$\text{minimize } [0, 60] + [1, 2]\delta^+_{x_m} + [1, -2]\delta^-_{x_m} + [0, 2]\delta^-_{x_l} + [0, 2]\delta^-_{x_r} \quad \text{subject to}$$

$$\begin{array}{rcrrrrrr}
x_m & = & 90 & & +\delta^+_{x_m} & -\delta^-_{x_m} & & \\
x_l & = & 80 & +s_3 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & & \\
x_r & = & 100 & -s_3 & & & & \\
\hline
s_1 & = & 10 & -2s_3 & -2\delta^+_{x_m} & +2\delta^-_{x_m} & & \\
\delta^+_{x_l} & = & 50 & +s_3 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & +\delta^-_{x_l} & \\
\delta^+_{x_r} & = & 10 & -s_3 & & & & +\delta^-_{x_r} \\
s_2 & = & 90 & +s_3 & +2\delta^+_{x_m} & -2\delta^-_{x_m} & &
\end{array}$$

The tableau is feasible (and of course still optimal) and represents the solution $\{x_m \mapsto 90, x_r \mapsto 100, x_l \mapsto 80\}$. So by sliding the midpoint further right, the rightmost point hits the wall and the left point slides right to satisfy the constraints. The resulting diagram is shown at the bottom of Figure 3.3.

To summarize, incrementally finding a new solution for new input variables involves updating the constants in the tableaux to reflect the updated stay constraints, then updating the constants to

```
re_optimize($C_S$,$f$)
    foreach stay : $v \in C$
        if $\delta_v^+$ or $\delta_v^-$ is basic in row $i$ then $c_i := 0$ endif
    endfor
    foreach edit : $v \in C$
        let $\alpha$ and $\beta$ be the previous and current edit values for $v$
        let $\delta_v^+$ be $y_j$
        foreach $i \in \{1, \ldots, n\}$
            $c_i := c_i + a_{ij}(\beta - \alpha)$
        endfor
    endfor
    repeat
        % Choose variable $x_I$ to become non-basic
        choose $I$ where $c_I < 0$
        if there is no such $I$
            return true
        endif
        % Choose variable $y_J$ to become basic
        if for each $j \in \{1, \ldots, m\}$ $a_{Ij} \leq 0$ then
            return false
        endif
        choose $J \in \{1, \ldots, m\}$ such that
            $d_J / a_{IJ} = \min_{j \in \{1,\ldots,m\}} \{d_j / a_{Ij} \mid a_{Ij} > 0\}$
        $e := (x_I - c_I - \sum_{j=1, j \neq J}^{m} a_{Ij} y_j) / a_{IJ}$
        replace $y_J$ by $e$ in $f$
        for each $i \in \{1, \ldots, n\}$
            if $i \neq I$ then replace $y_J$ by $e$ in row $i$ endif
        endfor
        replace the $I^{th}$ row by $y_J = e$
    until false
```

Figure 3.4: Dual Simplex Re-optimization

reflect the updated edit constraints, and finally re-optimizing if needed. In an interactive graphical application, this dual optimization method typically requires a pivot only when one part of the figure first hits or first moves away from a barrier. The intuition behind this is that when a constraint first becomes unsatisfied, the value of one of its error variables will become non-zero, and hence the variable will have to enter the basis; conversely, when a constraint first becomes satisfied, we can move one of its error variables out of the basis.

In the example, pivoting occurred when the right point x_r came up against a barrier. Thus, if we picked up the midpoint x_m with the mouse and smoothly slid it rightwards, 1 pixel every screen refresh, only one pivot would be required in moving from 50 to 95. This behavior is why the dual optimization is well suited to this problem and leads to efficient resolving of the hierarchical constraints.

3.2.5 *Incrementality: adding a constraint*

I now describe how Cassowary adds the equation for a new constraint incrementally. This technique is also used in the implementation to find an initial basic feasible solved form for the original simplex problem, by starting from an empty constraint set and adding the constraints one at a time.

As an example, suppose we wish to require that the midpoint be centered. That is, we wish to add a required constraint $x_m = 50$ to the final tableau given in Section 3.2.2. For reference, that tableau is:

$$\begin{array}{rcl} x_l & = & 90 - s_1 - s_3 \\ x_m & = & 95 - \frac{1}{2}s_1 - s_3 \\ x_r & = & 100 - s_3 \\ \hline s_2 & = & 100 - s_1 - s_3 \end{array}$$

If we substitute for each of the basic variables in $x_m = 50$ (namely x_m), we obtain the equation $45 - \frac{1}{2}s_1 - s_3 = 0$. In order to add this constraint straightforwardly to the tableau we create a new non-negative variable a called an *artificial variable*. (This technique is simply an incremental version of the operation used in Phase I of the two-phase simplex algorithm.) We let $a = 45 - \frac{1}{2}s_1 - s_3$ be added to the tableau (clearly this gives a tableau in basic feasible solved form) and then minimize the value of a. If a takes the value 0 then we have obtained a solution to the problem with the added constraint, and we can then eliminate the artificial variable altogether since it is a parameter (and hence takes the value 0). This is the case for our example; the resulting tableau is

$$\begin{array}{rcl}
x_l & = & 0 + s_3 \\
x_m & = & 50 \\
x_r & = & 100 - s_3 \\
\hline
s_1 & = & 90 - 2s_3 \\
s_2 & = & 10 + s_3
\end{array}$$

In general, to add a new required constraint to the tableau we first convert it to an augmented simplex form equation by adding a slack variable if it is an inequality. Next, we use the current tableau to substitute out all the basic variables. This gives an equation $e = 0$ where e is a linear expression. If the constant part of e is negative, we multiply both sides by -1 so that the constant becomes non-negative. If e contains an unrestricted variable we use e to substitute for that variable, and we add the equation to the tableau above the line (i.e., to C_U). Otherwise we create a restricted artificial variable a, add the equation $a = e$ to the tableau below the line (i.e., to C_S), and minimize e. If the resulting minimum is not zero then the constraints are unsatisfiable. Otherwise a is either parametric or basic. If a is parametric, the column for it can be simply removed from the tableau. If it is basic, the row must have constant 0 (since we were able to achieve a value of 0 for the objective function, which is equal to a). If the row is just $a = 0$, it can be removed. Otherwise, $a = 0 + bx + \cdots$ where $b \neq 0$. We can then pivot x into the basis using this row and remove the column for a.

If the equation being added contains any unrestricted variables after substituting out all the basic variables, as described above we do not need to use an artificial variable. Not only that, we *could not* use an artificial variable, since we cannot put an equation in C_S that contains an unrestricted variable. In some other cases we can avoid using an artificial variable for efficiency, even though it would be permissible to use one. We can avoid using an artificial variable if we can choose a subject for the equation from among its current variables. Here are the rules for choosing a subject. (These are to be used after replacing any basic variables with their defining expressions.)

We start with an expression e. If necessary, normalize e by multiplying by -1 so that its constant part is non-negative. We are adding the constraint $e = 0$ to the tableau. To do this, we want to pick a variable in e to be the subject of an equation, so that we can add the row $v = e'$, where e' the result of solving $e = 0$ for v.

- If e contains any unrestricted variables, we must choose an unrestricted variable as the subject.
- If the subject is new to the solver, we will not have to do any substitutions, so we prefer new variables to ones that are currently noted as parametric.
- If e contains only restricted variables, if there is a (restricted) variable in e that has a negative coefficient and that is new to the solver, we can pick that variable as the subject.
- Otherwise use an artificial variable.

A consequence of these rules is that we can always add a non-required constraint to the tableau without using an artificial variable, because the equation will contain a positive and a negative error or slack variable, both of which are new to the solver, and which occur with opposite signs. (Constraints that are originally equations will have a positive and a negative error variable, while constraints that are originally inequalities will have one error variable and one slack variable, with opposite signs.) This observation is good news for performance, since adding a non-required edit constraint is a common operation.

3.2.6 Incrementality: removing a constraint

We also want a method for incrementally removing a constraint from the tableaux. After a series of pivots have been performed, the information represented by the constraint may not be contained in a single row, so we need a way to identify the constraint's influence in the tableaux. To do this, we use a "marker" variable that is originally present only in the equation representing the constraint. We can then identify the constraint's effect on the tableaux by looking for occurrences of that marker variable. For inequality constraints, the slack variable s that we added to make it an equality serves as the marker (because s will originally occur only in that equation). For non-required equality constraints, either of its two error variables can serve as a marker (Section 3.2.3). For required equality constraints, we add a "dummy" restricted variable to the original equation to serve as a marker, which we never allow to enter the basis (so that it always has value 0). In our running example, then, to allow the constraint $2x_m = x_l + x_r$ to be deleted incrementally we would have added a dummy variable d, resulting in $2x_m = x_l + x_r + d$. The simplex optimization routine checks for dummy variables in choosing an entry variable and does not allow one to be selected. Dummy variables must be restricted, not unrestricted, because we might need to have some of them in the equations for restricted basic variables. (I did not include the variable d in the tableaux presented earlier to simplify the presentation.)

Consider removing the constraint that x_l is 10 to the left of x_r. The slack variable s_1, which we added to the inequality to make it an equation, records exactly how this equation has been used to modify the tableau. We can remove the inequality by pivoting the tableau until s_1 is basic and then simply drop the row in which it is basic.

In the tableau in Section 3.2.5 (obtained after adding the required constraint $x_m = 50$), s_1 is already basic, so removing it simply means dropping the row in which it is basic, obtaining

$$\begin{array}{rcrl} x_l & = & 0 & +s_3 \\ x_m & = & 50 & \\ x_r & = & 100 & -s_3 \\ \hline s_2 & = & 10 & +s_3 \end{array}$$

If we wanted to remove this constraint from the tableau before adding $x_m = 50$ (i.e., the final tableau given in Section 3.2.2, page 51), s_1 is a parameter. We make s_1 basic by treating it as an entry variable and (as usual) determining the most restrictive equation, then using that to pivot s_1 into the basis before finally removing the row.

There is such a restrictive equation in this example. However, if the marker variable does not occur in C_S, or if its coefficients in C_S are all non-negative, then no equation restricts the value of the marker variable. If the marker variable does occur in one or more equations in C_S, always with a positive coefficient, pick the equation with the smallest ratio of the constant to the marker variable's coefficient. (The row with the marker variable will become infeasible after the pivot, but all the other rows will still be feasible, and we will be dropping the row with the marker variable. In effect we are removing the non-negativity restriction on the marker variable.) Finally, if the marker variable occurs only in equations for unrestricted variables, we can choose any equation in which it occurs.

In the final tableau in Section 3.2.2, the row $s_2 = 100 - s_1 - s_3$ is the most constraining equation. Pivoting to let s_1 enter the basis and then removing the row in which it is basic, we obtain

$$\begin{array}{rcl} x_l & = & -10 + s_2 \\ x_m & = & 45 + \frac{1}{2}s_2 - \frac{1}{2}s_3 \\ x_r & = & 100 - s_3 \\ \hline \end{array}$$

In the preceding example the marker variable had a negative coefficient. Here is an example that only has positive coefficients. The original constraints are:

$$\begin{array}{rcl} x & \geq & 10 \\ x & \geq & 20 \\ x & \geq & 30 \end{array}$$

In basic feasible solved form, this is:

$$\begin{array}{rcll} x & = & 30 & +s_3 \\ \hline s_1 & = & 20 & +s_3 \\ s_2 & = & 10 & +s_3 \end{array}$$

where s_1, s_2, and s_3 are the marker (and slack) variables for $x \geq 10$, $x \geq 20$, and $x \geq 30$ respectively. This gives a solution for x of $x = 30$, which of course satisfies all of the original inequalities.

Suppose we want to remove the $x \geq 30$ constraint. We need to pivot to make s_3 basic. The equation that gives the smallest ratio is $s_2 = 10 + s_3$, so the entry variable is s_3 and the exit variable is s_2, giving:

$$\begin{array}{rcrl} x & = & 20 & +s_2 \\ \hline s_1 & = & 10 & +s_2 \\ s_3 & = & -10 & +s_2 \end{array}$$

This tableau is now infeasible, but we drop the row with s_3 giving

$$\begin{array}{rcrl} x & = & 20 & +s_2 \\ \hline s_1 & = & 10 & +s_2 \end{array}$$

which is, of course, feasible.

A beneficial result of using marker variables is that redundant constraints can be represented and manipulated. Consider:

$$\begin{array}{rcl} x & \geq & 10 \\ x & \geq & 10 \end{array}$$

When converted to basic feasible solved form, each $x \geq 10$ constraint gives rise to a separate slack variable, which is used as the marker variable for that constraint.

$$\begin{array}{rcrl} x & = & 10 & +s_1 \\ \hline s_2 & = & 0 & +s_1 \end{array}$$

To delete the second $x \geq 10$ constraint we would simply drop the $s_2 = 0 + s_1$ row. To delete the first $x \geq 10$ constraint we would pivot, making s_1 basic and s_2 parametric:

$$\begin{array}{rcrl} x & = & 10 & +s_2 \\ \hline s_1 & = & 0 & +s_2 \end{array}$$

and then drop the $s_1 = 0 + s_2$ row.

A consequence of directly representing redundant constraints is that they must all be removed to eliminate their effect. (This seems to be a more desirable behavior for the solver than removing redundant constraints automatically, although if the latter were desired the solver could be modified to do this.) Another consequence is that when adding a new constraint, we would never decide that it was redundant and not add it to the tableau.

Before we remove a constraint, there may be some stay constraints that were unsatisfied previously. If we just removed the constraint, these could come into play, so instead, we reset all of the stays so that all variables are constrained to stay at their current values.

Also, if the constraint being removed is not required we need to remove the error variables for it from the objective function. To do this we add the following to the expression for the objective function:

$$-1 \times e \times s \times w$$

where e is the error variable if it is parametric, or else e is its defining expression if it is basic, s is the unit symbolic weight for the constraint's strength, and w is its weight at the given strength. (In the implementation s is an instance of ClSymbolicWeight and w is a float—see Section 3.3.7.)

If we allow non-required constraints other than stays and edits, we also need to re-optimize after deleting a constraint, since a non-required constraint might have become satisfiable (or more nearly satisfiable).

3.3 Implementation details

3.3.1 Principal classes

The principal classes in the implementations are as shown in Figure 3.5. All the classes start with "Cl" for "Constraint Library" and are, of course, direct or indirect subclasses of Object in the Smalltalk and Java implementations.

Some of these classes make use of the *Dictionary* (or *Map*) abstract data type: dictionaries have keys and values and permit efficiently finding the value for a given key, and adding or deleting key/value pairs. One can also iterate through all keys, all values, or all key/value pairs.

```
Object
    ClAbstractVariable
        ClDummyVariable
        ClObjectiveVariable
        ClSlackVariable
        ClVariable
    ClConstraint
        ClEditOrStayConstraint
            ClEditConstraint
            ClStayConstraint
        ClLinearConstraint
            ClLinearEqualityConstraint
            ClLinearInequalityConstraint
    ClLinearExpression
    ClTableau
        ClSimplexSolver
    ClStrength
    ClSymbolicWeight
```

Figure 3.5: Principle classes in the Cassowary implementations

3.3.2 Solver protocol

The solver itself is represented as an instance of ClSimplexSolver. Its public message protocol is as follows.

addConstraint(ClConstraint cn)

> Incrementally add the linear constraint cn to the tableau. The constraint object contains its strength.

removeConstraint(ClConstraint cn)

> Remove the constraint cn from the tableau. Also remove any error variables associated with cn from the objective function.

addEditVar(ClVariable v, ClStrength s)

> Add an edit constraint of strength s on variable v to the tableau so that suggestValue (see

below) can be used on that variable after a beginEdit().

removeEditVar(ClVariable v)

Remove the previously added edit constraint on variable v. The endEdit message automatically removes all the edit variables as part of terminating an edit manipulation.

beginEdit()

Prepare the tableau for new values to be given to the currently-edited variables. The addEditVar message should be used before calling beginEdit, and suggestValue and resolve should be used only after beginEdit has been invoked, but before the required matching endEdit.

suggestValue(ClVariable v, double n)

Specify a new desired value n for the variable v. Before this call, v needs to have been added as a variable of an edit constraint (either by addConstraint of a hand-built EditConstraint object or more simply using addEditVar).

endEdit()

Denote the end of an edit manipulation, thus removing all edit constraints from the tableau. Each beginEdit call must be matched with a corresponding endEdit invocation, and the calls may be nested properly.

resolve()

Try to re-solve the tableau given the newly specified desired values. Calls to resolve should be sandwiched between a beginEdit() and an endEdit(), and should occur after new values for edit variables are set using suggestValue.

addPointStays(Vector points)

This method is a bit of a kludge, and addresses the desire to satisfy the stays on both the x and y components of a given point rather than on the x component of one point and the y component of another. The argument points is an array of points, whose x and y components are constrainable variables. This method adds a weak stay constraint to the x and y variables of each point. The weights for the x and y components of a given point are the same. However, the weights for successive points are each smaller than those for the previous point (half of the previous weight). The effect of this is to encourage the solver to satisfy the stays on both the x and y of a given point rather than the x stay on one point and the y stay on another. See Section 3.4 for more discussion of this issue.

setAutoSolve(boolean f)

Choose whether the solver should automatically optimize and set external variable values after each addConstraint or removeConstraint. By default, auto-solving is on, but passing false to this method will turn it off (until later turned back on by passing true to this method). When auto-solving is off, solve (below) or resolve must be invoked to see changes to the ClVariables contained in the tableau.

isAutoSolving() **returns** boolean

Return true if and only if the solver is auto-solving, false otherwise.

solve()

Optimize the tableau and set the external ClVariables contained in the tableau to their new values. This method need only be invoked if auto-solving has been turned off. It never needs to be called after a resolve method invocation.

reset()

Re-initialize the solver from the original constraints, thus getting rid of any accumulated numerical problems. (Such problems have not yet arisen in practice, but I provide the method just in case.)

3.3.3 Variables

ClAbstractVariable and its subclasses represent various kinds of constrained variables. ClAbstractVariable is an abstract class, that is, it is just used as a superclass of other classes: one does not make instances of ClAbstractVariable itself. ClAbstractVariable defines the message protocol for constrainable variables. Its only instance variable is name, which is a string name for the variable. (This field is used for debugging and constraint understanding tasks.)

Instances of the concrete ClVariable subclass of ClAbstractVariable are what the user of the solver sees (hence it was given a nicer class name). This class has an instance variable value that holds the value of this variable. Users of the solver can send one of these variables the message value to get its value. Each constraint variable has an optional attached object, and the constraint solver can be instructed to invoke a callback upon changing the value assigned to any variable and also upon completion of the re-solve phase (i.e., after all variable assignments are completed).

The other subclasses of ClAbstractVariable are used only within the solver. They do not hold their own values—rather, the value is just given by the current tableau. None of them have any additional instance variables.

Instances of ClSlackVariable are restricted to be non-negative. They are used as the slack variable when converting an inequality constraint to an equation and for the error variables to represent non-required constraints.

Instances of ClDummyVariable is used as marker variables to allow required equality constraints to be deleted. (For inequalities or non-required constraints, the slack or error variable is used as the marker.) These dummy variables are never pivoted into the basis.

An instance of ClObjectiveVariable is used to index the objective row in the tableau. (Conventionally this variable is named *z*.) This kind of variable is just for convenience—the tableau is represented as a dictionary (with some additional cross-references). Each row is represented as an entry in the dictionary; the key is a basic variable and the value is an expression. So an instance of ClObjectiveVariable is the key for the objective row. The objective row is unique in that the coefficients of its expression are ClSymbolicWeights in the Smalltalk implementation, not just ordinary real numbers. However, the C++ and Java implementations convert ClSymbolicWeights to real numbers to avoid dealing with ClLinearExpressions parameterized on the type of the coefficient (Section 3.3.7).

All variables understand the following messages: isDummy, isExternal, isPivotable, and isRestricted. They also understand messages to get and set the variable's name.

Table 3.1: Subclasses of ClAbstractVariable

Class	isDummy	isExternal	isPivotable	isRestricted
ClDummyVariable	true	false	false	true
ClVariable	false	true	false	false
ClSlackVariable	false	false	true	true
ClObjectiveVariable	false	false	false	false

For isDummy, instances of ClDummyVariable return true and others return false. The solver uses this message to test for dummy variables. It will not choose a dummy variable as the subject for a new equation, unless all the variables in the equation are dummy variables. (The solver also will not pivot on dummy variables, but this is handled by the isPivotable message.)

For isExternal, instances of ClVariable return true and others return false. If a variable responds true to this message, it means that it is known outside the solver, and so the solver needs to give it a value after solving is complete.

For isPivotable, instances of ClSlackVariable return true and others return false. The solver uses this message to decide whether it can pivot on a variable.

For isRestricted, instances of ClSlackVariable and of ClDummyVariable return true, and instances of ClVariable and ClObjectiveVariable return false. Returning true means that this variable is restricted to being non-negative.

A variable's significance is largely just its identity (as mentioned above, variables have little state: a name for debugging and a value for instances of ClVariable). The only other messages that variables understand are some messages to ClVariable for creating constraints (Section 3.3.6).

3.3.4 Linear Expressions

Instances of the class ClLinearExpression hold a linear expression and are used in building and representing constraints and in representing the tableau. A linear expression holds a dictionary of variables and coefficients (the keys are variables and the values are the corresponding coefficients). Only variables with non-zero coefficients are included in the dictionary; if a variable is not in this dictionary its coefficient is assumed to be zero. The other instance variable is a constant. So to represent the linear expression $a_1 x_1 + \cdots + a_n x_n + c$, the dictionary would hold the key x_1 with value a_1, etc., and the constant c.

Linear expressions understand a large number of messages. Some of these are for constraint creation (Section 3.3.6). The others are to substitute an expression for a variable in the constraint, to add an expression, to find the coefficient for a variable, and so forth.

3.3.5 Constraints

There is an abstract class ClConstraint that serves as the superclass for other concrete classes. It defines two instance variables: strength and weight. The variable strength is the strength of this constraint in the constraint hierarchy (and should be an instance of ClStrength), while weight is a float indicating the actual weight of the constraint at its indicated strength, or nil/null if it does not have a weight. (Weights are only relevant for weighted-sum-better comparators, not for locally-error-better ones.)

Constraints understand various messages that return true or false regarding some aspect of the constraint, such as isRequired, isEditConstraint, isStayConstraint, and isInequality.

ClLinearConstraint is an abstract subclass of ClConstraint. It adds an instance variable expression, which holds an instance of ClLinearExpression. It has two concrete subclasses. An instance of ClLinearEquation represents the linear equality constraint

    expression = 0.

An instance of ClLinearInequality represents the constraint

    expression $\geq$ 0.

The other part of the constraint class hierarchy is for edit and stay constraints (both of which are represented explicitly). ClEditOrStayConstraint has an instance field variable, which is the ClVariable with the edit or stay. Otherwise all that the two concrete subclasses do is respond appropriately to the messages isEditConstraint and isStayConstraint.

This constraint hierarchy is also intended to allow extension to include local propagation constraints (which would be another subclass of ClConstraint)—otherwise I could have made everything be a linear constraint, and eliminated the abstract class ClConstraint entirely.

3.3.6 Constraint Creation

This subsection describes a mechanism to allow constraints to be defined easily by programmers. The convenience afforded by my toolkit varies among languages. Smalltalk's dynamic nature makes it the most expressive. C++'s operator overloading still permits using natural infix notation. Java, however, requires using ordinary methods, and leaves us with the single option of prefix expressions when building constraints.

In Smalltalk, the messages +, -, *, and / are defined for ClVariable and ClLinearExpression to allow convenient creation of constraints by programmers. Also, ClVariable and ClLinearExpression, as well as Number, define cnEqual:, cnGEQ:, and cnLEQ: to return linear equality or inequality constraints. Thus, the Smalltalk expression

```
3*x+5 cnLEQ: y
```

returns an instance of ClLinearEquality representing the constraint $3x + 5 \leq y$. The expression is evaluated as follows: the number 3 gets the message * x. Since x is not a number, 3 sends the message * 3 to x. x is an instance of ClVariable, which understands * to return a new linear expression with a single term, namely itself times the argument. (If the argument is not a number it raises an exception that the expression is non-linear.) The linear expression representing $3x$ gets the message + with the argument 5, and returns a new linear expression representing $3x + 5$. This linear expression gets the message cnLEQ: with the argument y. It computes a new linear expression representing $y - 3x - 5$, and then returns an instance of ClLinearInequality with this expression.

(It is tempting to make this nicer by using the =, <=, and >= messages, so that one could write

```
3*x+5 <= y
```

instead but because the rest of Smalltalk expects =, <=, and >= to perform a test and return a boolean, rather than to return a constraint, this would not be a good idea.)

Similarly, in C++ the arithmetic operators are overloaded to build ClLinearExpressions from ClVariables and other ClLinearExpressions. Actual constraints are built using various constructors for ClLinearEquation or ClLinearInequality. An enumeration defines the symbolic constants cnLEQ and cnGEQ to approximate the Smalltalk interface. For example:

```
ClLinearInequality cn(3*x+5, cnLEQ, y);      // C++
```

build the constraint cn representing $3x + 5 \leq y$. In Java, the same constraint would be built as follows:

```
ClLinearInequality cn =
        new ClLinearInequality(CL.Plus(CL.Times(x,3),5), CL.LEQ, y);
```

Although the Java implementation makes it more difficult to express programmer-written constraints, this inconvenience is relatively unimportant when the solver is used in conjunction with graphical user interfaces for specifying the constraints.

3.3.7 Symbolic Weights and Strengths

The constraint hierarchy theory allows an arbitrary (although finite) number of strengths of constraint. In practice, however, programmers use a small number of strengths in a stylized way. The current implementation therefore includes a small number of pre-defined strengths, and the maximum number of strengths is defined as a constant. (This constant can be changed as discussed below.)

The strengths provided in the current release are:

required Required constraints must be satisfied exactly. A common use of a required constraint is to give a shorthand name to an expression such as: win.right = win.left + win.width

strong This strength is conventionally used for edit constraints.

medium This strength can be used for strong stay constraints; for example, we might put medium strength stay constraints on the width and height of an object and weak stay constraints on its position to represent our preference that the object move instead of change size when either is possible to maintain the stronger constraints.

weak This strength is used for most stay constraints.

Each strength category is represented as an instance of ClStrength.

A related class is ClSymbolicWeight. As mentioned in Section 3.2.3, the objective function is formed as the weighted sum of the positive and negative errors for the non-required constraints. The weights should be such that the stronger constraints totally dominate the weaker ones. In general, to pick a real number for the weight we need to know how big the values of the variables might become. To avoid this problem altogether, in the Smalltalk and C++ implementations we use symbolic weights and a lexicographic ordering for the weights rather than real numbers, which ensures that strong constraints are always satisfied in preference to weak ones.

Instances of ClSymbolicWeight are used to represent these symbolic weights. These instances have an array of floating point numbers, whose length is the number of non-required strengths (so three as described here). Each element of the array represents the value at that strength, so $[1.0, 0.0, 10.0]$ represents a weight of 1.0 strong, 0.0 medium, and 10.0 weak. (In Smalltalk, ClSymbolicWeight is a variable length subclass; it could have had an instance variable with an array of length 3 instead.) Symbolic weights understand various arithmetic messages (or operator overloading in C++) as follows:

+ w

w is also a symbolic weight. Return the result of adding w to self (or this in C++).

– w

w is also a symbolic weight. Return the result of subtracting w from self.

* n

n is a number. Return the result of multiplying self by n.

/ n

n is a number. Return the result of dividing self by n.

$<=$ n, $>=$ n, $<$ n, $>$ n, $=$ n

w is a symbolic weight. Return true if self is related to n as the operator normally queries.

negative

Return true if this symbolic weight is negative (i.e., it does not consist of all zeros and the first non-zero number is negative).

Instances of ClStrength represent a strength in the constraint hierarchy. The instance variables are name (for printing purposes) and symbolicWeight, which is the unit symbolic weight for this strength. Thus, with the 3 strengths as above, strong is $[1.0, 0.0, 0.0]$, medium is $[0.0, 1.0, 0.0]$, and weak is $[0.0, 0.0, 1.0]$.

The above arithmetic messages let the Smalltalk implementation of the solver use symbolic weights just like numbers in expressions. This interface is important because the objective row in the tableau has coefficients which are ClSymbolicWeights, but which are subject to the same manipulations as the other tableau rows whose expressions have coefficients that are just real numbers.

In both C++ and Java, an additional message called asDouble() is understood by ClSymbolicWeights. This converts the representation to a real number that approximates the total ordering suggested by the more general vector of real numbers. It is these real numbers that are used as the coefficients in the objective row of the tableau instead of ClSymbolicWeights (which the coefficients conceptually are). This kludge avoids the complexities that such genericity introduces to the static type systems of C++ and Java.

Also, since Java lacks operator overloading, the above operations are invoked using suggestive alphabetic method names such as add, subtract, times, and lessThan.

3.3.8 ClSimplexSolver *implementation*

Here are the instance variables of ClSimplexSolver. Some fields are inherited from ClTableau, the base class of ClSimplexSolver which provides the basic sparse-matrix interface (Section 3.3.9).

rows

A dictionary with keys ClAbstractVariable and values ClLinearExpression. This holds the tableau. Note that the keys can be either restricted or unrestricted variables, i.e., both C_U and C_S are actually merged into one tableau. This simplifies the code considerably, since most operations are applied to both restricted and unrestricted rows.

columns

A dictionary with keys ClAbstractVariable and values Set of ClAbstractVariable. These are the column cross-indices. Each parametric variable p is a key in this dictionary. The corresponding set includes exactly those basic variables whose linear expression includes p (p will of course have a non-zero coefficient). The keys can be either unrestricted or restricted variables.

objective

An instance of ClObjectiveVariable (named z) that is the key for the objective row in the tableau.

infeasibleRows

A set of basic variables that have infeasible rows. This field is used when re-optimizing with the dual simplex method.

prevEditConstants

An array of constants (floats) for the edit constraints on the previous iteration. The elements in this array must be in the same order as editPlusErrorVars and editMinusErrorVars, and the argument to the public resolve: message.

stayPlusErrorVars, stayMinusErrorVars

An array of plus/minus error variables (instances of ClSlackVariable) for the stay constraints. The corresponding negative/positive error variable must have the same index in stayMinusErrorVars/stayPlusErrorVars.

editPlusErrorVars, editMinusErrorVars

An array of plus/minus error variables (instances of ClSlackVariable) for the edit constraints. The corresponding negative/positive error variable must have the same index in editMinusErrorVars/editPlusErrorVars.

markerVars

A dictionary whose keys are constraints and whose values are instances of a subclass of ClAbstractVariable. This dictionary is used to find the marker variable for a constraint when deleting that constraint. A secondary use is that iterating through the keys will give all of the original constraints (useful for the reset method).

errorVars

A dictionary whose keys are constraints and whose values are arrays of ClSlackVariable. This dictionary gives the error variable (or variables) for a given non-required constraint. We need this if the constraint is deleted because the corresponding error variables must be deleted from the objective function.

3.3.9 ClTableau *(Sparse Matrix) Operations*

The basic requirements for the tableau representation are that one should be able to perform the following operations efficiently:

- determine whether a variable is basic or parametric
- find the corresponding expression for a basic variable
- iterate through all the parametric variables with non-zero coefficients in a given row

- find all the rows that contain a given parametric variable with a non-zero coefficient
- add/remove a row
- remove a parametric variable
- substitute out a variable (i.e., replace all occurrences of a variable with an expression, updating the tableau as appropriate).

The representation of the tableau as a dictionary of rows, with column cross-indices, supports these operations. Keeping the cross indices up-to-date and consistent with the row dictionary is error-prone. Thus, the solver actually accesses the rows and columns only via the following interface of ClTableau.

addRow(ClAbstractVariable var, ClLinearExpression expr)

Add the constraint var=expr to the tableau. var will become a basic variable. Update the column cross indices.

noteAddedVariable(ClAbstractVariable var, ClAbstractVariable subject)

Variable var has been added to the linear expression for subject. Update the column cross indices.

noteRemovedVariable(ClAbstractVariable var, ClAbstractVariable subject)

Variable var has been removed from the linear expression for subject. Update the column cross indices.

removeColumn(ClAbstractVariable var)

Remove the parametric variable var from the tableau. This operation involves removing the column cross index for var and removing var from every expression in rows in which it occurs.

removeRow(ClAbstractVariable var)

Remove the basic variable var from the tableau. Because var is basic, there should be a row var=expr. Remove this row, and also update the column cross indices.

substituteOut(ClAbstractVariable var, ClLinearExpression expr)

Replace all occurrences of var with expr and update the column cross indices.

3.3.10 Omissions

The solver should implement Bland's anti-cycling rule [101], but it does not at the moment. Adding this should be straightforward if it ever proves necessary.

3.4 Comparator details

Cassowary favors solutions that satisfies some of the constraints completely, rather than ones that, for example, partially satisfy each of two conflicting equalities. These are still legitimate locally-error-better and weighted-sum-better solutions. Cassowary's behavior is analogous to that of the simplex algorithm, which always finds solutions at a vertex of the polytope even if all the solutions on an edge or face are equally good. (Of course, Cassowary behaves this way because the simplex algorithm does.)

Such solutions are also produced by greedy constraint satisfaction algorithms, including local propagation algorithms such as DeltaBlue [127] and Indigo [11]: these algorithms try to satisfy constraints one at a time, and in effect the constraints considered first are given a stronger strength than those considered later.

However, there is an issue regarding comparators and Cassowary that has not yet been resolved in an entirely clean way. One of the public methods for Cassowary is addPointStays: points, as discussed in Section 3.3.2. This method addresses the desire to satisfy the stays on both the x and y components of a given point rather than on the x component of one point and the y component of another.

As an example of why this is useful, consider a line with endpoints p1 and p2 and a midpoint m. There are constraints (p1.x+p2.x)/2 = m.x and (p1.y+p2.y)/2 = m.y. Suppose we are editing

m. It would look strange to satisfy the stay constraints on p1.x and p2.y, rather than both stays on p1 or both stays on p2. (This claim has been verified empirically—in early prototype Cassowary implementations this happened, and indeed it looked strange.)

The current implementation of addPointStays: points uses different weights for the stay constraints for successive elements of points—a kludge that seems to work well in practice.

It was difficult to devise an example where it would give a bad answer—here is a contrived one. Suppose we have a line with endpoints p1 and p2 and a midpoint m, and we are moving m. Suppose also we have constraints p2.x = 2*p3.x and p2.y = 2*p3.y. (This example is a bit strange since here we are using p3 as a distance from the origin rather than as a location—otherwise multiplying it by 2 is problematic.) If we give these points to addPointStays: in the order p1, p2, and p3, then the stays on p1 will have weight 1, those on p2 will have weight 0.5, and those on p3 will have weight 0.25. Then, a one legitimate weighted-sum-better solution would satisfy the stays on p1.x and p1.y, but another legitimate weighted-sum-better solution would satisfy the stays on p1.x, p2.y, and p3.y.

Here is a cleaner way to handle this situation. We first introduce a new comparator with the dubious name of *tilted-locally-error-better* (TLEB). The set of TLEB solutions can be defined by taking a given hierarchy, forming all possible hierarchies by breaking strength ties in all possible ways to form a totally ordered set of constraints, and taking the union of the sets of solutions to each of these totally ordered hierarchies.

For example, consider the two constraints weak $x = 0$ and weak $x = 10$. The set of locally-error-better solutions is the infinite set of mappings from x to each number in $[0 \ldots 10]$. Assuming equal weights on the constraints, the (single) least-squares solution is $\{x \mapsto 5\}$. The TLEB solutions are defined by producing all the totally ordered hierarchies and taking the union of their solutions. In this case the two possible total orderings are:

weak $x = 0$, slightly_weaker $x = 10$
slightly_weaker $x = 0$, weak $x = 10$

These have solutions $\{x \mapsto 0\}$ and $\{x \mapsto 10\}$ respectively, so the set of TLEB solutions to the original hierarchy is $\{\{x \mapsto 0\}, \{x \mapsto 10\}\}$.

A *compound constraint* is a conjunction of primitive constraints, in this case linear equalities or inequalities. For compound constraints, when we break the strength ties in defining the set of tilted-locally-error-better solutions, we insist on mapping each linear equality or inequality in a compound constraint to an adjacent strength. (I have been a bit imprecise in the use of the term "constraint" in this chapter, sometimes using it to denote a primitive constraint and sometimes to denote a conjunction of primitive constraints. For the present definition, however, we need to distinguish compound constraints that have been specifically identified as such by the user from conjunctions of primitive constraints more generally, such as the constraints C_S and C_U discussed in Section 3.2.1.)

Now, to define addPointStays: in a more clean way, we could make each point stay a compound constraint. To illustrate why this works, consider the midpoint example again. We have two endpoints p1 and p2, and a midpoint m. There are constraints (p1.x+p2.x)/2 = m.x and (p1.y+p2.y)/2 = m.y, and we are editing m. Then the stays on p1 and p2 will each be compound constraints:

weak $(stay\ p1.x\ \&\ stay\ p1.y)$
weak $(stay\ p2.x\ \&\ stay\ p2.y)$

In defining the set of tilted-locally-error-better solutions, the total orderings of these constraints that we will consider have the stays on $p1.x$ and $p1.y$ both stronger than those on $p2.x$ and $p2.y$, or both weaker. Thus, we produce the desired result.

It is not sufficient just to define a notion of "compound constraint" without adding the notion of tilting. Otherwise if we were using locally-error-better, we would just sum the errors of the primitive constraints which would allow us to trade off the errors arbitrarily; we could still satisfy the stay on the x component of one point and the y component of another.

Regarding other comparators, tilting is incompatible with the weighted-sum comparator. Using tilting to break a tie between two constraints with the same strength and weight could lead to incorrect results because one of the constraints would dominate a third constraint with the same strength and larger weight. In any case, the weights already can be used to break ties among constraints with the same strength.

3.5 Empirical evaluation

Cassowary has been implemented in Smalltalk, C++, and Java. I ran some benchmarks using test problems that tried to add 300 randomly-generated constraints using 300 variables, and 900 randomly-generated constraints using 900 variables.

With the Smalltalk implementation of Cassowary on the 300-constraint benchmark problem, adding a constraint takes on average 38 msec (including the initial solve), deleting a constraint 46 msec, and resolving as the point moves 15 msec. (Stay and edit constraints are represented explicitly in this implementation, so there were also stay constraints on each variable, plus two edit constraints, for a total of 602 constraints minus the constraints that, if added, would have resulted in an unsatisfiable system.) For the 900 constraint problem, adding a constraint takes on average 98 msec, deleting a constraint 151 msec, and resolving as the point moves 45 msec. These tests were run using an implementation in OTI Smalltalk Version 4.0 running on a IBM Thinkpad 760EL laptop computer.

For the C++ implementation on the problem with 900 constraints and variables, adding a constraint takes 15 msec, deleting a constraint 1. 2 msec, and resolving as the point moves 1. 4 msec. These tests were run on a Pentium III/450 running Linux 2.2.5 and compiled with GCC-2.95.2. The Java implementation under the basic Sun JDK 1.2 (no JIT compiler) is about 3 to 8 times slower than the C++ implementation.

The various implementations of Cassowary are actively being used. A Scheme wrapping of the C++ implementation is used in SCWM, the Scheme Constraints Window Manager (Chapter 4). I have also embedded the C++ implementation in a prototype web browser that supports my constraint-based extension to Cascading Style Sheets (Chapter 5), and in a prototype implementation of a constraint extension to Scalable Vector Graphics, CSVG (Chapter 6). A demonstration Constraint Drawing Application using the Java implementation was written by Michael Noth and is included with the Cassowary toolkit. Another Cassowary application (developed using a different Java implementation) is a web authoring tool [16] in which the appearance of a page is determined by the combination of constraints from both the web author and the viewer. Cassowary has also been used in a non-interactive application to perform consistency checks in a planning application [145].

3.6 Summary

Cassowary is a constraint satisfaction algorithm specialized for interactive user interfaces that handles simultaneous linear equations and inequalities. Because of the minimal update of the tableau which is performed, it is (perhaps surprisingly) fast on the operation of incrementally resolving the system. That operation's efficiency is crucial for interactive redrawing diagrams during editing.

Additionally, because Cassowary handles cycles in the constraint graph without difficulty, users of the Cassowary toolkit can concentrate on exploiting the additional expressiveness that the library provides; the declarative nature of constraints is not undermined by a need to understand the algorithm. Cassowary has proven to be efficient and expressive enough to be used in many applications, including those described in the following three chapters.

Chapter 4

THE SCHEME CONSTRAINTS WINDOW MANAGER

4.1 Introduction

I desired a platform for researching advanced window layout paradigms, including the use of constraints. With constraints, related windows can be grouped together visually, subsets of windows can be tiled (instead of overlapping), and other layout desires can be maintained.

Typical window management applications for the X windows system are written entirely in a low-level systems language such as C or C++. Because the X windows libraries have a native C interface, using C is justified. However, a low-level language is far from ideal when prototyping implementations of sophisticated window manager functionality. For these purposes, a higher-level language is much more appropriate, powerful, and satisfying.

Using C to implement a highly-interactive application also complicates extensibility and customizability. To add a new feature, the user likely must write C code, recompile, relink, and restart the application before changes are finally available for testing and use. This development cycle is especially problematic for software such as a window manager that generally is expected to run for weeks at a time. Additionally, maintaining all the features that any user desires would result in terrible code bloat.

An increasingly popular solution to these problems is the use of a scripting language on top of a core system that defines new domain-specific primitives. A prime example of this architecture is Richard Stallman's GNU Emacs text editor [132]. In the twenty years since the introduction of Emacs, numerous extensible scripting languages have evolved including Tcl [117], Python [95], Perl [142], and Guile [67, 123]. Each of the first three languages was designed from scratch with scripting in mind. In contrast, Guile—the GNU Ubiquitous Intelligent Language for Extension—takes a pre-existing language, Scheme, and adapts it for use as an extension language.

Because I am interested in practical use of constraints, I decided to target the X windows system and build a complete window manager for X/11. I chose to use Guile/Scheme as the extension language for my project named SCWM—the Scheme Constraints Window Manager [5, 6]. The most notable feature of SCWM is constraint-based layout. Whereas typical window management systems use only direct manipulation [129] of windows, SCWM also supports a user-interface for specifying constraints among windows that it then maintains using Cassowary (Chapter 3). Much of the advanced functionality of SCWM is implemented in Scheme, thus exploiting the embedded-extension-language architecture.

The next section discusses some background issues. Section 4.3 describes the SCWM system in detail and discusses some of the benefits and difficulties that arise in using Guile/Scheme. Section 4.4 describes the constraint-based capabilities of SCWM. Section 4.6 mentions related work, and Section 4.7 discusses possible future work on SCWM.

4.2 Background

SCWM leverages numerous existing technologies to provide its infrastructure and support its advanced capabilities.

4.2.1 X Windows and `fvwm2`

A fundamental design decision for the X windows system [115] was to permit an arbitrary user-level application to manage the various application windows. This open architecture permits great flexibility in the way windows look and behave.

X window managers are complex applications. They are responsible for decorating top-level application windows (e.g., drawing labelled titlebars with buttons), permitting resizing and moving of windows, iconifying, tiling, cascading windows, and much more. Many Xlib library functions wrapping the X protocol are specific to the special needs of window managers. Because my goal is to do interesting research beyond that of modern window managers, I used an existing popular window manager, `fvwm2`, for a starting point [51]. In 1997 when the SCWM project began, `fvwm2` was arguably the most used window manager in the X windows community. It supports flexible configuration capabilities via a per-user `.fvwm2rc` file that is loaded once when `fvwm2` starts. To tweak parameters, end-users edit their `.fvwm2rc` files using an ordinary text editor, save the changes, then restart the window manager to activate the changes. The `fvwm2` configuration language supports a very restricted form of functional abstraction, but lacks loops and conditionals.

Despite these shortcomings, `fvwm2` provides a good amount of control over the look of windows. It also has evolved over the years to meet complex specifications (e.g., the Interclient Communication Conventions Manual [121]) and to deal with innumerable quirks of applications. By basing SCWM on `fvwm2`, I leveraged those capabilities and ensured that SCWM was at least as well-behaved as `fvwm2`. My fundamental change to `fvwm2` was to replace its ad-hoc configuration language with Guile/Scheme [67].

4.2.2 Scheme for extensibility

Guile [67] is the GNU project's R4RS-compliant Scheme [30] system designed specifically for use as an embedded interpreter. Scheme is a very simple, elegant dialect of the long-popular Lisp programming language. It is easy to learn and provides exceptionally powerful abstraction capabilities including higher-order functions, lexically-scoped closures and a hygienic macro system. Guile extends the standard Scheme language with a module system and numerous wrappers for system libraries (e.g., POSIX file operations).

4.3 The system

SCWM is a complex software system that emphasizes extensibility and customizability to enable sophisticated capabilities to be developed and tested quickly and easily. The window manager embraces the embedded-scripting language architecture first introduced by Emacs (Section 4.3.1) and it further exploits Guile/Scheme's support for dynamically-loadable binary modules (Section 4.3.2).

This extra power and extensibility can complicate using the window manager. To simplify configuration, SCWM provides developers with an easy way to define various options declaratively. SCWM then automatically builds a graphical user-interface that enables end-users to manipulate those parameters easily (Section 4.3.3).

Another challenge was embedding Cassowary in SCWM reasonably non-invasively. Avoiding altering all of the functions that access or mutate fields of the window structure required exploiting some of the extensibility features built into the constraint solving toolkit (Section 4.3.4).

The current implementation of SCWM contains roughly 32,500 non-comment, non-blank lines of C code, 800 lines of C++ code, and 25,000 lines of Scheme code. The Guile/Scheme system is about 44,000 lines of C code and 11,500 lines of Scheme code. Finally, the Cassowary constraint solving toolkit is about 9,500 lines of C++ code in its core, plus about 1,400 lines of C++ code in the Guile wrapper.

4.3.1 Basic philosophy

My first version of SCWM was a simple derivative of its predecessor, `fvwm2`, with the ad-hoc configuration language replaced by Guile/Scheme. Like `fvwm2`, SCWM reads a startup file containing all of the commands to initialize the settings of various options. Most `fvwm2` commands have reasonably straightforward translations to SCWM expressions. For example, the following `fvwm2` configuration lines set the default coloring style for windows, choose the colors for the highlighted window, define a function, and bind a key to that function.

```
Style "*" ForeColor black
Style "*" BackColor grey76

HilightColor  white navyblue

AddToFunc Raise-and-Stick
+ "I" Raise
+ "I" Stick

Key s WT CSM Function Raise-and-Stick
```

Those lines are rewritten for SCWM in Guile/Scheme as:[1]

```
(window-style "*" #:fg "black"
                  #:bg "grey76")

(set-highlight-foreground! "white")
(set-highlight-background! "navyblue")
```

[1]Because the `fvwm2` configuration language is so limited, it is possible to mechanically convert to SCWM commands; I provide a reasonably-complete automated translator for this purpose.

```
SCWM_PROC( X_property_get,
          "X-property-get",
          2, 1, 0,
      (SCM win, SCM name, SCM consume_p))
/** Get X property NAME of window WIN. */
#define FUNC_NAME s_X_property_get
{
 SCM answer;
 VALIDARG_WIN_ROOTSYM_OR_NUM_COPY(1,win,w);
 VALIDARG_STRING_COPY(2,name,aprop);
 VALIDARG_BOOL_COPY_USE_F(3,consume_p,del);
 ...
 XGetWindowProperty(...);
 ... answer = ...;
 return answer;
}
#undef FUNC_NAME
```

Figure 4.1: An example SCWM primitive.

```
(define*-public (window-class #&optional (win (get-window)))
  "Return the class of window WIN."
  (X-property-get win "WM_CLASS"))
```

Figure 4.2: The “`window-class`” procedure.

```
(define* (raise-and-stick
         #&optional (win (get-window)))
   (raise-window win)
   (stick win))

(bind-key ’(window title) "C-S-M-s"
          raise-and-stick)
```

The simpler and more regular syntax is more convenient for the end-user. An even greater advantage of using a real programming language instead of a static configuration language stems from the ability to extend the set of commands (either by writing C or Scheme code) and to combine those new procedures arbitrarily.

Adding a new SCWM primitive is easily done by writing a new C function that registers itself with the Guile interpreter. For example, after implementing an “`X-property-get`” primitive in C (Figure 4.1), we can write a new procedure to report a window’s class, which is just the value of its `WM_CLASS` property (Figure 4.2). Then we can use that `window-class` procedure interactively by writing:

```
(bind-key ’all "C-S-M-f"
 (lambda ()
  (let* ((win (window-with-focus))
         (class (window-class win)))
    (if (string=? class "Emacs")
        (resize-window 500 700 win)
        (resize-window 400 300 win)))))
```

The above expressions, when evaluated in SCWM’s interpreter, will make the user’s “`Control + Shift + Meta + f`” keystroke resize the window to 500 $\times$ 700 pixels if the currently-focused window is an `Emacs` application window, or 400 $\times$ 300 pixels otherwise.

SCWM’s extensible architecture also allows Guile extensions to be accessible from the window manager. Via standard Guile modules, SCWM can read and parse web pages, download files via ftp, do regular expression matching, and much more. In fact, nearly all of the user-interface elements in SCWM are built using `guile-gtk`, a Guile wrapper of the GTk+ toolkit.

4.3.2 Binary modules

Because each user only needs a subset of the full functionality that SCWM provides, it is important that users only pay for the features they require (in terms of size of the process image). Guile, unlike Emacs Lisp, allows new primitives to be defined by dynamically-loadable binary modules. Without this feature, all primitives would need to be contained in the SCWM core, thus complicating the source code and increasing the size of the resulting monolithic system.

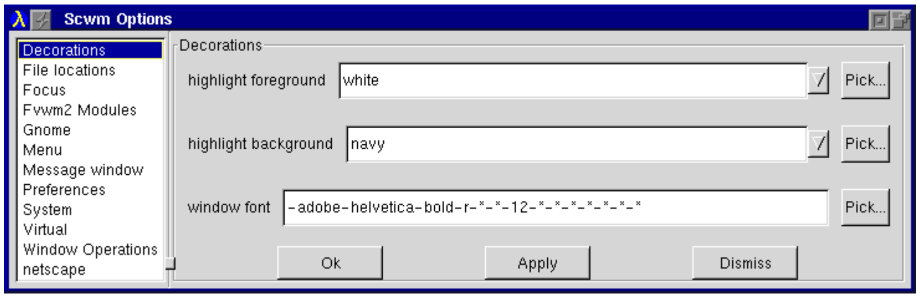
The voice recognition module based on IBM's ViaVoice^{tm} software illustrates the benefits of dynamically-loaded extensions. Some users do not to use that feature—perhaps because the library is not available on their platform or perhaps because they have no audio input device. Those users will never have the module's code loaded. Additionally, if ViaVoice does not exist at compile time, the voice recognition module will not even be built.

Implementing the module was also straightforward. After getting a sample program for the voice recognition engine working, it required less than six hours of development effort to wrap the core functionality of the engine with a Scheme interface. A grammar describes the various utterances that SCWM understands, and the C code asynchronously invokes a Scheme procedure when a phrase is recognized. Because those action procedures are written in Scheme, the responses to phrases can be easily modified and extended without even restarting SCWM.

4.3.3 Graphical configuration

Another example of the extensibility that Guile provides SCWM is the `preferences` system for graphical customization. Novice SCWM users are unlikely to want to write Scheme code to configure the basic settings of their window manager, such as the background color of the currently-active window's titlebar. A graphical user interface is necessary to manage these settings, but there are potentially a huge number of configurable parameters. Undisciplined maintenance of a user interface for those options would be tedious and error-prone.

Fortunately, SCWM can leverage its Scheme extension language to ease these difficulties. The `defoption` module provides a macro `define-scwm-option` that permits declarative specifi-



highlight foreground: white

highlight background: navy

window font: -adobe-helvetica-bold-r-*-*-12-*-*-*-*-*-*-*

Figure 4.3: The automatically-generated options dialog.

cation of a configuration option.[2] To expose a graphical interface to setting the `*highlight-background*` configuration variable, the SCWM developer need simply write:

```
(define-scwm-option
  *highlight-background* "navy"
 "The bg color for focused window."
 #:type 'color
 #:group 'face
 #:setter (lambda (v)
             (set-highlight-background! v))
 #:getter (lambda () (highlight-background)))
```

This code states that `*highlight-background*` is an end-user configurable variable that will contain a value that is a color. It also specifies that the variable can be grouped with other variables into a `face` category. Finally, setter and getter procedures are specified to teach SCWM how to alter and retrieve the value.

The `preferences` module then accumulates all of these specifications and dynamically generates the user interface shown in Figure 4.3.[3] This modular approach also enforces the separation of the visual appearance from the desired functionality—a visually-distinct notebook-style interface with the same functionality is also available.

[2]Recent versions of Emacs [132] provide a similar feature in their “customize” package. The layout of their user-interfaces is simpler, though, as no attempt is made to create a fully graphical interface.

[3]The user interface is written in guile-gtk, a Guile wrapper of the GTk+ widget toolkit [66] that integrates seamlessly with SCWM.

4.3.4 Connecting to Cassowary

The most important module for my research on advanced window layout paradigms is the wrapper of the Cassowary constraint solving toolkit. To connect the constraint solver with the window manager, the variables known to the solver must relate to aspects of the window layout. Each application window object contains four constrainable variables: `x`, `y` (the offsets of the window from the top-left corner of the virtual desktop); and `width`, `height` (the dimensions of the window frame in pixels). When Cassowary finds a new solution to the set of constraints, it invokes a hook for each constraint variable whose value it changes, and invokes another hook after all changes have been made. For SCWM, the constraint-variable-changed hook adds the window that embeds that constraint variable to its "dirty set," and the second hook repositions and resizes all of the windows in the dirty set.

In each window object, the constrainable variables that correspond to the window's position and size mirror the ordinary integer variables that the rest of the application uses. The hooks copy the new values assigned to the constrainable variables into the ordinary variables. This technique avoids modifying the vast majority of the code that manipulates and manages windows. (Bjorn Freeman-Benson discusses these issues in greater detail [47].)

To make it easy for developers to express constraints among windows, the constraint variables embedded in each window are available to Scheme code via the accessor primitives `window-clv-`{`xl,xr,yt,yb,width,height`}, where, for example, `-xl` names the `x` coordinate of the left side of the window and `-yb` abbreviates the `y` coordinate of the bottom of the window.[4] Thus, to keep the tops of two window objects aligned, we can use:

```
(cl-add-constraint solver
   (make-cl-constraint
      (window-clv-yt win1) =
      (window-clv-yt win2)))
```

Although these primitive constraint-creation constructs are sufficient for specifying desired relationships, end-users need a higher-level interface to make use of the solver in their daily activities. The next section describes the graphical interface I built on top of these primitives.

[4]For each window, explicit constraints `xr = x + width` and `yb = y + height` are added automatically by SCWM.

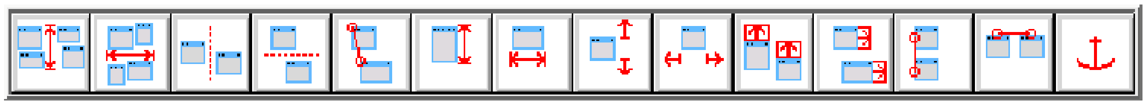


Figure 4.4: SCWM constraint toolbar. The text describes the constraint classes in the same order as they are laid out in the toolbar (from left to right).

4.4 Constraints for layout

Ordinary window managers permit only direct-manipulation as a means of laying out their windows. Although this technique is useful, a constraint-based approach provides a more dynamic and expressive system. In SCWM, I use my Cassowary constraint solving toolkit described in Chapter 3 and Section 4.3.4. On top of the primitive equation-solving capabilities of Cassowary, SCWM adds a graphical user interface that employs an object-oriented design. I specify numerous constraint classes representing kinds of constraint relationships, and instances of each class are added to the system for maintaining relationships among actual windows. The interface allows users to create constraint objects, to manage constraint instances, and to create new constraint classes from existing classes by demonstration.

4.4.1 Applying constraints

Applying constraints to windows is done using a toolbar. Each constraint class in the system is represented by a button on the toolbar (Figure 4.4). The user applies a constraint by clicking a button, then selecting the windows to be constrained. Alternatively, the user can first highlight the windows to be constrained and then click the appropriate button. Icons and tooltips with descriptive text assist the user in understanding what each constraint does. I consulted with a graphic artist on the design of the icons in an effort to make them intuitive and attractive. Preliminary user studies have demonstrated that users can determine the represented relationship reasonably well from the icons even without the supporting tooltip text.

I provide the following constraint classes in SCWM. Many interesting relationships are either present or can be created by combining classes in the list.

Constant Height/Width Sum Keep the total of the height/width of two windows constant.

Horizontal/Vertical Separation Keep one window always to the left of or above another.

Strict Relative Position Maintain the relative positions of two windows.

Vertical/Horizontal Maximum Size Keep the height/width of a window below a threshold.

Vertical/Horizontal Minimum Size Keep the height/width of a window above a threshold.

Vertical/Horizontal Relative Size Keep the change in heights/widths of two windows constant (i.e., resize them by the same amount, together).

Vertical/Horizontal Alignment Align the edge or the center of one window along a vertical/ horizontal line with the edge or center of another window.

Anchor Keep a window in place.

Some of these constraint types can constrain windows in several different ways. For example, the "Vertical Alignment" constraint can align the left edge of one window with the right edge of another or the right edge of one window with the middle of another. Users specify the parameters of the relationship by using window "nonants," the ninefold analogue of quadrants (Figure 4.5). The nonant that the user clicks in dictates the part of the window to which the constraint applies. For example, if the user selects the "Vertical Alignment" constraint and chooses the first window by clicking in any of the east nonants, and the second window by clicking on its left edge, the resulting constraint will align the right edge of the first window in with the left edge of the second. This technique makes some constraint classes, such as alignment, more generally useful. It also decreases the number of buttons on the toolbar, which could otherwise become unwieldy with many narrowly-applicable constraint classes.

4.4.2 Managing constraints

Once a constraint is applied, the user still needs to be able to manage it. Users may wish to disable the constraint temporarily or remove it entirely. They may encounter an odd behavior while they

NW 0	N 1	NE 3
W 3	C 4	E 5
SW 6	S 7	SE 8

Figure 4.5: The nine nonants of a window.

are moving or resizing a window and want to discover which constraint(s) caused the unexpected result, or they may simply be curious to know what constraints are applied to a given window and how that window will interact with other windows. My constraint investigation interface allows for all of these kinds of interactions.

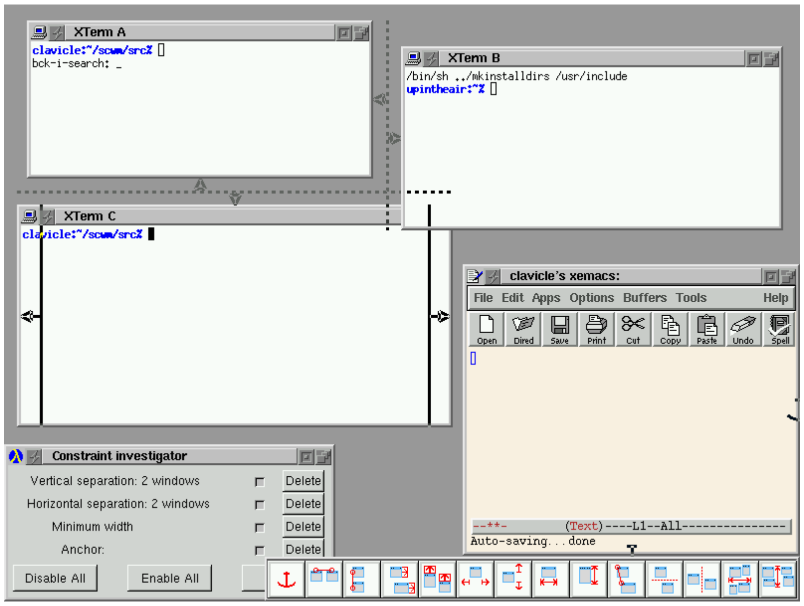
The constraint investigation window allows the user to enable or disable constraints using checkboxes, and to remove constraints using a delete button. The window is dynamically updated as constraints are applied and removed, and changes made in the investigator are immediately reflected in the layout of windows.

When the user moves her mouse pointer over a constraint in the investigator, the representation of that constraint is drawn directly on the windows related by the constraint (Figure 4.6). This hint makes it easy for the user to make the correct associations between windows and constraints. Each constraint class defines its own visual representation, which in most cases closely matches the icon in the toolbar.

Enabling or disabling constraints can result in global rearrangements of windows and large changes in position. To make these discontinuities less confusing, SCWM animates windows fluidly from their old positions and sizes to their new configuration. The animations borrow features from the Self programming environment that mimic cartoon-style animation [26].

4.4.3 Constraint abstractions

A problem with the interface as described thus far is that the basic constraint classes, such as "Vertical Alignment" and "Horizontal Separation," are not always sufficient to convey a user's


XTerm A
XTerm B
XTerm C
clavicle's xemacs:
File Edit Apps Options Buffers Tools Help
Open Dired Save Print Cut Copy Paste Undo Spell
Constraint investigator
Disable All
Enable All

```
clavicle:~/scwm/src%
bck-i-search: _
```

```
/bin/sh ../mkinstalldirs /usr/include
upintheair:~%
```

```
clavicle:~/scwm/src%
```

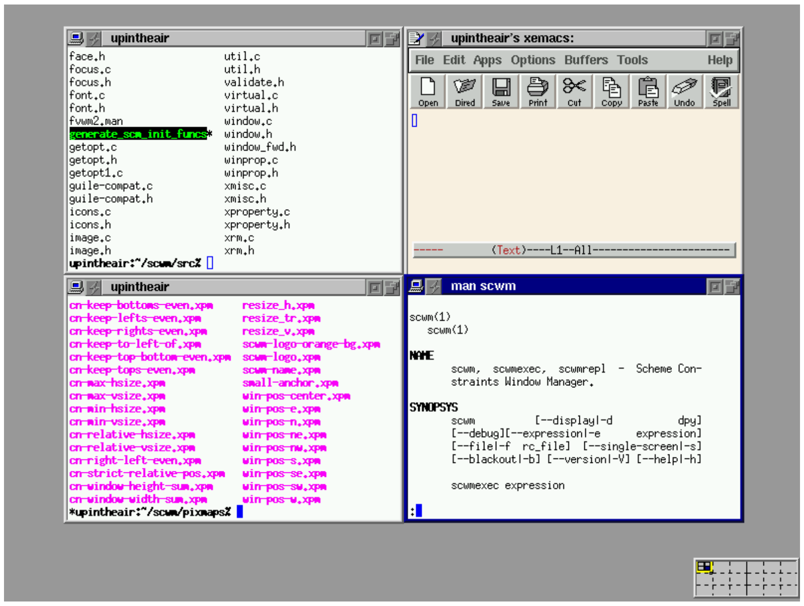
```
--**-        (Text)----L1--All---------------
Auto-saving...done
```

Vertical separation: 2 windows	☐	Delete
Horizontal separation: 2 windows	☐	Delete
Minimum width	☐	Delete
Anchor:	☐	Delete

Figure 4.6: Visual representation of constraints. XTerm A is constrained to be to the left of XTerm B, and above XTerm C. Additionally, XTerm C is required to have a minimum width, and the XEmacs window's south and east edges are anchored at their current locations. The constraint investigator that allows users to manage the constraints instances appears in the bottom left of the screen shot.

intention fully. My own use showed that often one needs to combine several constraints to obtain the desired behavior. A good example of this situation is tiling (Figure 4.7), where two or more windows are aligned next to each other such that they appear to become a window unit of their own. A tiling configuration for two windows can take from three to five constraints to implement. Adding the constraints is tedious when tiling many windows, or when repeatedly tiling and untiling two windows. Certainly a "tiled windows" constraint class could be hard-coded into the system, but that just postpones the problem—some means of abstracting relationships should be provided to the end-user.

A solution to this problem is to support constraint "compositions." A composition is created using a simple programming-by-demonstration technique. SCWM records the user applying a constraint arrangement to some windows in the workspace. The constraints used and the relationships created among the windows are saved into a new constraint class object, which then appears in the


upintheair
upintheair's xemacs:
File Edit Apps Options Buffers Tools Help
Open Dired Save Print Cut Copy Paste Undo Spell
upintheair
man scwm

```
face.h                    util.c
focus.c                   util.h
focus.h                   validate.h
font.c                    virtual.c
font.h                    virtual.h
fvwm2.man                 window.c
generate_scm_init_funcs*  window.h
getopt.c                  window_fwd.h
getopt.h                  winprop.c
getopt1.c                 winprop.h
guile-compat.c            xmisc.c
guile-compat.h            xmisc.h
icons.c                   xproperty.c
icons.h                   xproperty.h
image.c                   xrm.c
image.h                   xrm.h
upintheair:~/scwm/src%
```

```
-----        (Text)----L1--All----------------------
```

```
cn-keep-bottoms-even.xpm     resize_h.xpm
cn-keep-lefts-even.xpm       resize_tr.xpm
cn-keep-rights-even.xpm      resize_v.xpm
cn-keep-to-left-of.xpm       scwm-logo-orange-bg.xpm
cn-keep-top-bottom-even.xpm  scwm-logo.xpm
cn-keep-tops-even.xpm        scwm-name.xpm
cn-max-hsize.xpm             small-anchor.xpm
cn-max-vsize.xpm             win-pos-center.xpm
cn-min-hsize.xpm             win-pos-e.xpm
cn-min-vsize.xpm             win-pos-n.xpm
cn-relative-hsize.xpm        win-pos-ne.xpm
cn-relative-vsize.xpm        win-pos-nw.xpm
cn-right-left-even.xpm       win-pos-s.xpm
cn-strict-relative-pos.xpm   win-pos-se.xpm
cn-window-height-sum.xpm     win-pos-sw.xpm
cn-window-width-sum.xpm      win-pos-w.xpm
*upintheair:~/scwm/pixmaps%
```

```
scwm(1)
   scwm(1)

NAME
       scwm,  scwmexec,  scwmrepl  -  Scheme Con-
       straints Window Manager.

SYNOPSYS
       scwm          [--display|-d          dpy]
       [--debug][--expression|-e     expression]
       [--file|-f  rc_file]  [--single-screen|-s]
       [--blackout|-b] [--version|-V] [--help|-h]

       scwmexec expression

:
```

Figure 4.7: Four windows tiled together. Unlike tiled-only window managers, SCWM permits users to tile a subset of their windows; other windows could overlap arbitrarily.

toolbar like all other constraint classes. Clicking the button in the toolbar will prompt the user to select a number of windows equal to that used in the recording. The constraints will then be applied in the same order as before. Compositions allow users to accumulate a collection of often-used constraint configurations that can then be easily applied.

4.4.4 Inferring constraints

The toolbar-based user interface allows flexible relationships to be specified, but many common user desires reflect very simple constraints. For example, users may place a window directly adjacent to another window and want them to stay together. Some windowing systems provide a basic “snapping” behavior that recognizes when a user puts a window nearly exactly adjacent to another window and then adjusts the window coordinates slightly to have them snap together precisely.

In SCWM, I support a useful extension to basic snapping called “augmented snapping” [53]. Using this technique, the user has the option of transforming a snapped-to relationship to a persistent constraint that is then maintained during subsequent manipulations. When a snap is performed, instead of simply moving the window, the appropriate constraint object is created and added to the system. Such inferred constraints can be manipulated via the constraint investigator described earlier. They also can be removed by simply "ripping-apart" the windows by holding down the `Meta` modifier key while using direct manipulation to move them apart.

4.5 Usability study

I applied a discount usability approach [113] to improve the constraint interface to managing windows. My goal was for users to require no documentation to benefit from the constraint features of SCWM.

4.5.1 Methodology

Six advanced computer users were asked to think aloud while performing three tasks. Each task consists of two parts: discovery and re-creation. First, users manipulate windows with constraints already active to discover and describe those relationships (without use of the constraint investigator). After giving a correct description, they then use the interface on a second display to constrain a fresh set of windows identically. Users were given only a very minimal description of the interface.

The three constraint configurations tested were: 1) a Netscape Find dialog kept in the upper right corner of the main browser window; 2) three windows kept right-aligned along the edge of the screen such that none of the windows overlap nor leaves the top or bottom of the screen; and 3) two windows tiled horizontally.

4.5.2 Results

All users were able to complete their tasks. Discovering the constraints was straightforward—manipulating the windows and observing the behavior was sufficient to deduce the relationships already present. Re-creating the configurations was more troublesome, but users still succeeded. They often used the investigator to remove incorrect constraints, and then they continued onward with an alternate hypothesis.

4.5.3 Problems discovered

My study uncovered numerous usability issues. The most substantial problem involved selecting window parts for the alignment constraints. When performing a vertical alignment, all that matters is whether the user clicks on the left, center, or right third of the window—it is irrelevant whether the click is in the top, middle, or bottom of the window. The interface, however, still highlighted individual corners or edges as it does for anchor constraints where any of the nine positions is significant. Users were confused by the UI distinguishing along the irrelevant vertical dimension. I revised SCWM to highlight whole edges of windows when applying an alignment constraint.

When users began adding a constraint and wanted to cancel, they were unsure of how to abort their action. Some users clicked on the toolbar thinking that it is a special window. Others discovered that clicking on the background results in an error that terminates the operation. No user realized that a right-click aborts, and because several users tried to abort by pressing the `Escape` key, I now support that action to cancel a window selection.

4.5.4 Other observations

The users who performed best studied the tooltip help for each of the toolbar buttons before attempting their first re-creation sub-task. I was surprised at the variety of constraints used in re-creating the configurations: no user matched the expected solution on all three tasks. In particular, the “strict relative position” constraint was used especially advantageously by users who chose to configure windows manually before applying constraints to keep the windows as they were.

Not all users discovered the constraint-visualization feature of the investigator. I now draw the visualizations whenever the user points at any part of the description, not just the enable checkbox. Also, one user wanted to modify the parameters of a constraint in the investigator window directly.

4.6 Related work

There is considerable early work on windowing systems [63, 64, 97, 104, 106, 107]. Many of these projects addressed lower-level concerns that a contemporary X/11 window manager can ignore. An issue that does remain is tiled vs. overlapping windows. SCWM, like nearly all windowing interfaces of the 1990s, chooses overlapping windows for their generality and flexibility. However, unlike other systems, SCWM's constraint solver can permit arbitrary sets of windows to be maintained in a tiled format of a given size.

Although there are dozens of modern window managers in common use on the X windowing platform, only two (besides `fvwm2`) are especially related to SCWM. GWM, the Generic Window Manager, embeds a quirky dialect of Lisp called "WOOL" for Window Object Oriented Language [111]. It supports programmability, and some of its packages, such as directional focus changing, inspired similar modules in SCWM. Sawfish [69] (formerly, Sawmill) is a new window manager with an architecture similar to GWM and SCWM. Like GWM, it embeds its own unique dialect of Lisp (called "rep"). Both embrace the extensibility language architecture and provide low level primitives, then implement other features in their extension language. Neither GWM nor Sawfish has any constraint capabilities, though the hooks they provide can permit procedural implementations to approximate some of the simpler constraint-based behaviours that SCWM implements.

Various other scripting languages exist. As mentioned previously, GNU Emacs and its Emacs Lisp is similar to SCWM in philosophy. The earliest popular general-purpose scripting languages is Tcl, the tool command language [117]. John Ousterhout, Tcl's author, makes a compelling case for the advantages of scripting [118]. Tcl is an incredibly simple but under-powered language that only in the most recent versions includes real data structures. Subsequent similar languages include Python [95] and Perl [142]; both are far more feature-full languages than Tcl, but all three are more commonly used for scripting where the main control resides with the language. SCWM and Emacs both exploit their languages for embedding and invoke scripting code in response to events dispatched by C code.

There are also several other Scheme-based extension languages. Elk [42] is an early Scheme intended as an extension language but is no longer well supported. SIOD (Scheme In One Defun) [131] is an especially compact implementation of Scheme that in return compromises completeness and standards-compliance; it is embedded in the popular GIMP (GNU Image Manipulation Program) application [52] to support user-programmable transformations on images.

4.7 Summary and future work

One of the most useful aspects of this research has been the continuous feedback from end-users throughout the development of SCWM. Since 1997, I have made the latest versions of SCWM and Cassowary (along with all of their source code) available under the GNU General Public License on the Internet, and have actively solicited feedback on support mailing lists. Many of the high-level layout features were developed in response to real-world frustrations and annoyances experienced either by the authors or by the SCWM user community. Although cultivating that community has taken time and effort, I feel that the benefits from user feedback outweigh the costs.

Three years ago when I first began the SCWM project, `fvwm2` was a good choice as a starting point for a new window manager. Since then, though, several other window managers have matured and are far more feature-full than `fvwm2`. Most notably, Enlightenment [68] and WindowMaker [143] are popular powerful window managers that might prove useful as a starting point for a new version of SCWM.

Perhaps the most significant implementation issue for SCWM is its startup time of nearly 20 seconds on a Pentium III 450 class machine. Loading the nearly 20,000 lines of Scheme code at every restart is costly, and wasteful. To address this, one could add an Emacs-like “unexecing” capability to dump the state of a SCWM process that has all of the basic modules loaded. Although this would increase the size of the executable, it also would substantially reduce startup delays. Fortunately, after startup, SCWM’s performance is indistinguishable from other window managers that are written entirely in C.

Another rich area for future work involves the constraint interface. Currently, only constraints among windows are supported. It seems useful to permit the addition of “guide-line” and “guide-point” elements and allow windows to be constrained relative to them. These could, for example, be used to ensure that a window stays in the current viewport, or stays in a specific region of the display. It would also be intriguing to investigate the possibility of ghost-frame objects that are controlled exclusively by SCWM. These window frames could then hold real application windows by dragging them into the frame. This feature would permit hierarchically organizing windows, while still allowing full access to the constraint solver for non-hierarchical relationships.

I am also considering extending the voice-based interface to permit specifying constraints. In SCWM, a user can center a window simply by saying aloud “Center current window.” The voice recognition interface to window layout and control encourages the user to express higher level intention: it is far more awkward to say “move window to 379, 522” than it is to say “move window next to Emacs.” In this way, the voice interface usefully contrasts with direct manipulation where exact coordinates naturally result from the interaction technique. Additionally, voice-based interactions may prove especially valuable for disabled users for whom direct manipulation is difficult.

Discerning a user’s true intention is an interesting complexity of the declarative specification of the current constraints interface. Consider a user who is manipulating three windows, **A**, **B**, and **C**. Suppose the user constrains **A** to be to the left of **B**, and **B** to the left of **C**. Now suppose the application displaying in window **B** terminates, thus removing that window. Should window **A** still be constrained to be to the left of window **C**? In other words, should the transitive constraint that was implicit through window **B** be preserved? The answer depends on the user’s underlying desire. Providing higher-level abstractions for commonly-desired situations may alleviate this ambiguity. For example, if the user had pressed a button to keep three windows horizontally non-overlapping in a row, it is clear that window **B**’s disappearance should not remove the constraint that window **A** remain to the left of **C**.

Finally, I am especially interested in combining my work on constraints and the web (Chapters 5 and 6) with this work on window layout. Web, window, and widget layout are all fundamentally related, and their similarities should ideally be factored out into a unifying framework so that advances made in any area benefit all kinds of flexible, dynamic two-dimensional layout.

Chapter 5

CONSTRAINT CASCADING STYLE SHEETS

5.1 Introduction

Since the inception of the Web there has been tension between the “structuralists” and the “designers.” On one hand, structuralists believe that a Web document should consist only of the content itself and tags indicating the logical structure of the document, with the browser free to determine the document’s appearance. On the other hand, designers (understandably) want to specify the exact appearance of the document rather than leaving it to the browser.

With the recent championing of *style sheets* by the World-Wide-Web Consortium (W3C), this debate has resulted in a compromise. The web document proper should contain the content and structural tags, together with a link to one or more style sheets that determine how the document will be displayed. Thus, there is a clean separation between document structure and appearance, yet the designer has considerable control over the final appearance of the document. W3C has introduced *Cascading Style Sheets*, first CSS 1.0 and now CSS 2.0 [19], for use with HTML documents.

Despite the clear benefits of cascading style sheets, there are several areas in which the CSS 2.0 standard can be improved.

- The designer lacks control over the document’s appearance in environments different from her own. For example, if the document is displayed on a monochrome display, if fonts are not available, or if the browser window is sized differently, then the document’s appearance will often be less than satisfactory.

- CSS 2.0 has seemingly ad hoc restrictions on layout specification. For example, a document element’s appearance can often be specified relative to the parent of the element, but generally not relative to other elements in the document.

- The CSS 2.0 specification is complex and sometimes vague. It relies on procedural descriptions to understand the effect of complex language features, such as table layout. This makes it difficult to understand how features interact.

- Browser support for CSS 2.0 is still limited. This is due in part to the complexity of the specification, but also likely because the specification does not suggest a unifying implementation mechanism.

I argue that constraint-based layout provides a solution to all of these issues, because constraints can be used to specify *declaratively* the desired layout of a web document. They allow partial specification of the layout, which can be combined with other partial specifications in a predictable way. They also provide a uniform mechanism for understanding layout and cascading. Finally, constraint solving technology provides a unifying implementation technique.

I describe a constraint-based extension to CSS 2.0, called *Constraint Cascading Style Sheets* (CCSS). The extension allows the designer to add arbitrary linear arithmetic constraints to the style sheet to control features such as object placement, and finite-domain constraints to control features such as font properties. Constraints may be given a strength, reflecting their relative importance. They may be used in style rules in which case rewritings of the constraint are created for each applicable element. Multiple style sheets are available for the same media type (e.g., paper vs. screen) with preconditions on the style sheets determining which are appropriate for a particular environment and user requirements.

My main technical contributions described in this chapter are:

- A demonstration that constraints provide a powerful unifying formalism for declaratively understanding and specifying CSS 2.0.

- A detailed description of a constraint-based style sheet model, CCSS, which is compatible with virtually all of the CSS 2.0 specification. CCSS allows more flexible specification of layout, and also allows the designer to provide multiple layouts that better meet the desires of the user and environmental restrictions.

- A prototype extension of the Amaya browser that demonstrates the feasibility of CCSS. The prototype makes use of the Cassowary (Chapter 3) and a simple one-way binary acyclic finite-domain solver based on BAFSS [94].

5.2 Background

Cascading Style Sheets (CSS 1.0 in 1997 and CSS 2.0 in 1998) were introduced by the W3C in association with the HTML 4.0 standard. In this section, I review relevant aspects of CSS 2.0 [20] and HTML 4.0 [33].

CSS 2.0 and HTML 4.0 provide a comprehensive set of "style" properties for each type of HTML tag. By setting the value of these properties the document author can control how the browser will display each element. Broadly speaking, properties either specify how to position the element relative to other elements, e.g. `text-indent`, `margin`, or `float`, or how to display the element itself, e.g. `font-size` or `color`.

Although the author can directly annotate elements in the document with style properties, CSS encourages the author to place this information in a separate style sheet and then link or import that file. Thus, the same document may be displayed using different style sheets and the same style sheet may be used for multiple documents, easing maintenance of a uniform look for a web site.

A style sheet consists of *rules*. A rule has a *selector* that specifies the document elements to which the rule applies, and *declarations* that specify the stylistic effect of the rule. The declaration is a set of *property*/*value* pairs. Values may be either absolute or relative to the parent element's value.

For instance, the style sheet in Figure 5.2 has three rules. The first uses the selector `H1` to indicate that it applies to all elements with tag `H1` and specifies that those first-level headings should be displayed using a 13 pt font. The second rule specifies that paragraph elements should use an 11 pt font. The third rule specifies the appearance of text in a `BLOCKQUOTE`, specifying that the font-size should be 90% of that of the surrounding element.

We can use this style sheet to specify the appearance of the HTML document shown in Figure 5.1. Notice the link to the style sheet and that each element includes an `ID` attribute since I will

```
<HTML> <HEAD>
  <TITLE>Simple Example</TITLE>
  <LINK REL="stylesheet"
        HREF="simple.css"
        TYPE="text/css"> </HEAD>
 <BODY>
  <H1 ID=h>Famous Quotes</H1>
  <P ID=p> At a party at Blenheim Palace,
           Lady Astor said to
           Winston Churchill:
   <BLOCKQUOTE ID=q1>
    If I were married to you, I’d put
    poison in your coffee. And he responded:
    <BLOCKQUOTE ID=q2>
     If you were my wife, I’d drink it.
    </BLOCKQUOTE>
   </BLOCKQUOTE>
  </P>
 </BODY>
</HTML>
```

Figure 5.1: Example HTML Document

```
H1 { font-size: 13pt }
P { font-size: 11pt }
BLOCKQUOTE { font-size: 90% }
```

Figure 5.2: A simple Cascading Style Sheet example, simple.css

refer to them later.[1]

Selectors come in three main flavors: *type*, *attribute*, and *contextual*. These may be combined to give more complex selectors. We have already seen examples of a type selector in which the document elements are selected by giving the "type" of their tag. For example, the type-selector `H1` refers to all first-level heading elements. The wildcard type, "`*`", matches all tags.

Attribute selectors choose elements based on the values of two attributes that each element in the document tree may optionally provide: `CLASS` and `ID`. Multiple elements may specify the same `CLASS` value, while the `ID` value should be unique.

Selection based on the `CLASS` and `ID` attributes provides considerable power. By using `CLASS` attributes and selectors, the author can categorize various document elements into groups and then apply different formatting to each of the groups. Similarly, by using `ID` attributes and selectors, the author can single out document elements for special formatting and then refer to them from the style sheet. Elements with a specific class value are selected using the syntax "`.`*class*", while instance ids are selected with "#*id*".

Contextual selectors allow the author to take into account where the element occurs in the document, i.e. its context. They are based on the document's *document tree*, which captures the recursive structure of the tagged elements. A context selector allows selection based on the element's ancestors in the document tree.

For instance, the document in Figure 5.1 has the document tree shown in Figure 5.3. If we want to ensure the innermost block quote does not have its size reduced relative to its parent, we could use

```
BLOCKQUOTE BLOCKQUOTE { font-size: 100% }
```

Less generally, we could individually override the font size for the second `BLOCKQUOTE` by using a rule with an `ID` selector:

```
#Q2 { font-size: 100% }
```

[1]Marking all elements with `ID` attributes defeats the modularity and re-use benefits of CSS; I over-use `ID` tags here strictly as an aid to discussing the examples.

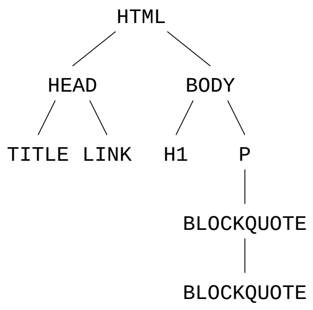


Figure 5.3: Document tree for the HTML of Figure 5.1.

Many style properties are inherited by default from the element's parent in the document tree. Generally speaking, properties that control the appearance of the element itself, such as `font-size`, are inherited, while those that control its positioning are not.

As another example, consider the HTML document shown in Figure 5.4. We can use a style sheet to control the width of the columns in the table. For example, `table.css` (Figure 5.5) contains rules specifying that the elements of the classes `medcol` and `thincol` have widths 30% and 20% of their parent tables, respectively. (Note the use of the class selector "`.`" syntax).

One of the key features of CSS is that it allows multiple style sheets for the same document. Thus a document might be displayed in the context of the author's special style sheet for that document, a default company style sheet, the user's style sheet and the browser's default style sheet. This is handled in CSS by *cascading* the style sheets, permitting each of the sheets to affect the final rendering.

Cascading, inheritance, and multiple style sheet rules matching the same element may mean that there are conflicts among the rules as to what value a particular style property for that element should take. The exact details of which value is chosen are complex. Within the same style sheet, inheritance is weakest, and rules with more specific selectors are preferred to those with less specific selectors. For instance, each of the rules

```
#Q2 { font-size: 100% }
BLOCKQUOTE BLOCKQUOTE { font-size: 100% }
```

is more specific than

```
<HTML> <HEAD>
  <TITLE>Table Example</TITLE>
  <LINK REL="stylesheet"
        HREF="table.css"
        TYPE="text/css"> </HEAD>
 <BODY>
  <TABLE ID=t>
   <COL ID=c1 CLASS=medcol>
   <COL ID=c2>
   <COL ID=c3 CLASS=thincol>
   <TR>
    <TD COLSPAN=2>
     <IMG ID=i1 SRC="back.gif"></TD>
    <TD><IMG ID=i2 SRC="next.gif"></TD>
   </TR>
   <TR>
    <TD>Text1</TD>
    <TD>Text2</TD>
    <TD>Text3</TD>
   </TR>
  </TABLE>
 </BODY>
</HTML>
```

Figure 5.4: Example HTML Document with a Table

```
.medcol { width: 30% }
.thincol { width: 20% }
```

Figure 5.5: Stylesheet for table: table.css

```
BLOCKQUOTE { font-size: 90% }
```

Among style sheets, the values set by the designer are preferred to those of the user and browser, and for otherwise equal conflicting rules, those in a style sheet that is imported or linked first have priority over those subsequently imported or linked. However, a style sheet author may also annotate rules with the strength `!important`, which will override this behavior. In CSS 2.0, for rules designated with strength `!important`, user-specified rules take priority over designer-specified rules.[2]

Despite its power, CSS 2.0 still has a number of limitations. One limitation is that a style property may only be relative to the element's parent, not to other elements in the document. This can result in clumsy specifications, and makes some reasonable layout constraints impossible to express. For example, it is not possible to require that all tables in a document have the same width, and that this should be the smallest width that allows all tables to have reasonable layout. With CSS 2.0, one can only give the tables the same fixed size or the same fixed percentage width of their parent element.

The other main limitation is that it is difficult for the designer to write style sheets that degrade gracefully in the presence of unexpected browser and user limitations and desires. For instance, the author has little control over what happens if the desired fonts sizes are not available. Consider the style sheet `simple.css` again. Imagine that only 10 pt, 12 pt, and 14 pt fonts are available. The browser is free to use 12 pt and 10 pt for headings and paragraphs respectively, or 14 pt and 12 pt, or even 12 pt and 12 pt. Part of the problem is that rules always give definite values to style properties. When different style sheets are combined only one rule can be used to compute the value. Thus a rule is either on or off, leading to discontinuous behavior when style sheets from the author and user are combined. For instance, a sight-impaired user might specify that all font sizes must be greater than 11 pt. However, if the designer has chosen sufficiently large fonts, the user wishes to use the designer's size. This is impossible in CSS 2.0.

[2]This seemingly-inconsistent relative ordering of the `!important` preferences was changed from CSS 1.0 to guarantee that the user has ultimate control over the appearance of a document.

5.3 Constraint Cascading Style Sheets

My solution to these problems is to use constraints for specifying layout. The major advantage of using constraints in this application is that they allow partial specification of the layout, which can be combined with other partial specifications in a predictable way. In this section, I describe my constraint-based extension to CSS 2.0, called *Constraint Cascading Style Sheets* (CCSS).

One complication in the use of constraints is that they may conflict. To allow for this we use the *constraint hierarchy* formalism [14]. In these examples we shall assume the *weighted-sum-better* comparator that sums the errors in satisfying each of the constraints, weighting each error by the strength of that constraint. By using an appropriate set of strength labels, I can model the behavior of CSS 2.0.

5.3.1 A constraint view of CSS 2.0

Hierarchical constraints provide a simple, unifying way of understanding much of the CSS 2.0 specification. This viewpoint also suggests that constraint solvers provide a natural implementation technique. Each style property and the placement of each element in the document can be modeled by a variable. Constraints on these variables arise from browser capabilities, default layout behavior arising from the type of the element, from the document tree structure, and from the application of style rules. The final appearance of the document is determined by finding a solution to these constraints.

The first aspect of CSS 2.0 I consider is the placement of the document elements (i.e., page layout). This can be modeled using linear arithmetic constraints. To illustrate this, let us examine table layout—one of the most complex parts of CSS 2.0. The key difficulty in table layout is that it involves information flowing bottom-up (e.g. from elements to columns) and top-down (e.g. from table to columns). The CSS 2.0 specification is procedural in nature, detailing how this occurs. By using constraints, we can declaratively specify what the browser should do, rather than how to do it. Furthermore, the constraint viewpoint allows a modular specification. For example, to understand how a complex nested table should be laid out, we simply collect the constraints for each component, and the solution to these is the answer. With a procedural specification it is much

(1)	#t[width] = #c1[width]+ #c2[width] + #c3[width]	REQUIRED
(2)	#c1[width] $\geq$ width(“Text1”)	REQUIRED
(3)	#c2[width] $\geq$ width(“Text2”)	REQUIRED
(4)	#c3[width] $\geq$ width(“Text3”)	REQUIRED
(5)	#c3[width] $\geq$ #i2[width]	REQUIRED
(6)	#c1[width] + #c2[width] $\geq$ #i1[width]	REQUIRED
(7)	#t[width] = 0	WEAK
(8)	#c1[width] = $0.3 *$ #t[width]	DESIGNER
(9)	#c3[width] = $0.2 *$ #t[width]	DESIGNER

Figure 5.6: Example layout constraints

harder to understand such interaction.

Consider the style sheet `table.css` (Figure 5.5) and the associated HTML document (Figure 5.4). The associated layout constraints are shown in Figure 5.6. The notation “#id[prop]” refers to the value of the property *prop* for the presentation element corresponding to the document element with ID *id*.[3] Since we are dealing with a table, the system automatically creates a constraint (1) relating the column widths and table width.[4] Similarly, there are automatically created constraints (2–6) that each column is wide enough to hold its content, and (7) that the table has minimal width. Constraints (8) and (9) are generated from the style sheet. Notice the different constraint strengths: from weakest to strongest they are WEAK, DESIGNER and REQUIRED. Since REQUIRED is stronger than DESIGNER, the column will always be big enough to hold its contents. The WEAK constraint `#t[width] = 0` cannot be satisfied exactly; the effect of minimizing its error will be to minimize the width of the table, but not at the expense of any of the other constraints.

These constraints provide a declarative specification of what the browser should do. This approach also suggests an implementation strategy: to lay out the table, we simply use a linear arithmetic constraint solver to find a solution to the constraints. The solver implicitly takes care of the flow of information in both directions, from the fixed widths of the images upward, and from

[3] I use associative array-like syntax for referring to properties of elements to avoid the confusion that the alternative “*.selector*” form would cause due to CSS’s pre-existing use of “.” as a class-name prefix in selectors of rules.

[4] For simplicity, I ignore margins, borders and padding in this example.

(1)	$\#h$[font-size] $\in \{9, 10, 12, 16, 36, 72\}$	REQUIRED
(2)	$\#p$[font-size] $\in \{9, 10, 12, 16, 36, 72\}$	REQUIRED
(3)	$\#q1$[font-size] $\in \{9, 10, 12, 16, 36, 72\}$	REQUIRED
(4)	$\#q2$[font-size] $\in \{9, 10, 12, 16, 36, 72\}$	REQUIRED
(5)	$\#h$[font-size] = 13	DESIGNER
(6)	$\#p$[font-size] = 11	DESIGNER
(7)	$\#q1$[font-size] = $0.9 * \#p$[font-size]	DESIGNER
(8)	$\#q2$[font-size] = $0.9 * \#q1$[font-size]	DESIGNER

Figure 5.7: Example finite domain constraints

the fixed width of the browser frame downward.

Linear arithmetic constraints are not the only type of constraints implicit in the CSS 2.0 specification. There are also constraints over properties that can take only a finite number of different values, including font size, font type, font weight, and color. Such constraints are called *finite domain* constraints and have been widely studied by the constraint programming community [101]. Typically, they consist of a *domain constraint* for each variable giving the set of values the variable can take (e.g., the set of font sizes available) and required arithmetic constraints over the variables.

As an example, consider the constraints arising from the document in Figure 5.1 and style sheet `simple.css` (Figure 5.2). The corresponding constraints are shown in Figure 5.7. The domain constraints (1-4) reflect the browser's available fonts. The remaining constraints result from the style sheet rules. Note that the third rule generates two constraints (7) and (8), one for each block quote element.

Both of the preceding examples have carefully avoided one of the most complex parts of the CSS 2.0 specification: what to do when multiple rules assign conflicting values to an element's style property. As discussed earlier, there are two main aspects to this: cascading several style sheets, and conflicting rules within the same style sheet.

We can model both aspects by means of hierarchical constraints. To do so, we need to refine the constraint strengths we have been using. Apart from REQUIRED, each strength is a lexicographically-ordered tuple

$\#h[\text{font-size}] = 13$	$\langle \text{DESIGNER}, 0, 0, 1\rangle$
$\#p[\text{font-size}] = 11$	$\langle \text{DESIGNER}, 0, 0, 1\rangle$
$\#q1[\text{font-size}] = 0.9 * \#p[\text{font-size}]$	$\langle \text{DESIGNER}, 0, 0, 1\rangle$
$\#q2[\text{font-size}] = 0.9 * \#q1[\text{font-size}]$	$\langle \text{DESIGNER}, 0, 0, 1\rangle$
$\#q2[\text{font-size}] = 1.0 * \#q1[\text{font-size}]$	$\langle \text{DESIGNER}, 0, 0, 2\rangle$

Figure 5.8: Example of overlapping rules

$$\langle cs, i, c, t\rangle.$$

The first component in the tuple, *cs,* is the *constraint importance* and captures the author-suggested strength of the constraint and its position in the cascade. The constraint importance is one of WEAK, BROWSER, USER, DESIGNER, DESIGNER-IMPORTANT, or USER-IMPORTANT (ordered from weakest to strongest). The importance WEAK is used for automatically generated constraints only. The last three components in the tuple capture the specificity of the rule that generated the constraint: *i* is the number of `ID` attributes, *c* is the number of `CLASS` attributes, and *t* is the number of tag names in the rule (i.e., the depth of the contextual selection).

As an example, consider the constraints arising from the document in Figure 5.1 with the style sheet

```
H1 { font-size: 13pt }
P { font-size: 11pt }
BLOCKQUOTE { font-size: 90% }
BLOCKQUOTE BLOCKQUOTE { font-size: 100% }
```

The constraints and their strengths for those directly generated from the style sheet rules are shown in Figure 5.8. Because of its greater weight, the last constraint listed will dominate the second to last one, giving rise to the expected behavior—that the longer contextual selection of a blockquote within a blockquote will govern the appearance of those nested blockquotes.

The remaining issue we must deal with is inheritance of style properties such as font size, and the expression of this inheritance within our constraint formalism. For each inherited property, we need to automatically create an appropriate constraint between each element and its parent. At first

BODY[font-size] = 12	⟨BROWSER, 0, 0, 0⟩
#*h*[font-size] = BODY[font-size]	⟨WEAK, 0, 0, 0⟩
#*p*[font-size] = BODY[font-size]	⟨WEAK, 0, 0, 0⟩
#*q*1[font-size] = #*p*[font-size]	⟨WEAK, 0, 0, 0⟩
#*q*2[font-size] = #*q*1[font-size]	⟨WEAK, 0, 0, 0⟩
#*q*1[font-size] = 8	⟨DESIGNER, 0, 0, 1⟩
#*q*2[font-size] = 8	⟨DESIGNER, 0, 0, 1⟩

Figure 5.9: Example of inheritance rules

glance, these should simply be WEAK equality constraints. Unfortunately, this does not model the inherent directionality of inheritance.

For instance, imagine displaying the document in Figure 5.1 with the style sheet

```
BLOCKQUOTE { font-size: 8pt }
```

where the default font size is 12 pt. The scheme outlined above gives rise to the constraints shown in Figure 5.9. One possible weighted-sum-better solution to these constraints is that the heading is in 12 pt and the rest of the document (including the paragraph) is in 8 pt. The problem is that the paragraph element #*p* has "inherited" its value from its *child*, the `BLOCKQUOTE` element #*q*1.

To capture the directionality of inheritance we use *read-only* annotations [14] on variables that represent inherited attributes. Intuitively, a read-only variable v in a constraint c means that c should not be considered until the constraints involving v as an ordinary variable (i.e., not read-only) have been used to compute v's value.

To model inheritance, we need to add the inheritance equalities with constraint importance of WEAK, and mark the variable corresponding to the parent's property as read-only. The read-only annotation ensures that the constraints are solved in an order corresponding to a top-down traversal of the document tree. To achieve this, we modify the constraints in Figure 5.9 so that each font size variable on the right hand side has a read-only annotation.

5.3.2 Extending CSS 2.0

I have shown how we can use hierarchical constraints to provide a declarative specification for CSS 2.0. There is, however, another advantage in viewing CSS 2.0 in this light. The constraint viewpoint suggests a number of natural extensions that overcome the limitations of the expressiveness of CSS 2.0 discussed previously.

As the above examples indicate, virtually all author and user constraints that are generated from CSS 2.0 either constrain a style property to take a fixed value, or relate it to the parent's style property value. One natural generalization is to allow more general constraints, such as inequalities. Another natural generalization is to allow the constraint to refer to other variables—both variables corresponding to non-parent elements and to "global" variables.

In CCSS, I allow constraints in the declaration of a style sheet rule. The CSS-style `attribute:value` pair is re-interpreted in this context as the constraint `attribute = value`. I prepend all constraint rules with the `constraint` pseudo-property so that CCSS is backwards compatible with browsers supporting only CSS. In a style sheet rule, the constraint can refer to attributes of `parent` and `left-sibling`. For example:

```
P { constraint: font-size <= (parent[parent])[font-size] + 2 }
```

is a rule that applies constraints that relate the font-size of a paragraph element to the font-size of its grandparent element.

CCSS style sheets also allow the author to introduce global constrainable variables using a new `@variable` directive. A variable identifier is lexically the same as a CSS `ID` attribute. The author can express constraints among global constrainable variables and element style properties using a new `@constraint` directive. There are also various global built-in objects (e.g., `Browser`) with their own attributes that can be used.

These extensions add considerable expressive power. For instance, it is now simple to specify that all tables in the document have the same width, and that this is the smallest width that allows all tables to have a reasonable layout:

```
@variable table-width;
TABLE { constraint: width = table-width }
@constraint table-width = 0 !weak;
```

Similarly we can specify two columns `c1` and `c2` in the same (or different) tables have the same width (the smallest for reasonably laying out both):

```
@constraint #c1[width] = #c2[width];
```

It also allows the designer to express preferences in case the desired font is not available. For example adding

```
H1 { constraint: font-size >= 13pt }
P  { constraint: font-size >= 11pt }
```

to `simple.css` (Figure 5.2) will ensure that larger fonts are used if 13 pt and 11 pt fonts are not available.

Finally, a sight-impaired user can express the strong desire to have all font sizes greater than 12 pt:

```
* {constraint: font-size >= 12pt !important}
```

As long as the font size of an element is 12 pt or larger it will not be changed, but smaller fonts will be enlarged.

The style sheet author is not allowed to explicitly specify a constraint to be REQUIRED—that capability would admit the possibility of an unsatisfiable constraint system. Instead, REQUIRED constraints are generated implicitly for capturing relationships inherent in the structure of the layout, such as a table's width being the sum of the widths of its columns.

Providing inequality constraints allows the author to control the document appearance more precisely in the context of browser capabilities and user preferences. Additionally, CCSS allows the author to give alternate style sheets for the same target media. Each style sheet can list preconditions for their applicability using a new `@precondition` directive. For efficiency, the precondition can only refer to various pre-defined variables. The values of these variables will be known (i.e. they will have specific values) at the time the precondition is tested. For example:

```
@precondition Browser[frame-width] >= 800px;
@precondition ColorMonitor = True;
```

I extend the style sheet `@import` directive to permit listing multiple style sheets per line, and the first applicable sheet is used (the others are ignored). If no style sheet's preconditions hold, none are imported. Consider the example directive

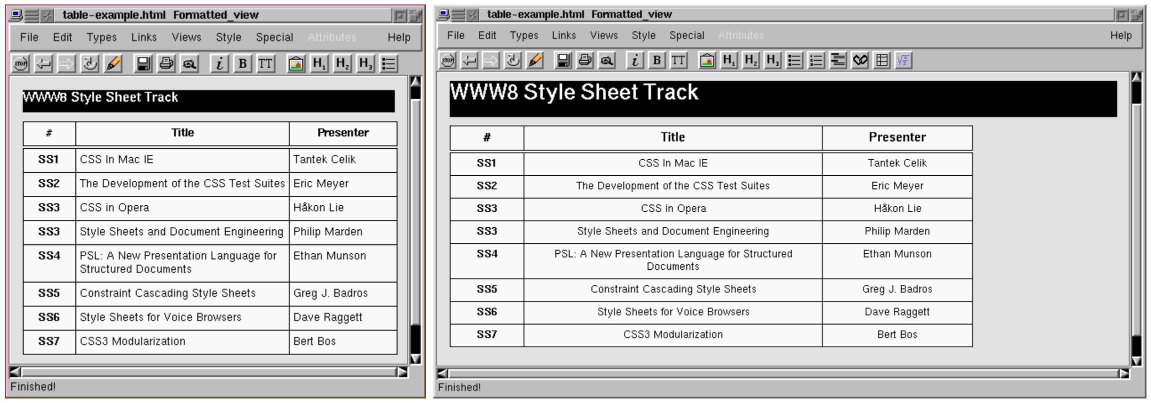


table-example.html Formatted_view
File Edit Types Links Views Style Special Attributes Help
Finished!
table-example.html Formatted_view
File Edit Types Links Views Style Special Attributes Help
Finished!

WWW8 Style Sheet Track

#	Title	Presenter
SS1	CSS In Mac IE	Tantek Celik
SS2	The Development of the CSS Test Suites	Eric Meyer
SS3	CSS in Opera	Håkon Lie
SS3	Style Sheets and Document Engineering	Philip Marden
SS4	PSL: A New Presentation Language for Structured Documents	Ethan Munson
SS5	Constraint Cascading Style Sheets	Greg J. Badros
SS6	Style Sheets for Voice Browsers	Dave Raggett
SS7	CSS3 Modularization	Bert Bos

WWW8 Style Sheet Track

#	Title	Presenter
SS1	CSS In Mac IE	Tantek Celik
SS2	The Development of the CSS Test Suites	Eric Meyer
SS3	CSS in Opera	Håkon Lie
SS3	Style Sheets and Document Engineering	Philip Marden
SS4	PSL: A New Presentation Language for Structured Documents	Ethan Munson
SS5	Constraint Cascading Style Sheets	Greg J. Badros
SS6	Style Sheets for Voice Browsers	Dave Raggett
SS7	CSS3 Modularization	Bert Bos

Figure 5.10: Screen shots of the prototype browser. In the view on the left, a narrow style sheet is in effect because the browser width $\leq$ 800 pixels, while on the right a wide style sheet is used. Interactively changing the browser width dynamically switches between these two presentations. In both figures, the first column is $\frac{1}{4}$ the width of the second column, which is twice the width of the last column. On the left, the table consumes 100% of the frame width, but on the right, the table width is the browser width minus 200 pixels. Also notice the changes in font size and text alignment.

```
@import "wide.css", "tall.css", "small.css";
```

If `wide.css`'s preconditions fail, but `tall.css`'s succeed, the layout uses `tall.css`. If, through the course of the user resizing the top-level browser frame, `wide.css`'s preconditions later become satisfied, the layout does not switch to that style sheet unless `tall.css`'s preconditions are no longer satisfied. That is, the choice among style sheets listed with one directive is only revisited when a currently-used style sheet is no longer applicable.

As an example, consider a style sheet for text with pictures. If the page is wide, the images should appear to the right of the text; if it is narrow, they should appear without text to the left; and if it is too small, the images should not appear at all. This can be encoded as:

```
/* wide.css */
@precondition Browser[frame-width] > 550px;
IMG { float: right}

/* tall.css */
@precondition Browser[frame-width] <= 600px;
@precondition Browser[frame-height] > 550px;
IMG { clear: both; float: none}
```

```
/* small.css */
IMG { display: none }
```

Preconditions become even more expressive in the presence of support for CSS positioning [50] and a generalized `flow` property [21].

5.4 Implementation

5.4.1 Prototype web browser

I have implemented a representative subset of CCSS to demonstrate the additional expressiveness it provides to web designers. My prototype is based on version 1.4a of Amaya [32], the W3 Consortium's browser. Amaya is built on top of Thot, a structured document editor, and has partial support for CSS1. Amaya is exceptionally easy to extend in some ways (e.g., adding new HTML tags), and provides a stable base from which to build.

My support for constraints in Amaya covers the two main domains for constraints that we have discussed: table widths (for illustrating page layout relationships) and font sizes (for illustrating the solution of systems involving inherited attributes). In the prototype, HTML and CSS statements can contain constraints and declare constrainable variables. In HTML statements, constrainable variables, in addition to specific values, can be attached by name to element attributes (e.g., to the "width" of a table column). When the constraints of the document force the values assigned to variables to change, the browser updates its rendering of the current page, much as it does when the browser window is resized (which often caused the re-solve in the first place).

I have also extended Amaya to support preconditions on style sheets and the generalized "`@import`" CCSS rule. The performance when switching among style sheets is similar to a reload, and when the style sheets are cached on disk, performance is good even when switching style sheets during an interactive resize. (It may be useful to provide background pre-fetching of alternate style-sheets to avoid latency when they are first needed.) See Figure 5.10 for screen shots of an example using the prototype's support for both table layout and preconditions. As the support for CSS improves in browsers, more significant variations will be possible through the use of the `@precondition` and extended `@import` directives.

I compared the performance of our prototype browser to an unmodified version of Amaya 1.4a, both fully optimized, running on a PII/400 displaying across a 10Mbit network to a Tektronix X11 server on the same subnet. My test case was a small example on local disk using seven style sheets. I executed 100 re-loads, and measured the total wall time consumed. The unmodified browser performed each re-load and re-render in 190 ms, while the prototype took only 250 ms even when sized to select the last alternative style sheet in each of three `@import` directives. This performance penalty is reasonable given the added expressiveness and features the prototype provides.

One of the most important benefits of re-framing CSS as constraints is that it provides an implementation approach for even the standard CSS features. To simplify the prototype and ensure it remains a superset of CSS functionality, I currently do not treat old-style declarations as constraints, but instead rely on the existing implementation's handling of those rules. However, if designed into a browser from the beginning, treating all CSS rules as syntactic sugar for underlying constraints will result in large savings in code and complexity. The cascading rules would be completely replaced by the constraint solver's more principled assignment of values to variables, and the display engine need only use those provided values, and redraw when the solver changes the current solution.

5.4.2 Constraint solving algorithms

As mentioned previously, the semantics of the constraints is independent of the algorithms used to satisfy them. Nevertheless must select an algorithm that is capable of efficiently finding solutions to the constraints at interactive speeds. My implementation uses two algorithms: Cassowary (Chapter 3) and a restricted version of BAFSS [94].

The Cassowary constraint solving toolkit handles linear arithmetic equality and inequality constraints. It supports the weighted-sum-better comparator [14]. As mentioned earlier, this comparator computes the error for a solution by summing the product of the strength tuple and the error for each constraint that is unsatisfied. To model the CSS importance rules in a hierarchy of constraint strengths, I encode the symbolic levels of importance as tuples as well; for example, USER-IMPORTANT is $\langle 1, 0, 0, 0, 0, 0 \rangle$ and BROWSER is $\langle 0, 0, 0, 0, 1, 0 \rangle$. Thus, no matter what scalar error a BROWSER constraint has, it will never be satisfied if doing so would force a USER-IMPORTANT constraint to not be satisfied. Similarly the last three components of the strength tuple are encoded as $\langle 10^i, 10^c, 10^t \rangle$.[5] Since the Cassowary toolkit operates on constraints with strengths that are a single n-tuple, I internally use 9-tuples to represent strengths—for example,

$$\langle 1, 0, 0, 0, 0, 0, 10^0, 10^1, 10^0 \rangle$$

is the strength of a user-specified `!important` constraint whose selector only contains a single class name.

BAFSS uses a dynamic programming approach to handle systems of font constraints which are binary (i.e., a constraint with two variables) and for which the associated constraint graph is acyclic. For the font constraints implied by CSS, we can simplify the algorithm because all of the constraints relate a read-only size attribute in the parent element to the size attribute of a child element. Given this additional restriction that all constraints are one-way, the algorithm is simple: visit the variable nodes in topological order and assign each a value that greedily minimizes the error contribution from that variable.

Both constraint solvers are implemented within the Cassowary Constraint Solving library described in Chapter 3.

5.5 Related work

The most closely related research is earlier work by Borning, Lin, and Marriott on the use of constraints for web page layout [15]. This system allowed the web page author to construct a document composed of graphic objects and text. The layout of these objects and the text font size were described in a separate “layout sheet” using linear arithmetic constraints and finite domain constraints. Like CCSS, layout sheets had preconditions, controlling their applicability.

The work reported here, which focuses on how to combine constraint-based layout with CSS,

[5]This does not exactly match the CSS specificity rules. For example if the error in a constraint with strength $\langle \text{WEAK}, 0, 0, 1 \rangle$ is 10 times greater than the error in a conflicting constraint with strength $\langle \text{WEAK}, 0, 0, 2 \rangle$, the first constraint will affect the final solution. By choosing appropriate error functions we can make this unlikely to occur in practice. However, the more general constraint hierarchy support may actually permit more desirable interactions rather than the strict strength ordering imposed by CSS.

is complementary to previous research. One of the major technical contributions here is to provide a declarative semantics for CSS based on hierarchical constraints; this issue was not addressed in the prior work [15]. There are two fundamental differences between layout sheets and CCSS. Layout sheets are not style sheets in the sense of CSS, since they can only be used with a single document. Constraints only apply to named elements, and there is no concept of a style rule that applies to multiple elements—the constraints that are used are exactly the constraints that the author has specified. The other fundamental difference between the earlier work [15] and CCSS is that the former has no analogue of the document tree. In essence, the document is modeled as a flat collection of objects; there is no notion of inheritance, and nearly all layout must be explicitly detailed in the layout sheet.

Cascading Style Sheets are not the only kind of style sheet. The Document Style Semantics and Specification Language (DSSSL) is an ISO standard for specifying the format of SGML documents. DSSSL is based on Scheme, and provides both a transformation language and a style language. It is very powerful but complex to use. More recently, W3C has begun designing the XSL style sheet for use with XML documents. XSL is similar in spirit to DSSSL. PSL [98] is another style sheet language; its expressiveness lies midway between that of CSS and XSL. The underlying application model for all three is the same: take the document tree of the original document and apply transformation rules from the style sheet in order to obtain the presentation view of the document, which is then displayable by the viewing device. In the case of XSL, the usual presentation view is an HTML document whose elements are annotated with style properties.

None of these other style sheet languages allow true constraints. Extending any of them to incorporate constraints would offer many of the same benefits as it does for CSS: the ability to flexibly combine user, browser, and designer desires and requirements, and a simple powerful model for layout of complex objects, such as tables. The simplest extension is to allow constraints in the presentation view of the document. (Providing constraints in the transformation rules would seem to offer little advantage.) In the case of DSSSL a natural way to do this is to embed a constraint solver into Scheme (as in SCWM, Chapter 4). In the case of XSL, since HTML is often used as the targeted visual rendering language, the simplest change is to augment that language to be HTML with CCSS style properties. Then the XSL translator would simply generate HTML and a CCSS style sheet, with a CCSS-enhanced browser still performing the dynamic constraint solving, rendering, and interaction.

5.6 Summary and future work

I have demonstrated that hierarchical constraints provide a unifying, declarative semantics for CSS 2.0 and also suggest a simplifying implementation strategy. Furthermore, viewing CSS from the constraint perspective suggests several natural extensions. I call this extension CCSS—Constraint Cascading Style Sheets. By allowing true constraints and style sheet preconditions, CCSS increases the expressiveness of CSS 2.0 and, importantly, allows the designer to write style sheets that combine more flexibly and predictably with user preferences and browser restrictions. I have demonstrated the feasibility of CCSS by modifying the Amaya browser. However, substantial work remains to develop an industrial-strength browser supporting full CCSS, in part because of Amaya's lack of support for CSS 2.0. A more complete implementation will be especially useful for investigating the important issue of how well the constraint systems and solver scale to larger, more complicated designs that further exploit the constraint extensions.

Apart from improving the current implementation, there are two principal directions for further extensions to CCSS. The first is to increase the generality and solving capabilities of the underlying solver. For example, style sheet authors should be able to arbitrarily annotate variables as read-only so that they have greater control over the interactions of global variables. Additionally, virtually all CSS properties, such as color and font weight, could be exposed to the constraint solver once other algorithms are integrated into the solving toolkit.

The second extension is to allow "predicate" selectors in style sheet rules. These selectors would permit an arbitrary predicate to be tested in determining the applicability of a rule to an element in the document structure tree. Predicate selectors can be viewed as a generalization of the existing selectors; an `H1 P` selector is applied only to nodes *n* for which the predicate "*n*[type] = `P` and $\exists m$ parent-of *n* such that *m*[type] = `H1`" holds. These predicate selectors would allow the designer to take into account the attributes of the selected element's parents and children, thus, for instance, allowing the number of items in a list to affect the appearance of the list (as in an example used to motivate PSL [98]).

A final important area for future work is the design, implementation, and user testing of graphical interfaces for writing and debugging Constraint Cascading Style Sheets and web pages that use them.

Chapter 6

CONSTRAINT SCALABLE VECTOR GRAPHICS

6.1 Introduction

Scalable Vector Graphics (SVG) [44] is a language developed by the World Wide Web Consortium (W3C) for describing two dimensional vector graphics. SVG is used for storage and distribution of images on the web, and is increasingly well-supported by both commercial and free software. In contrast with raster image formats such as GIF, JPEG, and PNG, which store a matrix of individual pixels that compose an image, a Scalable Vector Graphic image contains instructions for resolution independent rendering: the same SVG file will be shown in more detail when viewed at a higher resolution (e.g., on a 1200 dots per inch typesetting device rather than a 75 dpi screen). A sample SVG image appears in Figure 6.1.

SVG graphics provide numerous immediate benefits besides resolution independence. SVG files are often smaller than an analogous raster image, thus web pages using them may take less time to download. Because SVG is based on XML [22], SVG files are easy to exchange, process, and analyze. SVG integrates well with Cascading Style Sheets (CSS) [20] specifications, thus

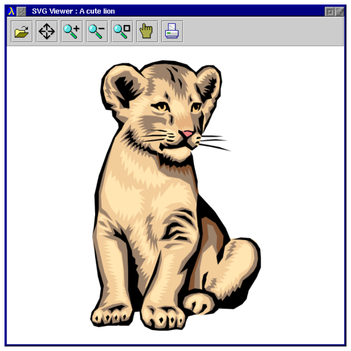


Figure 6.1: SVG image of a lion cub.

enabling some separation of the content of the graphic from the visual appearance of that image. For example, the colors of a graphic can be specified in a style sheet that is independent of the SVG file itself. SVG also preserves image structure at a higher level—for example, a web browser can directly read the text included in an SVG figure. This ability, along with the separation of style from content, dramatically improves the accessibility of images for users with color-blindness or other visual impairments. Additionally, the Document Object Model (DOM) [2] and the SVG DOM [44, Appendix B] can be used to manipulate the shapes in an image dynamically to create animations and other effects.

6.1.1 SVG is not enough

Although the SVG format is a huge step forward for many kinds of images, we can do even better for diagrammatic illustrations. Contrast the illustration in Figure 6.2 with the lion cub in Figure 6.1. Figure 6.2 is a simpler image in which I provide a visualization of a class hierarchy. With SVG, we have to specify the entire diagram fully and exactly by giving positions and sizes for all of the elements: precisely one class hierarchy diagram is described.

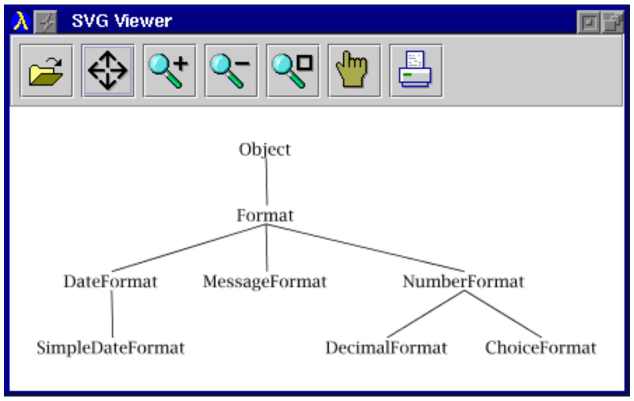


Figure 6.2: SVG image diagramming the object hierarchy surrounding the `Java.Text.Format` class. The SVG source for this image appears in Figure 6.3.

Full specification is important for a complex realistic image such as Figure 6.1, but is less important for many information visualization applications. Instead, in Figure 6.2, there are certain properties of the layout that are important in conveying the desired information. For example, we want the parent class “Object” to appear above its subclasses, and want lines to connect classes to denote the inheritance relationship. If we were able to describe what is actually semantically important about a figure, we could have a single description that preserves flexibility for the renderer and would generate Figure 6.2 or other variations of that illustration.

Constraints are a useful approach for allowing users to state their intentions more directly. For example, “`Format` appears above `DateFormat`” is a constraint. We can write the constraint mathematically as:

$$\text{Format}.y_{bottom} + \text{vert\_spacing} \leq \text{DateFormat}.y_{top}$$

By stating declaratively how the two object attributes are to relate, we avoid having to give explicit values to either. Instead, we can defer that task to a constraint satisfaction algorithm that will assign values to variables. In this example, we can then use those value assignments to determine where to position the names of the various classes in the hierarchy.

6.1.2 Contributions

I describe a constraint extension to Scalable Vector Graphics, called Constraint Scalable Vector Graphics (CSVG). The extension allows CSVG images to use arbitrary linear arithmetic constraints to control the layout of shapes, lines, paths, and font sizes. With constraints, diagrams can be under-specified, thus permitting the rendering engine greater flexibility when laying out the illustration.

The main contributions described in this chapter are:

- a motivation for using constraints for certain kinds of SVG illustrations;
- a description of Constraint Scalable Vector Graphics as an extension of SVG, including a Document Type Definition (DTD) for CSVG; and
- a prototype implementation of a CSVG viewer based on the CSIRO SVG viewer [120] and Cassowary (Chapter 3).

6.2 Background

The Scalable Vector Graphics (SVG) language [44] is based on the eXtensible Markup Language (XML) [22]. SVG also makes use of the Cascading Style Sheets (CSS) [20] standard for partially separating visual presentation information from the basic image description itself. In this section, I provide a brief overview of each of these standards.

6.2.1 XML: eXtensible Markup Language

XML is a standardized eXtensible Markup Language [22] that is a subset of SGML, the Standard Generalized Markup Language [85]. The World Wide Web Consortium (W3C) designed XML to be lightweight and simple, while retaining compatibility with SGML. Although HTML (HyperText Markup Language) is currently the standard web document language, the W3C is positioning XHTML, an XML-based language, to be its replacement. While HTML permits authors to use only a pre-determined fixed set of tags in marking up their document, XML allows easy specification of user-defined markup tags adapted to the document and data at hand [60, 61]. XML can thus be used as the basis for many languages describing arbitrary data, not just the single XHTML language.

An XML document consists simply of text marked up with tags enclosed in angle braces. A simple example appears in Figure 6.3.

The `<svg>` is an open tag for the `svg` element. The `</svg>` at the end of the example is the corresponding close tag. Text and other nested tags can appear between the open and close constructs. In the example, the `svg` contains 16 immediate children elements. Empty elements are allowed and can be abbreviated with a specialized form that combines the open and close tags: `<`*`tag-name`*`/>` (e.g., each of the `line` elements). Additionally, an XML open tag can associate attribute–value pairs with an element. For example, the first `text` element has the value `200` for its `x` attribute. Attributes of an element are unordered and multiple values for the same attribute name are disallowed. In contrast, child elements are ordered, and multiple child elements of the same type may be permitted (e.g., there are eight `text` children of the `svg` element).

For an XML document to be *well-formed*, the document must conform to the syntactic rules

```
<?xml version="1.0"?>
<!DOCTYPE svg SYSTEM "svg.dtd">
<svg width="4.5in" height="4in"
     viewBox="0 0 100 100"
     style="fill: none; font-size: 15;
            stroke-width: 1; stroke: black;
            text-anchor: middle">
  <desc>The object hierarchy surrounding
        the class "Java.text.Format"</desc>

  <text x="200" y="30">Object</text>
  <text x="200" y="90">Format</text>
  <text x="60" y="150">DateFormat</text>
  <text x="60" y="210">SimpleDateFormat</text>
  <text x="200" y="150">MessageFormat</text>
  <text x="380" y="150">NumberFormat</text>
  <text x="310" y="210">DecimalFormat</text>
  <text x="450" y="210">ChoiceFormat</text>
  <line x1="200" y1="32" x2="201" y2="75"/>
  <line x1="200" y1="92" x2="60" y2="135"/>
  <line x1="200" y1="92" x2="201" y2="135"/>
  <line x1="200" y1="92" x2="380" y2="135"/>
  <line x1="60" y1="152" x2="61" y2="195"/>
  <line x1="380" y1="152" x2="310" y2="195"/>
  <line x1="380" y1="152" x2="450" y2="195"/>
</svg>
```

Figure 6.3: SVG source of the class hierarchy illustration shown in Figure 6.2. SVG is based on XML.

required of XML documents (e.g., tags must be balanced and properly nested, and attribute values must be of the proper form and enclosed in quotes).

A more stringent characterization of an XML document is *validity*. An XML document is valid if and only if it both is well-formed and adheres to its specified *document type definition*, or *DTD*. A document type definition is a formal description of the grammar of the specific language to be used by a class of XML documents. It defines all the permitted element names and describes the attributes that each kind of element may possess. It also restricts the structure of the nesting within a valid XML document. Figure 6.3 is valid with respect to the DTD that describes Scalable Vector Graphics, `svg.dtd` [44, Appendix A].

6.2.2 SVG: Scalable Vector Graphics

SVG is an XML-based language for describing vector graphics. It was designed by the W3C and is intended to be the standard format for all images on the Internet. Vector graphics provide resolution independence—the description of the image is based on higher-level graphical elements, rather than the pixels used to describe a raster image. SVG uses XML elements to represent basic shapes, including rectangles, ellipses, lines, and polygons. It also supports the more general notion of an arbitrary path that can represent an outline to be filled, stroked, or clipped to. SVG is very similar in spirit to the PostScript page-description language [1], but uses XML syntax instead of postfix notation.

An SVG element describes a shape to be rendered. For example:

```
<rect x="20" y="10" width="10" height="5"/>
```

describes a rectangle whose top-left is positioned at coordinate (20,10) with a width of 10 units, and a height of 5 units. Lengths and coordinates can specify units explicitly, but when they are omitted, the user space coordinate system is used [44, Ch.7]. Unfortunately, all basic shape objects use their top-left as an anchor point, making it unduly cumbersome to position, for example, the center of an object at a specific location.

An especially powerful SVG element is `path`. Its `d` (for “data”) attribute contains a string that encodes a command-based description of an arbitrary outline. For example, the element:

```
<path d="M 20 10 L 30 10 L 30 15 L 20 15 Z"/>
```

describes a rectangle path equivalent to the preceding `rect` element: first **M**ove to (20, 10), then draw **L**ines to (30,10), (30,15), and (20,15), and finally close the path (**Z**). Uppercase command characters designate the use of absolute coordinates, while lowercase denotes relative coordinates. Other `path` sub-language commands include **C**urve-to, **S**mooth curve-to, **Q**uadratic Bezier curve-to, and more.

Other important elements include `defs` and `use` for defining objects and later referencing them, `image` for embedding legacy raster image files (e.g., PNG or JPEG graphics), `text` for including text, and `g` for grouping sub-elements to be rendered as a single entity.

A program that reads an SVG file has access to the internals of the image via the SVG Document Object Model [44, Appendix B]. The SVG DOM is compatible with the basic XML DOM [2] and is a proper extension of the DOM Core [77]. The DOM permits access to the SVG element tree, including allowing the manipulation of element attributes. For example, to increase the size of a text element, we can write the following code in ECMAScript [41] (a standardized version of JavaScript).

```
e = document.getElementById("TextElement");
e.setAttribute("transform", "scale(2)");
```

and the selected element will be scaled to twice its normal size. The SVG DOM can be used in combination with scripting and event handlers (e.g., `mousedown`, `onclick`) to permit some useful interactive capabilities.

SVG also contains several animation elements that describe time-based perturbation of the containing object. These elements can be used to achieve motion along paths, the fading in or out of objects, changes in color, and more. For example, to animate moving a rectangle horizontally across the viewport to the right, we write:

```
<rect x="20" y="10" width="10" height="5"/>
  <animate attributeName="x"
           attributeType="XML"
           begin="0s" dur="9s" fill="freeze"
           from="20" to="120"/>
</rect>
```

Most elements contain attributes to control especially important properties of the described object, such as its position and size. Numerous other properties of objects are set using a single attribute called `style`. That attribute is the access point to a powerful style description language called Cascading Style Sheets.

6.2.3 CSS: Cascading Style Sheets

The Cascading Style Sheets recommendation provides a rich set of “style” properties for various HTML and SVG tags. By setting the value of these properties, the document author can control how the browser will display each element.

SVG images can directly annotate elements in the document with style properties via the `style` attribute. Alternatively, the author can place this information in a separate style sheet and then link or import that file.[1] For example, in Figure 6.3 the `svg` element specifies a `style` attribute with the multi-part string value:

```
fill: none;                font-size: 15;
stroke: black;             stroke-width: 1;
text-anchor: middle
```

As usual, each of the above five CSS declarations is a property–value pair. For example "font-size: 15" specifies that the property "font-size" should take on the value "15". Because all of these style properties are specified on the `svg` root element, the styles they set are inherited by each child element, unless they are overridden.

6.3 *Adding constraints to Scalable Vector Graphics*

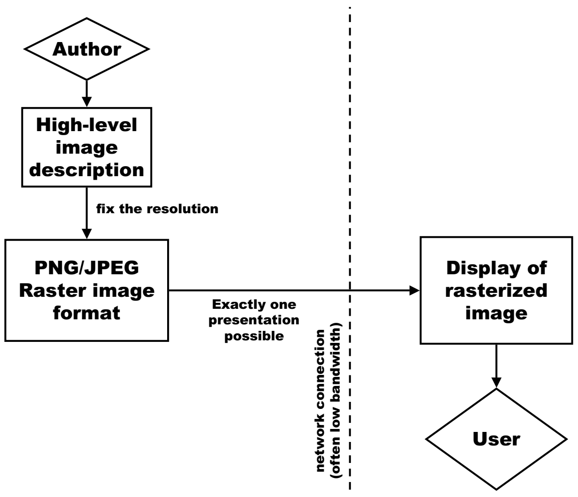

Author
High-level image description
fix the resolution
PNG/JPEG Raster image format
Exactly one presentation possible
network connection (often low bandwidth)
Display of rasterized image
User

Figure 6.4: The conventional process of delivering a raster image across the network.

[1]Unfortunately, few SVG renderers currently support separating the style sheet from the SVG document—with some implementations, only style properties set via the `style` attribute are honored.

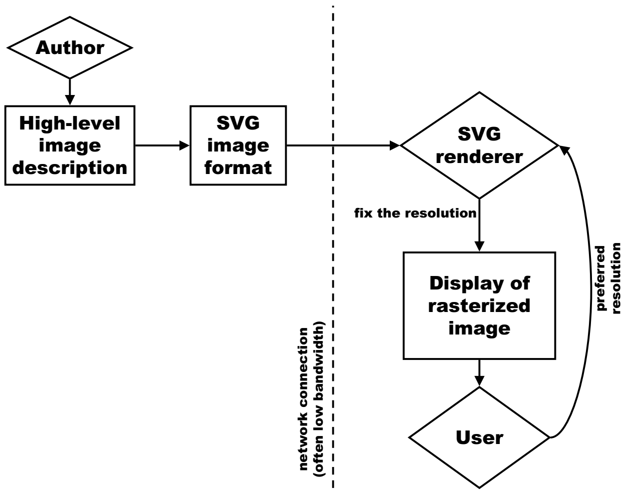

Author
High-level image description
SVG image format
SVG renderer
fix the resolution
Display of rasterized image
User
preferred resolution
network connection (often low bandwidth)

Figure 6.5: The process of delivering a resolution-independent SVG image across the network.

As previously mentioned, a primary advantage of Scalable Vector Graphics is resolution independence. The conventional means of delivering an image is to render the figure, then send the figure across the network in a rasterized image format such as PNG or JPEG (Figure 6.4). The resolution is fixed when that file is created, and the artifact the user receives is inflexible. The adoption of the SVG image format permits a different delivery mechanism (Figure 6.5). The high-level image description is stored in the SVG image format, preserving much of the semantic value provided by the author. That SVG file is then sent across the network, where an SVG renderer on the client side chooses the resolution and creates a rasterized display of that image specially-tuned for the display device and the desired size.

The key observation concerning the evolution from raster images to SVG is that we are sending a higher-level description across the network and moving some of the processing of the image from the server side to the client side. Thus, the artifact sent across the Internet is more flexible—it can be used as the source for generating a high-quality printout of the image, to create a low-resolution thumbnail of the image, or even to "render" the image aurally using speech synthesis to describe the diagram. The decision of how to present the image is made with input from the user, her browser, and other client-side software. Style sheets provide yet another way to increase the flexibility of the

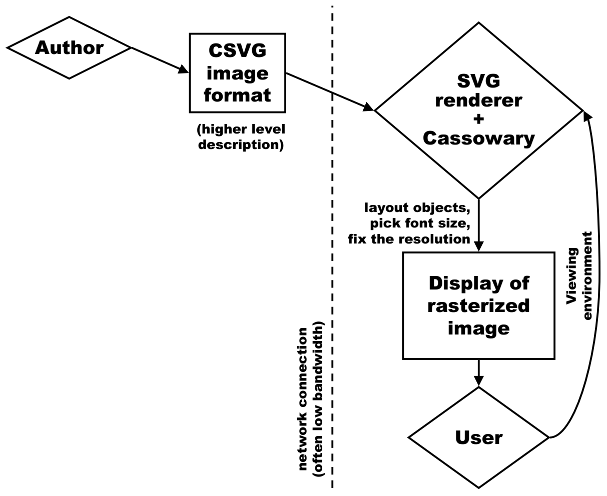

Author
CSVG image format
(higher level description)
network connection (often low bandwidth)
SVG renderer + Cassowary
layout objects, pick font size, fix the resolution
Display of rasterized image
Viewing environment
User

Figure 6.6: The process of delivering a CSVG image across the network.

image sent over the network: not only is the resolution left undetermined, but the final decision as to, for example, the coloring scheme, can be delayed until after applying style sheet declarations.

My constraint extension to SVG permits describing the author's layout intentions, and defers the actual positioning and sizing of the image's elements until just before final display for the user (Figure 6.6). To support this greater flexibility, I have made three extensions to the SVG language. First, I add a new element type called `constraint` and permit those elements to be children of the `svg` root element. Each `constraint` element has a required attribute, `rule`, and an optional attribute, `strength`. Second, I support identifier names in place of literal numbers in all attribute and style sheet values.

Thus, we can write:

```
<constraint rule="rect_w >= rect_h"
            strength="strong"/>
<rect x="10" y="20"
      width="rect_w" height="rect_h"/>
```

to express the desire that the rectangle be at least as wide as it is tall. The rule implicitly introduces new constraint variables.[2] Third, I add several built-in read-only constraint variables. (A read-only variable is one that cannot be changed by the solver to satisfy the constraint in which it occurs [14].) Two variables, `viewport_width` and `viewport_height`, are used to allow the image to be influenced by the size of the display area. I expose `current_time` and `current_time_squared` which are both ever-increasing read-only variables that allow CSVG to support the declarative specification of time-based animations more directly than the `animate` elements.

CSVG permits image descriptions to be at a higher level of abstraction than an ordinary SVG file. Instead of forcing the author to specify exact values for positions and sizes, the CSVG author can use meaningful names for values and enumerate desired relationships among those values. Similar to how SVG defers choosing the display resolution to later in the delivery pipeline, CSVG delays finalizing the *layout* of the illustration until the client side (Table 6.1).

Table 6.1: Where properties of a graphic becomes fixed.

Image format	**Resolution**	**Style**	**Layout**
PNG/JPEG	server	server	server
SVG	client	server	server
SVG + CSS	client	client	server
CSVG + CSS	client	client	client

6.3.1 A layout example

We can rewrite Figure 6.3 to specify constraints on the layout of the class hierarchy, rather than giving exact locations for all the parts of the illustration. The CSVG description of the image looks like the ordinary SVG image (Figure 6.2) under "ideal" viewing conditions. However, the CSVG file is far more flexible, and it will appear as shown in Figures 6.7 and 6.8 when the viewport dimensions are altered. An ordinary SVG file would always appear as just a uniformly scaled version of Figure 6.2.

[2]My syntax was chosen for simplicity. It may be useful to require explicit introduction of variables and to use a separate XML namespace for the extensions so that SVG renderers without a constraint engine could still handle CSVG images.

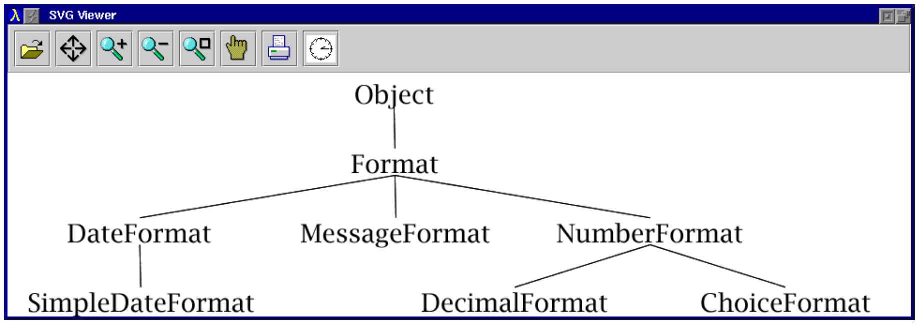


Figure 6.7: CSVG rendering of the `Format` class hierarchy inside a wide and short viewport.

For the CSVG version of the class hierarchy, I use a total of 77 constraints that reflect typical layout desires for viewing trees: nodes at the same level are aligned horizontally (4), different levels are spaced at equal vertical intervals (8), there is a minimum gap between adjacent nodes on the same level (4), and parent nodes are above and midway between their edge children (5) [101, p. 204]. Of the remaining 56 constraints, 32 are used to keep the text inside the viewport, 16 are used to declare connection points for the lines, and the last 8 are for setting the margin parameters and controlling the font size. An abridged version of the CSVG source is in Figure 6.10.

Of course, many of these constraints are redundant and could be eliminated through analysis. Because the Cassowary algorithm handles cycles without difficulty, the redundancies are not a problem, though they do impact performance. A CSVG image for frequent use would likely be optimized before distribution.

6.3.2 *An animation example*

Constraints relating object positions to the current time can be used to support simple animations. Constraints for layout are even more compelling when parts of the image are moving: the positions of the remaining objects can be described at a high level, knowing that the solver will animate whatever other objects need to move to maintain the specified constraints.

Figure 6.9 shows four screenshots of the CSVG prototype rendering an animation of a ball falling on a seesaw. The `seesaw.csvg` image contains 18 constraints to support the animation: 12 for the positions of the various elements, 1 relating the ball to the `current_time_squared`

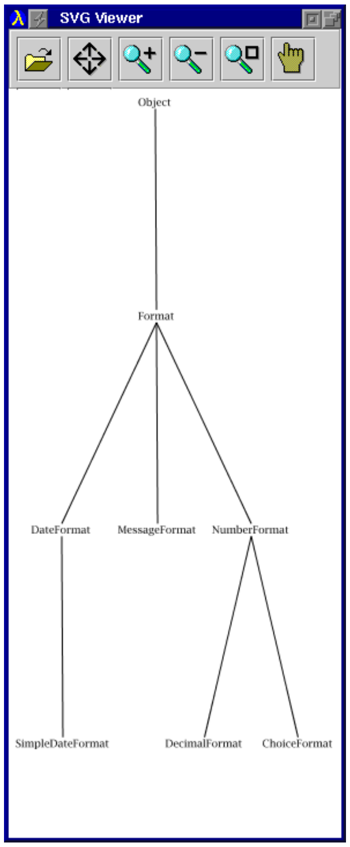


Figure 6.8: CSVG rendering of the `Format` class hierarchy inside a narrow and tall viewport.

built-in variable, 1 stating that the ball must remain above the left edge of the seesaw, and 4 describing that the seesaw cannot go through the floor nor through the fulcrum.

6.4 Implementation

On the client side of the pipeline, I have implemented a CSVG viewer to experiment with the additional expressiveness it provides. My prototype is based on version 0.71 of the CSIRO SVG Viewer [120]. That SVG viewer is implemented in Java, and it uses IBM's XML4J parser version 2.0.15 [83]. For parsing the constraint rule expressions, I use JLex [7], a lexical analyzer generator (similar to Lex), and CUP (Constructor of Useful Parsers) [80], an LALR parser generator (similar to YACC). For solving the constraint systems and laying out the figure, I embedded my Java

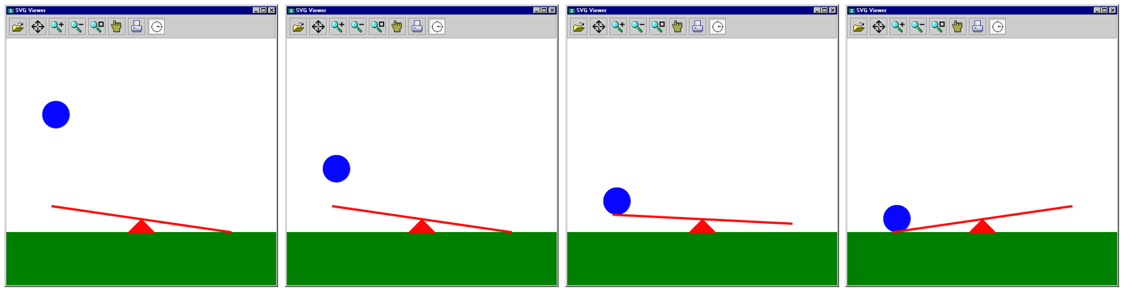


Figure 6.9: CSVG animation of a ball falling towards seesaw. The position of the ball is directly related to time, and the seesaw moves because of constraints describing its behavior.

implementation of the Cassowary Constraint Solving Toolkit (Chapter 3).

As with any XML language, CSVG is defined by its Document Type Definition. The CSVG DTD is a straightforward extension of the SVG DTD: I added the `constraint` element and specified its two attributes, `rule` (required) and `strength` (implicit, defaulting to strong):

```
<!ELEMENT constraint EMPTY >
<!ATTLIST constraint
   rule CDATA #REQUIRED
   strength CDATA #IMPLIED>
```

Additionally, I added the `constraint` element to the list of permissible children of `svg` elements:

```
<!ELEMENT svg (defs?,desc?,title?, (path|text|...|constraint)*)>
```

No other changes to the SVG DTD were necessary to support using identifiers inside of attribute expressions. (However, further changes would be necessary with the more sophisticated data description that XML Schema allows.)

After the XML parser reads in the SVG document, constraint elements create new constrainable variables for each unique identifier contained in a constraint rule. For each variable, we add a stay constraint on it to ensure stability of the resulting figure. Then, for each constraint element, we create a constraint object by parsing the rule attribute's string. Finally, we add each constraint to the global solver.

As the internal representation of the image is built, we store the names of variable identifiers that are used as an attribute's value. Then, whenever we render the figure, we retrieve the values

```
<?xml version="1.0"?>
<!DOCTYPE svg SYSTEM "csvg.dtd">
<svg width="4.5in" height="4in" viewBox="0 0 100 100"
     style="fill: none; stroke: black; stroke-width: 1">
 <desc>The object hierarchy surrounding class "Java.text.Format"</desc>
 <constraint rule="fh >= 9"/>
 <constraint rule="vert_spacing = vp_height / 3.5"/>
 <constraint rule="text_w * 4 = vp_width"/>
 ...
 <!-- stay inside viewport  -->
 <constraint rule="o_x >= side_margin + h_text_w"/>
 <constraint rule="o_x &lt;= vp_width - side_margin - h_text_w"/>
 <constraint rule="o_y >= top_margin + fh"/>
 <constraint rule="o_y &lt;= vp_height - top_margin"/>
 ...
 <!-- layout between children and parents -->
 <constraint rule="(dtf_x + nf_x) / 2 = f_x" strength="strong"/>
 <constraint rule="f_y >= o_y + vert_spacing" strength="strong"/>
 ...
 <!-- same level at same y coordinate -->
 <constraint rule="dtf_y = mf_y"/>
 ...
 <!-- same level spread out horizontally -->
 <constraint rule="dtf_x + text_w &lt;= mf_x"/>
 ...
 <!-- the text elements for each class -->
 <g style="font-size: fh; text-anchor: middle">
   <text x="o_x" y="o_y">Object</text>
   <text x="f_x" y="f_y">Format</text>
   <text x="dtf_x" y="dtf_y">DateFormat</text>
   <text x="mf_x" y="mf_y">MessageFormat</text>
   <text x="nf_x" y="nf_y">NumberFormat</text>
   ...
 </g>

 <!-- lines connecting parents to children -->
 <line x1="o_x" y1="o_y_b" x2="f_x" y2="f_y_t"/>
 <line x1="f_x" y1="f_y_b" x2="dtf_x" y2="dtf_y_t"/>
 ...
</svg>
```

Figure 6.10: CSVG source of the object hierarchy surrounding the `Java.text.Format` class. The `&lt;` inside of rule attribute values is an XML entity that represents the “<” symbol.

of attributes as usual, with one extra step: if the attribute is an identifier, we then look up that constraint variable's value and use it. For `path` elements, we prefix names of constraint variables with the `$` symbol to avoid ambiguity. For example, we write:

```
<path d="M $x $y l $dx $dy"/>
```

to move to the absolute coordinates held in `x` and `y`, and then draw a line to the relative coordinates contained in variables `dx` and `dy`.

On a Xeon Pentium III 550 MHz test machine running Java 1.3beta-0 with the HotSpot virtual machine under Windows NT 4.0, the performance of the prototype is very good. For the class hierarchy example that contains 77 constraints, the adding of the constraints and the initial solve requires only 360 ms. Subsequent re-solves of the constraint system after resizing the window require less than 200 ms each. Thus, re-rendering the figure after changing the viewport size takes only slightly longer than for the ordinary SVG viewer. Performance would be even better if I removed redundant constraints or further optimized the implementation.

On the server side, the class hierarchy diagram example was largely mechanically-derived from an XML-based representation of Java source code, JavaML [4]. Using XSLT [29], it is reasonably straightforward to generate CSVG from the JavaML representation.

6.5 Related work

As mentioned earlier, style-sheet technologies, such as CSS (Cascading Style Sheets) [20], DSSSL (Document Style Semantics and Specification Language) [86], PSL (Proteus Style Language) [98], and XSL (eXtensible Style Language) [29], each delay finalizing various presentational attributes of a figure until later in the delivery process, closer to the viewing user. None of these style languages, however, attempt to preserve layout desires to perform layout dynamically on the client side.

My CSVG motivation and philosophy is analogous to that of Constraint Cascading Style Sheets (Chapter 5), and CCSS is directly applicable to controlling style properties of CSVG documents as well. The primary addition of CSVG beyond CCSS is the ability to control non-style properties of SVG elements. This feature is necessary to control layout because the positions of those objects are determined not by style properties but by element attributes.

Kim Marriott (a co-author on papers describing the Cassowary and CCSS work) and his colleagues have independently done some preliminary work on constraint extensions to SVG. They use MathML to describe constraints (instead of a string), use the QOCA algorithm which uses a least-squares-better comparator but is otherwise similar to Cassowary, and support a limited form of disjunctions modeled after the preconditions for CCSS [137]. Diehl and Keller describe constraint extensions to the Virtual Reality Markup Language (VRML) based on a local propagation based solver that is unable to handle cycles or inequality constraints [39].

The animation aspects of SVG and CSVG are related to the Synchronized Multimedia Integration Langauge (SMIL) [78]. Another project called Madeus has used the Cassowary solver to handle a wider range of constraints in multimedia documents [136]. Madeus provides support for both temporal and spatial relationships, and it includes a rudimentary authoring environment.

6.6 Summary and future work

My constraint extension to SVG provides useful new expressiveness for describing illustration graphics at a higher semantic level. CSVG permits deferring the actual layout of the objects in the figure until the final rendering, thus resulting in greater flexibility in dealing with varied viewing environments and user desires. The implementation of the prototype system was straightforward because I was able to leverage the Cassowary constraint solving toolkit.

There are substantial opportunities for future improvements of CSVG. Currently, there are no authoring environments that preserve the author's intentions sufficiently well to generate CSVG at the appropriate level of abstraction. It is essential that a drawing program permit users to specify constraints interactively, dynamically maintain them throughout editing, and ultimately reflect those constraints in the saved CSVG file. Noth's CDA [114] or an SVG-capable editor such as Adobe Illustrator^{tm} or Sketch [75] may provide a useful starting point.

Even in the presence of graphical editing tools for CSVG, it may be beneficial to provide some syntactic sugar for CSVG. Future versions of CSVG could support referencing other elements' attributes directly. Additionally, CSVG could easily support using arbitrary expressions, instead of just identifiers, for attribute values. Such expressions would provide non-linear and non-numeric constraints over read-only variables. Extending the power of the constraint solving algorithms would permit some of these kinds of constraints over read-write variables. For example, a text element in a CSVG document could be constrained to display the coordinates of a circle: moving the circle would update the string, and editing the string would move the circle.

It may also be useful to permit even higher-level constraint abstractions in the CSVG source. For example:

```
<align dir="horizontal" anchor="middle">
  <!-- arbitrary basic shape objects here -->
</align>
```

would permit easier specification of the intention that a set of basic shapes are aligned in a row by their vertical centers. Constraints at this level also avoid problems that arise when object structure changes. Suppose a basic shape is removed from a diagram (e.g., using the SVG DOM): should indirect relationships through that object remain or be removed? If only the primitive constraints are present, the situation is ambiguous. With multiple objects being aligned with a single declaration, the answer is more clearly that those objects should remain aligned.

Another area for future work is to better describe the semantics of the SVG in terms of constraints and constraint hierarchy theory. This direction is similar to what I did for Constraint Cascading Style Sheets (Chapter 5) and it may provide a unifying implementation mechanism for parts of SVG as well. In particular, some of the scripting events, such as `onMouseMove`, may be handled within this framework: a discrete action (such as a button press) establishes a connection that then is managed via a constraint relationship until a subsequent action removes the constraint [87].

Overall, CSVG provides a surprising amount of expressiveness at a minimal implementation complexity, and at a low performance cost.

Chapter 7

CONCLUSIONS AND FUTURE WORK

This dissertation describes a number of interactive graphical applications that benefit from constraints, and demonstrates the practical applicability of sophisticated constraint solving techniques using the Cassowary constraint solving toolkit.

7.1 Summary

Constraints provide a means of separating the *what* of desired relationships from the *how* of maintaining those relationships. Over the years, constraints have been used with varying levels of success by numerous interactive graphical systems. A primary issue is managing the tradeoff between expressiveness and performance (Chapter 2).

The Cassowary linear arithmetic constraint solving algorithm provides a useful balance between the expressiveness it allows and the performance in finding solutions. My research introduces the Cassowary constraint solving toolkit, which demonstrates that an efficient, modular, reusable constraint-solving black-box software-component is possible. The toolkit was embedded in various interactive graphical applications that benefited from the constraint capabilities the toolkit provides (Chapter 3).

SCWM provides a demonstration that exposing end-users to the power of constraint solving is a realistic possibility. That system also gives a rich framework for further investigations of window layout policies, constraint-interaction paradigms, and programming by demonstration systems. Because SCWM has been developed in the Open Source community with constant feedback from real-world users, it is an excellent opportunity to expose more people to the world of declarative programming and its benefits (Chapter 4).

Constraints also have proven useful for formalizing some of the complicated procedural semantics of standards such as Cascading Style Sheets (CSS). Moreover, they can provide a unifying implementation mechanism for otherwise ad-hoc capabilities desired for page layout: instead of providing numerous specific features, we can instead provide a general technique that encompasses those techniques with fewer special cases and greater applicability (Chapter 5).

The constraint extension to SVG, CSVG, is especially notable for demonstrating how straightforward it is to augment an existing system with constraint capabilities. The engineering effort required to use the toolkit is small, and the addition of constraints permits varying the layout in more complex ways than simple scaling. Additionally, by exposing the read-only `current-time` variable, we can use CSVG to achieve simple animations, specified declaratively (Chapter 6).

In recent years, there has been a strong movement in the World Wide Web and documents communities for the separation of presentation details from content in pages delivered across the Internet. This separation is a move towards higher-level abstractions in the long-term storage format of data, and enables multiple presentations of the same core data. Constraints for layout can be usefully viewed in this framework: augmenting documents and images with information about the author's intentions for relationships among the included entities permits greater flexibility in the ultimate presentation. In addition, it adds value to the source format, and enables more semantically meaningful processing that can be especially important when the presentation medium differs dramatically from what the designer expects (e.g., aural instead of visual).

7.2 Limitations of Cassowary

The Cassowary algorithm suffers from three primary limitations. First, it can manage only linear constraints. Second, it cannot handle disjunctions of constraints. Third, its domain is limited to only real-valued variables.

The limitation to only linear constraints has proven somewhat restricting. In particular, Euclidean distance is non-linear, and is thus beyond the expressive power of Cassowary. For drawing programs, especially, this shortcoming is significant. For example, it disallows requiring that two line segments remain the same length, or ensuring that two objects are equidistant from a third. Fortunately, for aligned layouts, as is more common in publishing and diagrammatic applications, rectilinear (i.e., Manhattan) distances may suffice. Nevertheless, circumventing the linearity requirement for at least the special case of distance would definitely provide a useful boost in expressiveness.

The inability to handle disjunctions has also caused difficulties. Disjunctions arise frequently in authors' intentions for layout. For example, I may wish for a figure to be either to the left or to the right of the text. Similarly, I may wish for two windows to be non-overlapping. In the general case, it is NP-complete to handle arbitrary disjunctions. For the applications described here, domain-specific techniques were used to support the disjunction capabilities that were especially useful. For example, for CCSS, I added preconditions on style sheets, and the ability to do a linear search on a list of style sheets for the first one that had its preconditions met. Permitting some more general handling of disjunctions would be useful, and could reduce the need to address disjunctions on a per-application basis.

The limitation that Cassowary deals only with relationships among real-valued variables was the least troublesome for the applications discussed. In part, this was because my research focuses on layout applications where numerical constraints were most helpful. Nevertheless, there are certainly benefits to creating a constraint toolkit that is able to handle arbitrary domains. Enabling constraints over strings, enumerations, and more, would permit more of the system to be described declaratively. Ultimately, such an extension could even further simplify the implementation of an application by increasing the scope of the constraint solver's responsibilities.

7.3 Limitations of the applications

The biggest shortcoming of the SCWM work is that I have not done a study to better understand how real users actually use the constraint features in their daily work. In part, this is the result of an engineering decision to permit the window manager to be compiled either with or without constraint support. Although that decision has increased the total number of users, the additional complexity of building the software to support constraints has left some users willing to settle for using SCWM simply as a highly configurable and programmable system. Thus, there were fewer users of the constraint features than otherwise might have been possible.

Constraint Cascading Style sheets is compelling especially because it provides a framework for understanding CSS. However, the usefulness of the added expressiveness is somewhat uncertain. Over the years, countless many procedural hacks and special-purpose features have been built into the most popular web browsers to support the kinds of layout features that designers and users have desired. Because those capabilities already exist, there is less motivation to provide the constraint framework, despite its generality, and even though CCSS would be a significant simplification and improvement.

Constraint Scalable Vector Graphics suffers from a similar problem, but also has an additional limitation of too few inputs to the constraint system that affect the final rendering. Constraints provide the most value in the face of uncertainty in the final presentation environment. CSVG in its simple form only has two inputs available to alter the layout of the graphic: the viewport width and the viewport height. Once SVG renderers support the SVG DOM and the various interactive capabilities of the format, CSVG will become far more compelling as designers will be able to describe declaratively how the diagram should respond to those interactions, much like SCWM and users interacting with windows.

7.4 Research in the open source community

One unusual aspect of this research is that parts of it were executed in full view of the open source software community. The Cassowary toolkit, along with SCWM and the prototype implementations for CCSS and CSVG, are freely available. In particular, SCWM, from its inception was developed as a free software project. The latest and greatest “developer” version of the source code was always easily available online, and periodic stable releases were made throughout the development process.

Choosing to develop SCWM in full public view involved numerous tradeoffs. It was incredibly advantageous to have real users testing the code regularly. As the user base of SCWM grew, having dedicated users was also a useful motivator. Because SCWM was publicly recognizable, obtaining expert feedback and advice (e.g., on X/11 system intricacies) was perhaps easier. Comments and code patches received from users were invaluable in advancing the functionality of SCWM. Making the window manager available continually required greater engineering discipline and was more time consuming, but it resulted in a more robust platform. Oftentimes users desired features that were not directly relevant to the research goals of the project. When possible, users were encouraged to become developers by contributing changes to support those features, but I also often addressed the requests personally.

7.5 Compatibility of constraint extensions

Backward compatibility issues also present a significant design tradeoff. For SCWM, I chose a design that permits the window manager to be built without the constraint features included. This decision enabled many more users to use the constraint-disabled version of SCWM, as it was easier to build (it does not require a C++ compiler) and has lower resource requirements so it can be run on lower-end hardware. That choice also enabled a cleaner separation of the constraint functionality from the rest of the window manager. For example, as described in Section 4.3.4, instead of changing each window to directly use constrainable variables to determine its configuration, the constrainable variables actually shadow the ordinary primitive integer variables that are used internally. This architecture permits better control of when the windows react to new solutions to the constraints. From an engineering perspective, separating the constraint functionality from the core window manager was useful.

However, a significant problem with permitting SCWM to function without the constraint solver is that it complicates replacing procedural features with constraint-based versions. For example, SCWM still uses ordinary procedural code for direct-manipulation moving and resizing of windows. It would be better from a research perspective to replace that procedural functionality with constraints relating the user's pointer position and the window configuration. As with the CCSS work, this would let us eliminate substantial chunks of procedural code by instead leveraging the constraint solver's capabilities. As other highly programmable window managers such as Sawfish [69] gain popularity, it may be beneficial to revisit this design decision as SCWM grows into its unique niche as being the only constraint-enabled window manager for X/11.

7.6 Evolution of the toolkit

Having multiple different target applications has been useful in growing the Cassowary toolkit to meet the demands of real-world applications. Over the last three years, over a dozen versions of the toolkit have been released. Some of those releases simply fix bugs (often ones reported by users from the free software community). Others, though, add numerous new features to the toolkit to better support client applications.

For example, early versions of the Cassowary toolkit only permitted one set of edit constraints to be active at a time. After calling beginEdit, another invocation of beginEdit was disallowed. This restriction is unnecessary from the perspective of the solving algorithm, and the situation was encountered in SCWM when moving a window off the edge of the screen. During the interactive movement of the window, the window's position is a two-variable edit constraint, and when the edge of the screen is encountered, the viewport offset needs to be changed. This scenario requires a nested edit constraint. The change to the implementation to support such nesting was straightforward, but it was the experience of writing a sophisticated client of the library that brought this detail to my attention.

The CCSS project resulted in another feature enhancement for the toolkit. In CCSS, numerous constraints are added simultaneously as the style sheet rules are applied to the document tree. Cassowary originally would always optimize the tableau after each constraint addition. This approach is inefficient if we do not care about intermediate solutions and are only interested in the solution after all constraints have been added. Thus, I added the setAutoSolve function and the ability to short-circuit out of the optimization phase when adding constraints. This capability improves performance at minimal extra complexity, and it has also proven useful in the CSVG work.

7.7 Future work

In addition to overcoming the limitations described in the preceding section, there are numerous fruitful areas for future work enabled by the research described here.

SCWM provides a powerful and general infrastructure for all kinds of application and window management extensions. SCWM's ability to intercept keystrokes, simulate keystrokes, and observe the user's interactions with all of her application windows lets it be a useful, pro-active user agent. Features that span multiple applications can be developed for SCWM to help the user manage her conceptual tasks more directly (rather than dealing with each application in isolation).

SCWM's voice recognition support is currently only a proof of concept. That support opens up a world of possibilities for experimenting with multi-modal input techniques. In particular, it seems natural to express constraints verbally, and some combination of that with direct manipulation may prove especially worthwhile, particularly for disabled users for whom direct manipulation is difficult or impossible.

For Constraint Cascading Style Sheets, providing a first-class implementation inside of a popular web browser could increase acceptance of constraints and greatly simplify parts of the implementation. At the time the research was done, the Mozilla open-source browser was far too unstable to use as a starting point. Now, two years later, Mozilla is finally stable enough that it may warrant a re-implementation of CCSS to further demonstrate the practical applicability of these techniques.

Constraint Scalable Vector Graphics, too, will likely be worth re-implementing once an SVG renderer supports all of the advanced interaction and animation capabilities. Additionally, demonstrating how the existing complex features can be viewed as special cases of the general constraints framework (as was done with CCSS) would be of some value, and could similarly suggest worthwhile simplifications to the underlying implementation.

Finally, because of the power of abstraction and code reuse, extending the Cassowary constraint solving toolkit is an important goal that will benefit all of these constraint-enabled applications, and many more to come. Handling some important non-linear constraints (e.g., distance, parallelism), limited forms of disjunctions, and supporting integration with local-propagation based sub-solvers for non-numeric constraints would each be valuable extensions to the toolkit.

7.8 Conclusion

The benefits from the constraint features varied in usefulness. For existing systems that provide ad hoc support for a few especially-important constraint-like features, the general constraint mechanism is not likely to provide substantial commercial benefit in the short term. In the longer term, the value of constraints will be much greater when they are integrated with the design from the start. The existence of the Cassowary toolkit makes that a realistic possibility for future interactive graphical applications.

BIBLIOGRAPHY

[1] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1990.

[2] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document object model (DOM) level 1. W3C Recommendation, October 1998. `http://www.w3.org/TR/REC-DOM-Level-1`.

[3] Greg J. Badros. Cassowary constraint solving toolkit. Web page, 1998–2000. `http://www.cs.washington.edu/research/constraints/cassowary/`.

[4] Greg J. Badros. JavaML: A markup language for java source code. In *Proceedings of the Ninth International Conference on the World Wide Web*, Amsterdam, The Netherlands, May 2000. Elsevier Science B. V. `http://www.cs.washington.edu/homes/gjb/JavaML`.

[5] Greg J. Badros, Jeffrey Nichols, and Alan Borning. SCWM—an intelligent constraint-enabled window manager. In *Proceedings of the AAAI Spring Symposium on Smart Graphics*, March 2000.

[6] Greg J. Badros and Maciej Stachowiak. Scwm—The Scheme Constraints Window Manager. Web page, 1997-2000. `http://scwm.mit.edu/scwm/`.

[7] Elliott Berk and C. Scott Ananian. Jlex: A lexical analyzer generator for java. Web Page, 2000. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

[8] Sanjay Bhansali, Glenn A. Kramer, and Tim J. Hoar. A principled approach toward symbolic geometric constraint satisfaction. *Journal of Artificial Intelligence Research*, 4:419–443, 1996.

[9] Eric. A. Bier and Maureen C. Stone. Snap-dragging. In *Proceedings of SIGGRAPH 1986*, Dallas, August 1986.

[10] Alan Borning. *ThingLab—A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University, March 1979. A revised version is published as Xerox Palo Alto Research Center Report SSL-79-3 (July 1979).

[11] Alan Borning, Richard Anderson, and Bjorn Freeman-Benson. Indigo: A local propagation algorithm for inequality constraints. In *Proceedings of the 1996 ACM Symposium on User Interface Software and Technology*, pages 129–136, Seattle, November 1996.

[12] Alan Borning and Robert Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4):345–374, October 1986.

[13] Alan Borning and Bjorn Freeman-Benson. Ultraviolet: A constraint satisfaction algorithm for interactive graphics. *Constraints: An International Jounal*, 3:1–26, 1998.

[14] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992. `http://www.cs.washington.edu/research/constraints/theory/hierarchies-92.html`.

[15] Alan Borning, Richard Lin, and Kim Marriott. Constraints for the web. In *Proceedings of 1997 ACM Multimedia Conference*, pages 173–182, 1997.

[16] Alan Borning, Richard Lin, and Kim Marriott. Constraints for the web. In *Proceedings of ACM Multimedia 1997*, November 1997.

[17] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint hierarchies and logic programming. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 149–164, Lisbon, June 1989.

[18] Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology*, October 1997.

[19] Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. Cascading style sheets, level 2. W3C Working Draft, January 1998. `http://www.w3.org/TR/WD-css2/`.

[20] Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. Cascading style sheets, level 2. W3C Working Draft, January 1998. http://www.w3.org/TR/WD-css2/.

[21] Bert Bos, Dave Raggett, and Håkon Lie. Frame-based layout via style sheets. W3C Working Draft. http://www.w3.org/TR/WD-layout.

[22] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C Recommendation, February 1998. `http://www.w3.org/TR/REC-xml`.

[23] Arun N. Brotman, Lynne Shapiro ann Netravali. Motion interpolation by optimal control. In *Proceedings of SIGGRAPH 1988*, pages 309–315, Atlanta, Georgia, August 1988.

[24] Mark W. Brunkhart. Interactive geometric constraint systems. Master's thesis, University of California, Berkeley, Berkeley, California, May 1994.

[25] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. PWS Publishing Company, Boston, Massachusetts, fifth edition, 1985.

[26] Bay-Wei Chang and David Ungar. Animation: From cartoons to the user interface. In *Proceedings of the 1993 ACM Conference on User Interface Software and Technology*, pages 45–55, Atlanta, Georgia, November 1993. User Interface Software and Technology.

[27] Sitt Sen Chok and Kim Marriott. Automatic construction of user interfaces from constraint multiset grammars. In *Proceedings of IEEE International Symposium on Visual Languages*, pages 242–249, Los Alamitos, California, September 1995.

[28] Sitt Senn Chok and Kim Marriott. Automatic construction of intelligent diagram editors. In *Proceedings of UIST 1998*, San Francisco, California, November 1998.

[29] James Clark. XSL transformations. W3C Recommendation, November 1999. `http://www.w3.org/TR/xslt`.

[30] William Clinger and Jonathan Rees. *Revised 4 Report on the Algorithmic Language Scheme*, November 1991.

[31] Ellis S. Cohen, Edward T. Smith, and Lee A. Iverson. Constraint-based tiled windows. *IEEE Computer Graphics and Applications*, pages 35–45, May 1986.

[32] W3 Consortium. Amaya web browser software. Web page, October 1998. http://www.w3.org/Amaya.

[33] W3 Consortium. HTML 4.0 specification. Technical report, W3 Consortium, 1998. http://www.w3.org/TR/REC-html40.

[34] Isabel F. Cruz. Expressing constraints for data display specification: A visual approach. In *Principles and Practice of Constraint Programming*, chapter 23, pages 445–469. MIT Press, Cambridge, Massachusetts, 1995.

[35] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4), 1996.

[36] Ed Dengler, Mark Friedell, and Joe Marks. Constraint-driven diagram layout. In *Proceedings of the 1993 IEEE Symposium on Visual Languges*, pages 330–335, Bergen, Norway, August 1993.

[37] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphcs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, 1994.

[38] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, Englewood Cliffs, New Jersey, 1999.

[39] Stephan Diehl and Jörg Keller. VRML with constraints. In *Proceedings of the Web3D-VRML 2000 fifth symposium on Virtual reality modeling language*, Monterey, California, February 2000. `http://www.cs.uni-sb.de/RW/users/diehl/VRMLCONSTR/VRMLConstr.html`.

[40] Robert Adámy Duisberg. Animation using temporal constraints: An overview of the Animus system. *Human-Computer Interaction*, 3:275–307, 1987-1988.

[41] ECMAScript language specification, 3rd ed., December 1999. `ftp://ftp.ecma.ch/ecma-st/Ecma-262.pdf`.

[42] Elk—the extension language kit. Web page, 1999. `http://www-rn.informatik.uni-bremen.de/software/elk`.

[43] Danny Epstein and Wilf LaLonde. A Smalltalk window system based on constraints. In *Proceedings of the 1988 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 83–94, San Diego, September 1988. ACM.

[44] Jon Ferraiolo. Scalable vector graphics (SVG) 1.0 specification. W3C Working Draft, December 1999. `http://www.w3.org/TR/1999/WD-SVG-19991203/`.

[45] Roger Fletcher. *Practical Methods of Optimization*. John Wiley and Sons, New York, 1987.

[46] Barry Fowler and Richard Bartels. Constraint-based curve manipulation. *IEEE Computer Graphics and Applications*, pages 43–49, September 1993.

[47] Bjorn Freeman-Benson. Converting an existing user interface to use constraints. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 207–215, Atlanta, Georgia, November 1993.

[48] Bjorn Freeman-Benson, Molly Wilson, and Alan Borning. DeltaStar: A general algorithm for incremental satisfaction of constraint hierarchies. In *Eleventh Annual International Phoenix Conference on Computers and Communications*, pages 561–568, Phoenix, Arizona, April 1992.

[49] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[50] Scott Furman and Scott Isaacs. Positioning HTML elements with cascading style sheets. W3C Working Draft. http://www.w3.org/TR/WD-positioning.

[51] fvwm—the f? virtual window manager. Web page, 1999. `http://www.fvwm.org`.

[52] Gimp—GNU image manipulation program. Web page, 1999. `http://www.gimp.org`.

[53] Michael Gleicher. Integrating constraints and direct manipulation. In *Proceeding 1992 Symposium on Interactive 3D*, pages 171–174, 1992.

[54] Michael Gleicher. A graphics toolkit based on differential constraints. In *Proceedings of UIST 1993*, pages 109–120, Atlanta, Georgia, November 1993.

[55] Michael Gleicher. Practical issues in graphical constraints. In *Principles and Practice of Constraint Programming*, chapter 21, pages 407–426. MIT Press, Cambridge, Massachusetts, 1995.

[56] Michael Gleicher and Peter Litwinowicz. Constraint-based motion adaptation. Technical Report TR 96-153, Apple Computer, June 1996.

[57] Michael Gleicher and Andrew Witkin. Through-the-lens camera control. In *Proceedings of SIGGRAPH 1992*, July 1992.

[58] Michael Gleicher and Andrew Witkin. Supporting numerical computations in interactive contexts. In *Graphics Interface 1993*, 1993.

[59] Michael Gleicher and Andrew Witkin. Drawing with constraints. *Visual Computer*, 11(1):39–51, 1994.

[60] Charles F. Goldfarb and Paul Prescod. *The XML Handbook*. Prentice Hall PTR, 1998.

[61] Michael Goosens and Sebastian Rahtz. *The LaTeX Web Companion*. Addison Wesley Longman, 1999.

[62] James Gosling. *Algebraic Constraints*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1983.

[63] James Gosling. SunDew – a distributed and extensible window system. In *Methodology of Window Management*, chapter 5, pages 47–57. Springer Verlag, Heidelberg, Germany, 1986.

[64] James Gosling and David Rosenthal. A window manager for bitmapped displays and unix. In *Methodology of Window Management*, chapter 13, pages 115–128. Springer Verlag, Heidelberg, Germany, 1986.

[65] P. Griebel, G. Lehrenfeld, W. Mueller, C. Tahedl, and H. Uhr. Integrating a constraint solver into a real-time animation environment. Proceedings of IEEE Symposium on Visual Languages, September 1996.

[66] GTk+—the GIMP toolkit. Web page, 1999. `http://www.gtk.org`.

[67] Guile. Web page, 1999. `http://www.gnu.org/software/guile/guile.html`.

[68] Carsten Haitzler. Enlightenment. Web page, 1999. `http://www.enlightenment.org`.

[69] John Harper. Sawfish. Web page, 1999–2000. `http://sawmill.sourceforge.net/`.

[70] Warwick Harvey, Peter Stuckey, and Alan Borning. Compiling constraint solving using projection. In *Proceedings of the 1997 Conference on Principles and Practice of Constraint Programming (CP97)*, pages 491–505, October 1997.

[71] Weiqing He and Kim Marriott. Constrainted graph layout. In S. North, editor, *Proceedings of 1996 Graph Drawing Conference*, pages 217–232, Berkeley, California, September 1996. Springer Verlag.

[72] Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides. *An Object-Oriented Architecture for Constraint-Based Graphical Editing*, chapter 14, pages 217–238. Springer, 1995.

[73] Tyson R. Henry. *Interactive Graph Layout: The Exploration of Large Graphs*. PhD thesis, University of Arizona, Tucson, Arizona, June 1992. Also TR-92-03.

[74] Tyson R. Henry and Scott E. Hudson. Interactive graph layout. In *Proceedings of UIST 1991*, pages 55–64, November 1991.

[75] Bernhard Herzon. Sketch, a vector drawing program for unix. Web page, 2000. `http://sketch.sourceforge.net/`.

[76] Allan Heydon and Greg Nelson. The Juno-2 constraint-based drawing editor. Technical Report 131a, Digital Systems Research Center, Palo Alto, California, December 1994.

[77] Arnaud Le Hors, Mike Champion, Steve Byrne, Gavin Nicol, and Lauren Wood. Document object model core. W3C Working Draft, September 1999. `http://www.w3.org/TR/1999/WD-DOM-Level-2-19990923/core.html`.

[78] Philipp Hoschka. Synchronized multimedia integration language. W3C Recommendation, June 1998. `http://www.w3.org/TR/REC-smil/`.

[79] Hiroshi Hosobe, Ken Miyashita, Shin Takahashi, Satoshi Matsuoka, and Akinori Yonezawa. Locally simultaneous constraint satisfaction. In Alan Borning, editor, *Principles and Practice of Constraint Programming 1994*, pages 51–62, Orcas Island, Washington, 1994.

[80] Scott Hudson and C. Scott Ananian. CUP parser generator for Java. Web page, 1999–2000. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[81] Scott E. Hudson and Shamim P. Mohamed. Interactive specification of flexible user interface displays. *ACM Transactions on Information Systems*, 8(3):269–288, July 1990.

[82] Scott E. Hudson and John T. Stasko. Animation support in a user interface toolkit: Flexible robust and reusable abstractions. In *Proceedings of UIST 1993*, pages 57–67, Atlanta, Georgia, November 1993.

[83] IBM AlphaWorks. XML for Java. `http://www.alphaworks.ibm.com/tech/xml4j`.

[84] Takeo Igarashi, Satoshi Matsuoka, Sachiko Kawachiya, and Hidehiko Tanaka. Interactive beautification: A technique for rapid geometric design. In *Proceedings of UIST 1997*, pages 105–114, Banff, Alberta, Canada, October 1997.

[85] ISO. Standard generalized markup language (SGML). ISO 8879, 1986. `http://www.iso.ch/cate/d16387.html`.

[86] ISO/IEC. Document style semantics and specification language (DSSSL). ISO/IEC 10179, 1996.

[87] Robert J. K. Jacob, Leonida Deligiannidis, and Stephen Morrison. A software model and specification language for non-wimp user interfaces. *ACM Transactions on Computer-Human Interaction*, 6(1):1–46, March 1999. `http://www.acm.org/pubs/articles/journals/tochi/1999-6-1/p1-jacob/p1-jacob.pdf`.

[88] Tomihisa Kamada and Satoru Kawai. A general framework for visualizing abstract objects and relations. *ACM Transactions on Graphics*, 10(1):1–39, January 1991.

[89] Corey Kosak, Joe Marks, and Stuart Shieber. Automating the layout of network diagrams with specified visual organization. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(3):440–454, March 1994.

[90] Glenn A. Kramer. A geometric constraint engine. *Artificial Intelligence*, 58(1–3):327–360, December 1992.

[91] David Kurlander and Steven Feiner. Inferring constraints from multiple snapshots. Technical Report CUCS-008-91, Columbia University, New York, May 1991.

[92] David Kurlander and Steven Feiner. Interactive constraint-based search and replace. In *CHI 1992 Proceedings*, May 1992.

[93] David Joshua Kurlander. *Graphical Editing by Example*. PhD thesis, Columbia University, July 1993. `http://www.research.microsoft.com/~djk/chimera/chimera.htm`.

[94] Richard Lin, Kim Marriott, and Peter J. Stuckey. Flexible font-size specification in Web documents. In *Proceedings of the 22 Australasian Computer Science Conference*, Auckland, New Zealand, January 1999. Springer-Verlag.

[95] Mark Lutz. *Programming Python*. O'Reilly & Associates, Inc., Sebastopol, California, 1996.

[96] Blair MacIntyre. A constraint-based approach to dynamic colour management for windowing interfaces. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, 1991.

[97] Mark S. Manasse and Greg Nelson. *Trestle Reference Manual*. Digital Systems Research Center, December 1991. `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-068.html`.

[98] Philip M. Marden, Jr. and Ethan V. Munson. PSL: An alternate approach to style sheet languages for the world wide web. *Journal of Universal Computer Science*, 4(10), 1998. http://www.cs.uwm.edu/~multimedia.

[99] Kim Marriott. Constraint multiset grammars. In *Proceedings of IEEE Symposium on Visual Language*, pages 118–125, Los Alamitos, California, October 1994.

[100] Kim Marriott, Sitt Sen Chok, and Alan Finlay. A tableau based constraint solving toolkit for interactive graphical applications. In *International Conference on Principles and Practice of Constraint Programming*, 1998.

[101] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, Cambridge, Massachusetts, 1998.

[102] 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, Pittsburgh, Pennsylvania, July 1995.

[103] Nenad Medvidovic and Richard N. Taylor. Reuse of off-the-shelf constraint solvers in C2-style architectures. Technical Report UCI-ICS-96-28, University of California, Irvine, July 1996. `ftp://www.ics.uci.edu/pub/arch/papers/TR-UCI-ICS-96-28.fm.ps`.

[104] Brad Myers. Issues in window management design and implementation. In *Methodology of Window Management*, chapter 6, pages 59–71. Springer Verlag, Heidelberg, Germany, 1986.

[105] Brad Myers, Robert Miller, Rich McDaniel, and Alan Ferrency. Easily adding animations to interfaces using constraints. In *Proceedings of UIST 1996*, pages 119–128, Seattle, Washington, November 1996.

[106] Brad A. Myers. The user interface for Sapphire. *IEEE Computer Graphics and Applications*, 4(12):13–23, December 1984.

[107] Brad A. Myers. A taxonomy of user interfaces for window managers. *IEEE Computer Graphics and Applications*, 8(5):65–84, September 1988.

[108] Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, and John A. Kolojejchick. The Garnet toolkit reference manuals: Support for highly-interactive graphical user interfaces in Lisp. Technical Report CMU-CS-90-117, Computer Science Dept, Carnegie Mellon University, March 1990.

[109] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical highly interactive user interfaces. *IEEE Computer*, November 1990.

[110] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.

[111] Colas Nahaboo. GWM—the generic window manager. Web page, 1995. `http://www.inria.fr/koala/gwm`.

[112] Greg Nelson. Juno, a constraint-based graphics system. In *Proceedings of SIGGRAPH 1985*, San Francisco, July 1985.

[113] Jakob Nielson. *Usability Engineering*. Morgan Kaufmann, 1994.

[114] Michael Noth. Constraint drawing applet. Web page, 1998. http://www.cs.washington.edu/research/constraints/cda/info.html.

[115] Adrian Nye. *Xlib Programming Manual*. O'Reilly & Associates, Inc., Sebastopol, California, 1992.

[116] Gregory M. Oster and Anthony J. Kusalik. ICOLA—incremental constraint-based graphics for visualization. *Constraints: An International Jounal*, 3:32–59, 1998.

[117] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Massachusetts, 1994.

[118] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, March 1998.

[119] T. Pavlidis and Christopher J. Van Wyk. An automatic beautifier for drawings and illustrations. In *Proceedings of SIGGRAPH 1985*, July 1985.

[120] Bella Robinson and Dean Jackson. SVG toolkit. Web page, 1999–2000. `http://sis.cmis.csiro.au/svg/`.

[121] David Rosenthal. *Inter-client Communications Convention Manual*, version 2.0 edition, 1994. `http://www.talisman.org/icccm`.

[122] Kathy Ryall, Joe Marks, and Stuart Shieber. An interactive constraint-based system for drawing graphs. In *Proceedings of UIST 1997*, Banff, Alberta Canada, October 1997.

[123] Peter H. Salus, editor. *Functional and Logic Programming Languages*, volume 4 of *Handbook of Programming Languages*, chapter 4. MacMillan Technical Publishin, Indianapolis, Indiana, 1998.

[124] Michael Sannella. Analyzing and debugging hierarchies of multi-way local propagation constraints. In Alan Borning, editor, *Principles and Practice of Constraint Programming 1994*, pages 63–77, Orcas Island, Washington, 1994.

[125] Michael Sannella. *Constraint Satisfaction and Debugging for Interactive User Interfaces*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 1994.

[126] Michael Sannella. The SkyBlue constraint solver and its applications. In *Proceedings of the 1994 Workshop on Principles and Practice of Constraint Programming*, Cambridge, Massachusetts, 1994. MIT Press.

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

[128] Michael Sannella, John Maloney, Bjorne Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.

[129] Ben Schneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, August 1983.

[130] Ken Shoemake. Animating rotation with quaternion curves. In *Proceedings of SIGGRAPH 1985*, pages 245–254, San Francisco, California, July 1985.

[131] SIOD—scheme in one defun. Web page, 1999. `http://people.delphi.com/gjc/siod.html`.

[132] Richard M. Stallman. EMACS: The extensible, customizable display editor. Technical Report 519a, Massachusetts Institute of Technology Artificial Intelligence Laboratory, March 1981. `http://www.gnu.org/software/emacs/emacs-paper.html`.

[133] Gerald J. Sussman and Guy L. Steele Jr. CONSTRAINTS—a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14(1):1–39, August 1980.

[134] Ivan Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, Department of Electrical Engineering, MIT, January 1963.

[135] Shin Takahashi, Satoshi Matsuoka, Ken Miyashita, Hiroshi Hosobe, and Tomihisa Kamada. A constraint based approach for visualization and animation. *Constraints: An International Jounal*, 3:61–86, 1998.

[136] Laurent Tardif, Frédéric Bes, and Cécile Roisin. Constraints for multimedia documents. In *Proceedings of the Second International Conference and Exhibition on the Practical Application of Constraint Technology and Logic Programming*, Manchester, United Kingdom, April 2000.

[137] Jojada J. Tirtowidjojo, Kim Marriott, and Bernd Meyer. Extending svg with constraints. In *Proceedings of the Sixth Australian World Wide Web Conference*, Cairns, Queensland Australia, June 2000. Poster to appear. `http://www.dgs.monash.edu.au/~jojada/ConstraintSVG.html`.

[138] Christopher J. Van Wyk. A high-level language for specifying pictures. *ACM Transactions on Graphics*, 1(2):163–182, April 1982.

[139] Brad Vander Zanden. An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Transactions on Programming Languages and Systems*, 18(1):30–72, January 1996.

[140] Brad Vander Zanden, Brad A. Myers, Dario A. Giuse, and Pedro Szekely. Integrating pointer variables into one-way constraint models.

[141] Brad Vander Zanden and Scott A. Venckus. An empirical study of constraint usage in graphical applications. In *Proceedings of UIST 1996*, pages 137–146, Seattle, Washington, November 1996.

[142] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, California, 1996.

[143] Windowmaker. Web page, 1999. `http://www.windowmaker.org`.

[144] Andrew Witkin and Michael Kass. Spacetime constraints. In *Proceedings of SIGGRAPH 1988*, pages 159–168, Atlanta, Georgia, August 1988.

[145] Steve Wolfman and Dan Weld. The LPSAT engine and its application to resource planning. In *Proceedings of the 1999 International Joint Conference on Artificial Intelligence*, 1999.

VITA

Gregory Joseph Badros was born on 24 February 1973 in Buffalo, New York. In June 1991, he graduated as valedictorian from James M. Bennett High School in Salisbury, Maryland. In May 1995, he received *magna cum laude* a Bachelor of Science degree in both Mathematics and Computer Science from Duke University in Durham, North Carolina. Taking a one year hiatus from academic life, Greg then co-founded Transworld Numerics, Inc., and worked in Durham and Bermuda as the company's Senior Research Scientist. In September 1996, Greg started graduate school and in June 1998, he received his Master of Science degree in Computer Science and Engineering from the University of Washington. Greg is the primary author of the Scheme Constraints Window Manager and the Cassowary Constraint Solving Toolkit, and he is a contributor to numerous other Free software projects, including GNU Guile Scheme. His research interests include constraint technology, software engineering, programming languages, and the Internet.