

REUSABLE USER INTERFACE BEHAVIORS

A Dissertation

by

JOHN ALEXANDER FREEMAN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Chair of Committee,	Jaakko Järvi
Committee Members,	Lawrence Rauchwerger
	John Keyser
	Paul Gratz
Head of Department,	Duncan “Hank” Walker

May 2016

Major Subject: Computer Science

Copyright 2016 John Alexander Freeman

ABSTRACT

User interfaces often account for a majority of application code and defects. High quality user interfaces come with equally high development costs—lower than the cost of multitudes of users coping with low quality user interfaces, but higher than the mild frustration experienced by any individual user. Thus, the economics of software lead to a situation where barely passable user interfaces abound. That most user interface code comes from bespoke attempts to implement vague human interface guidelines is a leading cause of high cost and low quality.

This thesis introduces a novel formalism for user interfaces based on *ordered constraint systems*. Using explicit models for the values and relationships in a user interface, several reusable algorithms are defined for rich user interface behaviors, including value propagation, dataflow visualization, pinning, scripting, command activation, widget enablement, and context-sensitive help. Developers can leverage provably correct implementations of such desirable features for free, raising the quality of user interfaces while lowering their production cost.

Some of these behaviors have been implemented in a JavaScript framework, Hot-Drink, for web user interfaces, and a C++ framework, Adam, for desktop user interfaces. Experiments have demonstrated higher developer productivity, fewer lines of code, fewer defects, and fewer components when compared to conventional user interface frameworks.

DEDICATION

Dedicated to my parents, James and Frankie, for making all of this possible.

ACKNOWLEDGEMENTS

First, I want to thank my father, James, for sacrificing so that I could have the opportunities he never had, and my mother, Frankie, for the unconditional love and support that I never appreciated enough as a child. Special thanks to Memaw, Shylo, Mark, Dee, Bobby, Jared, Lisa, Robbie, Jenny, and the rest of my family for all the encouragement and support they've given me over the years.

My advisor, Jaakko, planted the initial idea, gave me funding (thank you!), guided me through the process of research and publication, handed out encouragement when I needed it, and never gave up on me. My lab mates—especially Jacob, Gabe, Yuriy, and Xiaolong—made it feel like we were all in it together. My faculty and committee—particularly Nancy, Lawrence, John, and Paul—are much appreciated for taking time out to help someone else's student.

Many thanks are due to the staff for accommodating me, especially when I forgot to register for the semester of my defense: Karrie, Kay, Kathy, Elena.

TABLE OF CONTENTS

	Page
ABSTRACT	ii
DEDICATION	iii
ACKNOWLEDGEMENTS	iv
TABLE OF CONTENTS	v
LIST OF FIGURES	vii
LIST OF TABLES	ix
1. INTRODUCTION	1
1.1 Motivation	1
1.2 Approach	5
2. SETTING	7
2.1 Definitions	7
2.2 Examples	10
2.2.1 Resizing an image	11
2.2.2 Booking a hotel room	12
2.3 State of the Art	12
3. FOUNDATION	15
3.1 Constraint System	15
3.2 Constraint Graph	19
3.3 Solving	21
3.4 Solution Graph	24
3.4.1 Solving algorithm	27
3.4.2 Uniqueness	30
3.4.3 When the solution graph does not change	32
3.4.4 When the solution graph does change	33
3.5 Evaluating	34
3.6 Evaluation Graph	38

3.7	Summary	39
4.	ALGEBRA	40
4.1	Constraint Systems as Monoids	40
4.2	Implementation in Haskell	45
4.3	Constraint System Properties as Monoid Properties	48
5.	BEHAVIORS	51
5.1	Value Propagation	52
5.2	Dataflow Visualization	54
5.3	Pinning	57
5.4	Scripting	60
5.5	Command Activation	61
5.6	Explaining Command Availability	62
5.7	Widget Enablement	64
5.8	Summary	66
6.	IMPLEMENTATION: HOTDRINK	67
6.1	Constructing View-Models	67
6.2	Adding Generic Behaviors	68
7.	EXPERIMENTS	73
7.1	ApplyTexas	73
7.2	TodoMVC	74
7.3	The Better Meal	78
7.4	Adobe	79
8.	RELATED WORK	82
9.	CONCLUSION	87
	REFERENCES	89

LIST OF FIGURES

FIGURE	Page
1.1 Sample section from ApplyTexas common college application.	2
1.2 Sample from Figure 1.1 amended with buttons for changing the order of entries and for inserting new entries (above or below).	4
2.1 Dialog for resizing an image.	11
2.2 Dialog for booking a hotel room.	13
2.3 Network of dependencies forming an incidental data structure among the UI components and event handlers in an object-oriented implementation of the image resize dialog shown in Figure 2.1.	14
3.1 Some possible method arrangements for a constraint among four variables.	18
3.2 Constraint graph for constraint system in Table 3.1.	20
3.3 Constraint graph from Figure 3.2 with stay constraints.	23
3.4 Possible solution graphs for the constraint graph in Figure 3.2.	25
3.5 Semantics for lazy evaluation of a solution graph G^s with valuation ν	36
3.6 Possible evaluation graph for the solution graph in Figure 3.4a.	39
4.1 Graph of a constraint composed of two other constraints.	43
4.2 All method graphs of the constraint graph in Figure 4.1.	44
4.3 Implementation of a data-flow constraint system planner in Haskell.	46
5.1 Implementation of the image resize dialog from Section 2.2.1 with an arrow visualizing one method in the evaluation graph.	56
6.1 Definition of three HotDrink variables.	68

6.2	Definition of a HotDrink constraint among the variables in Figure 6.1.	68
6.3	Implementation in HotDrink of the view-model for the image resize dialog from Figure 2.1.	69
6.4	Implementation in HotDrink of the view-model for the hotel booking dialog from Figure 2.2.	72
7.1	TodoMVC, a to-do list web application.	75
7.2	The Better Meal, a web user interface built with HotDrink for a popular online diet tracker.	78
7.3	The number of reported bugs during several months for three Adam engine teams (AE1–AE3) and a traditional framework team (TF). . .	80

LIST OF TABLES

TABLE	Page
3.1 Constraint system for image resize dialog in Figure 2.1.	16
7.1 Code statistics for TodoMVC implementations.	76
7.2 Counts of categories of JavaScript functions appearing in TodoMVC implementations.	77

1. INTRODUCTION

This thesis outlines novel abstractions for user interface behaviors that make it easier and cheaper to build high-quality user interfaces.

1.1 Motivation

Users pay a cost in time for user interfaces that are defective, confusing, or lacking crucial or useful functionality. Lazar et al. [48, 13] observe that one-third to one-half of the time spent on a computer is wasted due to frustrating experiences—and list poor user interfaces as one of the three main causes of user frustration.

Even a small waste of effort becomes significant when aggregated over a large number of users. For example, ApplyTexas (www.applytexas.org) is an online common application that must be used by all students wishing to matriculate to state universities in Texas. In one section, excerpted in Figure 1.1, students must include a list of up to ten extracurricular activities, 23 fields for each of them, in order of importance. For the student who fails to notice this last requirement, changes his/her mind, or forgets an activity, the user interface offers no other means to reorder activities than copy-pasting each field individually (a single swap of two activities requires $3 \times 23 = 69$ copy-paste operations). Similar issues occur practically everywhere: in e-commerce sites, travel bookings, tax form preparation software, “in-house” administrative applications, and so forth.

On the other hand, software budgets are strained by the high cost of developing graphical user interfaces. More than 50% of applications’ design and programming effort has been reported to be devoted to user interfaces [60]. Between 30% and 60% of application source code pertains to user interfaces [50, 60, 64] (a number as high as 85% has been reported for a large “shrink wrap” application [65]), and this

Activity 1

Organization / Activity
SPEECH CLUB

Description
EXTEMPORANEOUS SPEAKING

Activity 1 level
LOCAL

Participation Details for Activity 1 (Use whole numbers only, no fractions.)

Year	Position(s) Held	Were You Elected?	Hours/week	Weeks/year
<input type="checkbox"/> Fresh		Not Applicable	0	0
<input type="checkbox"/> Soph		Not Applicable	0	0
<input type="checkbox"/> Junior		Not Applicable	0	0
<input type="checkbox"/> Senior		Not Applicable	0	0

Activity 2

Organization / Activity

Description

Activity 2 level
LOCAL

Participation Details for Activity 2 (Use whole numbers only, no fractions.)

Year	Position(s) Held	Were You Elected?	Hours/week	Weeks/year
<input type="checkbox"/> Fresh		Not Applicable	0	0
<input type="checkbox"/> Soph		Not Applicable	0	0
<input type="checkbox"/> Junior		Not Applicable	0	0
<input type="checkbox"/> Senior		Not Applicable	0	0

Figure 1.1: Sample section from ApplyTexas common college application.

code has been observed to contain a disproportionately high number of defects [64]. Examples abound of software that is late to market, over budget, incomplete, and rife with bugs [42]. Myers' analysis [58] from two decades ago on the reasons for why user interface programming is difficult continues to be valid: with today's user interface frameworks and libraries, user interface programming remains complex and laborious.

One could expect to mitigate the problems by demanding that more programming resources were expended on systems with a large number of users. One finds, however, that commercially successful systems with very large user bases are certainly not immune to defects and problems in user interfaces. Paraphrasing the views from an industrial collaborator, user interface defects tend to be given low severity, and they often have a work-around. This means that they are the first defects to be deferred—and once a version of the product has shipped with user interface defects, unless there is an outcry from users, product management will not re-open the defect and the cycle perpetuates: bad user interfaces create expectations of bad user interfaces. [72]

The problem is that when the per-user cost of low quality is low, or seemingly low, software producers get away with quality that should be unacceptable, but is nevertheless accepted. A quick back-of-the-envelope calculation with conservative time estimates reveals that the amount of ApplyTexas users' time wasted collectively every year is at least a thousand times greater than the time the system's developers saved by not implementing a sorting feature and offering it to users, like the one that appears in Figure 1.2.

An individual user's reaction to a usability problem often involves some grumbling, an attempt to find a work-around, and then soldiering on. The blog post “Everything's broken and nobody's upset” [36] by Scott Hanselman characterizes this sentiment well (towards the state of software in general, not just user interfaces).

Activity 1 ± ▲ ▼ ▢

Organization / Activity
SPEECH CLUB

Description
EXTEMPORANEOUS SPEAKING

Activity 1 level
LOCAL ▾

Participation Details for Activity 1 (Use whole numbers only, no fractions.)

Year	Position(s) Held	Were You Elected?	Hours/week	Weeks/year
<input type="checkbox"/> Fresh	<input type="text"/>	Not Applicable ▾	0	0
<input type="checkbox"/> Soph	<input type="text"/>	Not Applicable ▾	0	0
<input type="checkbox"/> Junior	<input type="text"/>	Not Applicable ▾	0	0
<input type="checkbox"/> Senior	<input type="text"/>	Not Applicable ▾	0	0

Activity 2 ± ▲ ▼ ▢

Organization / Activity

Description

Activity 2 level
LOCAL ▾

Participation Details for Activity 2 (Use whole numbers only, no fractions.)

Year	Position(s) Held	Were You Elected?	Hours/week	Weeks/year
<input type="checkbox"/> Fresh	<input type="text"/>	Not Applicable ▾	0	0
<input type="checkbox"/> Soph	<input type="text"/>	Not Applicable ▾	0	0
<input type="checkbox"/> Junior	<input type="text"/>	Not Applicable ▾	0	0
<input type="checkbox"/> Senior	<input type="text"/>	Not Applicable ▾	0	0

Figure 1.2: Sample from Figure 1.1 amended with buttons for changing the order of entries and for inserting new entries (above or below).

The imbalance between the low per-user cost of experiencing low quality and the high per-developer cost of producing high quality rewards producing barely passable quality. We conclude that the way to improve the quality of user interfaces, and thus user experience, is to change this imbalance by lowering the cost of producing high-quality user interfaces.

1.2 Approach

One path to reducing the cost of development in user interface programming is to raise the level of reuse. Past experience has demonstrated that software systems utilizing reusable components tend to be more robust and less costly than their hand-crafted counterparts [6, 20, 62]. Software development resources can be re-targeted away from the costly and routine (and perhaps also intellectually less rewarding) user-interface programming, and our interactions with computers will be less frustrating and more productive.

In the prevailing approach to programming graphical user interfaces, a given framework [4, 53, 77] provides a selection of widgets as reusable software components, and the programmer implements a user interface as a composition of widgets by specifying the interactions between the components. The interactions are typically expressed using imperative object-oriented code placed in event handlers. Even in user interfaces with relatively simple functionality, interactions between components are often surprisingly complex. Consequently, the event-handling logic that expresses the interactions is similarly complex, often scattered to many locations in the program, and seldom reusable across user interfaces. A quote from the developer documentation of a widely used framework suggests this state of affairs is widely accepted: “*Since what a controller does is very specific to an application, it is generally not reusable even though it often comprises much of an application’s code.*” [2, §1]

The network of relationships among user interface components is an example of an *incidental data structure* [39]—a data structure that has neither an explicit encoding in the program nor an explicit run-time representation accessible to the rest of the program. Such data structures cannot be operated on by generic, reusable algorithms. Instead, they are manipulated with *incidental algorithms*, similarly emerging from the combined behavior of locally defined actions, and with no explicit encoding in the program. This thesis identifies the incidental data structures and algorithms common in user interfaces and develops abstractions that can encode those commonalities as explicit, reusable components. The benefit is that rich features, including some that are explored in computer-human interaction research but not widely adopted in mainstream software, can be included in user interfaces “for free”.

2. SETTING*

Clear, unambiguous definitions and running examples are essential to effectively conveying the concepts covered in this document. They are introduced here.

2.1 Definitions

Where possible, the definitions below respect common conventions of usage in the literature, but some are restricted to avoid ambiguity.

A *user interface* (UI) sits as a layer between a user and a system. If the user's *goal* is a specific system state, then they reach that goal by continually evaluating the state of the system and choosing an action that will transition the system to a new state. The user interface's role is to assist the user in that process by (1) executing actions on behalf of the user and/or (2) reporting to the user the effects of their actions and/or the current state of the system. A user interface need not perform both functions, but a generalized discussion, as presented here, will consider both. User interfaces exist in many domains, but this document is only concerned with software user interfaces.

Communication between user and user interface occurs in the context of *interface languages*, one each for input and output [37]. Each language has a vocabulary of terms from which expressions in the language are assembled. Terms come in many

* Portions of this chapter are reprinted from Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Property models: from incidental algorithms to reusable components. In *GPCE'08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 89–98, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-267-2. doi: 10.1145/1449913.1449927. URL <http://doi.acm.org/10.1145/1449913.1449927> Copyright 2008 ACM; John Freeman, Jaakko Järvi, Wonseok Kim, Mat Marcus, and Sean Parent. Helping programmers help users. In *GPCE'11: Proceedings of the 10th ACM international conference on Generative programming and component engineering*, pages 177–184, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0689-8. doi: 10.1145/2047862.2047892. URL <http://doi.acm.org/10.1145/2047862.2047892> Copyright 2011 ACM.

forms. As a simple example, a *command-line interface* may accept input terms as string arguments and produce output terms as a stream of text, or it may accept input terms as a stream of text and produce output terms as a collection of files.

Richer forms of interaction can be found in *graphical user interfaces (GUIs)*, which are the focus of this research. Input for GUIs includes discrete events such as keystrokes, mouse clicks, and touch gestures. Output is typically visual elements drawn on a screen, e.g., labels, buttons, or tables. Note that GUIs may provide visual representation of user input, e.g., by filling a textbox with characters as the user presses keys, but such representation should not be confused with input as defined here.

A family of design patterns has emerged to help structure user interfaces in software; one early and oft cited pattern is Model-View-Controller (MVC) [46]:

- The *model* is a collection of domain-specific objects, each of which has methods for inspecting or mutating its state. *Model objects* use the Observer pattern [30, §5] to notify dependents (e.g., views and controllers) of changes.
- *Views* are visual representations of data in the model. Upon being notified of a change in the model, each view consults the model for relevant data and then refreshes its display. Views are hierarchical and each controls the layout of its children. Views may commonly be called *widgets*, *controls*, or *elements*.
- *Controllers* are paired with views, and they handle user input. They trigger changes in the model by calling methods on the model objects. By responding to changes in the model, they can coordinate with each other and even change their own behavior.

In the basic MVC pattern, views are considered stateless and domain logic is fully encapsulated within the model. The model can be ignorant of the UI, allowing

it to be reused. MVC promotes a strong separation between the domain and its presentation, which was a significant technological advance when it originated.

The separation is not clean, however. First, views must assume that all of their state is stored in the model. Some state, however, is highly specific to the view (e.g., the currently selected item in a list widget) and does not belong in the model. Second, views must know how to retrieve their state from the model, which prevents them from being reused with other models. Lastly, views depend heavily on their controllers to handle input. Refinements of the basic MVC pattern have since appeared to cope with these shortcomings:

- A Presentation Model [18] is a layer between the model and the views. It isolates presentation-related state and logic, separating it from the model, and provides an interface around the model that is friendly to the views (as in the Adapter pattern [30, §4]).
- Passive Views [19] are “dumb” objects containing only display logic. Instead of observing the model and querying it for state, a Passive View waits for a controller to update it.
- In the Model-View-Presenter (MVP) pattern [67], views assume some of the responsibility of the controllers in MVC, converting all user input to a stream of *events* for the presenter. The *presenter* maps events from the views into *commands* (following the Command pattern [30, §5]) for the model.

Evolving MVC with important aspects of these patterns yields the setting for the research outlined in this document, a relatively recent pattern called Model-View-ViewModel (MVVM) [34]:

- Models are the same as in MVC.

- Views are passive and handle all user input. They expose an interface for observing interactions and updating the display.
- The *view-model* is a Presentation Model and presenter. It holds state specific to the presentation and acts as an adapter for the model.
- *Bindings* are an implicit component of MVVM that connect the views to the view-model. Bindings observe views for user input events and then call methods on the view-model. Similarly, they observe the view-model for changes and directly update the views.

This separation allows views (often packaged into *widget toolkits*) and models to be reused among multiple UIs. Only the view-model and bindings must be implemented separately for each UI.

2.2 Examples

Gaps may exist between the concepts in the user's mind and those in the UI. The user must translate his intentions into expressions in the UI's input interface language and likewise interpret expressions in the UI's output interface language. The quality of a UI can be judged by the level of assistance it gives the user in accomplishing a goal, that is, the extent to which it minimizes the cognitive effort required from the user to cross these gaps. Paraphrasing Hutchins et al [37], the *gulf of execution* is bridged by matching the facilities of the input language to the thoughts of the user, and the *gulf of evaluation* is bridged by creating expressions in the output language that are readily perceived and evaluated.

In MVVM, the nouns and verbs of the UI's interface languages are the values and commands in the view-model as represented by the views. Thus, one function of a high-quality software UI is to help the user construct parameters for a command.

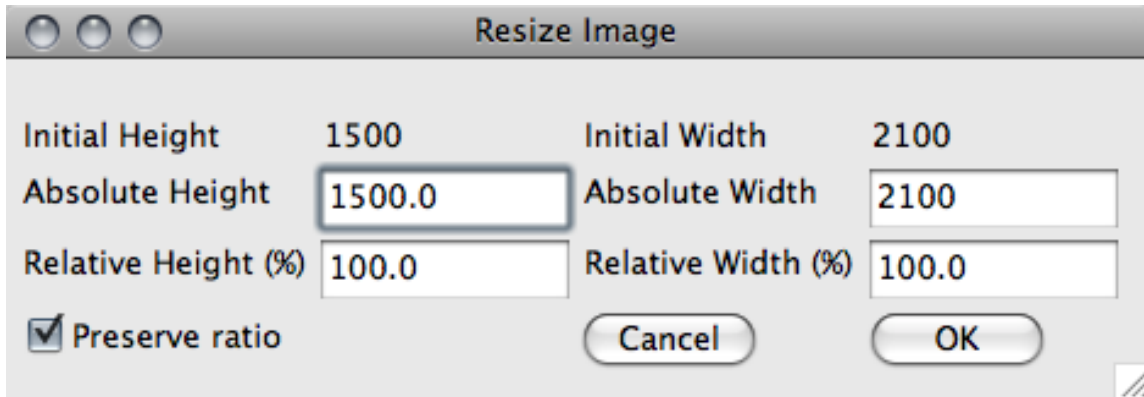


Figure 2.1: Dialog for resizing an image.

This section describes two small command dialogs demonstrating high-quality features. They will be used throughout the rest of this document to relate abstract concepts to real UIs.

2.2.1 Resizing an image

Consider the dialog appearing in Figure 2.1 for resizing an image. The associated command expects new horizontal and vertical dimensions for the image in pixels, but the dialog can be more flexible. Here, as an alternative to specifying new absolute dimensions, the user may scale dimensions as percentages relative to their initial values. Further, the user has the option to maintain the original aspect ratio so that changes in the height are proportionally reflected in the width, and vice versa. By offering a variety of means to choose parameters for the resize command, the UI bridges the gulf of execution by more directly supporting the different thought processes of users, e.g., “I want the image to have a fixed size” versus “I want the image to be twice as large”.

The UI keeps values consistent according to fixed relationships. If the user edits the relative height, the absolute height will change; if the option to preserve aspect

ratio is selected, then the relative and absolute widths will change as well. At any point, regardless of the path taken to a particular state of the UI, the user can see what argument values will be passed to the resize command, which helps span the gulf of evaluation.

In a large UI, the dataflow may become confusing to the user. A high quality UI may further reduce the gulf of evaluation by visualizing the dataflow, e.g., by drawing arrows overlaying the view to show the network of dependencies among values.

2.2.2 *Booking a hotel room*

Figure 2.2 shows a dialog that could be part of an application for booking a hotel room. The reservation service needs check-in and check-out dates to query room availability. By adding a field for the number of nights, the UI offers three ways to specify the dates—from any two fields, the third can be derived. The UI gets to pick which field is third, and the user may not like the choice, so the UI offers a toggle next to each field to lock its value.

Additionally, the command expects a list of ages for child guests, if any. The user must first select the number of children, after which the UI presents a corresponding number of age selections; thus, the UI avoids a clutter of unnecessary elements that may confuse the user.

Finally, the UI guards execution of the command by disabling its button whenever certain conditions are unmet—namely, whenever the number of nights to book is not positive.

2.3 State of the Art

The examples above illustrate a few UI features such as maintaining dependencies among values, disabling commands, and hiding irrelevant views. The main service offered by a typical GUI library is to translate user actions to *events* and deliver the

The image shows a light blue dialog box with a grid of input fields. The fields are: 'Check in' with a date field containing '25 March 2013'; 'Check out' with a date field containing '28 March 2013'; 'Nights' with a text field containing '3'; 'Number of children' with a spinner field containing '2'; 'Age of first child' with a spinner field containing '14'; and 'Age of second child' with a spinner field containing '10'. Each of the first three rows has a small square checkbox to its right. At the bottom center is a rounded rectangular button labeled 'Book'.

Figure 2.2: Dialog for booking a hotel room.

events to the correct *event handler* functions. A UI programmer writes the event handlers and registers them to listen to particular events. The UI logic is dispersed and often duplicated throughout the event handlers. Furthermore, the state of the UI is shared among them, eliciting a complex, ad hoc network of dependencies. Figure 2.3 illustrates such a network derived from an implementation, within an object-oriented GUI framework, of the image resizing dialog from Figure 2.1.

This thesis explains how the features described above (and others) can be implemented as generic algorithms and reused across UIs. Instead of writing event handlers, programmers declare values and define relationships among them with orthogonal, imperative functions. Event handlers become automatically generated boilerplate.

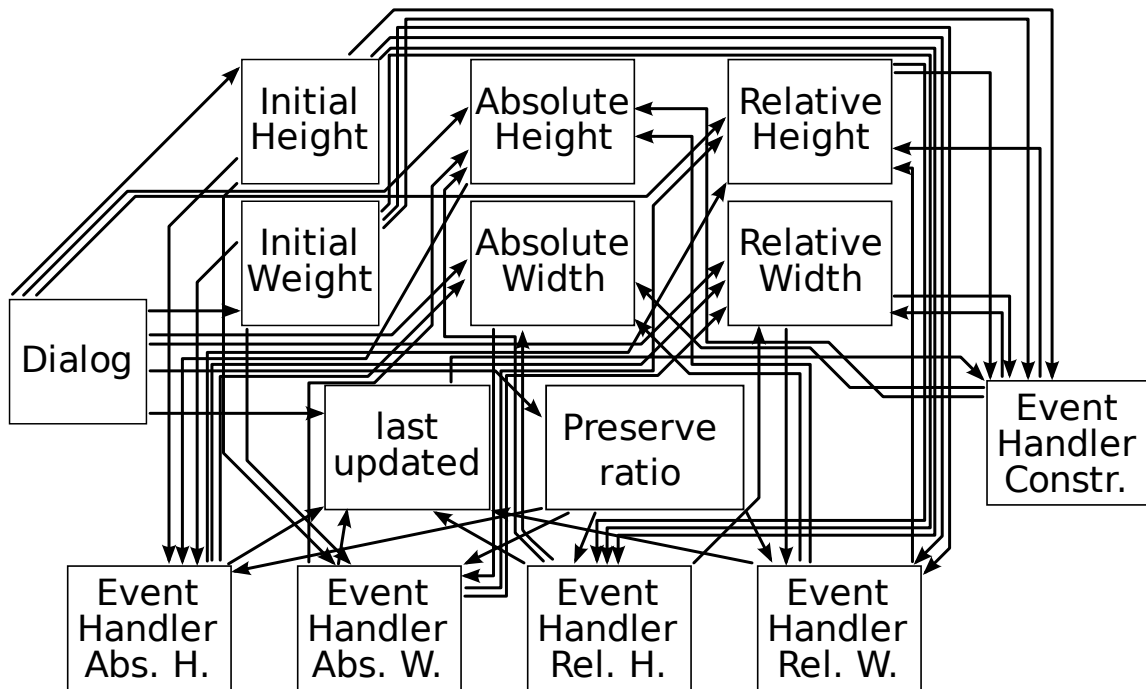


Figure 2.3: Network of dependencies forming an incidental data structure among the UI components and event handlers in an object-oriented implementation of the image resize dialog shown in Figure 2.1. Edges correspond to the relations “event handler writes a value” or “event handler reads a value”.

3. FOUNDATION*

The data structure used to implement the reusable algorithms presented in Chapter 5 is a collection of three specific graphs derived from a kind of *data-flow constraint system* [79] tailored to the domain of user interfaces, as described in this chapter.

3.1 Constraint System

A data-flow constraint system S is a tuple $\langle V, C \rangle$, where V is a set of *variables* and C a set of *constraints*. Each variable in V has an associated *value*. Each constraint in C is a tuple $\langle R, r, M \rangle$, where $R \subseteq V$, r is an n -ary relation ($n = |R|$) among variables in R , and M is a non-empty set of *constraint satisfaction methods*, or just *methods*. If the values of the variables in R satisfy r , the constraint is *satisfied*. Executing any method m in M *enforces* the constraint by computing values for some subset of R , using another disjoint subset of R as inputs, such that the relation r becomes satisfied. The input and output variables of a method m are denoted as $ins(m)$ and $outs(m)$, respectively, and determine its *direction*. A method is considered a “black box”—it is the programmer’s responsibility to ensure that a constraint is satisfied after any of its methods is executed.

* Portions of this chapter are reprinted from Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Property models: from incidental algorithms to reusable components. In *GPCE’08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 89–98, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-267-2. doi: 10.1145/1449913.1449927. URL <http://doi.acm.org/10.1145/1449913.1449927> Copyright 2008 ACM; Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Algorithms for user interfaces. In *GPCE’09: Proceedings of the 8th international conference on Generative programming and component engineering*, pages 147–156, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-267-2. doi: 10.1145/1621607.1621630. URL <http://doi.acm.org/10.1145/1621607.1621630> Copyright 2009 ACM; John Freeman, Jaakko Järvi, Wonseok Kim, Mat Marcus, and Sean Parent. Helping programmers help users. In *GPCE’11: Proceedings of the 10th ACM international conference on Generative programming and component engineering*, pages 177–184, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0689-8. doi: 10.1145/2047862.2047892. URL <http://doi.acm.org/10.1145/2047862.2047892> Copyright 2011 ACM.

V

name	description	value
v_{ih}	initial height	1500 (px)
v_{ah}	absolute height	1500 (px)
v_{rh}	relative height	100 (%)
v_{iw}	initial width	2100 (px)
v_{aw}	absolute width	2100 (px)
v_{rw}	relative width	100 (%)
v_{pr}	preserve ratio	true
v_{ok}	resize image	resize(v_{ah}, v_{aw}) (command)

(a) The variable set. There is one constraint system variable for each value and command in a view-model. Here, v_{ok} represents the command object executed whenever the “OK” button is clicked.

C

R	r	M
$\{v_{ih}, v_{ah}, v_{rh}\}$	$v_{rh} = 100 \frac{v_{ah}}{v_{ih}}$	$v_{rh} \leftarrow 100 \frac{v_{ah}}{v_{ih}}$ $v_{ah} \leftarrow \frac{v_{rh}}{100} v_{ih}$
$\{v_{iw}, v_{aw}, v_{rw}\}$	$v_{rw} = 100 \frac{v_{aw}}{v_{iw}}$	$v_{rw} \leftarrow 100 \frac{v_{aw}}{v_{iw}}$ $v_{aw} \leftarrow \frac{v_{rw}}{100} v_{iw}$
$\{v_{pr}, v_{rh}, v_{rw}\}$	$v_{pr} \Rightarrow v_{rh} = v_{rw}$	$v_{pr} \Rightarrow v_{rh} \leftarrow v_{rw}$ $v_{pr} \Rightarrow v_{rw} \leftarrow v_{rh}$
$\{v_{ok}, v_{ah}, v_{aw}\}$	$v_{ok} = f(v_{ah}, v_{aw})$	$v_{ok} \leftarrow f(v_{ah}, v_{aw})$

(b) The constraint set. Constraints tie together related values. In the last constraint, f is a non-invertible function that constructs a resize command object from its arguments.

Table 3.1: Constraint system for image resize dialog in Figure 2.1.

The view-model of a UI can be represented as a data-flow constraint system. For example, Table 3.1 shows the constraint system underneath the image resizing example from Section 2.2.1. There is one variable for each value and command. In this case, they correspond directly to elements of the UI, but in general a view-model may have values or commands with no corresponding view. Similarly, there may be views (e.g., the cancel button) with no corresponding variable in the view-model.

Constraints connect groups of related variables, and they offer a few degrees of flexibility:

- The relations represented by constraints can be arbitrary. They are often equalities (e.g., the associations among the initial, absolute, and relative values for each dimension), but not always (e.g., the implication which conditionally ties the relative values for each dimension).
- Methods may be *single-output* or *multi-output*.
- Constraints may have one or more methods. *One-way* constraints are typically suitable for relationships involving a non-invertible function (e.g., a checksum) or for tying a command to its parameters. *Multi-way* constraints support alternative constructions. Figure 3.1 demonstrates a few ways methods can be specified for a constraint among four variables.

As they are used here, constraints and methods are subject to a few restrictions, listed below and named for convenience. The reasons for these restrictions are explained in detail throughout the chapter. A constraint system satisfying these restrictions is *well-formed*.

(WF-1) Every method must use all of the variables in its constraint as either an

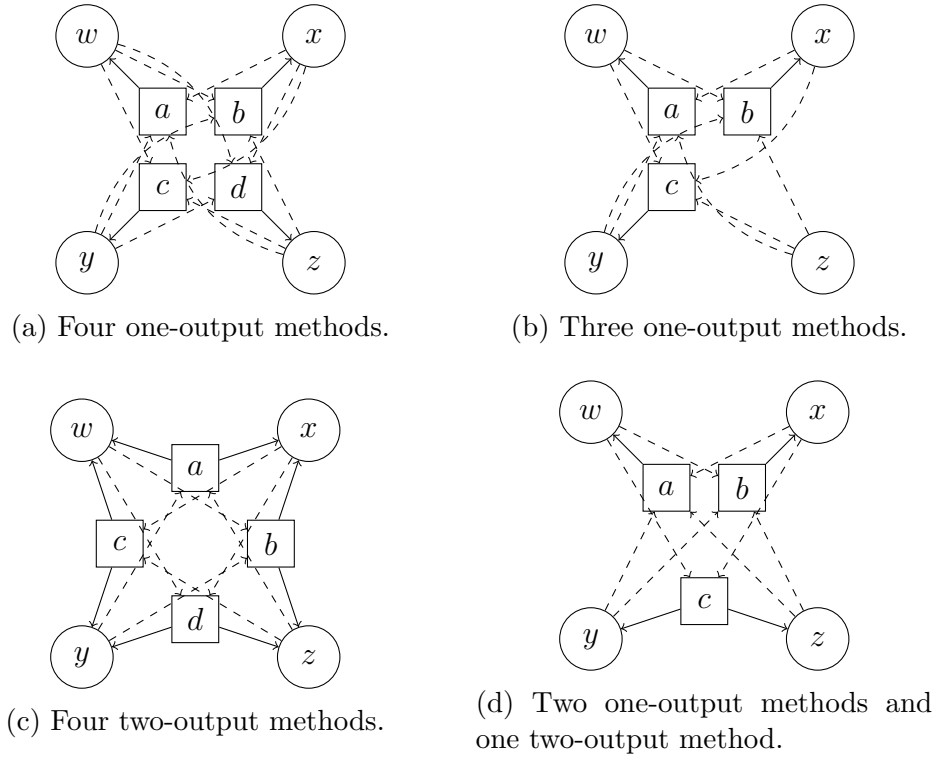


Figure 3.1: Some possible method arrangements for a constraint among four variables. In these diagrams, variables are circles, methods are squares, and methods are connected to their inputs with dashed lines and outputs with solid lines. The constraint can have one single-output method to compute each variable (a) or a subset of those (b). In the latter case, some variable(s) (z here) are used as inputs only. The constraint may instead have multi-output methods to some pairs of variables (c), or a mixture of single-output and multi-output methods (d).

input or an output (but not both).

$$\forall \langle R, \cdot, M \rangle \in C, \forall m \in M, \{ins(m), outs(m)\} \text{ is a partition of } R$$

This is known as *method restriction* [71, p. 56], and it is trivially satisfied: if a variable is not an output of a method, then it can be considered an (ignored) input without changing the method.

(WF-2) Two constraints may not share the exact same variables.

$$\forall \langle R_1, \cdot, \cdot \rangle, \langle R_2, \cdot, \cdot \rangle \in C, R_1 \neq R_2$$

(WF-3) One method's outputs may not be a subset of another method's outputs.

$$\forall m_1, m_2 \in M, outs(m_1) \not\subseteq outs(m_2)$$

3.2 Constraint Graph

This section develops the view of a constraint system as a graph. It begins with a “global” view of a graph for the entire constraint system and then discusses how to infer the individual constraints. The graph representation described differs slightly from prior works [79] where multi-way data-flow constraint systems are represented as undirected graphs with auxiliary information that expresses how the undirected graph can be directed. In the representation here, such auxiliary data is unnecessary.

A data-flow constraint system S is in a one-to-one correspondence with an *oriented, bipartite* graph $G = \langle V + M, E \rangle$, with vertex sets V and M representing the variables and methods of the system, respectively, and E the directed edges that connect each method with its input and output variables. Where $u, v \in V$ and $m \in M$, the edge (u, m) indicates that the variable u is an input of the method m , and (m, v) that m outputs to the variable v . The graph is oriented, that is, $(a, b) \in E \Rightarrow (b, a) \notin E$, because for each method m , $ins(m)$ and $outs(m)$ are disjoint according to WF-1. This graph is called the *constraint graph*. Figure 3.2 shows the constraint graph of the constraint system from Table 3.1.

The grouping of methods and variables into constraints is not explicit in the representation $G = \langle V + M, E \rangle$, but it is uniquely determined by G and the notion

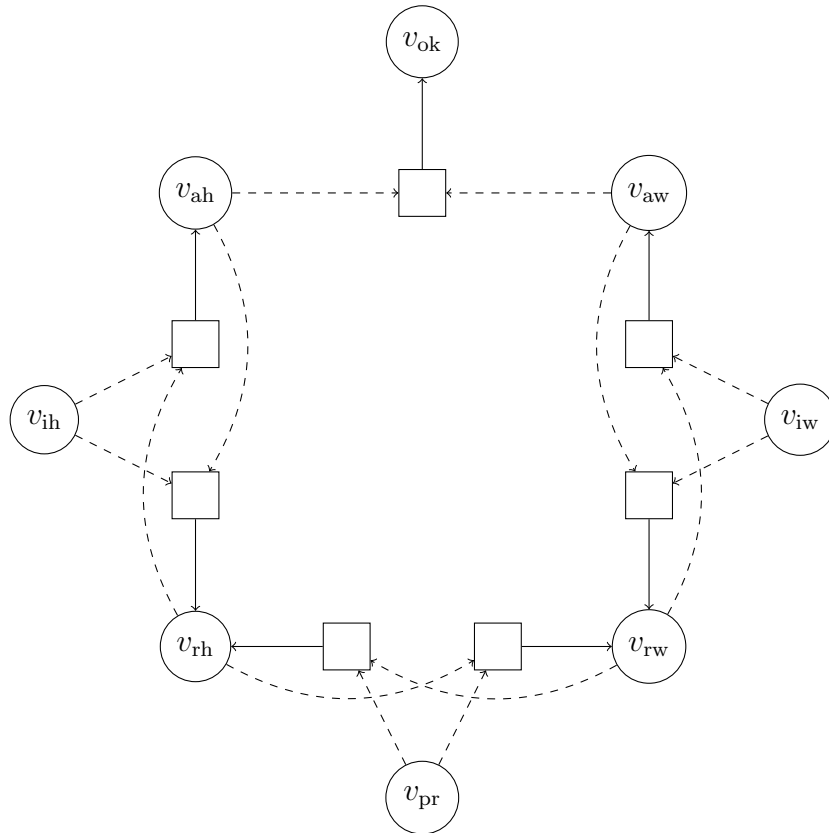


Figure 3.2: Constraint graph for constraint system in Table 3.1. The drawing conventions are the same used for Figure 3.1. Constraints are not marked explicitly; methods with the same neighborhood of variables belong to the same constraint.

of a vertex's *neighborhood* (nbh). For a graph $\langle V, E \rangle$ and vertex $a \in V$, $nbh(a) = \{b \in V \mid (a, b) \in E \vee (b, a) \in E\}$. Intuitively, method vertices with the same neighborhood all belong to the same constraint. Assuming method restriction (WF-1), all method vertices of the same constraint have the same neighborhood, and with uniqueness among constraints (WF-2), two methods from different constraints cannot have the same neighborhood. Formally, if \sim_{nbh} is the equivalence kernel of nbh in M , defined by $m_1 \sim_{nbh} m_2 \Leftrightarrow nbh(m_1) = nbh(m_2)$, then the method vertices of each constraint are an equivalence class in the quotient set M/\sim_{nbh} . Therefore, two methods m_1 and m_2 belong to the same constraint if and only if $[m_1]_{nbh} = [m_2]_{nbh}$. The rest of this document omits the subscript \cdot_{nbh} and just uses $[\cdot]$ and \sim .

Thus, the elements of the quotient set M/\sim correspond to the constraints of the original constraint system S ; in other words, for any method vertex m in G , $[m]$ is a set of method vertices in G , and it identifies a constraint in S .

3.3 Solving

The constraint satisfaction problem for a constraint system is to find a valuation of the variables such that each constraint is satisfied. Each constraint can be satisfied independently by executing one of its methods. However, to ensure consistency over the whole system, methods should be chosen and executed in an order such that once a variable has been read from or written to by one method, no other method will write to it.

A solution to a data-flow constraint system is thus characterized by a partially ordered set of methods. Each valid execution order of methods within a solution is often called a *plan* [28]. Depending on the constraint system, a solution may or may not exist, and may or may not be unique. If a solution exists for a given constraint system, the system is *satisfiable*; otherwise it is *over-constrained*. If more than one

solution exists, the system is *under-constrained*.

The algorithm to ensure consistency among the values in a UI maintains the view-model in a state where all constraints are satisfied. Whenever a variable’s value changes—possibly as the result of user interaction—some constraints may no longer be satisfied. The system is brought back to a satisfied state by computing and executing a new plan. New values of changed variables are then reflected back to the view.

Often the constraint system of a view-model is underconstrained. That is, many different solutions exist, and each solution corresponds to a different direction of flowing data in a user interface. For example, in the dialog for booking a hotel stay, data could flow from the check-in date and check-out date variables to the number of nights variable; or it could flow from the check-in date and number of nights variables to the check-out date variable.

Stay constraints and *constraint hierarchies* [11] are used to select among the many possible solutions that arise in a view-model. Each variable in the system is given a stay constraint. A stay constraint consists of a single *stay method* with one output and no inputs—it is a constant function. Formally, the set of stay constraints in a constraint graph G is:

$$\text{stays}(G) = \{ [m] \mid m \in M, |\text{outs}(m)| = 1, |\text{ins}(m)| = 0 \}.$$

Conceptually, every time the value of a variable changes, the stay method of that variable’s stay constraint is constructed anew, so that its constant function returns the current value of the variable. Thus, executing a stay method for a variable leaves the variable unchanged. The constraint graph from Figure 3.2 with stay constraints added appears in Figure 3.3. (Most figures will omit stay constraints as they are

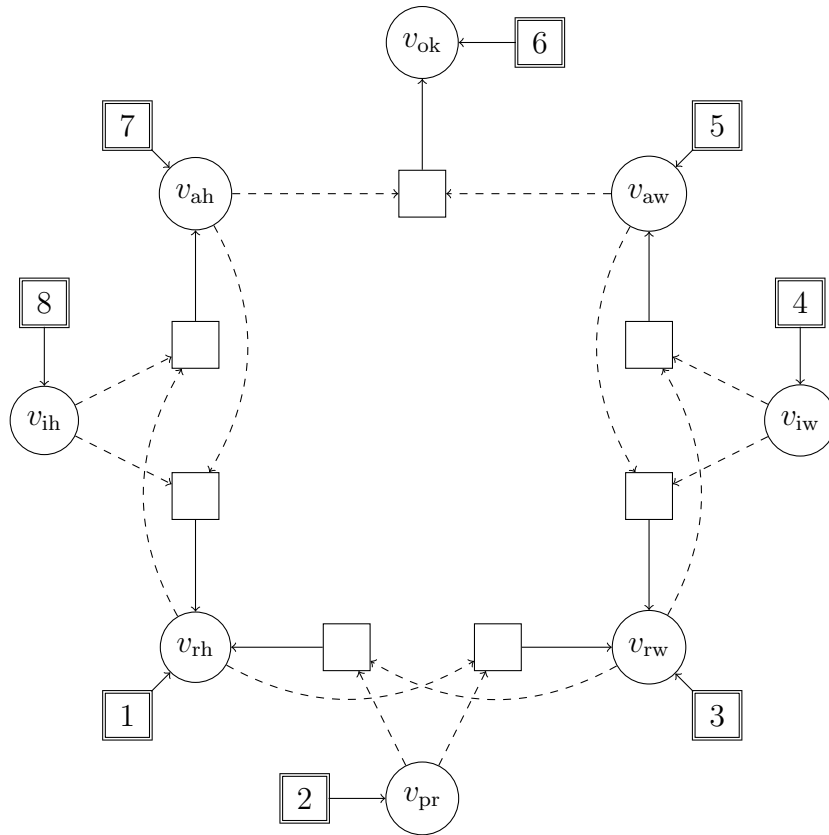


Figure 3.3: Constraint graph from Figure 3.2 with stay constraints. Stay methods are depicted as double-border rectangles.

implicit.)

Since not all stay constraints and programmer-defined constraints can be satisfied simultaneously, a solution can be found only after some constraints are *retracted*. To decide which constraints to retract, constraints are organized into levels within a *hierarchy*. The rank of a constraint within the hierarchy is called its *strength*. Multiple constraints may share the same strength, meaning they belong to the same level of the hierarchy. Intuitively, the *best* solution is the one allowed by retracting the fewest and weakest constraints. The “locally-predicate-better” comparator [27] defines this precisely: if one solution enforces a constraint that the other does not,

and every stronger constraint is either retracted in both solutions or enforced in both solutions, then the former solution is *locally-predicate-better* (hereafter, simply *better*) than the latter.

Hierarchical constraint systems are adapted here to make suitable models for UI behavior. The highest strength is assigned to the programmer-defined constraints to indicate that no solution can retract them. All stay constraints are weaker, and their strengths are unique and totally ordered according to their variable’s *priority*. Every time a variable is changed by a client, it is assigned the highest priority among all variables. Thus, the more recently edited variables will have higher priority, indicating that the preservation of those variables’ values should be favored. With this strength assignment, the best solution tends to follow the principle of least surprise [38] for the user.

Definition 1 (Ordered constraint system). The combination of (a) a constraint system with (b) stay constraints for each variable (c) whose strengths are unique and totally ordered is an *ordered constraint system*.

3.4 Solution Graph

A solution for a constraint system can be explicitly represented as a subgraph of the constraint graph, called a *solution graph*. The notation $G[V]$ indicates the *vertex-induced* subgraph of G : if V is a subset of G ’s vertex set, $G[V]$ is the graph whose vertex set is V and whose edge set includes all edges of G with both endpoints in V . Thus, if $G^c = \langle V + M, E \rangle$ is a constraint graph, then $G^s = G^c[V + M^s]$ is a *solution graph* of G^c iff (1) $M^s \subseteq M$; (2) G^s is *acyclic*; (3) $\forall v \in V, in-degree(v) \leq 1$; (4) $|M^s/\sim| = |M^s|$; and (5) $|M^s/\sim| = |M/\sim|$. The third condition establishes that no two methods output to the same variable, the fourth that every constraint in the solution graph has exactly one method, and the fifth that every constraint in the

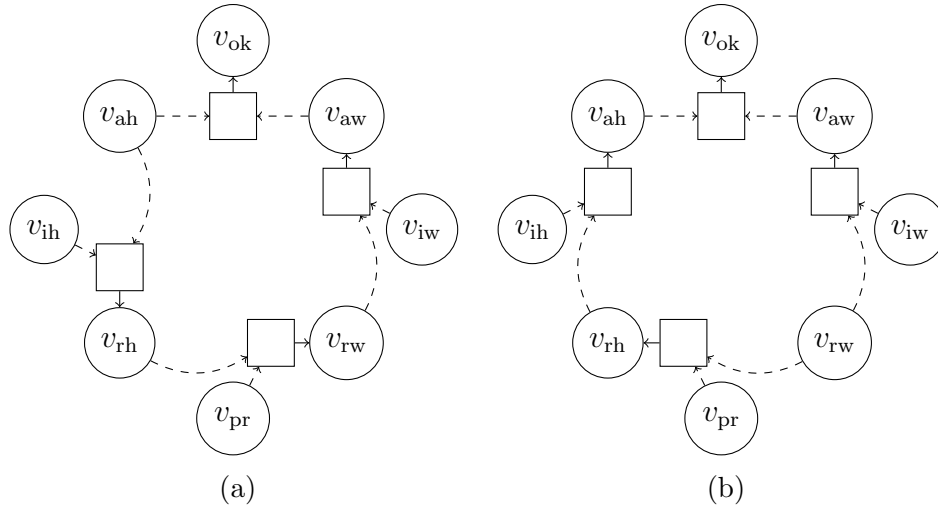


Figure 3.4: Possible solution graphs for the constraint graph in Figure 3.2. Stay constraints are implicit: one exists for every variable with in-degree 0.

constraint graph maps to exactly one constraint in the solution graph.

The last two conditions together guarantee that the solution graph contains exactly one method from every constraint in the constraint graph. This restriction must be relaxed for hierarchical constraint systems where some constraints may be retracted in a solution: (5) $|M^s/\sim| \leq |M/\sim|$.

In an ordered constraint system, every variable has a retractable stay constraint. Hence, if there exists a solution with a variable v with in-degree 0, then there exists a better solution that includes the stay method for v . In other words, for the best¹ solution to an ordered constraint system: (3) $\forall v \in V, in-degree(v) = 1$.

The order of methods in a plan is implicit in the solution graph, as a topological order of method vertices. A few solution graphs for the constraint graph in Figure 3.2 appear in Figure 3.4.

The locally-predicate-better ordering among solutions for a hierarchical constraint system can be expressed precisely in terms of solution graphs. Formally, let $s :$

¹Proved later to be unique.

$M/\sim \rightarrow \mathbb{Z}^*$ be a *strength assignment function* that maps a constraint to its strength. (Instead of \mathbb{Z}^* , any totally ordered set could be used.) Let $>_s$ define a relation among strengths such that, for $c_1, c_2 \in M/\sim$, if $s(c_1) >_s s(c_2)$ then c_1 is stronger than c_2 . Additionally, let $<_s$, $=_s$, $=/s$, \leq_s , and \geq_s have corresponding meanings. In the examples here, $>_s$ is equivalent to $<$ in \mathbb{Z}^* , e.g. 0 is the highest strength.

Let seq_s map a set of constraints to an ordered sequence of the constraint strengths assigned to each constraint by s :

$$seq_s : \{C\} \rightarrow (\mathbb{Z}^*)$$

$$\{[m_1], [m_2], \dots, [m_k]\} \mapsto (s([m_1]), s([m_2]), \dots, s([m_k])),$$

$$\text{where } s([m_1]) \geq_s s([m_2]) \geq_s \dots \geq_s s([m_k])$$

With this, the locally-predicate-better comparator for solution graphs (using $>_s^d$ to denote the lexicographical “greater than” operator between sequences according to the $>_s$ “greater than” operator between elements) can be defined as follows:

Definition 2 (“Locally-predicate-better” relation for solution graphs). Let $G^c = \langle V + M, E \rangle$ be a hierarchical constraint graph, $s : M/\sim \rightarrow \mathbb{Z}^*$ a strength assignment function, and $G_1^s = G^c[V + M_1]$ and $G_2^s = G^c[V + M_2]$ two solution graphs of G^c . G_1^s is *locally-predicate-better* than G_2^s (written $G_1^s \succ_s G_2^s$) iff $seq_s(M_1/\sim) >_s^d seq_s(M_2/\sim)$.

Intuitively, the set of methods included in a solution graph corresponds to a sequence of the methods’ constraints’ strengths, in decreasing order of strength. (Remember that decreasing order of strength will appear as increasing order of numerical value according to the definitions above.) These sequences are compared lexicographically: a greater sequence represents a stronger set of constraints. Such a comparison exhibits a couple important properties:

- After removing all common constraints from any two given solutions, the one left enforcing the strongest constraint is better:

$$(0, 1, 1) >_s^d (0, 1, 2, 3)$$

- A solution is better than every solution enforcing a proper subset of its constraints:

$$(0, 1, 1, 2) >_s^d (0, 1, 2)$$

$$(0, 1, 1, 2) >_s^d (0, 1, 1)$$

3.4.1 Solving algorithm

A derivative of the *QuickPlan* algorithm [79] is used to find the best solution graph for a given hierarchical constraint system and strength assignment function. QuickPlan is guaranteed to find such a graph if one exists, and fail otherwise. QuickPlan is “optimistic”, starting from an over-constrained system and temporarily retracting constraints until a solution can be found. Then, it improves the solution (if possible) by attempting to restore retracted constraints, one-by-one. The algorithm here omits the first stage, starting directly from a system that is known to be solvable and moving immediately to the improvement phase, and simplifying further because each stay constraint has a unique strength. Nevertheless, the essence of the algorithm remains the same.

The notions of *free* variable and method are essential in describing the algorithm:

Definition 3 (Free variable, free method). Let $G = \langle V + M, E \rangle$ be a constraint graph. A variable v is free in G if $\forall m_i, m_j \in M, m_i \in nbh(v) \wedge m_j \in nbh(v) \Rightarrow [m_i] = [m_j]$. A method m is free in G if $\forall v \in outs(m), v$ is free. The set of free

variables of a constraint $c \in M/\sim$ is denoted $frees(c) = \{v \in V \mid nbh(v) \subseteq c\}$.

A variable is free if it is connected to no more than one constraint; that is, any methods for which it is an input or output all belong to the same constraint. A free method outputs to only free variables, meaning it can satisfy a constraint without restricting which method is used to satisfy any other constraints in the system.

The PLANNER algorithm (Algorithm 1) finds a solution graph for a given constraint graph. It starts with an empty set of methods and adds a free method from the constraint graph. The free method’s constraint is removed from the constraint graph, and the algorithm iterates until the graph has no more constraints. The state of the algorithm is captured in two sets of methods: M_s are the methods that are part of the solution, and M_u the methods of the as yet unsatisfied constraints.

Algorithm 1 PLANNER($G\langle V + M, E \rangle$)

```

1:  $M_s \leftarrow \emptyset, M_u \leftarrow M$ 
2: while  $M_u \neq \emptyset$  do
3:   if no free methods in  $G[V + M_u]$  then
4:     return “no solution”
5:    $m \leftarrow$  some free method in  $G[V + M_u]$ 
6:    $M_u \leftarrow M_u \setminus [m]$ 
7:    $M_s \leftarrow M_s \cup \{m\}$ 
8: return  $G[V + M_s]$ 

```

Algorithm 2, HIERARCHY SOLVER, finds the best solution graph given a hierarchical constraint graph G and a strength assignment function s . The maximal value of s , i.e., the strength of non-retractable constraints, is denoted as $must$. The HIERARCHY SOLVER algorithm first divides the methods of the constraint system into those of non-retractable constraints, M_s , and retractable constraints, M_u . If

PLANNER cannot solve the graph of just the non-retractable constraints, then the entire constraint system is not satisfiable. Otherwise, M_s becomes the set of methods of satisfiable constraints, which the algorithm tries to expand.

On each iteration, the algorithm attempts to add to M_s the methods of the strongest retractable constraint(s), $M_u^>$. If the constraint graph remains satisfiable, the new methods are kept; otherwise, they are discarded. When there are no more constraints to try, the solution of the most recent satisfiable constraint graph is the best solution graph.

Although HIERARCHY SOLVER can be used to solve a general hierarchical constraint system, important properties of ordered constraint systems affect its behavior and serve to prove (further below) that the solution returned, if any, is unique. In particular, the initial retractable constraints M_u are the stay constraints, and during each iteration of the loop, $M_u^>$ is guaranteed to be a unique constraint—in fact, it is a set of exactly one stay method.

Algorithm 2 HIERARCHY SOLVER($G[V + M, E], s$)

```

1:  $M_s \leftarrow \{m \in M \mid s([m]) = \text{must}\}$ ,  $M_u \leftarrow M \setminus M_s$ 
2: if PLANNER( $G[V + M_s]$ ) has no solution then
3:   return “no solution”
4: while  $M_u \neq \emptyset$  do
5:    $M_u^> \leftarrow \{m \in M_u \mid \forall m_i \in M_u, s([m]) \geq s([m_i])\}$ 
6:   if PLANNER( $G[V + M_s \cup M_u^>]$ ) succeeds then
7:      $M_s \leftarrow M_s \cup M_u^>$ 
8:      $M_u \leftarrow M_u \setminus M_u^>$ 
9: return PLANNER( $G[V + M_s]$ )

```

With the assumptions that some (small) constants bound each of (1) the number of constraints to which any variable belongs, (2) the number of variables that any

constraint involves, and (3) the number of distinct constraint strength values, Zanden shows [79] that QuickPlan’s worst-case time complexity is $O(n^2)$, where n is the number of constraints in the system. In an ordered constraint system, condition (3) does not hold: the number of distinct constraint strengths is relative to the number of variables in the system. This assumption is not critical; even without it, $O(n^2)$ is an upper-bound of the asymptotic complexity of the above algorithm: HIERARCHY SOLVER is linear in the number of constraint hierarchies, and PLANNER is linear in the number of total constraints. Whether a solution exists at all can be determined in linear time. A polynomial bound is possible thanks in part to method restriction (WF-1) [78].

3.4.2 Uniqueness

Due to the special structure of ordered constraint systems, if a best solution graph exists, it is unique—this is a crucial property for the predictability of UIs built with constraint systems.

Lemma 3.4.1. *At most one solution graph $G^s = \langle V + M^s, E^s \rangle$ for the constraint graph $G^c = \langle V + M, E \rangle$ exists, such that $\forall v \in V, \exists m \in M^s, (m, v) \in E^s$.*

Proof. Zanden [79] showed that G^s has at least one free method. Let $F = \text{frees}([m])$ be the free variables for any free method m . Since m is free, $\text{outs}(m) \subseteq F$. Since $[m]$ is the only constraint attached to the variables F , and m is the only method in $[m]$ that exists in M^s , then $\forall v \in F \setminus \text{outs}(m), \text{ins}_{G^s}(v) = \emptyset$. However, by definition, $\forall v \in V, \text{ins}_{G^s}(v) \neq \emptyset$. Therefore $\text{outs}_{G^c}(m)$ cannot be a proper subset of F , and must be equal to F . Because of WF-3, $[m]$ does not contain other free methods in G^c .

Thus, every satisfiable constraint graph must contain at least one free method, which is the unique choice for its constraint in any solution graph where every variable

is an output of a method. This constraint can be removed from G^c , yielding a new, smaller, well-formed constraint graph $G'^c = G^c[V \setminus \text{frees}([m]) + M \setminus [m]]$. The lemma is true for G^c iff the lemma is true for G'^c . For an empty constraint graph (no variables, methods, or edges) the lemma is trivially true. \square

Theorem 3.4.2 (Uniqueness of best solution graph). *If $\langle G^c, s \rangle$ is an ordered constraint graph, then there exists at most one best solution graph of G^c with respect to s .*

Proof. Let $\langle G^c, s \rangle$ be an ordered constraint graph. Let $G_1^s = \langle V + M_1, E_1 \rangle$ and $G_2^s = \langle V + M_2, E_2 \rangle$ be best solution graphs of G^c with respect to s . Assuming $G_1^s \neq G_2^s$ leads to a contradiction.

First, G_1^s and G_2^s necessarily enforce the same constraints. Since all solution graphs enforce all non-retractable (i.e. non-stay) constraints, G_1^s and G_2^s could only differ in what stay constraints they enforce. Since the strength of each stay constraint is unique, $\text{stays}(G_1^s) \neq \text{stays}(G_2^s) \Rightarrow \text{seq}_s(\text{stays}(G_1^s)) \neq \text{seq}_s(\text{stays}(G_2^s))$, and thus either $G_1^s \succ_s G_2^s$ or $G_2^s \succ_s G_1^s$, violating the assumption that both solutions are best. Therefore, $\text{stays}(G_1^s) = \text{stays}(G_2^s)$.

It remains to show that $M_1 = M_2$, i.e., that if two best solution graphs enforce the same set of constraints, the methods selected from each constraint are the same. Every variable in a best solution graph is an output of some method. Consider the subgraph $G'^c = G^c[V + M']$ of $G^c = \langle V + M, E \rangle$ where $M' = M \setminus \{m \mid [m] \in \text{stays}(G^c) \wedge [m] \notin \text{stays}(G_1^s)\}$. That is, G'^c is G^c excluding the stay methods that do not appear in the solution G_1^s . The solution graphs of G'^c where $\forall v \in V, \exists m \in M', (m, v) \in \text{edges}(G'^c)$ (every variable is an output of some method) are the best solution graphs of G^c with respect to s . According to lemma 3.4.1, at most one such solution graph exists. \square

3.4.3 When the solution graph does not change

Certain changes to a constraint system do not alter the best solution graph. Namely, a solution can be reused when the strength of an enforced stay constraint increases. In other words, editing a variable with an enforced stay constraint does not change the solution. Among other applications, this analysis can be used to avoid the work of computing a new solution graph.

Lemma 3.4.3 (Increasing the strength of an enforced stay constraint does not change the best solution graph). *Let $\langle G^c, s_1 \rangle$ be an ordered constraint graph where $G^c = \langle V + M, E \rangle$, G^s its best solution graph, and c some constraint in $\text{stays}(G^c)$ that is enforced in G^s . Let s_2 be a strength assignment function such that $\forall c_1, c_2 \in M/\sim \setminus \{c\}, s_1(c_1) < s_1(c_2) \Rightarrow s_2(c_1) < s_2(c_2)$, $\forall c_3 \in \text{stays}(G^c) \setminus \{c\}, s_1(c) > s_1(c_3) \Rightarrow s_2(c) > s_2(c_3)$, and $\langle G^c, s_2 \rangle$ an ordered constraint graph. The best solution graph of $\langle G^c, s_2 \rangle$ is G^s .*

Proof. Let $G^c = \langle V + M, E \rangle$, $\langle G^c, s_1 \rangle$ and $\langle G^c, s_2 \rangle$ ordered constraint graphs, G_1^s the best solution graph of $\langle G^c, s_1 \rangle$, and c a constraint in G_1^s . The assumption $\forall c_1, c_2 \in M/\sim \setminus \{c\}, s_1(c_1) < s_1(c_2) \Rightarrow s_2(c_1) < s_2(c_2)$, states that the relative order of strengths of two constraints other than c does not change from s_1 to s_2 . Consider the case where the change from s_1 to s_2 is such that c “jumps” over exactly one constraint in the ordering induced by the strength assignment: $\exists! c' \in M/\sim \setminus \{c\}, s_1(c') > s_1(c) \wedge s_2(c) > s_2(c')$. Name the best solution graph of $\langle G^c, s_2 \rangle$ as G_2^s .

If G_1^s and G_2^s enforce the same stay constraints, then the second half of the proof for Theorem 3.4.2 shows that $G_1^s = G_2^s$. All constraints other than c and c' retain in G_2^s the enforcement status they held in G_1^s because their relative order with all other constraints is unchanged. It remains to show that c , after becoming stronger in s_2 , and c' , after becoming weaker, are the same way. A constraint (c) that is enforced when it is stronger than some set of constraints (call it C) should remain

enforced after becoming stronger than a superset of those constraints ($C \cup \{c'\}$). If c' is retracted in G_1^s , it cannot be enforced in G_2^s where it is weaker. If c' is enforced in G_1^s where c did not need to be retracted, then c' must be enforced in G_2^s as well.

Finally, the above change to s_1 where c moves over exactly one stronger constraint can be repeated arbitrarily many times, from which the lemma follows. \square

The lemma implies that several other transformations can preserve the best solution graph:

- a retracted stay constraint can be given a weaker strength;
- any contiguous region of retracted stay constraints can be permuted arbitrarily; and
- any contiguous region of enforced stay constraints can be permuted arbitrarily.

3.4.4 When the solution graph does change

It is possible to determine the extent of changes to the best solution graph induced by changes to the strength assignment function. Van Zanden observed that whenever QuickPlan attempts to enforce a previously retracted constraint, it must examine only the “upstream” constraints [79, §5.2]. A constraint c_1 is *upstream* from another constraint c_2 if, in the solution graph, there is a path from a method in c_1 to any of the output variables of any method in c_2 .

In the special case of constraint systems for UIs, the strength assignment function changes in a limited, predictable way: when a variable is **edited** (that is, its value is changed from outside of the constraint system), its stay constraint is given the highest strength; all other strengths are left unchanged. Since a stay constraint has only one method with one output, its upstream constraints are all the constraints

with methods in its variable’s *ancestor (directed acyclic) graph*: every variable and method reachable from the variable in the transpose of the solution graph. Therefore, after a variable is edited, the only constraints that may need different satisfying methods are those constraints with methods in the variable’s ancestor graph.

Note that such incremental solving does not change its complexity, which remains, in the worst case, quadratic in the number of constraints. In practice, however, it typically exhibits linear performance.

3.5 Evaluating

Execution of the methods in a plan, in order, is called *evaluation*. Because all methods in a constraint are assumed to enforce the constraint equally well, execution of a method can be omitted if the inputs that the method used in the last evaluation have not changed. This strategy is implemented in a lazy evaluation algorithm whose semantics are given in Figure 3.5.

For a constraint graph $G^c = \langle V + M, E \rangle$, ν is a valuation of variables in V and edges in E . The valuation ν maps a variable to the tuple $\langle t, p, q \rangle$, where t is the current value of the variable; p a flag indicating whether the value of the variable is “computed,” i.e., up-to-date; and q a flag indicating whether the value has changed since the last evaluation. For clarity, instead of Boolean values, p can have values **computed** and **uncomputed**, and q the values **changed** and **unchanged**. Further, ν maps all edges $e = (v, m) \in E$ to one of the values **used** or **unused**. The former signifies that when the code of the method m was executed, the value of the variable v was read, the latter that it was not. Consequently, ν is overloaded so that the expressions $\nu(v)$ and $\nu(e)$ are both valid. The notation $[v \mapsto val]\nu$ represents the valuation function identical to ν , except that the variable v maps to val ; the analogous notation applies for edges.

The following metavariables appear in the definition of the evaluation semantics, with primes, subscripts, and superscripts as appropriate: G^s for solution graphs; V for sets of variables; u and v for variables; m for methods (\cdot is a special value for m that indicates “no method”); t for values of variables; q for values of the changed-flag; ν for valuation functions; μ for mapping a method to the code of the method and to two sequences of variables indicating the input and output variables of the method; and f for the code of a method. The underscore symbol “_” is a placeholder variable that binds to anything, similarly to how it is used in, say, Haskell or ML.

Figure 3.5 defines the functions that evaluate new values for variables, both individually and altogether. The notation $func \mid \nu \rightarrow t \mid \nu'$ has the following meaning: the function $func$ (either `eval` or `evalmany`) is evaluated within the context of the current valuation ν , which produces a new valuation ν' and the result t . The symbol “.” indicates that the function has no result.

The `evalmany` function, defined by the rules `EVALMANY` and `EVALMANY-EMPTY`, simply invokes `eval` for each variable in a set in some order. Each call to `eval` traverses the dependencies in the solution graph upwards and evaluates all variables that are necessary for determining the value of the current variable, and then executes the method of the current variable. Along the way, the information on used input variables is collected and maintained to avoid recomputing a method if its inputs are known to have not changed.

The `eval` function defines how to obtain the value of a single variable. Besides the variable whose value should be obtained, `eval` has two other parameters: the method m that requested the value of the variable and the current constraint graph G^s . The method parameter accepts the value “.”, which indicates that the value of the variable is not requested by a method.

The rules `EVAL-COMPUTED` and `EVAL-COMPUTEDNOMETHOD` define the course of

$$\begin{array}{c}
\text{EVAL-COMPUTED} \\
\frac{\nu(v) = \langle t, \text{computed}, - \rangle \quad m \neq \cdot}{\text{eval}(v, m, G^s) \mid \nu \rightarrow t \mid [(v, m) \mapsto \text{used}] \nu} \\
\\
\text{EVAL-COMPUTEDNOMETHOD} \\
\frac{\nu(v) = \langle t, \text{computed}, - \rangle}{\text{eval}(v, \cdot, G^s) \mid \nu \rightarrow t \mid \nu} \\
\\
\text{EVAL-INPUTS} \\
\frac{\begin{array}{c} \nu(v) = \langle -, \text{uncomputed}, - \rangle \\ \{m'\} = \text{ins}_{G^s}(v) \quad V^{\text{in}} = \text{ins}_{G^s}(m') \quad v' \in V^{\text{in}} \quad \nu(v') = \langle -, \text{uncomputed}, - \rangle \\ \text{evalmany}(V^{\text{in}}, G^s) \mid \nu \rightarrow \cdot \mid \nu' \quad \text{eval}(v, m, G^s) \mid \nu' \rightarrow t \mid \nu'' \end{array}}{\text{eval}(v, m, G^s) \mid \nu \rightarrow t \mid \nu''} \\
\\
\text{EVAL-UNCHANGED} \\
\frac{\begin{array}{c} \nu(v) = \langle -, \text{uncomputed}, - \rangle \quad \{m'\} = \text{ins}_{G^s}(v) \quad V^{\text{in}} = \text{ins}_{G^s}(m') \\ \forall v_i^{\text{in}} \in V^{\text{in}}, \nu((v_i^{\text{in}}, m')) = \text{used} \Rightarrow \nu(v_i^{\text{in}}) = \langle -, \text{computed}, \text{unchanged} \rangle \\ \{v_1^{\text{out}}, \dots, v_l^{\text{out}}, \dots, v_k^{\text{out}}\} = \text{outs}_{G^s}(m') \\ v = v_l^{\text{out}} \quad \nu(v_1^{\text{out}}) = \langle t_1, -, q_1 \rangle \cdots \nu(v_k^{\text{out}}) = \langle t_k, -, q_k \rangle \\ v' = [v_1^{\text{out}} \mapsto \langle t_1, \text{computed}, q_1 \rangle, \dots, v_k^{\text{out}} \mapsto \langle t_k, \text{computed}, q_k \rangle] \nu \\ \text{eval}(v, m, G^s) \mid \nu' \rightarrow t' \mid \nu'' \end{array}}{\text{eval}(v, m, G^s) \mid \nu \rightarrow t' \mid \nu''} \\
\\
\text{EVAL-CHANGED} \\
\frac{\begin{array}{c} \nu(v) = \langle -, \text{uncomputed}, - \rangle \\ \{m'\} = \text{ins}_{G^s}(v) \quad \{v_1^{\text{in}}, \dots, v_l^{\text{in}}, \dots, v_n^{\text{in}}\} = \text{ins}_{G^s}(m') \\ \nu(v_1^{\text{in}}) = \langle -, \text{computed}, - \rangle \cdots \nu(v_n^{\text{in}}) = \langle -, \text{computed}, - \rangle \quad \nu((v_l^{\text{in}}, m')) = \text{used} \\ \nu(v_i^{\text{in}}) = \langle -, -, \text{changed} \rangle \quad \nu' = [(v_1^{\text{in}}, m') \mapsto \text{unused}, \dots, (v_n^{\text{in}}, m') \mapsto \text{unused}] \nu \\ \mu(m') = \langle f, (u_1^{\text{in}}, \dots, u_n^{\text{in}}), (u_1^{\text{out}}, \dots, u_k^{\text{out}}) \rangle \\ f(\nu', \lambda \nu. (\text{eval}(u_1^{\text{in}}, m', G^s) \mid \nu), \dots, \lambda \nu. (\text{eval}(u_n^{\text{in}}, m', G^s) \mid \nu)) \rightarrow (t'_1, \dots, t'_k) \mid \nu'' \\ \nu^{(3)} = [u_1^{\text{out}} \mapsto \langle t'_1, \text{computed}, \text{changed} \rangle, \dots, u_k^{\text{out}} \mapsto \langle t'_k, \text{computed}, \text{changed} \rangle] \nu'' \\ \text{eval}(v, m, G^s) \mid \nu^{(3)} \rightarrow t' \mid \nu^{(4)} \end{array}}{\text{eval}(v, m, G^s) \mid \nu \rightarrow t' \mid \nu^{(4)}} \\
\\
\text{EVALMANY-EMPTY} \\
\text{evalmany}(\emptyset, G^s) \mid \nu \rightarrow \cdot \mid \nu \\
\\
\text{EVALMANY} \\
\frac{\text{eval}(v, \cdot, G^s) \mid \nu \rightarrow \cdot \mid \nu' \quad \text{evalmany}(V', G^s) \mid \nu' \rightarrow \cdot \mid \nu''}{\text{evalmany}(\{v\} \sqcup V', G^s) \mid \nu \rightarrow \cdot \mid \nu''}
\end{array}$$

Figure 3.5: Semantics for lazy evaluation of a solution graph G^s with valuation ν . Functions ins_{G^s} and outs_{G^s} are overloaded for both methods and variables, so that they return the sets of incoming and outgoing vertices in G^s .

action in the case where the computed-flag of the requested variable is set. This indicates that the value of the variable is up-to-date and its value is returned immediately. The EVAL-COMPUTED applies when some method is asking for the value of a variable and thus it additionally sets the used-flag of the “variable to method” edge. EVAL-COMPUTEDNOMETHOD matches when the evaluation request comes from `evalmany`, rather than from executing a method.

The rules EVAL-INPUTS, EVAL-UNCHANGED, and EVAL-CHANGED define what to do when the value of a variable has not yet been computed. Consider evaluating the value of the variable v . All these three rules examine the method m' that outputs to v in the current solution graph, and the input variables of m' .

EVAL-INPUTS matches if any input variable of m' is still uncomputed. The rule invokes `evalmany` to evaluate the input variables, then invokes `eval` for v again.

EVAL-UNCHANGED matches when all the input variables of m' are computed and all the input variables that are used by m' are unchanged. In this case, it is not necessary to execute the method m' again. The new valuation marks all of the output variables of m' as computed, but does not change their values or changed-flags. Finally, `eval` is invoked again for v to effect the possible update of the used-flag for an edge from v to some method m . Note that the premise $v = v_i^{\text{out}}$ in EVAL-UNCHANGED is redundant; it is included to make it obvious that v is one of the output variables of m' . Further, if m' is a stay method, it has no inputs, and thus EVAL-UNCHANGED applies and retains v 's valuation unchanged.

EVAL-CHANGED matches when all the input variables of m' are computed and at least one input variable of m' has changed. If this is the case, the code of m' needs to be executed. This entails (1) retrieving the code f of m' , (2) constructing a callback function for each input variable of f that will obtain the value of the input variable by another call to `eval`, (3) passing the callbacks to f , and (4) executing f . If the

code of a method invokes any of its callbacks, a new call to `eval` results, where the method requesting the value of the variable is m' . This call will eventually reach `EVAL-COMPUTED`, which will set the corresponding used-flag of the edge to m' . Once the method f returns, the values in the tuple it returns are written to the output variables of m' (v is one of them) and the computed- and changed-flags of these output variables are set as well.

The correctness of the evaluation phase depends on methods never copying and saving the current valuation: the same shared state should always be used when invoking any of the callbacks of a method, and not held after the method returns. Note also that since arbitrary code is allowed to be executed in methods, termination cannot be guaranteed.

3.6 Evaluation Graph

The used/unused distinction for method input edges calculated during evaluation is captured in a subgraph of the solution graph called the *evaluation graph*. Input variables are passed to a method by name: to obtain a value of one of its input variables, a method must explicitly ask for it. Only if a method m asks for the value of its input variable v during execution of the method is the edge (v, m) included in the evaluation graph. The variable v and the input edge (v, m) are said to be *used* by the method m . Assuming a solution graph $G^s = \langle V + M, E_V + E_M \rangle$, where E_V are the edges whose target vertex is in V , and E_M the edges whose target vertex is in M , the evaluation graph G^e is the subgraph of G^s induced by the edges $E_V + E_u$ where $E_u \subseteq E_M$ are the used input edges. That is, $G^e = \langle V + M, E_V + E_u \rangle$.

A possible evaluation graph for the solution in Figure 3.4a appears in Figure 3.6.

4. ALGEBRA*

This chapter examines hierarchical multi-way data-flow constraint systems as a *commutative monoid* [41]. This correspondence appears to be missing from the early literature. Maybe it is not all that surprising, or maybe the authors working on constraint systems did not bother to make the connection explicit—the elements of such a connection, such as the operation of adding a constraint to an existing constraint system and its impact on the system’s solution, have been described [28, 25].

Investing in understanding how a data structure conforms to well-known algebras is beneficial. It can make properties of the data structure easily apparent. For example, when viewed as a monoid, solving a constraint system becomes a fold of the monoid’s binary operation over the system’s constraints. In turn, the connection simplifies and clarifies many aspects of hierarchical multi-way data-flow constraint systems, easing new paths of research.

4.1 Constraint Systems as Monoids

To view constraint systems as a commutative monoid, consider methods to be graphs directly following the construction of constraint graphs as given in Section 3.2. That is, the method graph of a method $m \in M$ of some constraint $C = \langle R, r, M \rangle$ is a graph with one method vertex m , variable vertices R , and edges (u, m) for each $u \in ins(m)$ and (m, v) for each $v \in outs(m)$. To not confuse the different views of

* Portions of this chapter are reprinted from Jaakko Järvi, Magne Haveræen, John Freeman, and Mat Marcus. Expressing multi-way data-flow constraint systems as a commutative monoid makes many of their properties obvious. In *Proceedings of the 8th ACM SIGPLAN workshop on Generic programming, WGP '12*, pages 25–32, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1576-0. doi: 10.1145/2364394.2364399. URL <http://doi.acm.org/10.1145/2364394.2364399> Copyright 2012 ACM.

a method, an undecorated method name, say m , refers to the single method vertex, and a method name decorated with the superscript \cdot^g , say m^g , refers to the method graph.

A single constraint like C is in itself a constraint system. C can be satisfied by executing any of its methods; any of C 's method graphs m^g will, by construction, meet the requirements of a solution graph, as described in Section 3.4 (in short, no variable is assigned a value twice, or assigned after being read). This set of solution graphs, each a directed acyclic graph with at most one in-edge to each variable vertex, represents all possible solutions of the constraint system arising from the single constraint C .

In our monoid, the carrier set is the set of all constraint systems, each represented by its set of solution graphs. A system of a single constraint is thus trivially represented by that constraint's method graphs.

The composition operator in this monoid is a refinement of graph union. Operationally, it takes two constraint systems, forms the cartesian product of their two sets of solution graphs, computes the graph union of each pair of this product, and discards those graphs that are not solution graphs. Formally, let $A = \{a_1^g, a_2^g, \dots, a_m^g\}$ and $B = \{b_1^g, b_2^g, \dots, b_n^g\}$ be constraint systems (sets of solution graphs). Then our monoid operation is defined as $A + B = \{c^g \mid a^g \in A, b^g \in B, c^g = a^g \cup b^g, c^g \text{ a solution graph}\}$. This operator is both associative and commutative because of the associativity and commutativity of graph union.

As a simplification, a constraint system can be regarded as a single constraint, and its solution graphs regarded as method graphs of that constraint. Consider a constraint system of two constraints, $A = \langle R_a, r_a, M_a \rangle$ and $B = \langle R_b, r_b, M_b \rangle$. When composed, they form a single constraint $C = \langle R_c = R_a \cup R_b, r_c = r_a \wedge r_b, M_c = M_a + M_b \rangle$. Each method $m_c \in M_c$ is the union of some two methods, call them $m_a \in M_a$

and $m_b \in M_b$. To consider m_c a method itself, we must define its sets of input and output variables. Because outputs cannot be shared when composing method graphs, we must keep track of the full set of outputs as we compose: $outs(m_c) = outs(m_a) \sqcup outs(m_b)$. Method restriction (WF-1: a method must use all of its constraint's variables as either input or output) then dictates that $ins(m_c) = R_c \setminus outs(m_c)$. Note that $ins(m_c) \subset ins(m_a) \cup ins(m_b)$ when any outputs of one method are inputs to the other.

The identity element is the singleton set, whose sole element is the null graph. This is easily seen:

$$\begin{aligned}
& \{a_1^g, a_2^g, \dots, a_m^g\} + \{(\{\}, \{\})\} \\
&= \{a_1^g \cup (\{\}, \{\}), a_2^g \cup (\{\}, \{\}), \dots, a_m^g \cup (\{\}, \{\})\} \\
&= \{a_1^g, a_2^g, \dots, a_m^g\}.
\end{aligned}$$

Note that the empty set is an *absorbing element* of the constraint composition operator; the cartesian product of a set with an empty set is an empty set. Further, the $+$ operation is *idempotent*: for all constraints C , $C + C = C$. This follows, as the union of two different method graphs from the same constraint is cyclic, and thus the only method graphs that remain in the result are the unions of each method graph with itself; and graph union is idempotent.

The size of $A + B$ can be up to $m \times n$ methods. For it to be interesting to compose constraints, the constraints must share variables, often in ways that will disqualify many combinations of methods. Figure 4.1 shows the constraint graph arising from the composition of two constraints, each containing three variables; two of the variables are shared. The example from which these two constraints arise is modeling the relation of a rectangle's width w , height h , and area A ; and, respectively, its width, height and perimeter p . Each constraint has three methods:

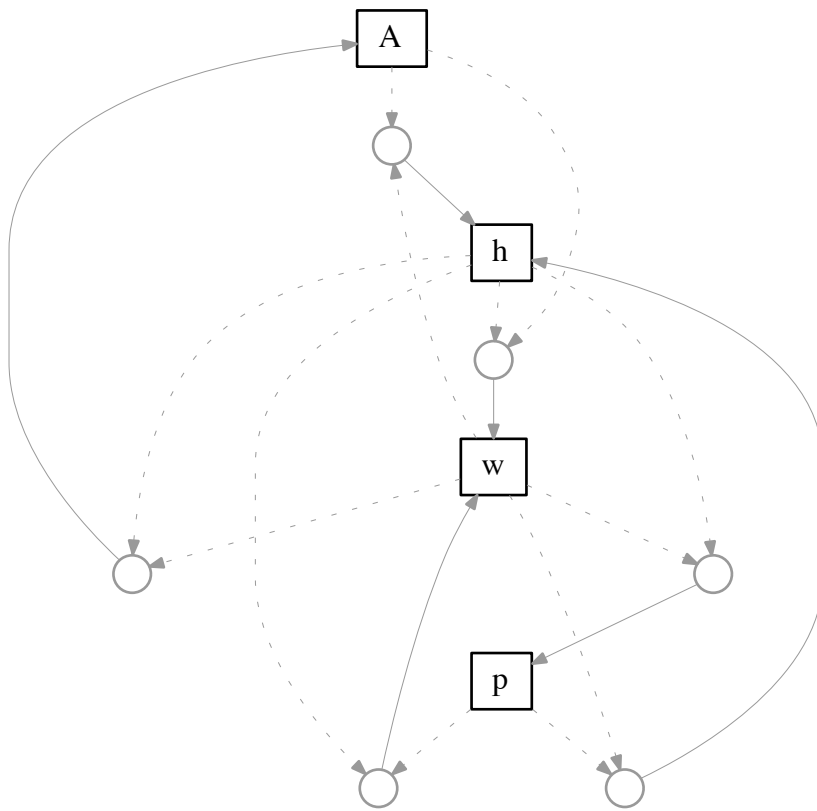


Figure 4.1: Graph of a constraint composed of two other constraints. The first represents a three-way relation between A , w , and h ; the second a three-way relation between p , w , and h .

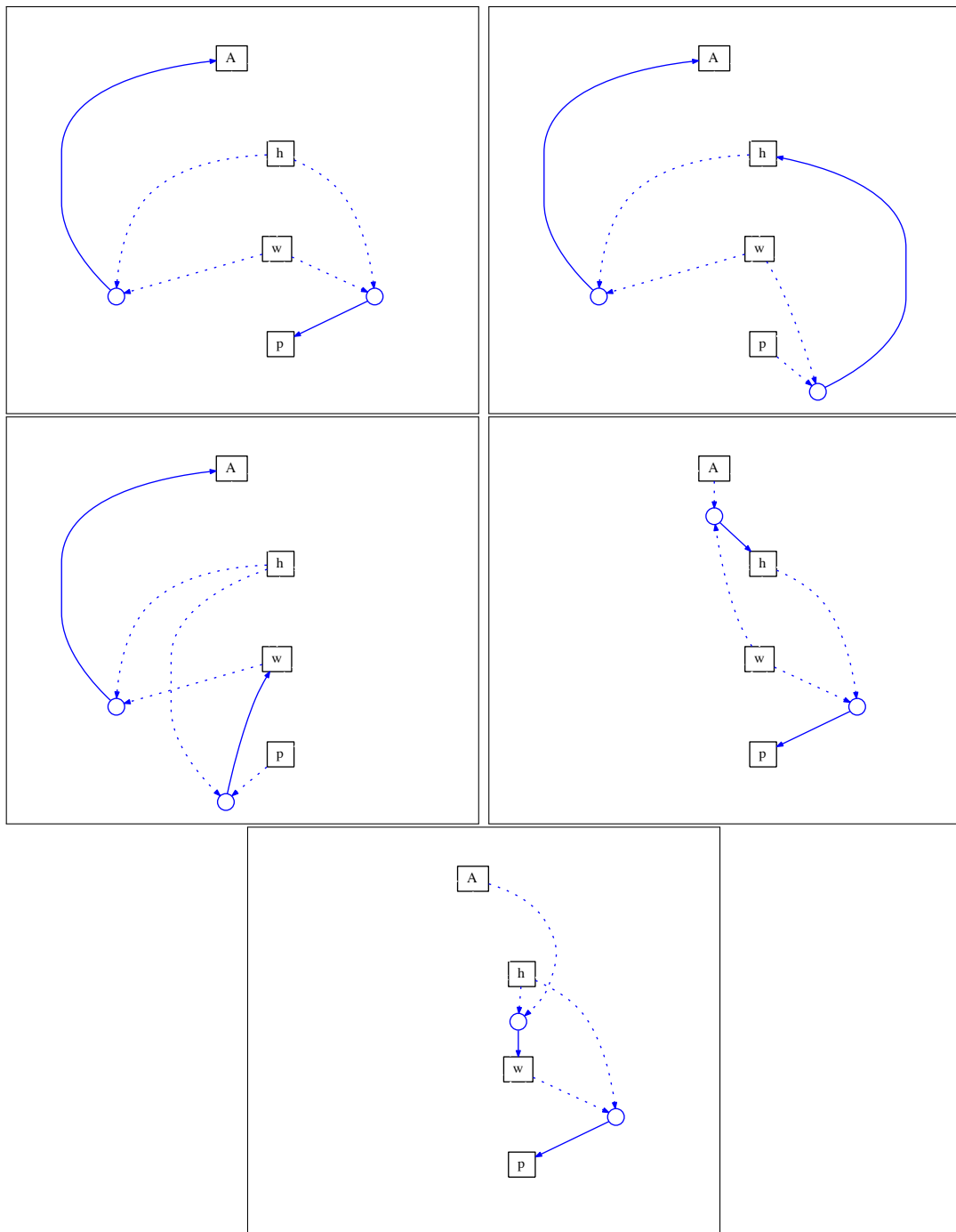


Figure 4.2: All method graphs of the constraint graph in Figure 4.1.

any one variable can be computed from the other two. Instead of 3×3 methods, the combined constraint has five methods. Figure 4.2 shows all these method graphs.

4.2 Implementation in Haskell

A concrete implementation of the abstract algebra can help demonstrate the correspondence. Figure 4.3 shows a Haskell implementation of the monoid. The necessary module `import` statements are included on lines 1–5 to make the code complete and executable. Various modules are imported as *qualified* to make it clear in which libraries various functions are defined.

As explained in Section 3.2, all structure of a constraint system can be recovered from the constraint graph. That is, one does not need to maintain auxiliary information of which method vertices belong to which constraints; or of which “elementary” constraints the system was composed. We can thus leverage a general purpose graph library, and directly represent methods, constraints, and constraint systems as graphs. Here, we use Haskell’s `Data.Graph.Inductive` library, based on Erwig’s *Functional Graph Library* (FGL) [16].

The graph, method, and constraint type definitions are shown on lines 7–11. The graph type is `G.Gr NodeKind ()`. The first type parameter of `G.Gr` represents the type of the label of nodes, the second the type of the label of edges. A node in the FGL graph is a pair of an integer and a label. The integer is a node’s unique identifier. The label type `NodeKind` indicates whether a node represents a variable or a method. An edge is a triple, whose elements are an integer (source), integer (target), and a label. The source and target integers each should refer to an existing node identifier. No additional information is necessary for edges, hence the label type is `unit`.

The method and constraint types directly follow the definitions above: a method

```

1 import Data.Monoid
2 import qualified Data.Graph.Inductive as G
3 import qualified Data.Graph.Analysis.Algorithms.Common as GA
4 import qualified Data.List as L
5 import Data.Maybe (catMaybes)
6
7 data NodeKind = VarNode | MetNode deriving (Eq, Show)
8
9 type Method = G.Gr NodeKind ()
10
11 data Constraint = Constraint [Method] deriving Show
12
13 methodUnion :: Method -> Method -> Maybe Method
14 methodUnion g1 g2 =
15   let ns1 = G.labNodes g1
16       ns2 = G.labNodes g2
17       es1 = G.labEdges g1
18       es2 = G.labEdges g2
19       common = L.intersect ns1 ns2
20       g = G.mkGraph (ns1 'L.union' ns2) (es1 'L.union' es2)
21   in
22     if all (<= 1) (map (G.indeg g . fst) common) &&
23       null (GA.cyclesIn g)
24     then Just g
25     else Nothing
26
27 instance Monoid Constraint where
28   mempty = Constraint [G.empty]
29   mappend (Constraint as) (Constraint bs) =
30     Constraint $ catMaybes [a 'methodUnion' b | a <- as, b <- bs]

```

Figure 4.3: Implementation of a data-flow constraint system planner in Haskell.

is a graph and a constraint a list of methods, also graphs. The general graph type `Gr.G` can represent any graphs, including ones that are not valid method graphs. Further, a list of method graphs might not constitute a valid constraint. We assume as an unchecked precondition that methods and constraints are always well-formed. Indeed, constraints are typically constructed with dedicated functions that guarantee well-formedness.

The workhorse of the monoid's binary operation is the function `methodUnion`.

This function computes the union of two graphs, but only accepts the result if it is a candidate solution graph. The FGL library does not provide a graph union function, so we implement here a simple graph union operation (`methodUnion`). The `methodUnion` function extracts both two lists of nodes and two lists of edges from its argument graphs, uses the list union function to combine the lists into one of each kind, and then reconstructs a graph from them.

The resulting graph is checked against the requirements for a solution graph on lines 22 and 23. The first checks that no variable is an output of more than one method. This check is only necessary for the variables that are **common** to both of the composed graphs. The function `fst` accesses the node index of a labeled node, `G.indeg` computes the number of incoming edges of a node. The second line checks whether the resulting graph contains cycles.

Taking advantage of the `methodUnion` function, `Constraint` can be made an instance of the `Monoid` type class without much effort. The identity element (called `mempty` in Haskell) is a singleton list of methods, where the method is the empty graph. The binary operation, `mappend`, applies `methodUnion` to each combination of the operands' methods, and collects the results in a list. The `catMaybes` function prunes the `Nothing` values that result from the failed method unions where the result does not satisfy the conditions of a solution graph.

With constraints defined as monoids, writing a planner algorithm becomes quite straight-forward. Assuming a constraint system is a list of constraints, the list of all possible plans/solution graphs for it, as a single `Constraint`, is obtained with the following function:

```

1 plans :: [Constraint] -> Constraint
2 plans = mconcat

```


Expanding `mconcat`, `plans = foldr mappend mempty`.

4.3 Constraint System Properties as Monoid Properties

The monoid-view makes it easy to study the strength assignment’s impact to the solution. To find the best locally-predicate-better solution, one starts from the original under-constrained system as the current solution, and adds stay constraints to it in the order from the strongest to the weakest. The results of additions that fail (return the absorber element) are ignored. For concreteness, we express this in Haskell. The order of the list of constraints passed to `prisolve` determines the strength assignment:

```
1 isAbsorber (Constraint []) = True
2 isAbsorber _ = False
3
4 prisolve :: [Constraint] -> Constraint
5 prisolve ls = foldl1 (\a b -> let sum = a `mappend` b in
6                          if isAbsorber sum then a else sum) ls
```

Understanding the mapping from strength assignments to the set of possible plans is desirable in user interface design. Questions such as “can the user interface ever override the value of any variable that a user is editing?” or “if variables `x`, `y`, and `z` are edited in this order, will the user interface modify either `x` or `y`; and if so, which?” are sometimes hard to answer conclusively based on complex event handling logic.

With the help of the monoid-view, the first question is answered by confirming that adding any of the stay constraints alone to the original under-constrained system yields a satisfiable constraint. Assuming the `stays` are the stay constraints of all variables, `cs` the original under-constrained constraint system (composed into one constraint), the query can be written in Haskell as `any (isAbsorber . (mappend cs)) stays`. The second question can be answered by adding the stay constraints of the variables

z and y to the original system, then those of z and x , and observing the possible solutions.

The impact of a change in the strength assignment is a source of interesting questions. In particular, what changes may require recomputing a new solution graph? Some properties that are obvious with the monoid view are:

- Increasing the strength of an enforced constraint does not change the best solution graph.
- Decreasing the strength of an unenforced constraint does not change the best solution graph.

Since the best solution graph is the sum of a sequence of constraints, which is commutative and associative, adding the same constraints in a different order yields the same solution. The question then is could strengthening an enforced constraint so that it surpasses some unenforced constraint in the preference order enable that constraint to be enforced. Let \perp denote the absorber constraint, and a , b , and c some constraints s.t. $s(a) \geq s(b) \geq s(c)$. If a and c are enforced in the best solution but b is not, then $a + b = \perp$ and $a + c \neq \perp$. Clearly $(a + b) + c = \perp = (a + c) + b$ and thus b cannot be enforced even if c is added to the solution first. The same reasoning applies to the second case, weakening an unenforced constraint, here moving b later than c . We note that in the two related cases, decreasing (respectively increasing) the strength of an enforced (unenforced) constraint, the best solution graph may change.

In constraint graphs for user interfaces, increasing the strength of an enforced constraint is a common occurrence. Every time a user changes focus and updates a new variable, this variable is given the highest strength. A determination must be made whether the change can impact the best solution graph, that is, the direction

of data propagation in the user interface.

Finally, the monoid view allows for specializing constraint systems. Even though there may be many constraints, the number of possible solution graphs of the entire constraint system may be small. In cases where the number of solutions is not small for the entire system, there may be subsets of constraints for which there are only a small number of solutions. Such subsets of constraints are candidates for composing into a single constraint. Both the determination of which constraints should be composed and the actual composition are easy in the monoid view. Determining the optimal subsets to compose, however, can be computationally expensive with the rather simplistic constraint representation here.

In some cases it might be beneficial to get rid of the entire constraint solver. For example, HotDrink [22] is a JavaScript library for building web user interfaces based on ordered constraint systems (described in Chapter 6). When loading a page, the browser loads an implementation of a constraint system solver and a specification of a constraint system, both in Javascript. Typically the constraint system stays unchanged after loading, in which case an alternative is to translate the entire constraint system into a specialized sequence of branching instructions. When the total number of plans for a constraint system is small, the branching logic may be quite simple. The monoid-view is helpful for implementing these kinds of analyses and translations.

5. BEHAVIORS*

With an ordered constraint system as a view-model, rich UI behaviors can be implemented as generic, reusable algorithms over the three graphs—constraint, solution, and evaluation—described in Chapter 3 [39, 40, 23]. Concretely, this means that a GUI programmer can create user interfaces with these features for little or no additional cost.

This chapter presents algorithms for propagating values, visualizing dataflow, pinning values, recording and replaying user interface scripts, activating and deactivating commands, generating context-sensitive help, and enabling and disabling values. For each of them, we define a formal semantics, which serves as a specification for the corresponding algorithm, allowing us to unambiguously verify its correctness. Contrast that with the typical means of implementing these behaviors: ad hoc estimations of human interface guidelines, different for each platform and open to interpretation [7, 52, 3].

* Portions of this chapter are reprinted from Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Property models: from incidental algorithms to reusable components. In *GPCE'08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 89–98, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-267-2. doi: 10.1145/1449913.1449927. URL <http://doi.acm.org/10.1145/1449913.1449927> Copyright 2008 ACM; Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Algorithms for user interfaces. In *GPCE'09: Proceedings of the 8th international conference on Generative programming and component engineering*, pages 147–156, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-267-2. doi: 10.1145/1621607.1621630. URL <http://doi.acm.org/10.1145/1621607.1621630> Copyright 2009 ACM; John Freeman, Jaakko Järvi, Wonseok Kim, Mat Marcus, and Sean Parent. Helping programmers help users. In *GPCE'11: Proceedings of the 10th ACM international conference on Generative programming and component engineering*, pages 177–184, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0689-8. doi: 10.1145/2047862.2047892. URL <http://doi.acm.org/10.1145/2047862.2047892> Copyright 2011 ACM.

5.1 Value Propagation

Consider the image resize example dialog from Section 2.2.1. If the user edits the absolute height, the relative height must change to preserve the three-way relationship among the initial, absolute, and relative height values. If the aspect ratio is preserved, the change will flow into the values for width as well, and on down to the parameters for the resize command itself. This process is called *value propagation*.

As discussed on page 9, in the MVVM pattern, the role of maintaining consistency between the internal view-model and its visual representation (the view) falls to *bindings*.

Bindings are typically implemented as event handlers. Each handler works in one of two directions, updating the view-model in response to activity in the view, or vice versa. Among other semantic events, a view might generate events from user interactions, and a variable might generate events from changes in its value. In this example, clicking the “OK” button leads directly to executing a “resize” command, but some handlers might have more responsibilities. Changing any single number field may result in changes to the three others, depending on the value of the “preserve ratio” option. As the values and relationships in a UI grow, the event handlers may need to be revisited and extended.

With a view-model based on an ordered constraint system, these event handlers can be implemented generically. Each event handler connects exactly one element in the view to exactly one variable (representing either a value or a command) in the view-model. No event handler is left with the responsibility of managing relationships among view-model values; that logic is left to the constraint system, and the handler is reduced to just modifying a view-model variable on behalf of the user (an *edit*) or refreshing a view to reflect changes in the view-model.

Formally, the state of a view-model's ordered constraint system, the *current configuration*, is a tuple $J = \langle G^s, s, \nu \rangle$, where G^s is a solution graph of the ordered constraint graph $G^c = \langle V + M, E \rangle$ with respect to s , a strength assignment function, and ν is a valuation of variables in V and edges in E .

Assuming a constraint graph $G^c = \langle V + M, E \rangle$ and a current configuration $J = \langle G^s, s, \nu \rangle$, assigning a new value, say t , for some variable v has the following effect on J :

1. A new strength assignment s' is computed from s , such that the stay constraint of v will become the strongest of the stay constraints, and the relative order of other stay constraints remains the same. Thus, variable v is given the highest priority.
2. If necessary (according to the analysis in Section 3.4.3), the solver algorithm is run to produce a new solution graph G'^s ; otherwise, $G'^s = G^s$.
3. A new valuation ν' is computed from ν as follows: the value of v is set to t ; the computed-flag of every variable is set to **uncomputed**; the changed-flag is set to **changed** for v and to **unchanged** for all the other variables; and the used-flag is set to **used** for all edges (v', m') from variables to methods for which (v', m') is not an edge in G^s but is an edge in G'^s . A method is not executed if it can be seen that its used inputs have not changed. The above treatment of used-flags makes sure that all new methods of G'^s that were not included in G^s will be executed during evaluation.
4. The **evalmany** function, shown in Figure 3.5, is applied to the set V . It produces a new valuation ν'' .

The result of the above steps is a new current configuration $\langle G'^s, s', \nu'' \rangle$. The event

handlers for each variable $v \in V$ such that $\nu(v) = \langle -, -, \text{changed} \rangle$ are then executed to refresh the corresponding views.

Going back to the example, editing the absolute height will assign the highest strength to the stay constraint for absolute height in the ordered constraint system. Whether or not the solution must be recomputed, it must include methods for the constraints that assign to the relative height, then to the relative width, then to the absolute width, then to the resize command. Evaluating that graph will calculate values for all of the other variables. Any changes in their values will lead to invoking bindings that update the views attached to them. The result is consistency among the variables in the view-model (the application’s perspective of the state) and the view objects (the user’s perspective).

5.2 Dataflow Visualization

When an edit of one value triggers changes in other values, it can be unclear to the user which values are changing and for what reasons: changes may happen too quickly or inconspicuously to notice; it may appear that they are all directly related to the edited value, which could be misleading; or values may change in ways unexpected by the user. Each of these complications contributes to a gulf of evaluation with respect to value propagation.

Consider the image resizing dialog. It provides two ways of editing the image dimensions—either absolutely, in pixels, or relatively, in percentages—with the option to preserve the ratio between the image’s height and width. The values of all four textboxes and the checkbox are tied together in a complex, multi-way relationship. If the user edits one of the numeric values, the other three could change. To understand this behavior, a user will need to know which values currently affect which others.

Visualization techniques could help bridge the gulf of evaluation by communicating this information to the user, but they are often tedious or complicated to implement in traditional GUI frameworks. Consequently, such features rarely get written. However, ordered constraint systems open new possibilities because the network of dependencies among variables is readily available in the evaluation graph: a variable's value is affected by its ancestors and affects its descendants. A generic implementation of a visualization can thus be written once and then reused among all UIs with an ordered constraint system view-model.

We implemented a generic visualization of dependencies for GUIs and conducted experiments with this behavior [23]. For each method in the evaluation graph, an arrow is drawn connecting the widgets bound to its inputs and outputs. (Consequently, this means the arrow might have multiple heads or multiple tails or both.) To prevent confusion, an arrow is drawn only if the outputs had changed as a result of the most recent evaluation. These arrows are displayed, one at a time, according to the order of the methods' evaluation; an example is shown in Figure 5.1. To avoid interrupting casual use of the interface, the dataflow illustration is triggered only upon the explicit request of the user, indicated by clicking a button.

Now, the animation makes the relationships behind the interface explicit and clear, and thus fosters understanding in the user. Further, this behavior works for all interfaces built with ordered constraint systems for no additional cost.

We identified some potential improvements. In one way, the animation seemed to play too quickly. The time an individual relationship was displayed seemed too short for digesting the information the visualization was trying to convey. In another way, the animation seemed to play too slowly. If the user wanted to focus on one particular relationship, which was often the case, then he would have to wait for the animation to get to that point, and then wait for it to complete before seeing

Height

Initial : 1500

Absolute :

Relative :

Width

Initial : 2100

Absolute :

Relative :

Constrain Image Proportions :




Figure 5.1: Implementation of the image resize dialog from Section 2.2.1 with an arrow visualizing one method in the evaluation graph.

it again. The animation could have conformed better to how the user wanted to consume the relationship information—typically the user was interested in just one or two relationships out of the whole graph.

Consequently, a more desirable alternative to animation may be to allow the user to glance at individual relationships separately and at their own pace. Contextual information for a field would include arrows connecting it to the the fields from which it was computed, and to the fields computed from it. This contextual information could be displayed, say, along with existing menus upon right-clicking a field. By displaying the network of dependencies in localized chunks, the UI avoids presenting too much information at once and cluttering the display. Finally, a more sophisticated layout algorithm could be used to produce more visually appealing arrows.

In the bigger picture, however, the experiment suffices to demonstrate that an ordered constraint system enables a data flow visualization to be described and implemented in a generic, reusable manner, and thus make it worth the while to invest more on improving the visualization, to eventually benefit a large class of user interfaces.

5.3 Pinning

A rich user interface may automatically, without consent from the user, change values in order to enforce relationships among them. Even though reasonable heuristics are applied, the behavior may be unexpected to the user. In the hotel reservation dialog from Figure 2.2, consider this sequence of user actions:

1. The user enters a check-in date.
2. The user enters a check-out date.
3. The user realizes he did not mean to stay for the calculated number of nights,

so he changes the number of nights value.

The user's expectations may or may not agree with the heuristic that an ordered constraint system follows. If the user is editing the nights in order to correct his last edit to the check-out date, then he might expect the check-out date to change accordingly. However, since the check-out date was edited more recently, the system assumes that the user would rather preserve that value, so it changes the oldest value: the check-in date.

In a more complicated interface with more relationships, one can imagine that such a turn counter to user's expectations could lead to the undoing of a larger portion of the user's work. No rule can be "correct" in all cases, as sometimes there is no single unsurprising dataflow. The expectation of the user interface's most natural behavior may, for different users (or even for the same user) in identical situations, be different. A rich UI should thus provide means to the user to control the preferred dataflow.

One possible control mechanism allows the user to "protect," or *pin*, certain values as he moves along. Pinning does not prevent the user from further editing the value. It simply guarantees that the system will not automatically change the pinned value as a reaction to the user changing some other value. In Figure 2.2, each of the check-in, check-out, and number of nights fields is accompanied by a checkbox that, when checked, pins the corresponding variable in the view-model.

Implementing pinning with event handlers is laborious, and thus it is seldom seen. We found only a single paper discussing it [44], and very few applications supporting it. However, pinning is straightforward in an ordered constraint system, which enables a generic implementation of the feature. As explained in Section 3.3, a stay constraint keeps a variable's value unchanged. The desired effect is thus attained

if the stay constraint of the pinned variable is promoted to the same strength as that of the programmer-defined constraints, that is, the highest (*must*) strength. This guarantees that the stay constraint will be enforced in all solution graphs, and thus the pinned variable is not overwritten by any method.

Pinning is not without complications, though. Pinning a variable expands the set of constraints that must be enforced in each plan. If enough variables are pinned, the property model could become overconstrained, leaving it unsolvable. To prevent this, the pinning option can be disabled for widgets bound to variables that, if pinned, would result in an overconstrained system. Additionally, since editing a variable has the same effect as pinning it (i.e. promoting its stay constraint to highest priority), input widgets attached to such variables can be disabled, indicating that the variable's value is derived in all solutions given the existing set of pinned variables.

Identifying a variable as “pinnable” is straightforward: a single run of the constraint solver suffices to determine if the system can still be solved after a particular variable is pinned. Furthermore, after pinning a variable, it may be that some other variables are *derived* in all possible plans. Any user edits to such values will be overwritten. To prevent confusion, widgets bound to such variables should be disabled. These variables can be identified as well: after removing from constraints any methods that write to pinned variables, any variables written by a method in a constraint with no other methods will be derived in all solutions.

To summarize, using an ordered constraint system to model a UI enables a simple, reusable implementation of pinning. From an application programmer's perspective, they need only create view objects to represent pins (checkboxes are a common choice) and bind them to the “pinned” flag for a variable. The solver, as part of its algorithm, promotes pinned variables to highest strength before computing a solution graph. The evaluator handles bindings that enable and disable the pin view objects

for pinnable and unpinnable variables, respectively.

5.4 Scripting

A *script* is a sequence of actions that may be re-executed, or “played back”, in a context different from which it was recorded. Scripting is a feature rarely found in user interfaces because in traditional frameworks it typically involves duplicating user interface code. Some applications try to implement scripting by recording a sequence of UI events, which can lead to a degraded user experience (often in the form of a flickering UI) as the script is replayed.

Consider the image resizing dialog in the case where the preserve ratio flag is unset, and the relative dimensions have been edited most recently, say to 50% each. In this case, the resulting absolute width and height, 750 and 1050 respectively, are *derived* variables. The initial height and width values were populated from the model. Such initial values, i.e., those that are picked up from a particular (model) context rather than the user, are *input* variables. The set of non-derived, non-input variables, that is, the set of variables supplied directly by the user that contribute in some way to the executed command, are referred to as *contributing* variables.

As described above, a script should record the relative dimensions and the preserve ratio flag. That way, when the script is played back later against an image with initial dimensions 600 by 800, it will have the intended effect of scaling the image by 50% in each dimension, down to 300 by 400. Had the script failed to recognize the user’s intent, and stored the absolute dimensions, it would have incorrectly sized the second image to 750 by 1050.

Solution graphs can distinguish derived variables from non-derived variables. Non-derived variables are those with no in-edges (except from a stay method). Of non-derived variables, those that have been edited by the user (a distinction that

can be tracked with a flag initialized to false and set to true upon editing) are contributing.

Playback of scripts can also utilize an ordered constraint system. A recorded action in a script specifies which command to execute together with the context-independent values that contribute to synthesizing the parameters for the command. When a script is executed in a particular context, the application (1) constructs an ordered constraint system for the command specified in the script, (2) populates the input variables from the context-dependent data and the contributing variables from values recorded in the script, (3) solves and evaluates the constraint system, and (4) executes the command.

5.5 Command Activation

Programmers often follow “rules of thumb” offered by user interface guidelines for the deployment platform [7, 52, 3] to guide the decisions on when to enable and disable widgets. The advice may differ between the guides, but certain universally accepted reasons can be identified. To discuss these reasons, two forms of enablement are distinguished: enablement of widgets that launch commands (such as command buttons) and enablement of widgets that allow users to interactively edit the widgets’ value (such as text and list boxes, and radio buttons). Although these behaviors fall under the general scope of enablement, the reasons for decisions to enable or disable a widget are unrelated, and the mechanisms governing those decisions completely different. The former mechanism is called *command activation* and the latter *widget enablement*. Both are significant sources of complexity in GUIs’ event handling logic in the predominant GUI programming paradigm.

The result of command parameter synthesis is a set of parameters, possibly subject to preconditions, for a command. Ideally, a user interface avoids constructing

command objects from parameters that do not satisfy the preconditions. A UI may thus be expected to provide a “latch” that controls when a command object can be constructed. The canonical form of such a latch is activating and deactivating the “OK” button of a dialog. Consider the hotel booking example from Section 2.2.2. The dialog in Figure 2.2 for booking a hotel room will launch a command to reserve a room when the “Book” button is clicked, but it may be desirable to keep this button inactive if the check-out date comes before the check-in date, or if the check-in date comes before the current day. Alternatively, the button could be kept activated, but an error diagnostic is shown to the user if it is clicked. The generic behavior for command activation outlined here is neutral on these matters; the behavior is a policy decision made in the presentation layer that sits on top of the view-model.

In the view-model, command objects are held in variables. Preconditions are expressed as Boolean variables computed from other variables and associated with command variables, establishing dependencies in the constraint system between each command and the variables that determine its availability. Thus, a command activation algorithm can determine that a command widget should be deactivated if the variable to which the widget is bound depends on a precondition variable with a false value.

5.6 Explaining Command Availability

From the perspective of the user, a couple of issues surround the behavior described above. First, a user may not understand why a particular command widget is deactivated. As stated above, programmers can express these reasons through preconditions in the property model. However, since they appear only in the user interface code, the user may not know what they are and, consequently, have a limited understanding why a command widget is deactivated. From a prior experience,

when trying to change a password to a web system, an error message simply repeated that the new password entered did not satisfy the requirements of a valid password. The requirements were mostly revealed through trial and error. A large portion of the users of that system might fail in the same task on the first, second, or even third try.

To alleviate the above problem, help text can be automatically generated that describes the reasons why a command widget is deactivated. If the programmer adds a natural language description to the precondition variable, then it can be presented to the user as an explanation for a deactivated command widget. The explanation could be included with other contextual information, such as the dataflow visualization described in Section 5.2.

The second problem is that even after reading an explanation, a user may not know the actions necessary to re-activate a command widget. The constraint system can reveal which variables are responsible (to varying degrees) for the failed precondition. The user can be directed to the widgets bound to those variables and expected to deduce how to interact with them to satisfy the precondition. The responsible variables are the ancestors, in the evaluation graph, of the failed precondition, i.e., the variables that contributed to its false value. A UI can find the interactive widgets that are bound to those variables and reference their labels in the help text.

A UI is not limited to just this implementation, however. To varying degrees, other variables could be considered responsible for a failed precondition, including all variables that can reach it in the constraint graph. In some cases, such variables could be edited to affect the precondition's value, providing an alternative means to satisfy it. Additionally, instead of just listing the interesting widgets in the explanation, a UI could highlight them when the user hovers over each name. This should resolve situations where it may not be clear to the user which widgets are being referenced.

The generation of help text is completely orthogonal to and works in harmony with the dataflow visualization described in Section 5.2. After marking values that are violating the precondition, the user can see the network of relationships among them to better determine a root cause.

5.7 Widget Enablement

The intuitive reason for disabling a value widget is when the value of the widget cannot affect any command in the view-model. Consider again the hotel booking dialog. Part of the form asks for the number and ages of child guests. Depending on the number of children, a corresponding number of age entry fields should be provided; in this particular example, that number is two. However, if the user selects a lower number, one or both of the age entry fields will be unnecessary because the view-model variables to which they are bound will not affect any command (here, there is only the “book” command); they are *irrelevant*. In response, the UI may choose to visually indicate their disablement, e.g., by hiding or shading them.

There are two reasons why editing the value of some variable v could affect a variable o : (1) there exists a currently active functional dependency from v to o , or (2) editing v could create a functional dependency from v to o .

Again, the current functional dependencies are represented by the evaluation graph, and the solution and constraint graphs can be analyzed to predict changes to the functional dependencies triggered by an edit of a variable. Rephrasing the rationale above in terms of the graphs, a variable v could affect a variable o if (1) v reaches o in the evaluation graph, or if (2) editing v would change the strength assignment in a way that could result in an evaluation graph in which v reaches o . These conditions can be queried for every variable in an ordered constraint system after every user interaction. In this way, widget enablement can be completely

automated, with no additional effort from application programmers.

The most precise algorithm to determine these conditions is to, in turn, (1) give each variable the highest priority; (2) generate a new solution graph; (3) generate a new evaluation graph with the edges present in both the old evaluation graph and the new solution graph, and the edges present in the new solution graph that are not present in the old solution graph; and (4) determine if v reaches o . This option can be expensive to compute. A generic, close approximation follows:

A value of a variable v *cannot* affect another variable o if for every variable w that is (1) an ancestor of v in the solution graph, and (2) reachable from v in the constraint graph, then (3) w does not reach o in the evaluation graph.

To see why this is correct, note that the solution graph is a directed acyclic graph. Each variable v in the solution graph sits at the root of an *ancestor (directed acyclic) graph* consisting of every variable reachable from v (including v itself) in the transpose of the solution graph. The *ancestors* of v are the variables in its ancestor graph. Discussion of methods and constraints in the ancestor graph refers to the methods, and their constraints, in the solution graph that are connected only to members of the ancestor graph.

The ancestor graph of v partitions the variables of the solution graph: partition A consists of the ancestor graph, partition B is everything else. There may exist paths from A to B , but there can be no path from B to A : because every variable in A can reach v , any variable in B with a path to A would be an ancestor of v and thus belong to A .

Any ancestor reachable from v in the *constraint graph* is called a *reachable ancestor*. If v is edited (and thus given highest priority), then the only constraints

that could have different methods selected in the new solution graph must be in v 's ancestor graph. This implies the following: in an evaluation graph, v can reach some set of variables; if v is edited, it could reach a superset of those variables, and the difference must come from its reachable ancestors. In other words, if none of its reachable ancestors reaches a particular variable, then there is no way that editing v can make it reach that variable.

5.8 Summary

The above sections discuss several rich GUI features that are tedious to implement. With an ordered constraint system, though, there exists a generic, reusable algorithm for each of them. These algorithms are possible because an ordered constraint system *explicitly* models the dependencies within a UI, unlike conventional event handlers, and because we have devised formalisms for the behaviors. With reusable implementations, programmers can greatly reduce the cost of building high quality user interfaces.

6. IMPLEMENTATION: HOTDRINK*

HotDrink [22, 24] is an open-source JavaScript library for constructing web UIs based on ordered constraint systems and the MVVM pattern. For UI programmers, it provides an embedded domain-specific language for incrementally building the variables, constraints, and methods in an ordered constraint system, as well as facilities for binding (in HTML or JavaScript) view elements to view-model variables.

Rich UI behaviors are implemented as generic algorithms over the three graphs of the constraint system (as described in Section 3.7). Application developers can generally take advantage of them with no additional code, or at most a single statement “switching on” the behavior. Researchers can add new behaviors, and for ones that fit a specific pattern, HotDrink provides a plugin API.

Throughout the code examples below, all aspects of the HotDrink API live in the `hd` namespace.

6.1 Constructing View-Models

Consider implementing with HotDrink the view-model for the image resizing dialog from Figure 2.1 whose constraint system appears in Table 3.1. For each of the (identical) relationships among the initial, absolute, and relative values for each dimension, the programmer needs to define three variables, as in Figure 6.1. In HotDrink, variables hold values. Each variable’s value can be read by calling the variable as a function with no arguments or written by calling it with the new value as the only argument. Variables may be initialized upon definition (with the

* Portions of this chapter are reprinted from John Freeman, Jaakko Järvi, and Gabriel Foust. Hotdrink: a library for web user interfaces. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 80–83, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1129-8. doi: 10.1145/2371401.2371413. URL <http://doi.acm.org/10.1145/2371401.2371413> Copyright 2012 ACM.

```
1 var init = hd.variable(1500);
2 var abs  = hd.variable(1500);
3 var rel  = hd.variable();
```

Figure 6.1: Definition of three HotDrink variables.

```
1 hd.constraint()
2   .method(abs, function () { return init() * rel() / 100; })
3   .method(rel, function () { return 100 * abs() / init(); });
```

Figure 6.2: Definition of a HotDrink constraint among the variables in Figure 6.1.

argument to `hd.variable`) or during the first evaluation.

After defining variables, the programmer may define a constraint among them and incrementally add methods; Figure 6.2 shows how to relate the variables from Figure 6.1. Each method is a function that should compute some outputs (given as the first argument of the `method` function call) from some inputs. Instead of writing the values of output variables directly, a method may opt to return their new values.

The full listing of the HotDrink view-model for the image resizing example appears in Figure 6.3. All of the variables and constraints are grouped within a constructor for a view-model “class” (at least, JavaScript’s approximation of a class). The above constraint is abstracted into a function, `threeway`. Another constraint ties together the relative dimensions and the preserve ratio option. The final call to `hd.command` constructs both a variable and a one-way constraint from a single method definition; the variable returned is the output of the method.

6.2 Adding Generic Behaviors

Generic UI behaviors in HotDrink typically have a graph algorithm component. Algorithms are fed information about incremental changes in the ordered constraint

```

1  var threeway = function(abs, init, rel) {
2    hd.constraint()
3      .method(abs, function () { return init() * rel() / 100; })
4      .method(rel, function () { return 100 * abs() / init(); });
5  };
6
7  var ViewModel = hd.model(function () {
8
9    this.initial_width = hd.variable(2100);
10   this.relative_width = hd.variable(100);
11   this.absolute_width = hd.variable();
12
13   threeway(
14     this.absolute_width,
15     this.initial_width,
16     this.relative_width);
17
18   this.initial_height = hd.variable(1500);
19   this.relative_height = hd.variable(100);
20   this.absolute_height = hd.variable();
21
22   threeway(
23     this.absolute_height,
24     this.initial_height,
25     this.relative_height);
26
27   this.preserve_ratio = hd.variable(true);
28
29   hd.constraint()
30     .method(this.relative_width, function () {
31       if (this.preserve_ratio()) {
32         this.relative_width(this.relative_height());
33       }
34     })
35     .method(this.relative_height, function () {
36       if (this.preserve_ratio()) {
37         this.relative_height(this.relative_width());
38       }
39     });
40
41   this.ok = hd.command(function () {
42     return hd.fn(resize)(
43       this.absolute_width(), this.absolute_height());
44   });
45
46 });

```

Figure 6.3: Implementation in HotDrink of the view-model for the image resize dialog from Figure 2.1.

system: variables with changed priorities and/or values, and methods that have been executed since the last evaluation. They produce output in the form of markers on variables, and events associated with changes in those markers. View elements can react to or ignore such events as appropriate. In most cases, HotDrink provides a sensible default reaction for view elements as part of the bindings.

For example, the widget enablement behavior tags each variable according to whether it can reach a command variable in either the current evaluation graph or in the solution graph resulting from their edit (see Section 5.7 for details). HotDrink bindings for input widgets will watch for changes to that tag and enable or disable their widgets as appropriate. In our prototype implementation, pinning was implemented in the same way. When the user pinned a variable, it would directly set a flag on that variable. Then, the pinning behavior would identify which variables could not be pinned or edited without overconstraining the system, and mark them so. Any input widgets bound to such variables, in addition to their pin markers, would all be disabled by their bindings.

Behaviors are not limited to this pattern, however. In the example above, after constructing the view-model, no special code was needed to enable value propagation or dataflow visualization. Some behaviors, however, provide methods for creating special variables that are to be mixed in the view-model definition; still others must be “switched on” as they impose some run-time cost that isn’t suitable for every UI.

Consider the hotel booking example, whose view-model is given in Figure 6.4. The **precondition** function creates a special Boolean variable, a one-way constraint that outputs it with the method given as the second argument, and connects it to the command given as the first argument. Bindings created by the library know how to check the command’s activation status and reflect it in the view as appropriate. Further, the enablement behavior described in Section 5.7 must be enabled itself

after the view-model definition.

It is important to note that, other than corresponding HTML descriptions of the views, these JavaScript listings are complete implementations of the example user interfaces. Further, they have no trace of code managing the details of command activation or widget enablement; such features are provided by the framework.


```

1  var ViewModel = hd.model(function () {
2
3    this.checkin = hd.variable(new Date("25_March_2013"));
4    this.checkout = hd.variable(new Date("28_March_2013"));
5    this.nights = hd.variable();
6
7    hd.constraint()
8      .method(this.nights, function () {
9        return (this.checkout() - this.checkin()) / MSIN_DAY;
10     })
11     .method(this.checkin, function () {
12       return new Date(
13         this.checkout().getTime() - (this.nights() * MSIN_DAY));
14     })
15     .method(this.checkout, function () {
16       return new Date(
17         this.checkin().getTime() + (this.nights() * MSIN_DAY));
18     });
19
20    this.nkids = hd.variable(0);
21    this.ages = [
22      hd.variable(0),
23      hd.variable(0),
24      hd.variable(0)
25    ];
26
27    this.book = hd.command(function book() {
28      return hd.fn(book)(this.checkin(), this.checkout(),
29        hd.toJS(this.ages.slice(0, this.nkids())));
30    });
31
32    hd.precondition(this.book, function () {
33      return this.nights() > 0;
34    });
35
36    hd.precondition(this.book, function () {
37      return this.checkin() >= today();
38    });
39
40  });
41
42  ViewModel.behaviors(hd.enablement);

```

Figure 6.4: Implementation in HotDrink of the view-model for the hotel booking dialog from Figure 2.2.

7. EXPERIMENTS*

The value in reusable user interface behaviors is measured by how much easier application development comes, primarily in terms of programmer time and defect count for a given level of functionality. To measure results, we compared several GUIs written with ordered constraint systems to their counterparts built with conventional approaches, both qualitatively and quantitatively (where possible).

7.1 ApplyTexas

As described in the introduction (§ 1.1), the common application for Texas high school students matriculating to college, known as ApplyTexas, has a section where applicants can list their extracurricular activities, up to ten with 23 individual fields for each, in order of importance. The form has a fixed structure, meaning it cannot expand or reorder to accommodate users, and it uses no JavaScript, so the HTML for each activity is duplicated. It is difficult for users to use, and it was tedious for the author to write.

When written with HotDrink, a direct comparison is not useful for demonstrating the exclusive benefits of HotDrink. The form was originally written in such a primitive fashion that the form's author could have seen large benefits from using any library for writing dynamic forms, not just HotDrink. However, it may still be

* Portions of this chapter are reprinted from Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Property models: from incidental algorithms to reusable components. In *GPCE'08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 89–98, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-267-2. doi: 10.1145/1449913.1449927. URL <http://doi.acm.org/10.1145/1449913.1449927> Copyright 2008 ACM; Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Algorithms for user interfaces. In *GPCE'09: Proceedings of the 8th international conference on Generative programming and component engineering*, pages 147–156, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-267-2. doi: 10.1145/1621607.1621630. URL <http://doi.acm.org/10.1145/1621607.1621630> Copyright 2009 ACM.

useful to offer an assessment of writing part of a real-world application in HotDrink.

The data model behind ApplyTexas is very simple. There are no obvious relationships among the values. Still, writing it with HotDrink allowed me to easily implement missing features that aided users, e.g. inserting, removing, and reordering activities. For example, after I had recreated the original form, adding buttons to promote and demote activities required adding two one-liner JavaScript methods and two one-liner HTML buttons. HotDrink was able to automatically take care of renumbering the activities after reordering.

ApplyTexas is a good example of the poor economics of user interfaces. The frustrations of dealing with a list that cannot expand or reorder are small for any one user, but add up to large amounts of wasted time that far surpass the development effort required to alleviate them. With its support for automating rich behavior, HotDrink is able to bring the cost of development low enough to justify the work in more cases than before.

7.2 TodoMVC

The TodoMVC project [63] specifies a benchmark application (a simple to-do list, pictured in Figure 7.1) and hosts example implementations of it using various JavaScript MVC frameworks. TodoMVC provides a means to compare the strengths, weaknesses, and styles of those frameworks.

Here, HotDrink is compared with the more popular frameworks featured in TodoMVC, as well as a “vanilla” implementation that uses no framework. Each candidate implements the full TodoMVC application, but without the optional routing feature¹ (which is unsupported in at least HotDrink). Because each candidate was pulled directly from the TodoMVC project, this comparison assumes that it was

¹“Routing” describes the practice of mapping URIs to internal states of the application.



Figure 7.1: TodoMVC, a to-do list web application.

Candidate	HTML		JavaScript			Total Bytes
	Lines	Bytes	Files	Lines	Bytes	
vanilla	37	1163	1	312	7382	8545
HotDrink [22]	56	2297	1	96	2226	4523
Knockout [69]	53	2448	1	123	2990	5438
Angular [33]	61	2609	5	120	2654	5263
Agility [1]	47	1661	2	169	4283	5944
Backbone [5]	55	1905	5	254	4883	6788
Ember [43]	63	2073	7	334	8689	10762

Table 7.1: Code statistics for TodoMVC implementations. Includes measurements for lines and bytes of HTML code; files, lines, and bytes of JavaScript code; and total bytes of code. Each implementation had a single HTML file.

written to follow its framework’s recommended best practices, and that it represents the most common design decisions among developers working with its framework.

Each candidate features a single HTML file and one or more JavaScript files (excluding imported libraries). The number of lines and bytes in these files is charted in Table 7.1. None of these measures can precisely capture qualitative metrics like “programmer effort expended” or “readability”, but together they may lend an approximate sense of the landscape. The numbers of bytes among the files were included because, while each candidate follows the coding style of its framework, the frameworks do not all share the same line length conventions, and thus shorter line counts may belie the extra effort required.

HotDrink allowed the shortest implementation among all candidates. It was middle of the pack for lines and bytes of HTML, but had 20% fewer lines and 16% fewer bytes of JavaScript than the next shortest competitor, Angular. It also tied for the fewest JavaScript files, at one.

In each candidate’s JavaScript files, there may appear some number of functions, categorized in the following list, along with shorthand names:

Candidate	Total	CTOR	MTHD	DECL	EVNT	LMBD	FREE
vanilla	28	2			9		17
HotDrink	18	2	5	8		3	
Knockout	24	2	9	5	3	4	1
Angular	20	1	8	2	4	2	3
Agility	20		2		13	3	2
Backbone	27		7		17	3	

Table 7.2: Counts of categories of JavaScript functions appearing in TodoMVC implementations.

CTOR Class constructors.

MTHD Methods, in the object-oriented sense.

DECL Methods, in the constraint system sense.

EVNT Event handlers.

LMBD Arguments to higher-order functions, e.g. map, fold, filter.

FREE Free functions.

The counts in each of these function categories for the TodoMVC frameworks examined appear in Table 7.2. The key comparisons are between imperative event handlers, a common source of software complexity, and declarative constraints, a common approach to reduce that complexity (and the one chosen by HotDrink).

Like HotDrink, Knockout and Angular support declarative relationships among values. However, they suffer limitations (e.g. allowing only one-way constraints), and thus still require event handlers for some functionality. HotDrink is the only framework seen that can successfully eliminate the use of all event handlers through constraint based programming.

	Time	Servings	Description	Fat (g)	Carb (g)	Prot (g)	Cals	
<input type="checkbox"/>			Total	46.3	83.5	94.2	1127	
<input type="checkbox"/>	breakfast	2	slice of plain french toast	12.3	40.0	11.2	317	<input type="checkbox"/>
<input type="checkbox"/>	lunch	2	serving of chicken	13.0	2.0	64.0	380	<input type="checkbox"/>
<input type="checkbox"/>	lunch	1	serving of tomato salsa	0.0	4.0	1.0	20	<input type="checkbox"/>
<input checked="" type="checkbox"/>	lunch	1	serving of cheese	8.5	0.0	8.0	100	<input type="checkbox"/>
<input type="checkbox"/>	lunch	1	serving of black beans	1.0	23.0	7.0	120	<input type="checkbox"/>
<input type="checkbox"/>	lunch	.5	serving of rice	1.5	11.5	1.0	65	<input type="checkbox"/>
<input checked="" type="checkbox"/>	lunch	1	serving of sour cream	10.0	2.0	2.0	120	<input type="checkbox"/>
<input type="checkbox"/>	lunch	1	serving of lettuce	0.0	1.0	0.0	5	<input type="checkbox"/>

OR changes to the diary.

selected entries.

Figure 7.2: The Better Meal, a web user interface built with HotDrink for a popular online diet tracker. This screenshot features the daily log.

7.3 The Better Meal

As a testament to its real-world suitability, HotDrink is used in at least one web application. Called The Better Meal, it is a user interface for a popular online diet tracker. The most extensive use of HotDrink in The Better Meal is for the daily log, as seen in Figure 7.2. It consists of a table of food entries and a few commands.

Each entry is a food item with mutable fields for meal time, number of servings, and description, as well as immutable fields for chosen nutrients (fat, carbohydrates, protein, and calories). Each nutrient on a food item exists in a one-to-one relationship with its number of servings, and each nutrient group altogether exists in a many-to-one relationship with the total for that nutrient, displayed in the top row of the table.

Each food item tracks whether its mutable fields have changed since the last page load or save, using a custom behavior. That status is displayed in a narrow colored bar between each item’s checkbox and its meal time dropdown; green indicates “saved”, and yellow indicates “unsaved”. The “Save” and “reset” commands below

the table are enabled only whenever the table has any unsaved changes. The “Save” command commits any waiting changes while the “reset” command reverts them.

Food items can be selected individually for deletion. Those selections exist in a many-to-one relationship with a “select all” checkbox in the top row. The “Delete” command below the table becomes enabled when any items are selected and, when clicked, removes from the table not just those items, but also their internal relationships and their participation in external relationships. The nutrient totals, “select all” checkbox, and enablement of each command automatically adjust to match the new status.

A notable aspect of creating The Better Meal was that the developer did not need to worry about cleaning up the relationships for removed items. They could instead focus on correctly implementing the usual dataflow: confirm that the item has been removed in the remote database, and then reflect that in the user interface by simply removing the item from a list.

7.4 Adobe

Our collaborators at Adobe deployed a C++ library implementing a similar approach to that described in this thesis. It incorporated domain-specific languages for building constraint systems (Adam) and for laying out and binding view elements (Eve). Teams using it to reform user interface dialogs within Adobe software saw improvements in code size, programmer productivity, and defect rates.

In one application, the event handling and scripting code for a single dialog, which accounted for 781 statements and contained five known logic defects, was replaced by an Adam specification with 46 statements and no reported defects [39]. Generally, Adobe has seen reductions in code size of a factor of 8–10 and improved quality.

One experiment observed four teams of roughly three engineers each, balanced

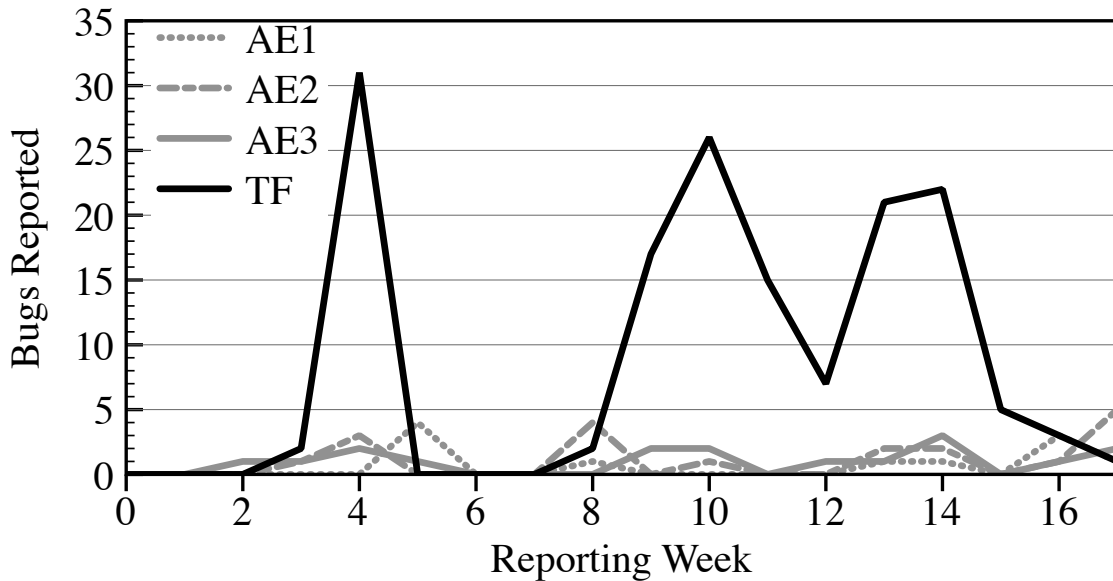


Figure 7.3: The number of reported bugs during several months for three Adam engine teams (AE1–AE3) and a traditional framework team (TF). The teams were tasked with rewriting code for a large number of dialogs of varying complexity. The teams each consisted of roughly three engineers, balanced in programming skills.

in programming skill as judged by their managers [40]. Each team was tasked with rewriting dialogs and palettes (varying in complexity but distributed evenly among the teams) for an application that formerly depended on a 32-bit-only user interface framework. Three of the teams used the new Adam library while the fourth used a modern, vendor-supplied, 64-bit object-oriented user interface framework.

Figure 7.3 compares the number of reported bugs over several months for the three Adam-Eve teams (AE1–AE3) and the traditional framework team (TF). It was rare for more than 2–3 defects per week to be found for a given Adam team, while a conservative estimate for the defect rate for the TF team was 10–20 defects per week. That is, the teams working in the Adam architecture produced defects at less than twenty percent of the rate of that of the traditional framework team.

The commonly high rates of defects among user interfaces reinforces the impor-

tance of these results. In a sample of defects in a 20,000-bug database of a major Adobe desktop application, roughly half were in the user interface layer that this research targets. Reviewing bugs across a range of Adobe products revealed that roughly 40% of Adobe products' bugs are "behavioral" in nature. Half of those were such that they could not even have existed if the Adam library was employed.

As with any prototype system, there were challenges in adopting the Adam library. Ordered constraint systems offer a new way of approaching user interface programming, and there is a learning curve. The supporting tool ecosystem, though more powerful than traditional user interface builders, is not as mature as with established frameworks. Notwithstanding these limitations, in the above study, Adam programmers were substantially more productive than their counterparts on the TF team. In the time period studied, the three AE teams combined completed roughly 75 dialogs and palettes, with another 50 or so underway. The TF team completed fewer than 10 altogether. The product team did not believe they would be able to succeed in porting to 64-bits in a single release without the Adam library.

8. RELATED WORK*

The basic architecture of a HotDrink user interface is based on the Model-View-Controller pattern [45]. This pattern is identified as being important for the separation of concerns in user interfaces in a recent work [32] aimed at untangling application logic from user interfaces.

Declarative layout specifications mixed with procedural behavior specifications are common in the area of user interfaces. Examples include the QtK module [35] in Mozart/Oz, Glade [17], XUL [15, 56], XAML [81], and XForms [12]. Many more, like HotDrink [22], are based on a combination of HTML, CSS, and JavaScript. Popular frameworks include Angular [33], Backbone [5] (sometimes enhanced with Marionette [8]), Ember [43], Agility [1], and Meteor [51].

All of these systems require user interface authors to write event handlers to define user interface behavior. HotDrink eschews event handlers in favor of declarative relationships, which provides a few advantages. With event handlers, authors must specify when any given method is executed by attaching handlers to the correct events. With HotDrink, authors need to define *how* to compute variables from each other, but not *when*.

* Portions of this chapter are reprinted from Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Property models: from incidental algorithms to reusable components. In *GPCE'08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 89–98, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-267-2. doi: 10.1145/1449913.1449927. URL <http://doi.acm.org/10.1145/1449913.1449927> Copyright 2008 ACM; Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Algorithms for user interfaces. In *GPCE'09: Proceedings of the 8th international conference on Generative programming and component engineering*, pages 147–156, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-267-2. doi: 10.1145/1621607.1621630. URL <http://doi.acm.org/10.1145/1621607.1621630> Copyright 2009 ACM; John Freeman, Jaakko Järvi, Wonseok Kim, Mat Marcus, and Sean Parent. Helping programmers help users. In *GPCE'11: Proceedings of the 10th ACM international conference on Generative programming and component engineering*, pages 177–184, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0689-8. doi: 10.1145/2047862.2047892. URL <http://doi.acm.org/10.1145/2047862.2047892> Copyright 2011 ACM.

In most of these systems, the effects of event handlers do not trigger other event handlers, in order to guarantee termination. Consequently, each event handler must wholly contain every possible dataflow that can originate with its corresponding event. Angular is one notable exception, where change events are fired in waves until a fixed point in the user interface state is reached. The number of waves is limited in order to guarantee termination, restricting the possible length of dataflows. HotDrink allows authors to think locally, within the confines of individual constraints, while placing no limits on the length of a dataflow.

Capturing user intent (i.e. preserving the values of more recently edited variables) entails more state (e.g. user editing history) shared among the event handlers, as well as branching within each dataflow that grows deeper as the flow grows longer. That complexity may be why no event handler framework even discusses capturing user intent. With HotDrink, user intent is captured by the semantics of an ordered constraint system.

Knockout [69] lets authors define “computed” (dependent) variables whose values are calculated from other variables, and “writable” computed variables effectively implement two-way constraints where one method has exactly one input and the other has exactly one output. HotDrink supports generalized multi-way constraints.

Angular’s “watchers” allow users to implement multi-way constraints, but doing so will cause at least two methods to execute whenever a user edits a variable: one whose inputs include the variable the user edited, and one whose inputs include the changed output(s) of that method. HotDrink efficiently evaluates the solution graph so that the minimum number of methods are executed. Further, Angular detects changes in variables by comparing the current value of a variable with its last known value, a potentially expensive operation, whether the variable was assigned or not. HotDrink instead compares values only when a variable is assigned in a method.

Rule-based systems such as Drools [68], Jess [29], and R++ [49], also support the specification of rules for maintaining consistency across values in user interfaces. These systems, like HotDrink, offer the ability to express certain relationships concisely. While they are expressive, the active dependencies are not explicitly represented in code or exposed in the runtime, unlike in the three graphs we use.

The above rule-based systems do not restrict the expressive power of the language specifying the rules. For example, XForms supports declarative (one-way) constraints, but also provides open-ended support for user scripts (in Javascript) to be bound to events that can, in turn, read and write arbitrary variables in the model. In addition to the dependencies between variables arising from the constraints, the scripts can hide arbitrary dependencies, leading to incidental data structures that cannot be analyzed. HotDrink controls the mutability of its variables and carefully tracks all of the dependencies in the functions users write.

HotDrink goes further than these systems by providing an explicit model of the dependencies among variables that enables generic algorithms for rich user interface behaviors like scripting, enablement, and automatic help generation.

Constraint systems have been studied extensively for use in user interfaces, mostly for automated element layout, but also for maintaining consistency across data in user interface elements, as in command parameter synthesis. HotDrink draws from this line of work; in particular, we use the algorithms and representations for hierarchical multi-way dataflow constraint systems [79].

A large number of declarative, constraint-based GUI systems have been proposed, for example, Sketchpad [76], Amulet [61], Garnet [57], ThingLab I [9, 10] and ThingLab II [54], DeltaBlue [26, 28], and SkyBlue [70]. A survey of model-based design environments for user interface construction can be found in [66]. More recent active projects that support (one-way) data-flow constraints include the OpenLaszlo

framework [47] for developing rich Internet applications. Constraints in these systems are mainly used for layout where the simpler one-way constraints are rather standard, e.g., in many diagram drawing tools [80]. Based on extensive experience with the Amulet system, its authors conclude that it is unlikely that constraint systems will ever be used for much other than layout [83]. Our experience indicates a more positive picture. Amulet and the related systems integrate a constraint solver into a general purpose programming language, but the state of the network of functional dependencies is hidden from the programmer, whereas it is explicitly modeled and made accessible by HotDrink. This, we believe, is why we can benefit from the constraint system formalisms and algorithms, and apply them in an area where previously their success has been limited.

Regarding enablement logic, the Jade interactive dialog creation tool for Garnet [82], and similar work by Myers et al. [59] and Frank et al. [21], target the expression of enablement logic using a constraint system. These works do not, however, attempt to devise a generic enablement algorithm. Further, apart from relieving the programmer from coding explicit enablement and activation logic, our analysis clearly defines the reasons why a widget should be enabled/activated or disabled/deactivated. To improve usability, a user interface could be instrumented to show these reasons to its user, as described in Section 5.6.

UIDE [74, 31] and HUMANOID [55] use preconditions on commands to disable widgets and to generate helpful explanations. UIDE also uses postconditions to explain how to enable a command widget [73, 14, 75]. The authors acknowledge that their system is not prepared to handle situations with complex, multi-way dependencies among actions and widgets; such dependencies are supported in an ordered constraint system. Unlike UIDE, our generated help text does not attempt to provide a precise sequence of interactions for re-activating a command widget (in some

cases, there may be many sequences to choose from), and we have not investigated its (in)feasibility. Further, we try not to burden the programmer with specifying postconditions for user interactions. We believe that the information available in an ordered constraint system is enough to provide sufficient help.

The Heracles system supported pinning of its controls [44]. In Heracles, not all constraints are enforced after each user edit, and multi-way constraints are not supported. Thus, some of the issues facing pinning in ordered constraint systems are avoided. Heracles also opts to pin a variable automatically upon user edit, whereas we pin a variable upon explicit request only. The authors of Heracles, like us, cite potentially unclear dataflow and overwritten values as motivations for pinning.

9. CONCLUSION

Ask nearly any software engineer what they most dislike in their work and the answer will most likely be “building the user interface.” Even working on products where an automatic layout library frees the engineer from mundane tasks such as placing a button at just the right pixel location, the effort to build a user interface is onerous. For many applications, the user interface accounts for a significant portion of the code and a majority of the defects.

A common shortcoming in modern user interfaces is that much of the functionality is bespoke to each application. Increased reuse in the domain of user interfaces can thus notably improve programmer productivity, improve software quality, and free programmers to work on more rewarding tasks. With higher quality user interfaces, our daily encounters with computer systems can become more productive, more pleasant, and less frustrating.

Modern GUIs are typically implemented as a web of shared state and event handlers, giving rise to incidental data structures and algorithms. Some more advanced frameworks leverage simple single-way constraint systems (losing any distinction between constraint and solution graphs), but do not track which dependency edges are used (losing any distinction between solution and evaluation graphs). Further, many common user interface behaviors are ad hoc implementations derived from published human interface guidelines, different among each platform and open to interpretation. As an alternative, an ordered constraint system provides an explicit model of the values and relationships in a user interface in addition to its interaction history. Such an explicit model enables the development of generic algorithms for behaviors based on well-defined formal specifications; this thesis presents examples for value

propagation, dataflow visualization, pinning, scripting, command activation, widget enablement, and context-sensitive help.

Reusable behaviors can change the economics of user interface construction: developers no longer need to decide whether it is worth the effort to implement advanced features when they come built-in. “Nice-to-have” features then become “routine” (and maybe eventually “essential”). In fact, the ordered constraint system approach has the potential to transform the entire GUI construction process by simplifying it to a lower class of problem. A constraint system allows creators to think locally by defining relationships among small subsets of variables without worrying about their interaction with other segments of the system. At that level of abstraction, mockups from GUI designers may no longer need to be implemented by programmers, but instead become implementations themselves.

Experiments with ordered constraint systems, both in the lab and in industry, confirm the approach leads to increased programmer productivity, less code, and fewer defects when compared to the employment of more traditional graphical user interface frameworks.

REFERENCES

- [1] Artur Adib et al. Agility, March 2015. URL <http://agilityjs.com>.
- [2] Apple, Inc. *Cocoa Application Tutorial*, October 2007. URL <http://developer.apple.com/documentation/Cocoa/Conceptual/ObjCTutorial/>.
- [3] Apple, Inc. *Apple Human Interface Guidelines: User Experience*. 1 Infinite Loop, Cupertino, CA 95014, June 2008.
- [4] Apple, Inc. Apple developer connection: Cocoa, March 2009. URL <http://developer.apple.com/cocoa/>.
- [5] Jeremy Ashkenas et al. Backbone, March 2015. URL <http://backbonejs.org>.
- [6] Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo. How reuse influences productivity in object-oriented systems. *Commun. ACM*, 39(10):104–116, 1996.
- [7] Calumn Benson, Adam Elman, Seth Nickel, and Colin Z. Robertson. *GNOME Human Interface Guidelines 2.2*, March 2008. URL <http://library.gnome.org/devel/hig-book/stable/index-info.html.en>.
- [8] Peter Blazejewicz, Sam Saccone, Joanne Daudier, et al. Marionette, March 2015. URL <http://marionettejs.com>.
- [9] Alan Borning. Thinglab: an object-oriented system for building simulations using constraints. In *Proceedings of the 5th international joint conference on Artificial intelligence-Volume 1*, pages 497–498. Morgan Kaufmann Publishers Inc., 1977.

- [10] Alan Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *ACM Trans. Program. Lang. Syst.*, 3(4):353–387, October 1981. ISSN 0164-0925. doi: 10.1145/357146.357147. URL <http://doi.acm.org/10.1145/357146.357147>.
- [11] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. *SIGPLAN Not.*, 22(12):48–60, 1987.
- [12] John M. Boyer, Micah Dubinko, Jr. Leigh L. Klotz, David Landwehr, Roland Merrick, and T. V. Raman. XForms 1.0 (Third Edition), October 2007. URL <http://www.w3.org/TR/2007/REC-xforms-20071029/>.
- [13] Irina Ceaparu, Jonathan Lazar, Katie Bessiere, John Robinson, and Ben Shneiderman. Determining causes and severity of end-user frustration. *International Journal of Human-Computer Interaction*, 17(3):333–356, 2004. doi: 10.1207/s15327590ijhc1703_3. URL http://www.tandfonline.com/doi/abs/10.1207/s15327590ijhc1703_3.
- [14] Johannes J. de Graaff, Piyawadee Sukaviriya, and Charles van der Mast. Automatic generation of context-sensitive textual help. Technical Report GIT-GVU-93-11, Georgia Institute of Technology, April 1993.
- [15] Neil Deakin. XUL tutorial. Webpage, February 2006. URL <http://www.xulplanet.com/tutorials/xultu/>.
- [16] Martin Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, September 2001. ISSN 0956-7968. URL <http://dx.doi.org/10.1017/S0956796801004075>.
- [17] E. Feldman. Create user interfaces with Glade. *Linux Journal*, 2001(87):4, 2001.

- [18] Martin Fowler. Presentation Model Pattern, 2004. URL <http://martinfowler.com/eaDev/PresentationModel.html>.
- [19] Martin Fowler. Passive View, 2006. URL <http://martinfowler.com/eaDev/PassiveScreen.html>.
- [20] William B. Frakes and Giancarlo Succi. An industrial study of reuse, quality, and productivity. *Journal of Systems and Software*, 57(2):99–106, June 2001.
- [21] Martin R. Frank, J. J. de Graaff, Daniel F. Gieskens, and James D. Foley. Building user interfaces interactively using pre- and postconditions. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 641–642, New York, NY, USA, 1992. ACM. ISBN 0-89791-513-5. doi: 10.1145/142750.143063. URL <http://doi.acm.org/10.1145/142750.143063>.
- [22] John Freeman and Gabriel Foust. HotDrink, an open source JavaScript GUI framework, June 2012. URL <https://github.com/HotDrink>.
- [23] John Freeman, Jaakko Järvi, Wonseok Kim, Mat Marcus, and Sean Parent. Helping programmers help users. In *GPCE'11: Proceedings of the 10th ACM international conference on Generative programming and component engineering*, pages 177–184, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0689-8. doi: 10.1145/2047862.2047892. URL <http://doi.acm.org/10.1145/2047862.2047892>.
- [24] John Freeman, Jaakko Järvi, and Gabriel Foust. Hotdrink: a library for web user interfaces. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 80–83, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1129-8. doi: 10.1145/2371401.2371413. URL <http://doi.acm.org/10.1145/2371401.2371413>.

- [25] B. N. Freeman-Benson. A module mechanism for constraints in smalltalk. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '89, pages 389–396, New York, NY, USA, 1989. ACM. ISBN 0-89791-333-7. doi: 10.1145/74877.74918. URL <http://doi.acm.org/10.1145/74877.74918>.
- [26] Bjorn N. Freeman-Benson and John Maloney. The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver. *Computers and Communications, 1989. Conference Proceedings., Eighth Annual International Phoenix Conference on*, pages 538–542, March 1989.
- [27] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Commun. ACM*, 33(1):54–63, 1990.
- [28] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Commun. ACM*, 33(1):54–63, 1990.
- [29] Ernest Friedman-Hill. Jess 7, February 2008. URL <http://www.jessrules.com/jess/charlemagne.shtml>.
- [30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.
- [31] Daniel F. Gieskens and James D. Foley. Controlling user interface objects through pre- and postconditions. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '92, pages 189–194, New York, NY, USA, 1992. ACM. ISBN 0-89791-513-5. doi: 10.1145/142750.142787. URL <http://doi.acm.org/10.1145/142750.142787>.

- [32] Sofie Goderis, Dirk Deridder, Ellen Van Paesschen, and Theo D'Hondt. DEUCE: A declarative framework for extricating user interface concerns. *Journal of Object Technology*, 6(9):87–104, October 2007. URL http://www.jot.fm/issues/issue_2007_10/paper5/. Special Issue: TOOLS EUROPE 2007.
- [33] Google, Inc. Angular, March 2015. URL <https://angularjs.org>.
- [34] John Gossman. Introduction to Model/View/ViewModel pattern for building WPF apps, October 2005. URL <http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx>.
- [35] Donatien Grolaux and Peter Van Roy. QtK — an integrated model-based approach to designing executable user interfaces. In *8th Workshop on Design, Specification, and Verification of Interactive Systems (DSVIS 2001)*, Lecture Notes in Computer Science, Glasgow, Scotland, June 2001. Springer-Verlag.
- [36] Scott Hanselman. Everything's broken and nobody's upset, September 2012. URL <http://www.hanselman.com/blog/EverythingsBrokenAndNobodysUpset.aspx>.
- [37] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. *Human-Computer Interaction*, 1:311–338, December 1985. ISSN 0737-0024.
- [38] Geoffrey James. *The Tao of Programming*. Info Books, September 1986. ISBN 9780931137075. URL <http://amazon.com/o/ASIN/0931137071/>.
- [39] Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Property models: from incidental algorithms to reusable components. In

- GPCE'08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 89–98, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-267-2. doi: 10.1145/1449913.1449927. URL <http://doi.acm.org/10.1145/1449913.1449927>.
- [40] Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Algorithms for user interfaces. In *GPCE'09: Proceedings of the 8th international conference on Generative programming and component engineering*, pages 147–156, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-267-2. doi: 10.1145/1621607.1621630. URL <http://doi.acm.org/10.1145/1621607.1621630>.
- [41] Jaakko Järvi, Magne Haverlaen, John Freeman, and Mat Marcus. Expressing multi-way data-flow constraint systems as a commutative monoid makes many of their properties obvious. In *Proceedings of the 8th ACM SIGPLAN workshop on Generic programming, WGP '12*, pages 25–32, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1576-0. doi: 10.1145/2364394.2364399. URL <http://doi.acm.org/10.1145/2364394.2364399>.
- [42] Jim Johnson. CHAOS: the dollar drain of IT project failures. *Application Development Trends*, pages 41–47, January 1995.
- [43] Yehuda Katz, Tom Dale, et al. Ember, March 2015. URL <http://emberjs.com>.
- [44] Craig A. Knoblock, Steven Minton, José Luis Ambite, Maria Muslea, Jean Oh, and Martin Frank. Mixed-initiative, multi-source information assistants. In *Proceedings of the 10th international conference on World Wide Web, WWW '01*, pages 697–707, New York, NY, USA, 2001. ACM. ISBN 1-58113-348-0. doi: 10.1145/371920.372185. URL <http://doi.acm.org/10.1145/371920.372185>.

- [45] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.
- [46] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.
- [47] Laszlo Systems. OpenLaszlo: a rich internet application development framework, Accessed May 2012. URL <http://www.openlaszlo.org>.
- [48] Jonathan Lazar, Adam Jones, Mary Hackley, and Ben Shneiderman. Severity and impact of computer user frustration: A comparison of student and workplace users. *Interact. Comput.*, 18(2):187–207, March 2006. ISSN 0953-5438. doi: 10.1016/j.intcom.2005.06.001. URL <http://dx.doi.org/10.1016/j.intcom.2005.06.001>.
- [49] D. Litman, P. F. Patel-Schneider, A. Mishra, J. Crawford, and D. Dvorak. R++: Adding path-based rules to C++. *IEEE Trans. on Knowl. and Data Eng.*, 14(3):638–658, 2002.
- [50] Ferren MacIntyre, Kenneth W. Estep, and John M. Sieburth. The cost of user-friendly programming: Macimage as example. *J. FORTH Appl. Res.*, 6(2): 103–115, June 1990. ISSN 0738-2022. URL <http://dl.acm.org/citation.cfm?id=83218.83217>.
- [51] Meteor Development Group. Meteor, March 2015. URL <https://www.meteor.com>.
- [52] Microsoft Corporation. *Windows Vista UX Guide: User Experience*

- Guidelines*, 2008. URL <http://download.microsoft.com/download/e/1/9/e191fd8c-bce8-4dba-a9d5-2d4e3f3ec1d3/uxguide.pdf>.
- [53] Microsoft, Inc. System.Windows.Forms, March 2009. URL <http://msdn.microsoft.com/en-us/library/system.windows.forms.aspx>.
- [54] J. Moloney, A. Borning, and B. Freeman-Benson. Constraint technology for user-interface construction in thinglab ii. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 381–388, New York, NY, USA, 1989. ACM. ISBN 0-89791-333-7. doi: 10.1145/74877.74917. URL <http://doi.acm.org/10.1145/74877.74917>.
- [55] Roberto Moriyon, Pedro Szekely, and Robert Neches. Automatic generation of help from interface design models. In *Proceedings of the SIGCHI conference on Human factors in computing systems: celebrating interdependence*, CHI '94, pages 225–231, New York, NY, USA, 1994. ACM. ISBN 0-89791-650-6. doi: 10.1145/191666.191751. URL <http://doi.acm.org/10.1145/191666.191751>.
- [56] Mozilla. XML user interface language (XUL) 1.0. Mozilla Foundation, March 2006. URL <http://www.mozilla.org/projects/xul/xul.html>.
- [57] B.A. Myers, D.A. Giuse, R.B. Dannenberg, B.V. Zanden, D.S. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85, November 1990.
- [58] Brad A. Myers. Why are human-computer interfaces difficult to design and implement? Technical Report CMU-CS-93-183, Carnegie Mellon University Computer Science Department, July 1993.
- [59] Brad A. Myers and David S. Kosbie. Reusable hierarchical command objects. In

- CHI '96: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 260–267, New York, NY, USA, 1996. ACM. ISBN 0-89791-777-4. doi: 10.1145/238386.238526. URL <http://doi.acm.org/10.1145/238386.238526>.
- [60] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, pages 195–202, New York, NY, USA, 1992. ACM. ISBN 0-89791-513-5. doi: 10.1145/142750.142789. URL <http://doi.acm.org/10.1145/142750.142789>.
- [61] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrenco, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The Amulet environment: New models for effective user interface software development. *Software Engineering*, 23(6):347–365, 1997. URL <http://citeseer.ist.psu.edu/article/myers96amulet.html>.
- [62] Derek L. Nazareth and Marcus A. Rothenberger. Assessing the cost-effectiveness of software reuse: A model for planned reuse. *Journal of Systems and Software*, 73(2):245–255, October 2004.
- [63] Addy Osmani et al. TodoMVC: A common learning application for popular JavaScript MV* frameworks, 2012. URL <http://addyosmani.github.com/todomvc/>.
- [64] Sean Parent. A possible future for software development. Keynote talk at the Workshop of Library-Centric Software Design 2006, at OOPSLA'06, Portland, Oregon, 2006. URL lcsd.cs.tamu.edu/2006.
- [65] Sean Parent. Science of ‘shrink wrap’: A look inside adobe photoshop,

2008. URL http://stlab.adobe.com/wiki/images/c/c9/2008_01_18_indiana_shrink_wrap.pdf.
- [66] Paulo Pinheiro da Silva. User interface declarative models and development environments: A survey. *Interactive Systems Design, Specification, and Verification*, pages 207–226, 2001. URL http://dx.doi.org/10.1007/3-540-44675-3_13.
- [67] Mike Potel. MVP: Model-view-presenter: The Taligent programming model for C++ and Java, 1996. URL <http://www.wildcrest.com/Potel/Portfolio/mvp.pdf>.
- [68] Mark Proctor, Michael Neale, Bob McWhirter, Kris Verlaenen, Edson Tirelli, Fernando Meyer, Alexander Bagerman, Michael Frandsen, Geoffrey De Smet, Toni Rikkola, Steven Williams, Ben Truit, Ritu Jain, Chinmay Nagarkar, and Denis Ahearn. Drools, 2008. URL <http://www.jboss.org/drools/>.
- [69] Steven Sanderson et al. Knockout, March 2015. URL <http://knockoutjs.com>.
- [70] Michael Sannella. Skyblue: A multi-way local propagation constraint solver for user interface construction. In *UIST '94: Proceedings of the 7th annual ACM symposium on User Interface Software and Technology*, pages 137–146, New York, NY, USA, 1994. ACM.
- [71] Michael John Sannella. *Constraint satisfaction and debugging for interactive user interfaces*. PhD thesis, University of Washington, Seattle, WA, USA, 1994.
- [72] Adobe Software Technology Lab Sean Parent, Principal Scientist & Manager. Private communication, 2012.
- [73] Piyawadee Sukaviriya and Johannes J. de Graaff. Automatic generation of context-sensitive "show and tell" help. Technical Report GIT-GVU-92-18, Georgia Institute of Technology, July 1992.

- [74] Piyawadee Sukaviriya and James D. Foley. Coupling a UI framework with automatic generation of context-sensitive animated help. In *Proceedings of the 3rd annual ACM SIGGRAPH symposium on User Interface Software and Technology*, UIST '90, pages 152–166, New York, NY, USA, 1990. ACM. ISBN 0-89791-410-4. doi: 10.1145/97924.97942. URL <http://doi.acm.org/10.1145/97924.97942>.
- [75] Piyawadee Noi Sukaviriya, Jeyakumar Muthukumarasamy, Anton Spaans, and Hans J. J. de Graaff. Automatic generation of textual, audio, and animated help in uide: the user interface design. In *Proceedings of the workshop on Advanced visual interfaces*, AVI '94, pages 44–52, New York, NY, USA, 1994. ACM. ISBN 0-89791-733-2. doi: 10.1145/192309.192322. URL <http://doi.acm.org/10.1145/192309.192322>.
- [76] Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. In *DAC '64: Proceedings of the SHARE design automation workshop*, pages 6329–6346, New York, NY, USA, 1964. ACM.
- [77] Troll, Inc. Qt: A cross-platform application and UI framework, March 2009. URL <http://www.qtsoftware.com/products>.
- [78] Gilles Trombettoni and Bertrand Neveu. Computational complexity of multiway, dataflow constraint problems. In *IJCAI: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 358–365, August 1997.
- [79] Brad Vander Zanden. An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Trans. Program. Lang. Syst.*, 18(1):30–72, January 1996. ISSN 0164-0925. doi: 10.1145/225540.225543. URL <http://doi.acm.org/10.1145/225540.225543>.

- [80] Michael Wybrow, Kim Marriott, Linda McIver, and Peter J. Stuckey. Comparing usability of one-way and multi-way constraints for diagram editing. *ACM Trans. Comput.-Hum. Interact.*, 14(4):1–38, 2008. ISSN 1073-0516. doi: <http://doi.acm.org/10.1145/1314683.1314687>.
- [81] XAML. XAML: Extensible application markup language. Microsoft Developer Network (MSDN), 2008. URL <http://msdn.microsoft.com/en-us/library/ms747122.aspx>.
- [82] Brad Vander Zanden and Brad A. Myers. Automatic, look-and-feel independent dialog creation for graphical user interfaces. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 27–34, New York, NY, USA, 1990. ACM. ISBN 0-201-50932-6. doi: 10.1145/97243.97248. URL <http://doi.acm.org/10.1145/97243.97248>.
- [83] Brad Vander Zanden, Richard Halterman, Brad Myers, Rob Miller, Pedro Szekely, Dario Giuse, David Kosbie, and Rich McDaniel. Lessons learned from users experiences with spreadsheet constraints in the garnet and amulet graphical toolkits. May 2002. URL <ftp://cs.utk.edu/pub/TechReports/2002/ut-cs-02-488.pdf>.