

GUARANTEEING RESPONSIVENESS AND CONSISTENCY IN DYNAMIC, ASYNCHRONOUS GRAPHICAL USER INTERFACES

A Dissertation

by

CHARLES GABRIEL FOUST

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,	Gabriel Dos Reis
	Paul Gratz
	Frank Shipman
Head of Department,	Dilma Da Silva

May 2016

Major Subject: Computer Science

Copyright 2016 Charles Gabriel Foust

ABSTRACT

This dissertation proposes a programming model for Graphical User Interfaces (GUIs) that relieves the programmer of a difficult and error-prone task: orchestrating concurrent responses to events to ensure data dependencies are always enforced correctly. In this programming model, rather than defining program responses to events, the programmer defines the data dependencies that exist in the GUI and the methods by which those dependencies may be enforced—a run-time system uses this specification to generate responses to events. The approach gives the following guarantee: the same sequence of events produces the same results, regardless of the timing of those events. The dissertation demonstrates the benefits of the proposed programming model with implementations of several example user interfaces.

At the core of this programming model is a data structure known as a property model. A property model composes responses to individual events into a single reactive program that runs asynchronously. The program’s results are used to update the GUI. The program is constructed in a manner that respects all data dependencies, thereby guaranteeing that results are consistent regardless of the length of time taken by individual responses. The core reactive program may be extended with features that support additional functionality, such as access to prior variable values, optional data dependencies, and identifying unused variables. The dissertation defines the semantics of the construction and execution of this reactive program formally.

The dissertation shows how property models may be defined as a composition of reusable components. This is essential for modeling GUIs whose structures change in response to user events by the addition or removal of components. Components can contain data and dependencies as well as templates that describe how dependencies arise from composition with other components. Furthermore, templates can be written for arrays of components to define dependencies that arise among them.

One key task of the property model is planning by which methods dependencies will be enforced. The dissertation describes how a specialized planner can be constructed that is able to create a plan for a specific property model. This specialized planner is essentially a Deterministic Finite-state Automaton (DFA), and can be orders of magnitude faster than a general-purpose planner.

DEDICATION

This dissertation is dedicated to my dear wife Shannon, whose constant love and support enabled me to work; and to my children Deborah, Brianna, Jonathan, and Melody, who were always ready to help me take a break.

TABLE OF CONTENTS

	Page
ABSTRACT	ii
DEDICATION	iv
TABLE OF CONTENTS	v
LIST OF FIGURES	viii
LIST OF TABLES	x
LIST OF ALGORITHMS	xi
1. INTRODUCTION	1
1.1 Motivation	1
1.2 Approach	5
1.3 Implementation	8
2. BACKGROUND AND RELATED WORK	9
2.1 MVVM	9
2.2 Dataflow Constraints	10
2.2.1 Multi-way Dataflow Constraints	11
2.2.2 Hierarchical Multi-Way Dataflow Constraint Systems	14
2.2.3 Dataflow Graphs	15
2.3 Related Work	17
3. THE CORE REACTIVE PROGRAM	20
3.1 Solving the Constraint System Asynchronously	22
3.1.1 Variables	23
3.1.2 Methods	23
3.1.3 Constructing the Reactive Program	25
3.1.4 The Reactive Program Graph	26
3.1.5 The Resulting GUI	28
3.2 Operations and Commands	30
3.2.1 Operations in the Reactive Program	30
3.2.2 Operations as Commands	35
3.3 Detecting Unreachable Program Elements	36
3.4 Failure in an Activation	38

3.5	Implementing Property Models with HotDrink	40
4.	EXTENSIONS TO THE CORE PROGRAM	46
4.1	Accessing Prior Values	46
4.2	Managing In-Place Modifications	49
4.3	Optional Constraints	53
4.4	Promise Forwarding	56
4.5	Adjusting Variable Priorities	58
4.6	Detecting Irrelevant Variables	62
4.6.1	Contributing and Relevant Variables	62
4.6.2	Creating the Evaluation Graph	64
5.	OPERATIONAL SEMANTICS	67
5.1	About the Formalism	67
5.1.1	Notational Conventions	67
5.1.2	Symbols and Values	68
5.1.3	Evaluation Environment	70
5.1.4	The Callback Set	72
5.2	The Evaluation Rules	74
5.2.1	Editing the Property Model	74
5.2.2	Scheduling Methods	76
5.2.3	Variables and Promises	80
5.2.4	Promise and Edge Usage	81
5.2.5	Lifting Functions	85
5.3	Summary	86
6.	COMPONENTS AND DYNAMIC ELEMENTS	87
6.1	Components and Composition	89
6.2	Templates and Dynamic Elements	92
6.3	Templates as Signals	96
6.3.1	Signals	96
6.3.2	Labels	97
6.3.3	Paths	98
6.3.4	Templates	99
6.3.5	Resulting Elements	100
6.3.6	Implementation	101
6.4	Dynamic Elements Using Arrays	102
6.5	Array Components as Signals	105
6.5.1	Array Components	105
6.5.2	Indexing Expressions	106
6.5.3	Paths	108
6.5.4	Templates	110
6.5.5	Resulting Elements	111

6.5.6	Implementation	111
6.6	Signature Variations	112
6.6.1	Constants and Partial Instantiation	113
6.6.2	Nested Signatures and Array Slices	115
6.7	Proper Placement of Property Model Modifications	118
7.	STATIC ANALYSIS: SPECIALIZING PLANNERS	121
7.1	Supplementary Background Material	122
7.1.1	Planning Algorithms	122
7.1.2	Constraint Systems as a Monoid	123
7.1.3	Planning of Constraint Hierarchies as Monoid Composition	126
7.2	Properties of Constraint Compositions	128
7.3	Specializing a Constraint System	129
7.3.1	Planner as a DFA	130
7.3.2	Generating the DFA	131
7.4	Experiments	136
7.4.1	Generator Implementation	136
7.4.2	Methodology	139
7.4.3	Results	142
8.	CONCLUSION AND FUTURE WORK	149
	REFERENCES	152
	APPENDIX A. LISTING OF OPERATIONAL SEMANTICS	159
A.1	Overloaded Signatures	159
A.2	Evaluation Environment	160
A.3	Editing the Property Model	161
A.4	Scheduling Methods	162
A.5	Variables and Promises	164
A.6	Promise and Edge Usage	165
A.7	Lifting Functions	167

LIST OF FIGURES

FIGURE		Page
1.1	An example of an auto-complete text box, and a diagram showing the data dependencies involved in its implementation.	3
2.1	A hypothetical GUI for determining prices in a shipping application.	12
2.2	Two dataflow graphs for the shipping GUI example.	16
3.1	The execution model for a GUI based on a property model.	20
3.2	The synchronous execution model for a property model.	21
3.3	The asynchronous execution model for a property model.	22
3.4	A reactive program graph based on the shipping constraint system.	27
3.5	A “top-down” view of the reactive program graph from Figure 3.4.	28
3.6	The shipping form after Generation 3 of Figure 3.4, while activation F_3 is still running.	30
3.7	The GUI for a distance calculator.	33
3.8	The reactive program graph showing operations in the shipping form example.	34
3.9	Activation failure in the reactive program.	39
3.10	The JavaScript definitions for three methods of the shipping form constraint system.	41
3.11	JavaScript that creates the property model for the shipping form example with HotDrink.	43
3.12	Example HTML containing binding declarations for the shipping form example.	44
4.1	Reactive program graph for a constraint that uses prior values.	48
4.2	Implementation of prior values in HotDrink.	49
4.3	Reactive program graph using barriers.	51

4.4	A program graph containing an optional constraint.	55
4.5	Reactive program graphs demonstrating how failure may propagate further than expected.	57
4.6	An example of confusing method selection.	59
6.1	A hypothetical scheduling application.	88
6.2	Creating and composing components in HotDrink.	91
6.3	Representation of a component containing a template.	93
6.4	Representation of multiple components containing instantiated templates. .	94
6.5	Constraints modified by inserting a component into a list of components. .	95
6.6	Creating a constraint template in HotDrink.	96
6.7	Representation of a constraint containing an array and an array template. .	103
6.8	Creating and composing components in HotDrink.	104
6.9	Partial instantiation of a template in HotDrink.	114
6.10	A template with an array slice in HotDrink.	118
6.11	GUI state transition, this time including structural modifications of the property model.	119
7.1	A user interface for ordering customized picture frames. The cost is determined as a function of perimeter (for the frame itself) and area (for the glass.)	125
7.2	Constraints arising from the picture frame GUI.	126
7.3	Constraint graph resulting from composing the constraints in Figure 7.2. . .	126
7.4	A planner for the four-variable system of Figure 7.2, implemented as a decision tree.	128
7.5	A DFA equivalent to the decision tree of Figure 7.4. The sums at state labels are assumed to be restricted to C	131
7.6	Code generated for a DFA.	138

LIST OF TABLES

TABLE		Page
7.1	Average planning times for QuickPlan planner, specialized planners, and planner DFAs.	143
7.2	Maximum planning times for QuickPlan planner, specialized planners, and planner DFAs.	144
7.3	Standard deviation and relative standard deviation of running times for QuickPlan planner, specialized planners, and planner DFAs.	145

LIST OF ALGORITHMS

ALGORITHM	Page
4.1 Enforce topological ordering on a constraint hierarchy	60
7.1 Calculate DFA state for a given path	133

1. INTRODUCTION

This work aims at improvements in Graphical User Interface (GUI) programming. It contributes to an ongoing project to formulate an alternative to the traditional GUI programming model. In this new programming model, GUI behavior is defined by a declarative specification of the data in the GUI, the relationships between pieces of the data, and the actions which the user may perform using the data. This approach yields concise program specifications which translate to rich program behavior. More importantly, the effects of *asynchronous events*—events which occur while older events are still being processed—on program behavior can be controlled, and in many cases eliminated. This is a desirable quality for GUIs, which frequently deal with a large number of events occurring at unpredictable times.

1.1 Motivation

User interfaces are costly to develop and difficult to get correct. Studies have shown that more than 50% of an application’s design and programming effort [36] and between 30% and 60% of an application’s source code [31, 36, 42] are devoted to user interfaces. And yet, despite the effort that goes into these GUIs, a disproportionately high number of defects arise from this code [42]. According to a study by Lazar et al. [29, 7], one-third to one-half of the time spent on a computer is wasted due to frustrating experiences—and poor user interfaces is one of the three main causes of that frustration.

We believe a key contributing factor to this problem is the traditional GUI programming model: the *event-driven model*, more commonly called *event-driven programming*. In this model, the programmer defines GUI behavior by a collection of callback functions, each assigned to an event that may occur during the life of the GUI. Although this programming model may work well for simple GUIs, it does not scale well as GUI complexity grows, leading to what has been called a “spaghetti of call-backs.” [33] Here, we identify two specific shortcomings of the event-driven model.

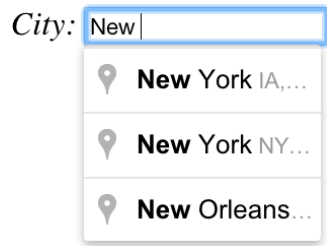
The first shortcoming is that, by requiring program logic to be organized around events, the event-driven model interferes with abstractions for other program concerns. Modular design and abstraction are well-established tools in computer science for managing program complexity: we must be able to divide the program into smaller pieces, and we must be able to use one of these pieces without full knowledge of its inner workings. However, in event-driven programming, any abstractions created by the programmer must be broken down and spread across multiple event handlers. Generally this breaks abstractions boundaries, forcing the programmer to have full knowledge of their workings and manage the details of their interactions.

Failure to support the hiding of implementation behind an abstract interface contributes directly to a lack of support for reusable components. It has long been known that software systems utilizing reusable components tend to be more robust and less costly than their hand-crafted counterparts [4, 16, 37]. Indeed, most GUI frameworks provide many reusable components implementing individual elements of the interface—elements such as text-edit fields, check-boxes, and buttons, collectively known as *widgets*. Such component libraries provide GUIs with reusable *widget behavior*, yielding robust widgets with consistent look and performance.

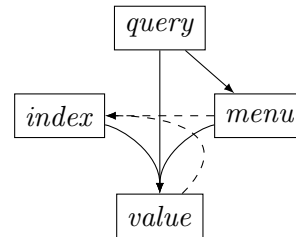
It is rare, however, to find components which encapsulate *GUI behavior*—that is, the coordinated response and interaction of multiple widgets to user input. Such components are difficult to produce in the event-driven model because GUI behavior is so closely tied to specific event handlers which may vary from GUI to GUI. In fact, general assumption seems to be that GUI behavior is, by nature, non-reusable, and must be written from scratch for each GUI. This sentiment is reflected by a quote from the developer documentation of a widely used GUI framework; referring to the “controller,” which is the application code controlling widget interactions, the documentation reads: “*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]

The second shortcoming of the event-driven model is its lack of assistance to the pro-

grammer in dealing with asynchronous program execution. Asynchronous execution is often necessary in GUIs to ensure responsiveness in the face of lengthy program operations: operations proceed in the background as the GUI continues to accept user input. Current GUI programming techniques offer a wide range of options for initiating asynchronous execution. However, asynchronous execution can become complicated when there are data dependencies between asynchronous tasks. Such dependencies mean that the execution of one task may affect the outcome of another; therefore running tasks in different orders or in parallel may yield different outcomes. In event-driven programming, data dependencies arise implicitly whenever different event-handlers access the same variables. Ensuring that data dependencies are enforced in every possible interleaving of events is not a trivial job.



(a) An auto-complete text box.



(b) The data dependencies.

Figure 1.1: An example of an auto-complete text box, and a diagram showing the data dependencies involved in its implementation. These data dependencies are not trivial to enforce, especially when asynchronous execution is involved.

By way of illustration, consider one common GUI element: the auto-complete text box. This element helps the user produce a string to be used as input by some part of the application. Text entered by the user becomes the value of the input string, but is also used as a parameter in an asynchronous search for related input strings. Typically the search results are listed below the text box as a menu from which the user, with mouse or keyboard, may select an alternate input string. Figure 1.1a shows an auto-complete text box being used to select a city as a travel destination.

Figure 1.1b shows the dependencies that emerge in this seemingly simple GUI element. Text entered by the user becomes the query parameter, which determines the menu items. If a menu item is selected, the index of the selected item and the contents of the menu determine the input string; if no item is selected, the query parameter itself becomes the input string. Finally, a change in the contents of the menu affects the selected index: if the previously selected city is in the new menu, its new index should be used; otherwise the index should be reset. We show this dependency with a dashed line, as it is only in effect when the menu changes.

The dependencies reflected in this diagram are non-trivial, and writing code that enforces them is difficult using the traditional event-driven programming model. To test this claim, we performed an informal survey of six popular commercial travel sites (www.expedia.com, www.orbitz.com, www.aa.com, www.united.com, www.hotels.com, and www.yahoo.com/travel) and found that all six contained auto-complete text boxes exhibiting *inconsistent behavior*. We define inconsistent behavior as the same sequence of editing operations producing different outcomes. In all cases, inconsistent behavior was triggered by a rapid succession of input events: presumably newer events were handled before all dependencies had been enforced by previous event handlers. Such behavior can lead to ignored input. In one representative case, we typed “TKU” as the airport code and initiated a search; the results were not for flights from Turku, Finland, but rather (incorrectly) from Tampa, Florida.

In these particular applications the consequences of inconsistencies are not that severe. If an error manifests, it is relatively easy to detect and correct. Furthermore, users learn to avoid errors by adapting their use—in these applications, by waiting for the GUI to catch up. Yet, requiring users to synchronize their usage negates many of the benefits of asynchronous execution. And it is not hard to find more harmful problems in other widely used applications. In a recent interaction with the Blackboard system (www.blackboard.com, used by many US Universities) we noticed that entering students’ grades quickly leaves behind numerous erroneous entries as keystrokes get ignored or assigned to wrong

entries. We argue that these examples are not anomalies, but rather indicative of a larger problem—these applications have millions of users and they can be expected to have been developed with ample resources and by competent programmers, yet they manifest glaring errors in their most basic functionalities. The problem of GUI inconsistencies is clearly systemic.

1.2 Approach

The programming model proposed by my research group is built around a data structure we call a *property model* [24]. A property model serves as an abstraction of the GUI, much like a *View-Model* in the MVVM design pattern [19] (see Section 2.1 for an overview of MVVM). As such, it contains the data used by the GUI, and the logic responsible for GUI behavior; it does not, however, contain any presentation logic, nor any event handlers. Any necessary event handlers are generated automatically according to *binding specifications* indicating how the data of the property model is connected to the widgets of the GUI.

The data of the GUI is held in variables of the property model. The value of these variables, together with certain dependency information discussed later, define the *state* of the property model. The logic of the GUI consists of functions provided by the programmer. These functions may be divided into two categories. The first category modify variables of the property model to create a *consistent* state. What it means for a state to be consistent is defined by the programmer; generally speaking, it means that all invariants for the data are satisfied. The second category of functions implement application-specific operations using data of the property model. We refer to these functions as *commands*. A command may be as simple as modifying a single variable, or as complicated as submitting the data of the GUI to a server for processing.

The life of a property model proceeds much like a state machine. It begins with the property model executing functions which produce a consistent state. Once this is done, the property model is available to execute a command in response to events in the application. A command may not modify any variables of the property model while executing; however, it may end by assigning new values to some variables of the property model. The

property model responds to any assignments by once again executing functions to produce a consistent state. Taken together, the command and the property model’s response to assignments define a transition from one consistent state of the property model to another. At this point, the property model is ready to execute another command, and the cycle repeats itself.

The property model executes logic of the GUI in two phases. The first phase involves planning out which functions will be executed and scheduling them to be run. This phase executes synchronously, beginning with the invocation of a command and ending with the scheduling of all functions needed to transfer the property model to its next consistent state. The second phase involves the scheduled functions being executed. This second phase proceeds asynchronously, with each function executing as its inputs become available. Because the second phase always follows the plan laid out during the first phase, its results are unaffected by later operations of the property model. After the first phase finishes, the property model is immediately available to invoke new operations. An upper limit for the execution time of the first phase can be established for each constraint system. In practice the execution is often instantaneous [22], so the GUI is guaranteed to always remain responsive.

One way to characterize the two phases is that the first phase generates a reactive program that executes asynchronously in the second phase. A *reactive* program is one which responds to changes in variables by automatically recalculating values for functionally dependent variables. Various reactive programming frameworks have recently gained popularity in GUI programming (see, e.g. [3, 27, 32]). These frameworks build reactive programs from a static description of data dependencies. Data dependencies in a GUI, however, often change based on user interaction. This would require programmers to define multiple reactive programs and coordinate their use. By dynamically generating a reactive program from the description of a constraint system, property models simplify GUI programming while guaranteeing consistent results.

My work contributes to this new programming model in the following areas.

- *Generating a reactive program given a sequence of user commands and a multi-way dataflow constraint system.* This work defines the core reactive program that governs all GUI behavior. This program behaves as a state machine, transferring the property model from one consistent state to the next. The program is a function of the commands and the order in which they were executed; it is not affected by the timing between commands or the time required for execution. This work is discussed in Chapter 3.
- *Additional features for generated reactive programs.* This work focuses on extending a generated reactive program with new features, thereby allowing the reactive program to implement additional program behaviors. This work is discussed in Chapter 4.
- *Operational semantics for the dynamic program.* This work provides a formalism for the behavior of a property model. This gives precise definition to the concepts discussed in Chapters 3 and 4 and serves as a foundation for a correct property model implementation. This work is discussed in Chapter 5.
- *Reusable components and dynamic elements.* This work focuses on expressing a property model as a composition of reusable components. Composition of components involves, not only combining their members, but also the generation of new *dynamic elements* as a product of the relationships between components. This work is discussed in Chapter 6.
- *Specializing planners for hierarchical multi-way dataflow constraint systems.* When data dependencies in a GUI are made explicit, as they are with property models, it creates new opportunities for static analysis and optimization. This work is an example of such an optimization, involving the offline construction of a planner for a specific constraint system as a finite state automaton. This work is discussed in Chapter 7.

1.3 Implementation

The work for this dissertation includes an implementation of property models for web-based applications called *HotDrink*[21]. Although a property model implementation exists from previous work, this new work involves such a drastic departure from previous work that a complete rewrite was justified; attempting to retrofit the existing implementation with support for asynchronous execution would have been difficult. This work includes a collection of unit tests and an introductory tutorial.

We chose TypeScript as the implementation language. TypeScript is a statically typed variant of JavaScript, created by Microsoft, which is compiled to JavaScript for execution. The benefit of TypeScript is that it provides a more structured programming environment than JavaScript, while remaining close enough to JavaScript that the compiled result is very easy to map back to the source code. We distribute HotDrink as the JavaScript code generated from the TypeScript sources.

Although this document is not intended as a full introduction or reference to HotDrink and its use, we will refer to it throughout this document in order to explain how the concepts discussed in this thesis can be realized in the implementation. For further information on HotDrink, including its source code and a tutorial, please see <http://github.com/HotDrink/hotdrink>.

2. BACKGROUND AND RELATED WORK

As mentioned in Section 1.1, the traditional event-based programming model has several shortcomings when it comes to programming complex GUIs. Because of this, there has been much work towards supplemental or alternative programming techniques for GUIs. This section gives an overview of two such techniques which are foundational to the property-model-based programming model: the MVVM design pattern and dataflow constraints. It also provides an overview of related work and how it compares to our technique.

2.1 MVVM

Much work has been done on the separation of concerns in GUIs, resulting in design patterns such as *Model/View/Controller* [28], *Model/View/Presenter* [44], and *Presentation-Model* [15]. Each of these patterns organizes a GUI-based application into clearly defined components, promoting modularity and encapsulation. Our programming model follows the organization of a more recent pattern: *Model/View/View-Model* [20], or *MVVM*.

The MVVM pattern divides the application into three major components. The *Model* is responsible for the “business” data and logic in the application. This component is largely domain-specific; it has no knowledge of, nor direct interaction with, the GUI itself. The *View* is responsible only for presentation logic and for accepting user input. More abstractly, the view is responsible for implementing widget behavior: drawing the widgets to the screen and translating user input into events. The *View-Model* is the application’s model, or abstraction, of the View. It maintains the data used by the GUI and provides the logic implementing GUI behavior.

The data and logic of the View-Model are connected to the widgets and events of the View through connections called *data bindings*, or simply *bindings*. The purpose of the bindings is to keep the data of the View “in sync” with the data of the View-Model: changes to one are automatically propagated to the other. Bindings use event-handlers to translate events of the view into changes in the variables of the View-Model. Similarly, the property

model produce events whenever its variables are changed; bindings use event-handlers to translate these events to updates in the View.

Describing our programming model in terms of MVVM, a property model is an implementation of the View-Model generated from a declarative specification. The primary focus of our work, then, is in defining how the property model can represent the data and behavior of a GUI, with a secondary focus on defining the bindings that connect the data and behavior to View. The View and the Model components of an application are outside the scope of our programming model. In HotDrink the View is constructed from HTML, along with any JavaScript code that builds HTML elements. The Model is any code implementing business logic, most likely run on a server in response to requests generated by the web page.

An additional element of the MVVM pattern which is relevant to property models is a *command*. A command is simply a function of the View-Model that can be invoked from the view—e.g., by clicking a button. Typically commands are associated with a property indicating whether the command can be executed. If the GUI is in an invalid state, e.g., due to invalid data, a command can be disabled until the state is corrected.

2.2 Dataflow Constraints

The use of dataflow constraints to manage data dependencies has become increasingly popular, as evidenced by the many modern GUI frameworks which make use of them, such as Knockout [27], Ember [13], and ConstraintJS [40]. A dataflow constraint is typically defined by providing a function which may be used to calculate the value of one variable based on the value of others. Once a constraint is defined, the run-time system is responsible for updating the variable by calling the function any time one of its inputs change. Property models make use of dataflow constraints to ensure that invariants are maintained between variables. In property models, however, the constraints are *multi-way dataflow constraints* [6] managed by a *hierarchical multi-way dataflow constraint system*.

2.2.1 Multi-way Dataflow Constraints

A *multi-way dataflow constraint* represents a relation over a set of variables. When the relation holds, we say the constraint is *satisfied*. The relation of the constraint is not defined explicitly, but rather implicitly through a set of *constraint satisfaction methods*, or simply *methods*. Each method is a function which calculates new values for some variables of the constraint using the remaining variables of the constraint as inputs. A method should *enforce* the constraint; that is, after execution of the method, the constraint should be satisfied.

In general, there is no restriction on how many methods a constraint may have, nor on how each method may use the variables of the constraint. A method may have multiple inputs and multiple outputs. Some variables may be used as input by every method; some may be used as output by every method. The only requirements for a constraint are, first, that the output variables of one method may not be a superset of the output variables of another, and, second, that every method must use all of the constraint's variables. A method violating the first requirement would be useless; it would never be selected as part of any plan for solving the system (see Section 2.2.2). The second requirement, known as *method restriction* [46], ensures that a plan is found in polynomial time [50].

Of all the variables of the property model, a method should read only its input variables, and write to only its output variables; it should, furthermore, write to each output variable exactly once. We refer to such a function as a *dataflow function*. In general, dataflow functions are not required to be referentially transparent.¹ Thus, a method may enforce the constraint differently each time it is run. For example, a method which involves a database query will return different results depending on the contents of the database. After a method of a constraint is executed, a property model assumes its constraint is enforced and will remain enforced until some variable of the constraint changes.

Defining the constraints present in a GUI is not always a trivial task. As an exercise

¹A function is *referentially transparent* if it has no external effects, and if the same inputs always result in the same outputs.

Class:

Volume: m³ **Weight:** kg

Dimensions: cm × cm × cm

Distance: km

Price: \$ USD

Figure 2.1: A hypothetical GUI for determining prices in a shipping application. Note the large number of data dependencies; this means changing one field often results in updates to many fields.

in constraint design, let us consider the constraints present in an example GUI shown in Figure 2.1. In this hypothetical shipping application, packages are classified by weight and volume into one of several “package classes;” the price of shipping a package is a function of its package class and the distance to be shipped. We assume that any field in the GUI may be edited and other fields will automatically adjust accordingly. This results in a very flexible GUI which supports several modes of interaction: the user may determine the price of shipping a package a certain distance, or how far a package can be shipped for a certain price, or even what shipping class may be shipped a certain distance for a certain price. Furthermore, the user may enter package class directly, or choose to enter the weight and volume of the package instead, or the weight and dimensions.

First, consider the constraint between the volume v of a package and its dimensions x , y , and z . The relation for this constraint is $v = xyz$. We may define this constraint using the four methods $x \leftarrow v/yz$, $y \leftarrow v/xz$, $z \leftarrow v/xy$, and $v \leftarrow xyz$. Each of these methods enforce the constraint by assigning a new value to one variable calculated from the remaining three. If we label these methods A , B , C , and D , respectively, then we may write them as $x \leftarrow A(v, y, z)$, $y \leftarrow B(v, x, z)$, $z \leftarrow C(v, x, y)$, and $v \leftarrow D(x, y, z)$. This representation reveals the dataflow while omitting details of method implementation.

Next, consider the volume, weight w , and class c of a package. This constraint may be

expressed informally as follows: a package having weight w and volume v falls into package class c . Note that constraints do not need to be expressed as mathematical equations, nor do methods need to be expressed as mathematical expressions. Methods are arbitrary functions of the implementation language. For this constraint, assume the weight and volume may determine the class of a package according to some function E , and we may also query for a representative weight and volume for a given class according to another function F . Disregarding here the implementation of these functions, we write these methods as $(w, v) \leftarrow E(c)$ and $c \leftarrow F(w, v)$.

The third constraint in our shipping form is between the package class, shipping distance d , and shipping price p . A simple formulation of this constraint would be as follows: a package of class c can be shipped a distance of d for a price of p . However, this formulation has a difficulty: it is problematic to determine a package class for a given price and distance since many price/distance pairs have no matching shipping class. We could simply define the constraint without a method that outputs to c , or we could find an alternative relation which supports output to c . If we add to our constraint the maximum allowed price m , then we may take this relation: a package of class c can be shipped a distance d for a price p which is no greater than m . We can implement this constraint with three methods. The first method, $(c, p) \leftarrow G(d, m)$, determines the largest package class c which can be shipped a distance d for a price no greater than m , as well as the actual price p of the shipping. The second method, $(d, p) \leftarrow H(c, m)$ determines the greatest distance for which a package of class c can be shipped for a price no greater than m , and the actual price p of the shipping. And the third method, $(m, p) \leftarrow I(c, d)$ determines the price of shipping a package of class c a distance d ; this becomes both the maximum price m and the actual price p .

To use this alternate relation, we must decide how m is to be presented in the GUI. For example, one might create an additional text box and bind it to m . For this discussion, however, we assume the binding of the *Price* text box has been altered so that it reads from p and writes to m . In this way, the user enters the maximum price and sees the actual calculated price.

2.2.2 Hierarchical Multi-Way Dataflow Constraint Systems

A *multi-way dataflow constraint system* is responsible for ensuring a collection of multi-way dataflow constraints are satisfied. It does this by executing one method from each constraint, taking care that, once a method for a constraint has been executed, no variables of that constraint are modified. Practically speaking, this means executing the methods in an order such that no method outputs to a variable after it has been used (either as input or output) by another method. We call any execution order that satisfies this condition a *valid execution order*. Note that an arbitrary set of methods may not have a valid execution order; for example, there can be no valid execution order for a set containing two methods which output the same variable, nor for a set containing a cycle in the dependencies.

Satisfying all constraints in the constraint system—known as *solving* the system—involves two steps. First, selecting the methods to be executed: one from each constraint, such that a valid execution order exists. This set of methods is called the *plan*. Second, executing the methods of the plan in a valid execution order.

Multi-way dataflow constraint systems can be *underconstrained*; multiple plans may exist for solving the system. Each plan, however, is unique in the set of variables not used as output by any method. A ranking of variables thus gives a ranking for plans, so that a unique “best” solution can be chosen. This is accomplished in the following three steps.

First, we add a *stay constraint* for each variable in the system. A stay constraint has one variable, and one method that outputs to the variable. The method is a constant function with the variable’s current value. Adding stay constraints makes the system *overconstrained*; no plan can enforce all stay constraints.

Second, we prioritize the stay constraints, making them totally ordered. When a variable is edited, its stay constraint is promoted to the highest priority. Thus, the hierarchy of stay constraints corresponds roughly to the order in which variables have been edited.²

Third, we use the hierarchy of constraints to select the plan that enforces the highest

²There are other occasional circumstances in which a constraint’s priority may be altered; these conditions are discussed in Section 4.5.

priority constraints. More precisely, if we characterize each plan by a sequence of the constraints it enforces, in order from highest to lowest priority, then the constraint system selects the plan which is lexicographically greatest. Because our hierarchy reflects the editing order, the system will have a bias towards preserving variables more recently edited by the user.

We define two editing operations for the constraint system. The *touch* operation promotes a constraint to the highest priority, and the *set* operation assigns a new value to a variable and also touches the stay constraint for that variable. An *edit* of the system may be defined as a sequence of one or more touch and set operations. The constraint system is solved after each edit.

2.2.3 Dataflow Graphs

The key to managing asynchronous computations in a GUI is making all dependency information explicit. Property models capture this information in three directed graphs, defined in terms of the constraint system's variables, methods, and constraints. To define these graphs, let V represent the set of all variables and M the set of all methods (every method of every constraint) in the system.

Constraint graph The constraint graph, $G = \langle N, E \rangle$, is a bipartite directed graph, where the node set $N = V \cup M$ consists of the variables and methods of the constraint system, and the edge set E contains an edge (v, m) if variable v is an input of method m , and an edge (m, v) if a variable v is an output of method m . The constraint graph represents every potential dependency that exists in the constraint system.

The constraint graph for the shipping form example is shown in Figure 2.2a. This graph reflects the three constraints we defined in Section 2.2.1; stay constraints are omitted to avoid complication. Notice that constraints are not made explicit in this graph. However, because of method restriction, we may deduce the constraints using the following rule: two methods m and n are in the same constraint if and only if they each have the same

Evaluation graph In some cases a method may run without reading all of its input variables. In such cases, the solution graph would reflect dependencies that were not actually enforced. The evaluation graph is identical to the solution graph except that it does not contain an edge (v, m) if method m did not read variable v during its most recent execution. Thus, the evaluation graph gives the most accurate representation of the dependencies currently in place.

The evaluation graph is used only in certain algorithms which require detailed dependency information, such as the algorithm for automatic disabling of irrelevant widgets.[25]. This work includes a complete rewriting of this algorithm, and with it, a complete redefinition of the evaluation graph. For this reason, further discussion of the evaluation graph is deferred to Section 4.6.

2.3 Related Work

Over the years there has been much research into the use of constraints and constraint systems in GUI implementation, resulting in many GUI toolkits and frameworks in which constraints play some part, whether large or small. On one end of the continuum, are GUI frameworks designed entirely around constraint systems such as SkyBlue [46], Garnet [34], and Amulet [35]. These frameworks primarily focus on visual tasks such as component layout and graphics. More recently, the Subtext [11] framework aims at simplified GUI implementation by providing automatic data layout and constraint satisfaction, much like a spreadsheet.

At the other end of the continuum are existing GUI frameworks which have added individual dataflow constraints to their toolkits—frameworks such as OpenLaszlo [41], Flex [1], and JavaFX [26]. The focus of these constraints is generally propagating changes in dependencies to the View. Often these constraints are solely to update the View, not for updating arbitrary variables in the program; the programmer simply binds an expression to the View, and, as variables in that expression change, the expression is recalculated and the View is updated with the result.

In between one finds many toolkits and frameworks focusing on dataflow constraints, but intended to be used in combination with some other GUI framework to provide traditional GUI functionality. These frameworks tend to be closer in spirit to property models. For example, Knockout [27] and ConstraintJS [40] both bring dataflow constraints to JavaScript, allowing the user to define dependencies which are automatically updated and propagated to the View. Microsoft’s Reactive Extensions (Rx) [32] can translate changes in variables to automatic queries using LINQ. Babelsberg [14] integrates constraints with the Ruby language.

Functional Reactive Programming (FRP) [12, 52] is reactive programming based on purely-functional abstractions of events and of values which change over time, known as *behaviors* or *signals*. There are several GUI frameworks designed around FRP. Frameworks such as FranTk [45], Fruit [9], and Yampa [10] are embedded in Haskell, making them somewhat difficult to integrate with imperative GUI frameworks. Elm and Flapjax, on the other hand, are two languages based on the concepts of FRP, but which compile to JavaScript. There are also libraries that allow FRP-style programming in imperative languages; for example, Bacon [3] brings FRP to JavaScript, and Frappé [8] brings it to Java.

Property models distinguish themselves from other approaches in three ways. The first is in the use of multi-way constraints. The majority of the above systems use one-way constraints, in which data always flows from one set of variables to a second set. A few systems, such as Knockout and Subtext, also support two-way constraints—i.e., a one-way constraint with an inverse function. ConstraintJS allows the programmer to define different program states and specify which constraints are active in each state. In theory, this mechanism could be used to simulate multi-way constraints by making a separate state for each possible dataflow. Additionally, Multi-Garnet [48] extended Garnet with multi-way constraints.

The second distinguishing feature is the explicit representation of dependency information as a data structure. This information is the basis for several reusable algorithms. For

example, we may determine which variables are not being used, allowing widgets bound to them to be disabled. Or we may determine *user intent*, defined by which variables were edited directly by the user vs. calculated by the system, which is needed for recording the use of a GUI to a script. [24, 25] Additionally, as described in this paper, the dependency information guides constructing the reactive program.

The third distinguishing feature is its approach to asynchronous execution of dependencies. Many of these systems represent dependencies as a function, making them strictly synchronous. Those which allow asynchronous execution, such as Microsoft Rx and Parallel FRP [43], are targeted for situations in which every event must be handled, and the order of results is unimportant—e.g., a web service replying to requests. A property model respects the ordering in which edits are made, yet allows computations to proceed as soon as their inputs are available. Furthermore, it ensures that each variable reflects the most current known value, and that computations made irrelevant by more recent results are unscheduled.

Elm permits an alternate type of asynchronicity by allowing the user to specify that certain computations are to be performed before all dependencies have been updated; if a dependency has not been updated, its last known value is used. For example, suppose a variable x is the output of a function f and the input of a function g . If g is ready to be computed before f has finished calculating the value of x , then g runs using the last known value of x ; later, when f produces a new value for x , function g is run again. This prevents the lengthy calculation of a single dependency from blocking the flow of computation. In a property model, we achieve this effect by creating two variables—one for the output of the first computation, and one for the input of the second—and binding the first to the second.

3. THE CORE REACTIVE PROGRAM*

As discussed in Chapter 1, a *property model* fills the role of View-Model in a GUI by storing the data used by the GUI and capturing the logic used to implement GUI behaviors. Data is held in variables of the property model; the values of these variables define the state of the GUI. The programmer defines invariants for the data in the form of constraints in a multi-way dataflow constraint system. When the values of the variables satisfy all constraints, we say the property model, and therefore the GUI, is in a *consistent* state. As the GUI runs, user actions in the View are translated by bindings into changes in the value of one or more variables. We refer to these changes as *edits*, and to the acts of making edits as an *editing operations*. The property model responds to edits by solving the constraint system, thereby enforcing all constraints. Taken together, an editing operation and the solving of the constraint system constitute a transition from one consistent state of the GUI to another. This transition is illustrated in Figure 3.1.

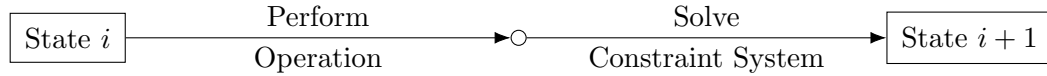


Figure 3.1: The execution model for a GUI based on a property model. Each editing operation is followed by solving the constraint system, thereby transitioning the GUI to a consistent state.

Chapter 2 describes the process by which a multi-way dataflow constraint system is

*Portions of this chapter are reprinted with permission from “Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems” by Gabriel Foust, Jaakko Järvi, and Sean Parent. *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pp. 121–130, Copyright 2015 by ACM. <http://dx.doi.org/10.1145/2814204.2814207>

Portions of this chapter reprinted with permission from “Responsive and Consistent User Interfaces with Multi-Way Dataflow Constraint Systems” by Gabriel Foust, Jaakko Järvi, and Sean Parent. Under review for inclusion in *Computer Languages, Systems and Structures (COMLAN)*, Elsevier.

solved. The process can be summarized in two steps. The first step is calculating a *plan*: a set of methods, one from each constraint, with a valid execution order. The second step is executing the methods of the plan following a valid execution order. In this way, all constraints in the system are enforced.

The discussion in Chapter 2 assumes that methods are executed *synchronously*—that is, once the execution of a method begins, no other code may execute until that method is finished. The synchronous execution of methods prevents the GUI from responding to new user events until all methods have finished execution. This execution model is depicted in Figure 3.2; it replaces the single step “Solve Constraint System” in Figure 3.1 with two steps, “Calculate Plan” and “Execute Methods.” As shown by this figure, the GUI does not arrive at a consistent state, and thus is unable to receive new user events, until all methods have finished execution. This conflicts with a key design goal for GUIs: a GUI should always remain responsive to user events.

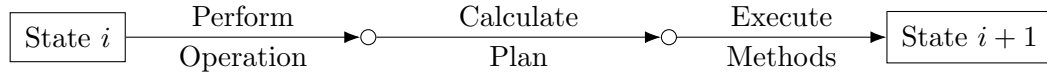


Figure 3.2: The synchronous execution model for a property model. Solving the constraint system requires calculating a plan and executing its methods. The GUI is not ready to accept new editing operations until all methods of the plan have finished execution.

This chapter describes how methods may be executed asynchronously. Asynchronous execution allows the GUI to respond to user events even as methods are executing. The key idea behind our approach is that the property model does not execute the methods of the plan while solving the constraint system, but rather schedules them to be run at a later time. The solution graph, described in Section 2.2.3, is used to guide scheduling so that no method will be executed before its inputs have been calculated. Furthermore, we can ensure that a method will *only* wait on its dependent values; that is, it is free to begin execution as soon as those values are ready. This execution model is illustrated in

Figure 3.3; it replaces the step “Execute Methods” with “Schedule Methods.”

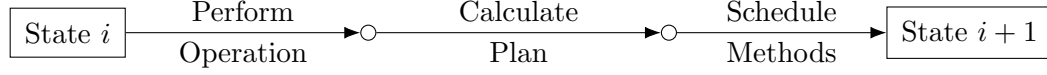


Figure 3.3: The asynchronous execution model for a property model. Solving the constraint system requires calculating a plan and scheduling its methods for execution. Methods execute asynchronously as their inputs become available.

In effect, this execution model constructs a reactive program to respond to edits by updating functional dependencies. This program runs asynchronously at its own pace. Solving the constraint system results in adding a new “layer” to the reactive program—that is, a new set of methods scheduled to be executed. Once the methods of a plan are scheduled, the GUI is considered to be in a consistent state even though some variable values may have yet to be calculated. Any attempts to use such variables, e.g., by other methods, will simply be added to the reactive program and scheduled to run as soon as the value is ready. Thus, the GUI is available to respond to user events immediately after all methods of the plan have been scheduled for execution. In this way, the GUI remains responsive at all times.

This chapter also examines how editing operations may be generalized and included as elements of the reactive program, and the way in which method failure affects the reactive program as a whole.

3.1 Solving the Constraint System Asynchronously

This section discusses the representation of variables and methods in a property model, and how these representations may be used to construct a reactive program. This section considers only one kind of editing operation: assigning a new value to a single variable. Section 3.2 shows how we may generalize this definition to include more complex editing operations.

3.1.1 Variables

A variable of a property model represents a value that changes over time. Unlike variables of some reactive systems, the variables of a property model do not change continuously; they change only in response to an edit: either as a direct result of the edit, or as a result of solving the constraint system in response to the edit. Once the constraint system has been solved, all variable values will remain constant until the next editing operation.

We represent individual variable values using a well-known asynchronous programming construct: promises [30]. A *promise* represents a value which may not be currently available, but which will be available at some point in the future. We distinguish promises from values: a *value* is considered to be a value of the implementation programming language and *not* a promise. A promise whose value is not yet available is said to be *pending*. Once the value becomes available, the promise is said to be *fulfilled*. We may *subscribe* to a promise by providing a callback function to be invoked with the value of the promise once it is fulfilled. A fulfilled promise remembers its value; subscribing to a fulfilled promise simply results in the callback being invoked at the next opportunity. A promise may also be *rejected* indicating that the process intended to produce its value has failed and therefore the value will never be available. Section 3.4 discusses rejected promises.

For each variable, we define a sequence of promises called the *promise history*. The promises of the promise history correspond to every value given to the variable over the life of the program, ordered by the time they were assigned. The last promise in the promise history, known as the *current promise*, represents the current value of the variable. To assign a new value to a variable, we add a new promise to the end of the promise history. This promise may be pending, allowing us to effectively assign a new value to a variable before the value is known. Promises are only added to the history; they are never removed.

3.1.2 Methods

Asynchronous execution in property models is supported by scheduling, rather than executing, methods while solving the constraint system. Concretely, this is achieved by

representing each method with a function whose inputs and outputs are promises. When called by the property model, this function does no real work: it merely subscribes to the input promises, constructs the output promises, and schedules the work which will fulfill the output promises once input promises are fulfilled. Since the actual work of a method will occur later, we say that, when the constraint solver executes this function, it *schedules* the method; thus, the function is called the method’s *scheduling function*.

It is through methods that the reactive program may achieve asynchronous execution. Methods can offload work to, e.g., other threads or remote servers, thus freeing the current thread to execute other methods or respond to user events. We make no assumptions regarding the manner in which this “offloading” is accomplished; the mechanisms used are assumed to belong to the implementation language. In our JavaScript implementation a method may schedule work on a different thread using web workers, on a remote server using Ajax, or simply at a later point in the current thread using a timer event. It may also forgo asynchronous execution and execute immediately. Which approach, if any, a method uses does not affect any other part of the reactive program.

It is the programmers job to supply the scheduling function for a method. However, the scheduling function is frequently formulaic enough to be generalized by a function we call *liftP*. The *liftP* function takes a function over values and “lifts” it to produce a function over promises; e.g., lifting a function of type $T \rightarrow U$ produces a function of type $P\langle T \rangle \rightarrow P\langle U \rangle$, where $P\langle T \rangle$ represents a promise for a value of type T . The effect of this new function is the same as the original, except that it accepts and returns promises instead of values. When called, the new function subscribes to all input promises and returns output promises. Once all input promises have been fulfilled, the new function calls the original function with their values; the return values of this call are then used to fulfill the output promises. This is exactly the work of a scheduling function. Thus, *liftP* allows us to generate scheduling functions for methods implemented by a single function.

We define the term *method activation* as a single execution of a method: what starts with scheduling the method and ends when all output promises are resolved. Although

a method activation is a computation and does not have concrete representation in our system, it is useful to think of it as an entity. Like promises, activations are elements in our reactive program, each representing the execution of a certain method with certain inputs which produced certain outputs. And, just as we associate every variable with a promise history, so may we associate every constraint with a sequence of activations, the *activation history*. The activation history represents every activation of any method of the constraint, ordered by the time they were scheduled. The last activation in the activation history, known as the *current activation*, represents the activation currently responsible for enforcing the constraint.

3.1.3 Constructing the Reactive Program

The reactive program is generated incrementally in response to edits to the variables of the property model. The very first edit, and therefore the beginning of the reactive program, occurs as variables are initialized according to the property model’s specification. Subsequent edits correspond to editing operations triggered in response to events in the life of the GUI. The property model responds to each edit by generating a plan for enforcing the constraints of the system, as described in Section 2.2.2. It then schedules methods of the plan in a valid execution order. Scheduling a method involves three steps: first, the current promise for each input variable is retrieved; second, the method’s scheduling function is invoked, passing in the retrieved promises as arguments; and third, each promise returned by the function is added to the end of the corresponding output variables’ promise history, thereby becoming the current promise for that variable.

Some methods of the plan may not need to be scheduled, because executing them is known to have no effect. Given a method m belonging to constraint c , if the current activation of c is an activation of m , and if the inputs of m remain unchanged since that activation, then we may assume that the constraint c is already enforced by m . Thus, the methods which must be scheduled are those *not* selected in the previous generation, and those whose inputs have changed since the previous generation. The selection of methods for execution is explored further in Section 4.5.

After all methods have been scheduled, the property model has completed its response to the edit; the property model is free to accept other edits while method execution proceeds asynchronously. As input promises are fulfilled, method activations perform their work and fulfill their output promises. In this way, the constraints of the system will eventually be enforced.

The property model’s response to an edit, planning and scheduling methods, is atomic—editing operations are queued if planning and scheduling for prior edits have not been completed. After the response, every variable which will receive a new value because of the edit has been given a promise for that value. Should some other code request the value of one of these variables, it will receive the promise for the updated value, regardless of whether or not its calculation has completed. Thus, the results of any two equal sequences of edits will always be the same reactive program, regardless of the differences in time intervals between the edits in each sequence. If all methods of the constraint system are referentially transparent, then so is the reactive program.

3.1.4 The Reactive Program Graph

As described above, the life span of our constraint system may be characterized as a sequence of edits, each followed by an update in which the constraints of the system are enforced. We refer to these successive updates as *generations* of the system. The solution graph describes dependencies between variables for a single generation, but not the dependencies that may arise between generations.

To capture the dependencies over the lifespan of the GUI, we define the *reactive program graph*. The nodes of the graph consist of promises and activations. The graph contains a directed edge from every promise to every activation taking the promise as input, as well as an edge from every activation to every promise it produced as output. Although the entire graph is simply a DAG, it is illustrative to visualize it in layers stacked on top of each other: one layer for every generation of the constraint system, containing all activations and promises generated while solving the constraint graph during that generation.

By way of example, Figure 3.4 shows the reactive program graph of one possible se-

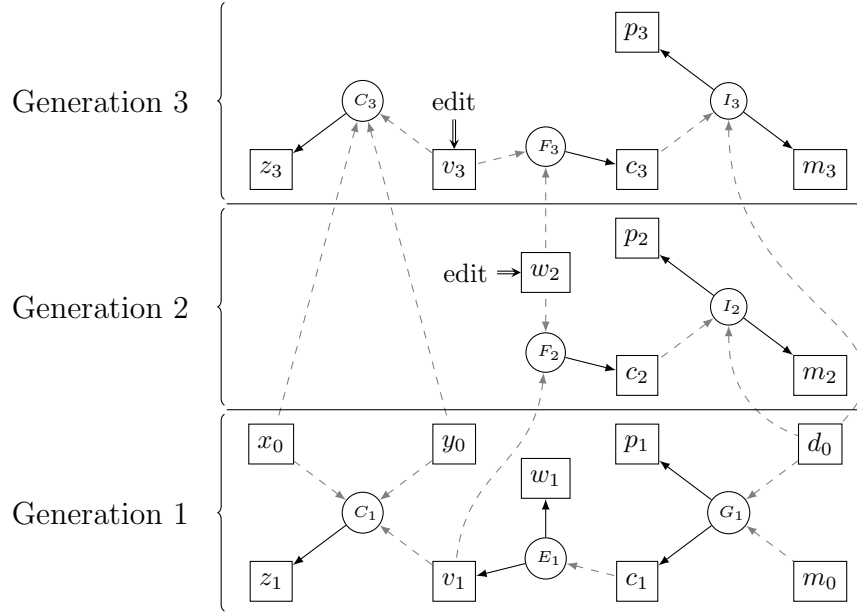


Figure 3.4: A reactive program graph based on the shipping constraint system. Rectangles represent promises; circles represent method activations. Nodes from the same generation are grouped together in layers, which are ordered by time along the y -axis.

quence of edits in the shipping form example. Promise nodes are labeled by the variable to which the promise is assigned; activation nodes are labeled by the method which was executed. Each label is sub-scripted with the generation in which it occurs. Generation 1 shows that the constraint system was initially solved using the plan $\{C, E, G\}$. This plan generates new promises for the variables z , v , w , p , and c , as indicated by the nodes sub-scripted with 1. Variables x , y , d , and m retain their initial values given when the property model was defined as indicated by the nodes sub-scripted with 0.

Generation 2 is the result of editing variable w . This triggers an update producing the plan $\{C, F, I\}$, which gives new promises to variables p , c , and m . Variable v did *not* receive a new promise; therefore, activation F_2 uses promise v_1 , which is the current promise for v , as input. Similarly, activation I_2 uses promise d_0 , which is the current promise for d . Because the inputs to method C are unchanged, it does not need to be scheduled in this generation.

Generation 3 is the result of editing variable v . This does not alter the plan; however, all methods have at least one changed input, and thus must be scheduled. Note that activations in this generation use promises from both previous generations.

The reactive program graph provides a clear visualization of the flow of data over the lifetime of the application. We imagine it in three-dimensional space: each generation laid out in its own plane, stacked on top of the previous generation. Arranging the nodes so that promises for the same variable are directly over one another reveals the promise history for each variable as a vertical column. Arranging method activations for the same constraint similarly reveals the activation history for each constraint. Viewing the graph in this arrangement from above, we see only the current promise for each variable and the current activation for each constraint. Such a “top-down” view of the graph found in Figure 3.4 may be seen in Figure 3.5. Once all activations are complete and all promises fulfilled, then this is the view the GUI will make visible to the user.

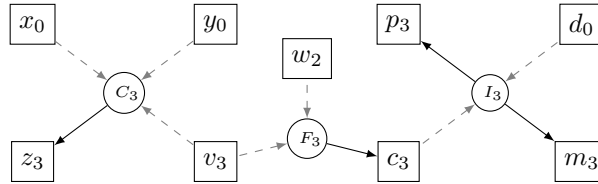


Figure 3.5: A “top-down” view of the reactive program graph from Figure 3.4. This view shows the most recent activation for each constraint and the most recent promise for each variable.

3.1.5 The Resulting GUI

As mentioned in Section 2.1, variables of the property model are connected to the view through data bindings. A variable publishes its value every time it changes; a binding subscribes to these notifications and updates the View when they occur. In this way the results of the property model are made visible in the GUI.

A variable can obtain a new value when a promise in its promise history is fulfilled.

Often it is the current promise, but in general a variable’s promise history may contain several pending promises which can be fulfilled in any order. The view should reflect the most current value of a variable, which is the value of the last fulfilled promise in the variable’s promise history. Therefore, when a promise of the promise history is fulfilled, the variable will publish the promise’s value as its own value *unless* a later promise in the promise history has already been fulfilled.



Consider again the “top-down” view of the reactive program graph shown in Figure 3.5. We may imagine that promises which are pending are transparent, so that for each variable we see the topmost fulfilled promise. As the reactive program’s execution proceeds, we can see its progress as more recent promises become updated, resulting in new values for the variable. This is the view reflected in the GUI.

We call a variable *pending* or *fulfilled* depending on whether its current promise is pending or fulfilled, respectively. It is helpful to the user to know whether the View shows a pending value that might change, or a value that will not change until an edit occurs. To this end, for each variable we define a property named *pending* which is true when the variable is pending and false when it is fulfilled. This property publishes its value in the same way a variable does, and it is thus suitable for binding to the View.

Figure 3.6 shows the shipping form GUI as it appears just after Generation 3 in Figure 3.4: the activation I_3 has completed, F_3 is running, and C_3 is scheduled. The promises c_3 , p_3 , and m_3 are pending, and therefore the drop-down list for the shipping class reflects the value of c_2 and the text box for *price* the value of p_2 . These elements will be updated when p_3 and m_3 are fulfilled.

To give a different appearance to GUI elements whose corresponding variables are pending, we used a binding that updates the element’s CSS class based on the variable’s *pending* property; we then used a CSS stylesheet to attach a different background color and add a “spinning” graphic to elements with the appropriate class.

Note the work required by the programmer to get this advanced GUI behavior: define the view, the constraint system, and the data bindings. From these specifications, shown

Class:  C 

Volume: m³ **Weight:** kg

Dimensions: cm × cm × cm

Distance: km


Price: \$  USD

Figure 3.6: The shipping form after Generation 3 of Figure 3.4, while activation F_3 is still running.

in part in Section 3.5, a property model derives a GUI implementation which schedules asynchronous computations, remains responsive while the computations are running, and gives notifications of their progress and results. Not only that, but the implementation guarantees that any sequence of edits will always produce the same results, no matter how quickly (or slowly) those edits occur.

3.2 Operations and Commands

To this point, the only editing operation we have considered is the assigning of a new value to a single variable. However, in certain cases more complex editing operations may be required. For example, we may wish to assign values to multiple variables as a single operation, without updating the property model between assignments. We may also wish an assignment to be a modification of a previous value of a variable, e.g., in the case of incrementing a counter. This section examines how we may generalize editing operations so that they may be included in the reactive program. We refer to this generalization simply as an *operation* of the property model.

3.2.1 Operations in the Reactive Program

We define an operation of the property model to be a dataflow function, like a method of the constraint system. As Section 2.2.1 describes, a dataflow function is a computation whose inputs and outputs are variables of the constraint system. Of all the variables in the property model, a dataflow function reads only its inputs and writes to only its outputs,

writing to each output only once. Operations are scheduled for execution in the same manner as methods. Specifically, each operation is represented by a scheduling function whose parameters are promises for input variables and which returns promises for output variables. Once the operation’s input promises are fulfilled, the work of the operation may commence, leading to the fulfillment of its output promises.

Operations differ from methods in their intended purpose. The purpose of a method is to enforce a constraint; after the method has executed, the constraint is assumed to be satisfied. In general, the purpose of an operation is to carry out an action in response to a user event. In particular, an operation does not enforce a constraint; after an operation has executed, constraints that use the output variables of the operation are assumed no longer to be satisfied after the operation. In other words, the output of an operation is an edit of the property model. Thus, the constraint system must be solved immediately after scheduling an operation.

As shown in Figure 3.3, the operation and the methods scheduled by the constraint system constitute a transition from one consistent state to the next. The construction of this transition is atomic. Once an operation is accepted by the property model, no other code that could affect the property model may execute until the operation has been scheduled, a new plan for the constraint system has been calculated, and the necessary methods of that plan have been scheduled. Once the transition is complete, the property model is ready to respond to new events.

An operation may serve many purposes in a property model. For example, we can represent a variable assignment as a constant operation—i.e., an operation that always yields the same output—with no inputs and a single output. We can set multiple variables at once using an operation with multiple outputs. We can also perform incremental updates by reading a variable’s value as input, altering it, and writing it back. (This requires accessing the prior value of a variable, which is discussed in Section 4.1.) Other variations are possible as well, such as using the value of one variable to decide what to write to another variable.

Because the scheduling of an operation is atomic, operations themselves may be considered atomic: it is impossible that between the reading of the inputs and the writing of the outputs some other process may alter any variables. Scheduling the operation immediately results in the output promises being assigned to their variables. Therefore, any subsequent alterations will be forced to come after the output of the operation in the promise history.

Not all operations are editing operations: we may create an operation with no output variables. Such an operation would presumably use the values of its inputs to perform some task unrelated to the GUI—e.g., submit data to a server. By performing this task as an operation we ensure that the values of all inputs to the operation are taken from the same consistent state. In fact, as a rule we restrict the reading of variable values by code to operations. In this way, we can be sure no code will make the mistake of using inconsistent variable values. In terms of state transitions, an operation with no outputs is considered a “self-loop”, i.e., a transition from a state back to itself.

As an example of operations, let us return to the shipping form of Figure 2.1. As presented in Chapter 2, a *set* operation is invoked by a data binding every time the user changes an element of the View bound to a variable. Because these operations happen so frequently, rather than generating unique labels for each one, any *set* of a variable is labeled as ε (for “edit”) in the reactive program graph. These operations have no inputs and a single output.

Let us add to the shipping form example an operation that submits the form to a server for processing, perhaps to place a shipping order. We will label this operation *J*. Assume the only values required by the server are the shipping class and distance. Thus *J* is an operation whose inputs are *c* and *d*; it has no outputs.

Furthermore, as an example of an operation with both inputs and outputs, let us add to the shipping form a “distance calculator” used to calculate the distance between two U.S. cities. The GUI for such a calculator can be seen in Figure 3.7. It contains drop-down lists for selecting the departure and arrival cities. We may add to the property model variables *l* (for “leaving”) and *a* (for “arriving”) to hold these values. Clicking the “Calculate” button

causes the application to calculate the distance between the selected cities and assign the result to the distance variable d of our existing property model. If we label the operation that performs this as κ , we may describe it as $d \leftarrow \kappa(l, a)$.

Figure 3.7: The GUI for a distance calculator. Clicking the button causes the application to calculate the distance between the selected cities.

Note here the difference between an operation and a constraint. A constraint between l , a , and d would define an invariant requiring that d *always* equals the distance between l and a . This is not the desired behavior; the user may choose to ignore l and a and edit d directly. An operation allows us to enforce the dependency between l , a , and d only upon request.

Figure 3.8 shows the reactive program graph for a hypothetical execution of this GUI. As with Figure 3.4, Generation 1 is created as the property model initially solves the constraint system to create a consistent state. Generation 2 is created in response to a modification of the arrival city by the user, resulting in the executing of a *set* operation. Because there are no constraints using a , executing this operation has no effect beyond updating a .

Generation 3 is created when the user clicks the “Calculate” button, thereby executing operation κ . This operation reads the current values of l and a and outputs a new value to d . The property model responds to this edit by solving the constraint system, resulting in fresh activations of methods G , E , and C .

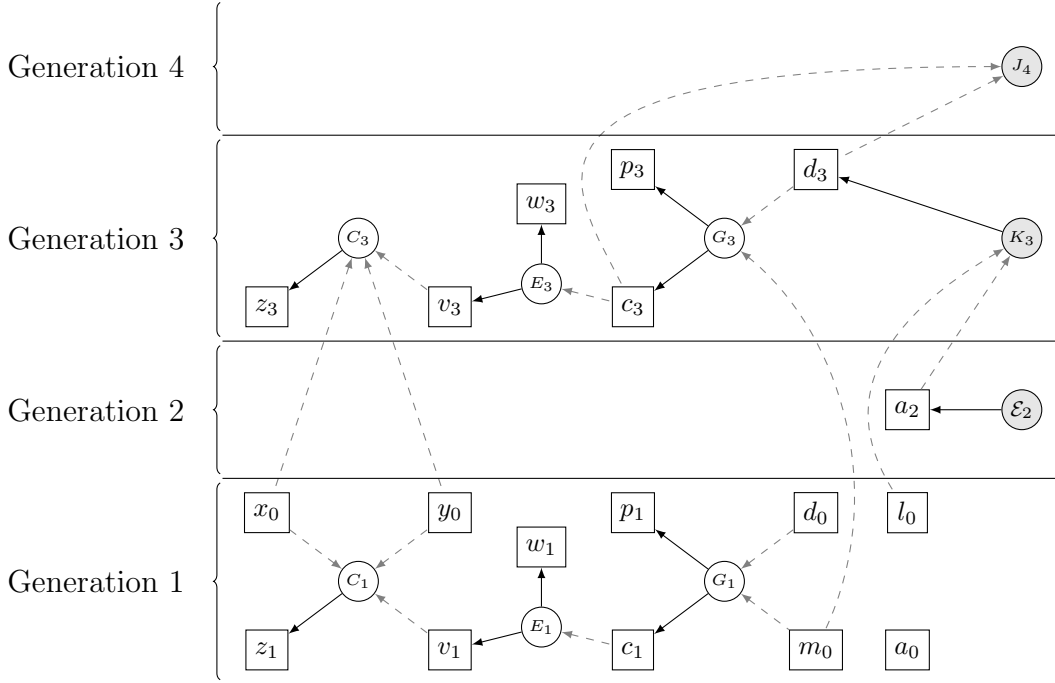


Figure 3.8: The reactive program graph showing operations in the shipping form example. Rectangles represent promises; circles represent activations: white circles are method activations, shaded circles are operation activations. Nodes from the same generation are grouped together in generations, which are ordered by time along the y -axis.

Generation 4 is created when the user submits the form, thereby issuing operation J . This operation reads the current value of variables c and d . However, it has no effect on the variables property model. Thus, the state immediately after Generation 4 is identical to the state immediately after Generation 3.

The execution of operation J in Generation 4 receives as input the *promises* for c and d from the previous generation. These promises may still be pending: both K and G are likely to involve asynchronous server calls which may need time to complete. However, if either of these promises are pending, J will wait until they are ready; it will *not* attempt to use the current values of c and d . We may contrast this behavior with the auto-complete text box discussed in Section 1.1, which generated inconsistent behavior when an operation was submitted before all dependencies had been updated.

3.2.2 Operations as Commands

Methods of the constraint system are scheduled automatically as the property model solves the constraint system in response to an edit. However, operations are not scheduled unless requested. Such a request is made through an additional property model construct known as a *command*.

A command is simply a function which can be invoked in response to an event. As such, it is similar to an event handler in the traditional event-driven programming model. However, whereas a traditional event handler generally represents a response to a View-related event, such as a key press or a mouse-button click, a command represents a response to a View-Model-related event. These View-Model events are defined by the programmer, and represent actions which may be taken by the user. For example, the *set* command for a variable represents the ability of the user to edit that variable.

A command can execute arbitrary code. For example, Chapter 4 discusses how we can use commands to alter the property model itself, adding or removing variables and constraints. However, we require that any interaction with variable values take place inside an operation: only an operation may read from or write to variables. Further, we require that scheduling of an operation be the last thing a command does, to avoid any structural changes to the property model after the operation has been scheduled. To enforce these requirements, we define the return value of a command to be an operation. The operation returned by a command will immediately be scheduled by the property model; this is the only means of scheduling an operation. Of course, a command may be a constant function, in which case it is represented simply as an operation.

We refer to View-related events as *user events* and to View-Model-related events as *logical events*. One way to define a *binding* is as a translation from user events to logical events—mapping, e.g., mouse moves and key presses to actions in the View-Model. Thus, a full description of an edit would be as follows. First, the user performs some input action, such as a key press. A binding translates the user event into a logical event: a set command for a variable. The command itself does nothing but return an operation with

no inputs and a single output. This operation is scheduled, causing the constraint system to be solved and corresponding methods to be scheduled. This whole process is atomic.

We may now more accurately describe the life of a property model as a sequence of commands. The guarantee provided by a property model is that *any sequence of commands will generate the same reactive program, regardless of the timing of those commands*.

3.3 Detecting Unreachable Program Elements

Up to this point we have described a property model as if it stored every promise of a promise history and every activation of an activation history. This is unnecessary, as once a promise is no longer the last promise in the history, it is no longer reachable and therefore no new tasks may be scheduled for it. It is also undesirable as it would result in a linear growth in program memory. Therefore, we consider here when elements of the reactive program may be released and their resources collected.

We define a program element (i.e., a promise or activation) to be *live* when its value (if a promise) or execution (if an activation) can affect the GUI, and *dead* otherwise. As mentioned previously, commands may have side effects which affect program state; for this reason, we must always assume commands to be live. However, the methods of the property model do not have side effects; the only external effect they have is fulfilling output promises. Promises themselves may affect program state only if their values are propagated by bindings to the view. We define a promise to be *visible* if, when fulfilled, its value may be propagated to the view. A promise is visible when all more recent promises in the same promise history are still pending.

Just because a promise is not visible does not mean it is dead; it may still have dependencies which are visible. We can use the reactive program graph to trace all the dependencies of a program element. This leads us to the following definition: a program element is live if and only if there is a path in the reactive program graph from that element to a command or a visible promise.

Promises which are dead may be discarded. Activations which are dead may be *un-scheduled* so that they will no longer respond to the fulfillment of their input promises,

then discarded.¹ Thus, while new program elements are added to the “top” of the reactive program graph in response to edits, old program elements may be removed from the “bottom” of the graph after they transition from live to dead. In this way, memory usage can be generally kept constant. Note that, in practice, it is not actually necessary to perform graph searches to determine dead elements. Our implementation uses a simple reference counting scheme in which promises keep track of their subscribers; when a promise loses all subscribers it dies and may be discarded.

There are still two ways in which memory usage by a property model may grow unchecked. The first may occur when there are methods which run exceedingly slow. Our approach to binding requires that promises be preserved until a more recent promise has been fulfilled. Thus, repeated invocations of slow methods could add new unfulfilled promises without rendering any old program elements dead. In general, if methods are taking so long to complete that this is an issue, it seems likely there are other problems the GUI needs to address. However, if desired we could eliminate this potential memory leak with an alternate binding approach in which only the most recent promise in each history is visible. In this way, even adding unfulfilled promises will result in dead program elements that may be disposed.

The second way in which memory may grow unchecked is if there are methods which begin execution and never terminate, even when they die. This is a drawback of our decision not to have the property model directly involved with offloading work to, e.g., other threads: because the property model is unaware of them, it cannot halt them. The best we can do in this case is allow a method to detect when it dies. Our promise implementation defines an event that occurs when the promise dies. Thus, for example, a method which intends to initiate work on a separate thread may register a callback for this event with its output promises. If all output promises are dead, the method itself is dead, and may thus go about halting the thread so that its resources may be released. This task is well suited to

¹Note that not all promise implementations support unsubscribing; we use our own promise implementation which does.

a library implementation; however it is outside the scope of the property model.

3.4 Failure in an Activation

A method activation executes arbitrary user code, and thus is subject to, e.g., resource allocation failures, network failures, exceptions, etc. We can equate all these different error states with a failure to terminate, since the end result is the same: the activation's output promises are never fulfilled. From the property model's perspective, a promise which is pending behaves as if it is in an error state. No computations which depend on a promise may proceed while it is pending. Therefore, the outputs of an activation will be pending as long as any of its inputs are pending. If a method activation fails to terminate, its output promises remain stuck in this error state forever.

If a variable's most recent promise is stuck in a pending state, then the variable itself will be as well. To recover from this error state, a variable must be provided with a new promise that can be fulfilled. For example, Figure 3.9 shows a representation of the reactive program graph from Figure 3.4 in which activation G_1 fails. Because of this failure, p_1 and c_1 will remain pending forever, as will their dependencies. After Generation 1, p , c , w , v , and z are all stuck in the pending state.

In Generation 2, we make an edit to variable w , resulting in promise w_2 . Assuming this promise is fulfilled, w is no longer pending. This also makes w_1 invisible and (because the only promise reachable from w_1 is itself) dead. This generation has new activations of F and I . However, because F_2 must wait on v_1 , and I_2 must wait on c_2 , these activations never begin execution. Thus, variables p and c remain stuck in the pending state, as do v and z , which are unchanged from the previous generation. Not only that, m becomes stuck in the pending state as well. This shows how the pending state naturally spreads: any variable whose value depends on a pending variable becomes pending itself.

In Generation 3, we make an edit to variable v , resulting in promise v_3 . Again assuming this promise is fulfilled, activations C_3 and F_3 have no pending inputs and they may execute; once F_3 finishes, I_3 may execute as well. If these three activations complete successfully, then all pending variables will have their most recent promise fulfilled, meaning they will no

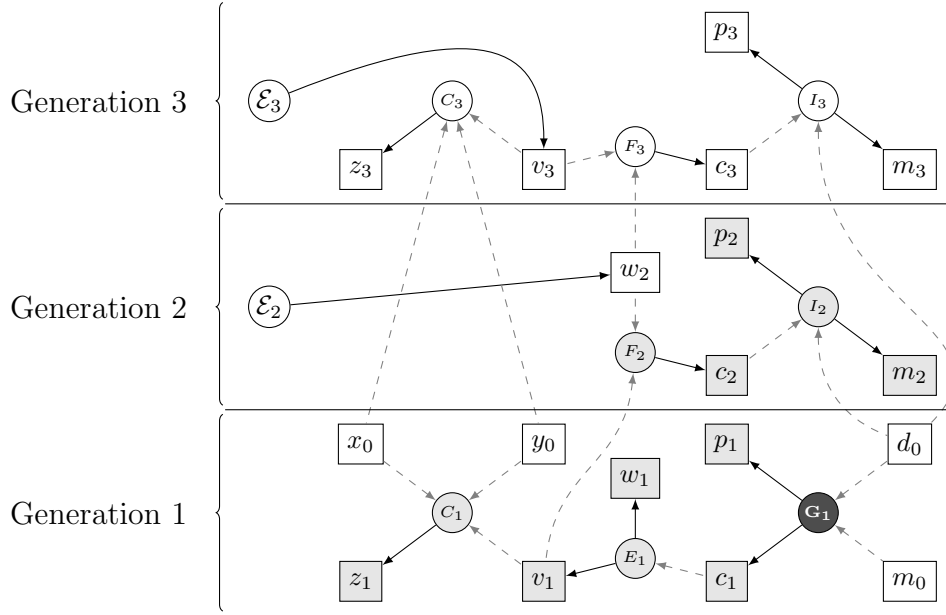


Figure 3.9: Activation failure in the reactive program. Rectangles represent promises; circles represent activations. The black circle represents an activation failure. The shaded rectangles represent promises that will never be fulfilled. The shaded circles represent methods that never run due to pending inputs.

longer be pending. Also, promises z_1 , v_1 , p_1 , p_2 , c_1 , c_2 , m_0 , and m_2 will become invisible. A graph search will show that they are also dead, as are activations C_1 , E_1 , G_1 , F_2 , and I_2 . Of these, only G_1 had begun execution; the remaining activations may simply be unscheduled.

At this point, all effects of the failed activation have been covered up. There are no longer any pending promises or scheduled activations. The state of the property model is exactly the same as it would have been if G_1 had succeeded. We were able to recover from the error state *without* re-executing G , the method that caused the failure. This illustrates the resilience of property models: even if a method is poorly written so that it at times crashes or never terminates, the property model continues to operate, and the GUI it governs stays responsive and in a well-defined state.

We can make the following guarantee about the consistent behavior of a property model: *a property model is deterministic upon success*. Here is what this means. Suppose we have a property model in which any method may fail non-deterministically, and we

apply a sequence of editing operations. There may be many possible outcomes for the property model depending on which method activations succeed and which ones fail. For any variable, there may be outcomes in which the variable is fulfilled (and thus its current value is known), and others in which it is stuck as pending (and does not have a current value). However, in every possible outcome in which the variable is fulfilled, it will have the same value.

Although *forever-pending* serves well as an error state for the property model, it may be helpful to the user to differentiate between a pending state and an error state. A GUI should provide notification when a method has failed so that the user can stop waiting for a value, and start to take action to correct the error. To enable such notifications, a method may indicate failure by rejecting its output promises; a rejected promise is treated as a pending promise that will never be fulfilled. If any input promise of a method activation is rejected, the activation rejects all of its output promises, thus spreading the error state the same way that the pending state spreads.

We call variables whose current promise has been rejected *stale*. Stale variables are in an invalid state and will require a new value before they can be used as input to any methods. To assist Views with indicating when a variable is stale, we define a property named *stale* which is true exactly when the variable is stale. This property is suitable for binding to the View; thus, we may alter the appearance of elements bound to stale variables, just as we did elements bound to pending variables in Figure 3.6.

When a method rejects a promise, it may provide—as an alternative to the promise value—an error value. In stale variables, the *error* property contains the error value with which the current promise was rejected. This property can also be bound to the View, providing a mechanism by which a method can communicate the cause of failure. By reflecting variables' *stale* and *error* properties, rich feedback on errors is easy to automate.

3.5 Implementing Property Models with HotDrink

The above sections describe how a property model implements the View-Model of a GUI. To give a clear picture of how a programmer might define and use a property model,

we present an implementation of the shipping example of Figure 2.1 using HotDrink, our JavaScript property models library.

To implement GUI behavior with HotDrink requires three basic steps: defining the View, defining the View-Model, and defining the bindings between the two. The View is implemented by the HTML Document Object Model (DOM), and is usually defined using HTML. The View may also include JavaScript code that creates or manipulates elements of the DOM. The View-Model is implemented as a property model, and is defined by a specification of the variables, constraints, and commands which compose it. Bindings connect the variables and commands of the property model to elements of the DOM. Bindings may be written in JavaScript or embedded in the HTML defining the View.

We first examine the process of defining the property model, beginning with method implementation. Figure 3.10 shows the code for three of the methods of the constraint system, D , F , and H , following the method definitions given in Section 2.2.1. So that it easy to match the code with the method definitions, the three functions are simply named D , F , and H . The parameter names match the names of the corresponding variables, though this is not required.

```
1 function D(x, y, z) { return x*y*z; }
2
3 function F(v, w) {
4   var p = new hd.Promise();
5   performShippingClassQuery(v, w,
6     function(c) { p.resolve(c); });
7   return p; }
8
9 function H(c, m) {
10  var p1 = new hd.Promise(), p2 = new hd.Promise();
11  performShippingRateQuery(c,
12    function(r) {
13      var d = Math.floor(m/d);
14      p1.resolve(d); p2.resolve(r*d); })
15  return [p1, p2]; }
```

Figure 3.10: The JavaScript definitions for three methods of the shipping form constraint system.

As explained in Section 3.1.2, HotDrink uses a “lifting” mechanism to convert functions over values to functions over promises, thereby creating the scheduling functions needed for these methods. This avoids boilerplate code for methods that do not perform any asynchronous computations, allowing them to be written as functions whose inputs and outputs are values; for example, method *D* is written on line 1 as a function whose inputs and output are numbers.

The lifting mechanism works for functions whose inputs and outputs are any mixture of values and promises. The function *F* on line 3, for example, takes values as inputs but returns a promise. It uses an auxiliary function named `performShippingClassQuery` (not shown) to look up a shipping class for the given volume and weight, presumably using an Ajax call. Once the Ajax call returns, the provided callback fulfills the output promise, thus completing the method’s duties.

The function *H* on line 9 also takes values as inputs and returns promises as outputs. This function shows our convention for returning multiple return values using arrays. Like function *F*, function *H* uses an auxiliary function to perform an Ajax call, this time to retrieve the shipping rate for the given shipping class. It then uses that value to calculate the maximum distance that can be shipped for a price less than *m*, and the actual price of the shipping.

Figure 3.11 shows how we define the various elements of a property model. These are created as fields in a special object called a *component*. Components are discussed in detail in Chapter 6. Briefly, a component of the property model serves as a container of property model elements such as variables and constraints, and also as a namespace, mapping names to its members. Components are generally created using a `ComponentBuilder` factory object. This temporary object is created in line 1. Its member functions are invoked on lines 2–15. The call to `component` member function on line 16 finalizes the construction.

`ComponentBuilder`’s members define an embedded DSL for creating elements of a property model. The `variables` function on line 2 creates the property model’s variables; the parameters to this function are the variable names and a map of initial values. Variables are

```

1 var model = new hd.ComponentBuilder()
2   .variables("d, c, m, p, v, w, x, y, z, l, a",
3     {x: 25, y: 50, z: 40 w: 10, d: 1500})
4   .method("v, y, z → x", A)
5   .method("v, x, z → y", B)
6   .method("v, x, y → z", C)
7   .method("x, y, z → v", D)
8   .constraint("v, w, c")
9   .method("c → [v, w]", E)
10  .method("v, w → c", F)
11  .constraint("d, p, m, c")
12  .method("d, m → [c, p]", G)
13  .method("c, m → [d, p]", H)
14  .method("d, c → [m, p]", I)
15  .command("K", "l, a → d", K)
16  .component();
17
18 var pm = new hd.PropertyModel();
19 pm.addComponent(model);
20 pm.update();
21
22 window.addEventListener("ready", function() {
23   hd.createDeclaredBindings(
24     component, document.body); });

```

Figure 3.11: JavaScript that creates the property model for the shipping form example with HotDrink.

initialized in the order declared, which is also the initial priority ordering of the variable's stay constraints. To create a constraint, `constraint` is first called to establish the constraint's variables, then repeated calls to `method` define its methods. Creating a method requires a signature that defines the method's inputs and outputs, and the method's function. Similarly, a command is defined using the `command` function with a name for the command, a signature, and the command's function.

Once the component is created, it is added to the property model, as shown on line 19. The call to `update` on line 20 enforces all constraints, creating the first generation of the reactive program. Finally, the property model's variables are bound to the View. In a web application, elements of the view are not available until the Document Object Model, has been built. Thus, line 22 registers a function for callback when the DOM is ready. This

```

1 Class :
2 <select data-bind="bd.value(c)" >...</select>
3
4 Volume :
5 <input type="text" data-bind="bd.num(v)"/>
6
7 Price :
8 <input type="text" data-bind="bd.num(bd.rw(p, m)),
9   bd.cssClass(p.pending, 'pending'),
10  bd.cssClass(p.stale, 'stale')"/>
11
12 <span data-bind="bd.text(p.error)"></span>

```

Figure 3.12: Example HTML containing binding declarations for the shipping form example.

function creates the data bindings by calling `createDeclaredBindings` with two arguments: a component and a node of the DOM—here, `document.body` representing the entire body of the HTML document. `HotDrink` searches the contents of the specified node, looking for HTML elements containing binding specifications. It then attempts to bind according to found specifications, using the given component to look up any names it encounters.

Implementing the View is done entirely apart from `HotDrink`, and it is therefore beyond the scope of this document. However, to illustrate binding specifications, Figure 3.12 shows excerpts of the HTML that creates the View of the shipping form; the full HTML is a bit long to include. Binding specifications are given as `data-bind` attributes of tags; their value is JavaScript code that customizes the binding for the tag.

`HotDrink` contains several functions for specifying common bindings; we use some of these in Figure 3.12. The call to `bd.value` on line 2 specifies that the variable `c` should be bound to the value of the select box widget; the call to `bd.num` on line 5 specifies that the variable `v` should be bound to the value of the text box widget, and that the string representation of the value should be converted to a number in the property model. (Bindings can include operations such as data conversion, formatting, and validation.) The call to `bd.cssClass` on line 9 specifies that the CSS class `pending` should be added to the

element whenever the *pending* property of the variable `p` is true. The call to `bd.text` on line 12 specifies that `p`'s *error* property should be bound to the contents of the `span` element.

As can be seen on lines 8–10, a tag can have multiple binding specifications, allowing it to reflect multiple values. This particular binding is for the *price* text box, which reads from the variable `p` but writes to the variable `m`. Rather than creating two separate bindings, we bind to a construction, created by the `bd.rw` function, that will read from `p` and write to `m`. The `bd.num` function again specifies the need to convert between a string representation in the View and a numeric representation in the property model.

The code in Figures 3.10, 3.11, and 3.12, along with the method definitions and HTML that were not shown, are a complete implementation of the shipping form example using HotDrink.

4. EXTENSIONS TO THE CORE PROGRAM*

The previous chapter describes the fundamental operation of a property model and how it threads methods and operations together into a single reactive program. This section describes several additional features that extend this basic model. These features have been developed to address specific issues that have arisen while using property models.

4.1 Accessing Prior Values

As stated previously, the life of a property model proceeds like a state machine, transitioning from one consistent state to another. That transition begins with a operation making changes to variables of the property model, and is completed with the constraint system enforcing constraints so that a consistent state is created. As in any state machine, transitions between states depend both on the input—in this case the values provided by a command—and the current state of the property model. We have already encountered one way in which the previous state affects the next: inputs which are not also outputs of the next generation are taken from the previous generation. This mechanism can be generalized: a method can use the values of variables from the previous generation even when those variables are outputs of the next generation.

While creating a new generation, we make the following distinction: the *prior value* of a variable is its value *before* the new generation; the *current value* is its value *after* the new generation. In cases where the new generation makes no changes to a variable, the variable’s prior and current value are the same. Generally, a method activation is passed the current value of each of its inputs. However, some methods may need to access the prior

*Portions of this chapter are reprinted with permission from “Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems” by Gabriel Foust, Jaakko Järvi, and Sean Parent. *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pp. 121–130, Copyright 2015 by ACM. <http://dx.doi.org/10.1145/2814204.2814207>

Portions of this chapter reprinted with permission from “Responsive and Consistent User Interfaces with Multi-Way Dataflow Constraint Systems” by Gabriel Foust, Jaakko Järvi, and Sean Parent. Under review for inclusion in *Computer Languages, Systems and Structures (COMLAN)*, Elsevier.

values of inputs even when the variable is updated by the new generation. For example, a method may wish to access the prior and current value of a variable to calculate the difference between the two; or a method may wish to access the prior value of one of its outputs to use as a “hint” in creating the output.

Methods in a property model can specify whether they want the current or prior value of each input variable, and they will be passed the appropriate promise when the method is scheduled. The reactive program graph gives a clear interpretation for current and prior values. The current value is always represented by the current promise (the topmost element of the promise stack); the prior value by either the promise immediately below the current promise, if the variable is written to in the new generation, or the current promise, if it is not. In the latter case, the current and prior values are represented by the same promise.

As an example of a constraint accessing a prior value of a variable, consider the shipping form example shown in Figure 2.1; specifically, the constraint between the volume, width, height, and length of a package. In our previous specification of this constraint (see Section 2.2.1), setting volume resulted in an update to just one dimension with the other two dimensions were used as inputs. An alternative formulation of this constraint might distribute a change in volume proportionally among all three dimensions. This new constraint consists of the old method $v \leftarrow xyz$, which we previously labeled D , and a new method $J : (x, y, z) \leftarrow (rx', ry', rz')$ where x' , y' , and z' are the previous values of, respectively, x , y , and z , and $r = \sqrt[3]{v/(x'y'z')}$.

Because no method may output to a prior value, the use of prior values by a method has no effect on the plan of the constraint system, nor on the valid execution orders of a plan. Thus, we add no edges to the constraint graph or the solution graph for uses of prior values. However, we do add edges to the reactive program graph between promises for prior values and the activations that use them, representing additional data dependencies. These dependencies are identical to dependencies for current values: they restrict the method from executing until the promise is fulfilled, and provide paths by which program

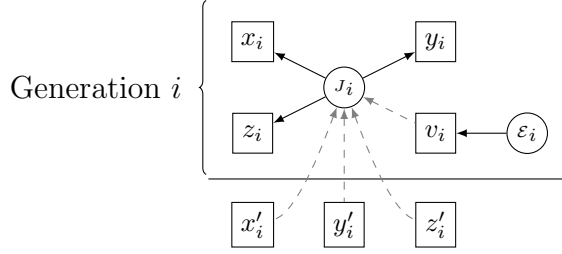


Figure 4.1: Reactive program graph for a constraint that uses prior values. Nodes x'_i , y'_i , and z'_i represent the values of, respectively, the variables x , y , and z prior to Generation i .

elements may be considered relevant. Figure 4.1 shows the reactive program graph for an activation of our new method, L . We do not specify the generations that x'_i , y'_i , or z'_i come from; we know only that they are from some prior generation.

Figure 4.2 illustrates the use of prior values in HotDrink. Figure 4.2a shows the implementation for method J : a function which takes four values and returns (an array of) three. This function gives no indication that any of its parameters are prior values. To see that they are, we must look to the method signature. Figure 4.2b shows an excerpt from a property model definition—the same property model from Figure 3.11 except that methods A , B , and C have been replaced with method J from Figure 4.1. The exclamation marks in the signature mark these parameters as being prior values. Otherwise, the method definition follows the same pattern as the ones we have seen previously.

The use of prior values enables the programmer to express a richer class of constraints. For example, suppose we have variables n and t , representing, respectively, a numeric value and an incrementing timer. Assume these variables are set together to indicate the value of n at time t . We may define a constraint which calculates i , the integral of n , like so: $i \leftarrow i' + n'(t - t')$. However, it should be noted that the use of prior values introduces synchronization into the system: a method that depends on a prior value cannot execute before the prior value has been computed. Also, the use of prior values can lead to cascading failures, in which failure in one generation leads to failure in the next. We will examine this problem, and a possible solution, in Section 4.4.

```

function J(x, y, z, v) {
  r = Math.cbrt(v/(x*y*z))*100;
  return [r*x, r*y, r*z]; }

```

(a) Method implementation

```

var model = new hd.ComponentBuilder()
  :
  .constraint("v, x, y, z")
    .method("!x, !y, !z, v → [x, y, z]", J)
    .method("x, y, z → v", D)
  :
  .end();

```

(b) Property model definition

Figure 4.2: Implementation of prior values in HotDrink.

4.2 Managing In-Place Modifications

Up to this point, we have assumed a *value semantics* for our reactive program: variable values are copied to the method, return values are copied back. In some cases, however, value semantics is impractical. For example, a variable in an image-processing application may contain data for a photographic image; such data may be large enough that we wish not to make frivolous copies of it. In such cases, it may be desirable to simply modify the variable’s existing value rather than produce a new one. We refer to this as an *in-place* modification of the variable.

The ability to reference prior values offers a simple solution to in-place modifications. Rather than storing a value in a variable, we store a reference to a value. That reference is provided to a method as an input; the method modifies the value being referenced, then returns the same reference. This is a natural extension to our existing property model semantics, especially in languages which use reference semantics (e.g., JavaScript). However, in order to allow in-place modifications and still guarantee correctness, we must make sure that no method is allowed to modify a value until all other methods that rely

on its current value have completed.

To accomplish this, we create an extra dependency in the reactive program graph that we refer to as a *barrier*. A barrier may be described as an activation of a dataflow function that takes one or more parameters and always returns the first one. In this case, we are using the dataflow function, not to perform any computation, but for its synchronizing effect: it effectively “holds” the first parameter until all other parameters are ready. We refer to this first parameter as the *target* of the barrier.

As an example, we revise the constraints from the shipping form constraint system of Figure 2.2a so that the package class variable c is an in-place variable. Figure 4.3 shows an example reactive program graph from this system. The graph contains only the constraints that contain c . Notice that we have altered the methods which output to c so that they also read the prior value of c in order to modify it.

Generation i schedules an activation of method F which writes to c . When preparing to schedule a dataflow function which could modify a variable in-place, we create a single barrier targeting the current promise for that variable. In this case, we create the barrier B_i targeting the prior value of c , represented in the figure by c'_i . The other inputs to this activation are any promises that depend *directly* on the target—that is, any promise whose value is calculated by reading the value of the target. The output of the barrier, \hat{c}_i , is used as the input for F_i . Although the value of promise \hat{c}_i will be identical to that of c'_i , the promise itself will not be fulfilled until all other inputs to B_i have been fulfilled. This ensures that no modifications will be made to the value referred to by c'_i until all direct dependencies are resolved.

When a barrier is created, we need to know which promises other than the target should be inputs. To this end, for each variable that may be modified in-place we keep a list of promises we call the *dependency list*. The promises of the dependency list are those that depend directly on the current promise for the variable. Every time the current promise is used as an input to an activation, the output promises for that activation are added to the variable’s dependency list. For example, in Generation i , c_i is used as input to an activation

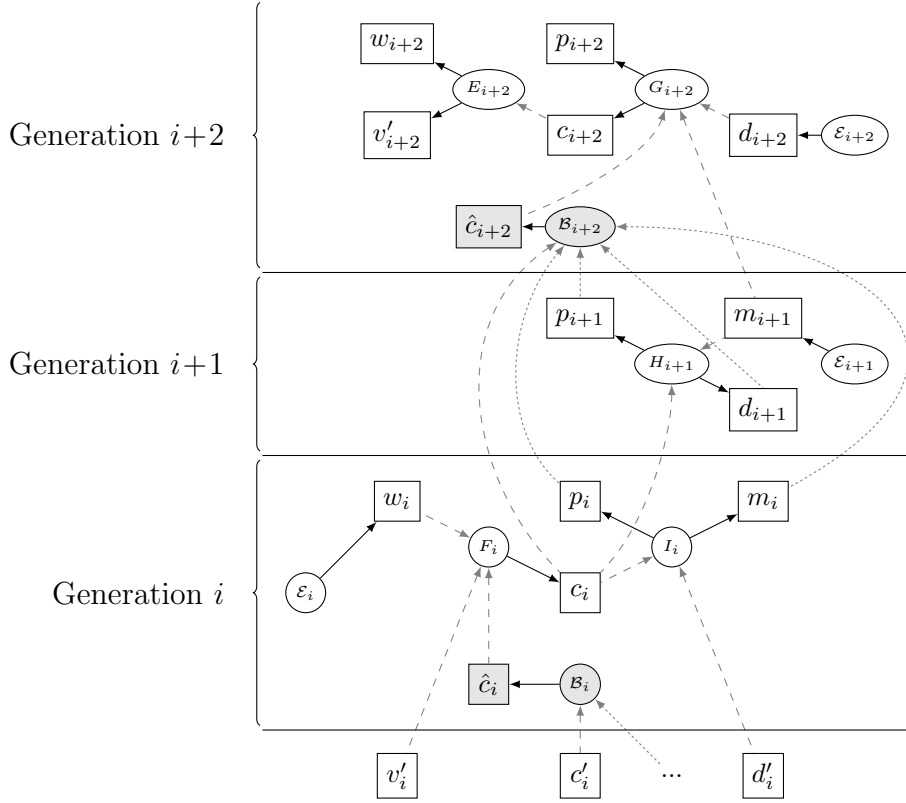


Figure 4.3: Reactive program graph using barriers. Barriers and their outputs are shaded.

of I ; therefore the outputs of that activation, p_i and m_i , are added to the dependency list for c . Similarly, in Generation $i+1$ activation H_{i+1} uses c_i as input, so promises p_{i+1} and d_{i+1} are added to the list.

Generation $i+2$ contains an activation of a method, this time G , that writes to c ; therefore we must create a new barrier for c . This barrier takes c_i as its target, and all promises on the dependency list for c —that is, p_i , m_i , p_{i+1} , and d_{i+1} —as additional inputs. Its output, \hat{c}_{i+2} , is used as the input to activation G_{i+2} . This activation outputs a new promise for c . Whenever a new promise is created for a variable, its dependency list is reset.

The drawback of in-place modifications is that barriers introduce more synchronization. They require waiting not only for the target to be calculated but also for all promises which

depend directly on the target. So that these dependencies are not unnecessarily added to the reactive program graph, we require that variables which may be modified in-place be identified when defining the property model. We call such variables *in-place* variables.

Once a programmer indicates a variable as in-place, the property model handles all aspects of creating and enforcing its barriers. When an activation uses an in-place variable as input, all outputs of the activation are added to the variable's dependency list. Any method that reads the prior value and writes the current value of the variable is assumed to perform an in-place modification. This is a conservative assumption; the property model does not examine the code of the method to see whether an in-place modification actually occurs. When such a method is scheduled, if the variable's dependency list is non-empty, a barrier is created as described above.

With the exception of the target, a barrier does not actually use the value of its inputs; it needs the promises only to determine when all computation directly depending on the target has finished. For this reason, the barrier does not distinguish between these promises being fulfilled or rejected: in either case the computation using the target is no longer running. Furthermore, the edges corresponding to these inputs are not used to determine whether program elements are live. If one of these input promises becomes irrelevant, it is simply removed as an input to the barrier. In Figure 4.3 the edges corresponding to these inputs are drawn as dotted lines instead of dashed.

It is possible for an in-place variable to be used in methods that do not modify it in-place. As mentioned above, if a method reads the current value of the variable, any output promises of that method are added to the dependency list. If a method outputs to the variable without reading its prior value, that method must create a fresh value for the variable; therefore, no barrier is created. As usual, when the method's output promise is generated, the dependency list for the variable is reset. This is appropriate, since there will be no in-place modification of the prior value of the variable.

If a method reads the prior value of the in-place variable without writing to it, behavior varies depending on where the method lies with respect to the dataflow of the current plan.

If the method lies downstream of some method that writes to the in-place variable, it is assumed that the method generates a fresh value for the variable; therefore, no action is required.¹ If the method does not lie downstream of some method that writes to the in-place variable, then the use is treated like a use of the current value, i.e., all output promises of the method are added to the variable’s dependency list.

A problem can arise if a plan contains one method which reads the prior value of the variable, another method which performs an in-place modification of the variable, and the two methods are not topologically ordered by the plan. The problem is that the method which reads the prior value must be executed before the method which performs the in-place modification. Enforcing this execution order requires that extra dependencies be added to the solution graph; in theory, these dependencies could cause cycles in the solution graph. We currently have no solution for this, but note that the situation cannot arise if the only methods which read the prior value of an in-place variable are those that modify the variable in-place.

4.3 Optional Constraints

As discussed in Section 2.2.2, stay constraints are considered optional constraints: they may or may not be enforced. Which stay constraints are enforced depends on the current constraint hierarchy: the constraint system attempts to enforce stay constraints that are higher in the hierarchy. This ability for a constraint to be optional is useful for constraints in general, not just for stay constraints.

As an example, consider again the “Distance Calculator” shown in Figure 3.7. This calculator contains a potential dependency between the city from which the package leaves, l , the city at which the package arrives, a , and the shipping distance, d . As presented in Section 3.2, the user must click the “Calculate” button to indicate that the dependency between variables l , a , and d should be enforced.

Instead of using a button, we can use the constraint hierarchy to determine whether

¹If the method performs an in-place modification of the variable, the property model is invalid. It is impossible to read the prior value of a variable after an in-place modification.

this dependency should be enforced. If the user edits a or l , we assume that we should enforce the dependency to calculate d . However, if the user edits d directly, we should not enforce the dependency and simply take the value entered by the user. To specify this behavior, we make the constraint between l , a , and d optional and add it to the constraint hierarchy. Whether or not the constraint is enforced depends on its relation to the other constraints in the hierarchy.

Stay constraints are automatically promoted to the top of the hierarchy whenever their corresponding variables are edited. Other optional constraints do not have an automatic-promotion event (though they may still be promoted manually using the *touch* operation discussed in Section 2.2.2.) Most commonly, we want to promote optional constraints in conjunction with stay constraints. For that reason, we allow the defining of *touch dependencies*. A touch dependency is a non-symmetric relationship between two constraints. If there is a touch dependency from constraint c_1 to constraint c_2 (written $c_1 \Rightarrow c_2$), then any time constraint c_1 is promoted, constraint c_2 is promoted just beneath it.

Touch dependencies are considered transitive: a touch dependency $c_1 \Rightarrow c_2$ together with a touch dependency $c_2 \Rightarrow c_3$ imply a touch dependency $c_1 \Rightarrow c_3$. When a constraint is promoted, we calculate the transitive closure of all touch dependencies, then promote those constraints immediately beneath the originally promoted constraint. If the transitive closure includes more than one constraint, the promoted constraints keep the same relative ordering.

As an example of this, consider a hypothetical constraint hierarchy: in order from highest to lowest, (a, b, c, d, e, f) . Now suppose we have defined touch dependencies $d \Rightarrow a$, $d \Rightarrow f$, and $f \Rightarrow c$. If d is touched, the resulting hierarchy would be (d, a, c, f, b, e) . Constraint d is on top as it was the constraint which was touched. Next come the touch dependencies of d —namely a , c , and f —in their relative order from the original hierarchy. Finally are the remaining constraints, also in their relative order from the original hierarchy.

Figure 4.4 shows a reactive program graph for the shipping example (with the distance calculator) in which we replace the command κ with an optional constraint, call it C . This

constraint consists of the single method $d \leftarrow \kappa(l, a)$. Notice that κ is reused as a method here; the only difference between a command and a method is that a method is part of a constraint. Let us add touch dependencies from the stay constraints for l and a to C so that C will be promoted whenever these stay constraints are promoted.

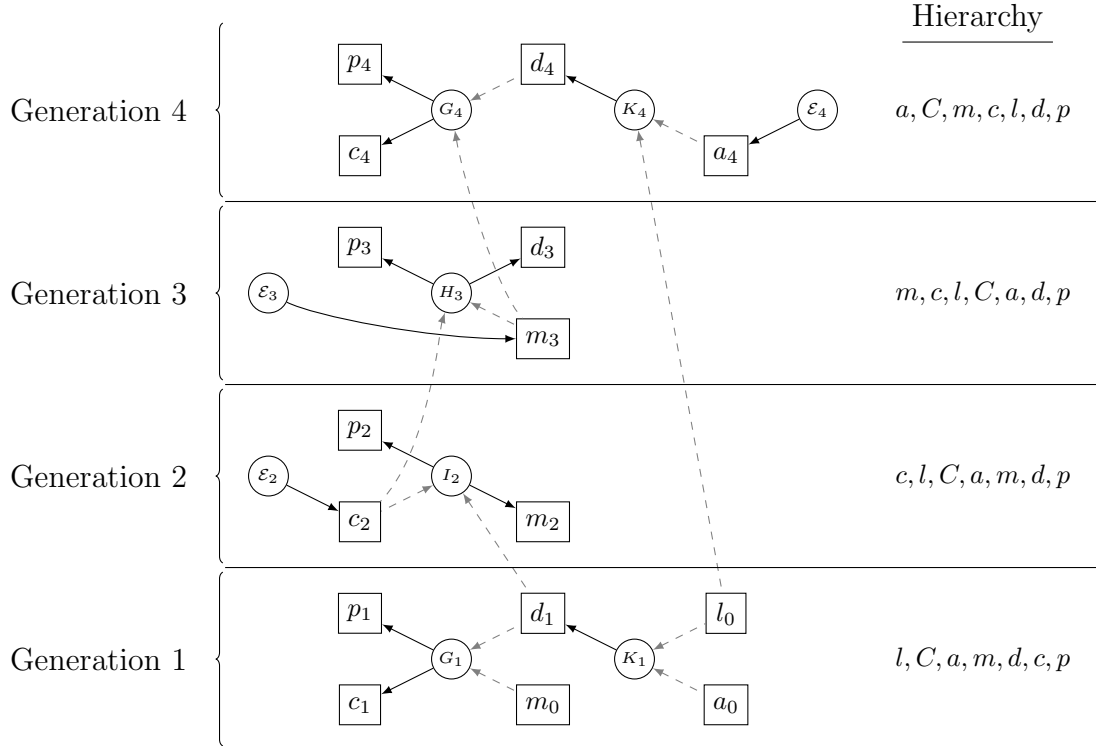


Figure 4.4: A program graph containing an optional constraint.

Generation 1 of the figure begins by solving the constraint system. The constraint hierarchy is shown to the right of the generation, in order from highest to lowest. Stay constraints are referred to by the name of their corresponding variable—e.g., a refers to the stay constraint for a . In Generation 1, C has a higher priority than the stay constraints for d , p , or c ; therefore it is enforced, as evidenced by the inclusion of κ in the program graph.

Generation 2 is the result of an edit to variable c . This edit promotes the stay constraint

for c . However, C is still greater than the stay constraints for p or m ; thus, it remains enforced as it was in Generation 1.

Generation 3 is the result of an edit to variable m . This edit promotes the stay constraint for m . At this point, every possible plan that preserves C leaves out a stay constraint that is greater than C . Thus, the constraint is unenforced. Although the graph does not make this explicit, we can see H writing to d , overwriting the value that came from K .

Generation 4 is the result of an edit to variable a . This edit promotes the stay constraint for a . However, the touch dependency $a \Rightarrow C$ means that C is promoted to just beneath a . Now, C is high enough in the hierarchy to be enforced, resulting in the execution of K .

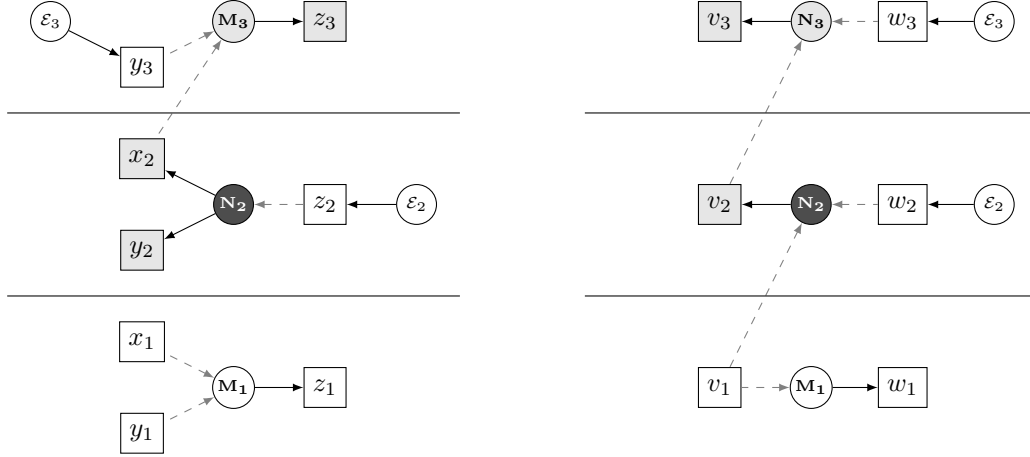
4.4 Promise Forwarding

Section 3.4 discusses how failure of an activation spread through the reactive program graph, leaving all dependent variables in a *stale* state: they received promises for new values, but those promises will never be fulfilled. In order to recover from this error state, a stale variable must be assigned a new promise and that new promise must be fulfilled. This approach has the advantage of making the property model deterministic upon success.

However, the recovery work may include assigning variables the same values they already have, and thus seem odd, or even pointless, to the user. Take as an example the reactive program graph in Figure 4.5a. This graph comes from a property model with three variables, x , y , and z , and a single constraint with two methods, $z \leftarrow M(x, y)$ and $(x, y) \leftarrow N(z)$. In the second generation, a failure in method N results in rejected promises for x and y . This means that *both* of these variables must be given new values in order to recover from this error.

Consider this example from the user's perspective. In Generation 2 the user edits variable z , resulting in a failure. He attempts to correct the error by providing a new value for y , resulting in Generation 3. At this point it may be unclear to the user as to why failure persists: both x and y have valid values, but method M will not execute.

Figure 4.5b shows another example, this time using a self-loop. This graph comes from a property model with two variables, w and v , and a single constraint with two methods,



(a) Failure persists due to multiple stale variables. (b) Failure persists due to a “self-loop.”

Figure 4.5: Reactive program graphs demonstrating how failure may propagate further than expected. The black circles represent failed activations; shaded rectangles stale variables; and shaded circles activations that never run due to stale inputs.

$w \leftarrow N(v)$ and $v \leftarrow M(w, v')$ Here, method M fails in Generation 2, resulting in a stale promise for v . The user attempts to fix this error by providing a new value for w . However, because method N depends on the previous value of v , this method cannot run. The only way to recover from this error is to provide a new value for v .

Both of these examples may be confusing because the View reflects an old value of the variable that cannot be used because the current value of the variable is a stale promise. While it is true that, as discussed in Section 3.4, the View may provide visual feedback indicating that the variable is stale, the concept of a stale variable may be unfamiliar to the user; the user may not understand why he must edit the variable when it already seems to contain a valid value.

We may address this problem by allowing rejected promises, when used as input to a method, to assume the value of the most prior promise of the promise history which has been successfully fulfilled. We refer to such a promise as a *forwarded* promise. For example, in Figure 4.5a the promise x_2 , when used as input by M_3 , would take the value of x_1 , thereby allowing the method to execute. Similarly, in Figure 4.5b the promise v_2 ,

when used as input by N_3 , would take the value of v_1 .

There are two cases when the use of forwarded promises may affect the execution of the reactive program. The first is when a variable goes from being the output of an activation in one generation to being a *source* variable (a variable which is not the output of any activation) in the next generation. This case is illustrated by variable x in Figure 4.5a. The second is when a variable is used as both input and output by the same method. This case is illustrated by variable v in Figure 4.5b.

Forwarded promises, while useful, complicate the programming model; we have yet to determine all the ramifications of this feature when used in combination with other features such as detecting irrelevant variables as discussed in Section 4.6. Further, it is important to note that the use of forwarded promises removes the guarantee of a property model being deterministic upon success. In Figure 4.5a, the use of a forwarded property would allow us to successfully calculate all values in Generation 3; however, the values of variables x and z could be different than they would have been if N_2 had succeeded. We consider it the programmer’s job to determine whether the loss of this guarantee makes up for the improvement in usability. In our current implementation, the property model may be configured to either use or not use forwarded promises; it is possible that in the future this could be further refined as a per-variable option.

4.5 Adjusting Variable Priorities

As mentioned in Section 3.1.3, not every method of the plan must be scheduled for execution. The only methods which must be scheduled are those which were not selected in the previous generation, and those whose inputs have changed since the previous generation. If a method is scheduled, its outputs are considered changed; it follows inductively that all methods downstream of an edited variable in the solution graph will necessarily be scheduled. Ideally, these would be the only methods scheduled: these are the methods affected by the edited value. However, if the constraint hierarchy (described in Section 2.2.2) is determined solely by editing order, it is possible that other methods may require scheduling as well.

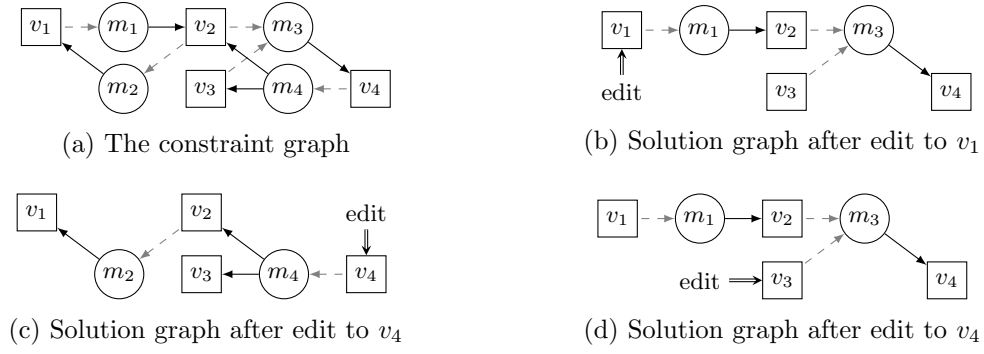


Figure 4.6: An example of confusing method selection. Figure (a) shows a constraint system; figures (b), (c), and (d) show plans corresponding to a sequence of edits. The last edit, shown in (d), results in the selection of method m_1 ; this is confusing, as it is a change which is not downstream from the edit.

As an example of such a situation, consider the constraint system shown in Figure 4.6a. This hypothetical system consists of four variables, v_1 , v_2 , v_3 , and, v_4 , as well as two constraints, $C_1 = \{m_1, m_2\}$ and $C_2 = \{m_3, m_4\}$. Figures 4.6b, 4.6c, and 4.6d show the plans resulting from three successive edits to this system. In the plans of Figures 4.6b and 4.6c, all methods requiring scheduling lie downstream of the edited variable. In the plan of Figure 4.6d, however, method m_1 is newly selected, so it must be scheduled, yet this method is not downstream of the edited variable. This may lead to GUI behavior that is confusing to the user: the edited value does not contribute to the inputs of method m_1 , making it unclear why it should be executed.

Considering this scenario in closer detail, the edit of v_1 in Figure 4.6b causes the stay constraint for v_1 to be promoted to the top of the hierarchy. We say variable v has a higher *priority* than variable w if the stay constraint for v is higher than the stay constraint for w in the constraint hierarchy; thus, the edit of v_1 gives that variable the highest priority. At this point, if we were to consider the constraint C_1 in isolation, we would expect method m_1 to be selected (m_1 preserves v_1 , which has a higher priority). We may describe this informally by saying that C_1 “prefers” method m_1 . In Figure 4.6c, the edit of v_4 gives it the highest priority. In order to preserve v_4 , we must select m_4 for the plan; this, in turn,

Input: $S = \langle V + M, E \rangle$, a solution graph
Input: P , a sequence of variables sorted in descending priority order
Output: P' , a sequence of variables sorted in updated priority order
Local: Q , a priority queue of variables sorted according to P
Local: D , a map from graph nodes to integer (representing *in-degree*)

```

1 Algorithm: ADJUSTPRIORITIES
2 foreach  $m \in M$  do
3    $D[m] \leftarrow \text{in\_degree}(m)$ 
4 foreach  $v \in V$  do
5    $D[v] \leftarrow \text{in\_degree}(m)$ 
6   if  $D[v] = 0$  then  $\text{enqueue}(Q, v)$ 
7 while  $\neg \text{empty}(Q)$  do
8    $v \leftarrow \text{dequeue}(Q)$ 
9    $\text{push}(P', v)$ 
10  foreach  $m \in M \mid (v, m) \in E$  do
11     $D[m] \leftarrow D[m] - 1$ 
12    if  $D[m] = 0$  then
13      foreach  $v' \in V \mid (m, v') \in E$  do
14         $D[v'] \leftarrow D[v'] - 1$ 
15        if  $D[v'] = 0$  then  $\text{enqueue}(Q, v')$ 

```

Algorithm 4.1: Enforce topological ordering on a constraint hierarchy

forces the selection of m_2 . Even so, variable v_1 still has a higher priority than v_2 , so C_1 still “prefers” m_1 . Thus, when v_3 is edited in Figure 4.6d, allowing either m_1 or m_2 to be selected, it is m_1 that is chosen.

We can prevent scenarios such as this one by adjusting the constraint hierarchy after the constraint system is solved so that each method selected in the plan becomes the “preferred” method of its constraint. To do this, we adjust the priority order so that it is topologically ordered with respect to the solution graph; that is, if there is a path from v to w in the solution graph, then v has a higher priority than w . If a method m is selected for the plan, this adjustment will ensure that each of its inputs will have a higher priority than any of its outputs. Because every other method of the constraint will output to at least one of the inputs of m , m will be the “preferred” method.

Algorithm 4.1 shows how this adjustment is performed. The algorithm takes a list of variables, sorted by priority, and a solution graph; it returns a new list of variables representing the adjusted priority order. The algorithm is based on a straightforward approach to topological sorting: repeatedly picking a *source* node—i.e., a node with no incoming edges—then removing it and all its edges from the graph. However, when picking a source node, the algorithm always selects a method if there is one, and, if not, the variable with the highest original priority.

To understand the effects of this algorithm, first consider the case where the solution graph contains no edges. In this case, the algorithm reduces to picking variables in priority order—i.e., the new priority order is identical to the original. Now suppose there is an edge from v to w . This means w cannot be added to the new priority order until v is picked and removed from the graph. If v has a higher priority than w , then this restriction is irrelevant: v would be picked first regardless. If w has a higher priority than v , then this restriction will force the new priority of w to be just below that of v . Generalizing, this algorithm decreases the priority of every variable until it is less than all of the variable’s ancestors in the solution graph. The relative ordering of all variables with the same ancestors in the solution graph remains unaffected by this adjustment.

One important property of this adjustment is that it has no effect on the current plan; that is, planning again immediately after the adjustment results in the same plan. The adjustment only affects the subsequent plan resulting from the next edit. It ensures that the only differences between that plan and the current plan lie downstream in the solution graph of a newly promoted constraint. As a result, we may be sure the only methods of that plan needing to be scheduled are those lying downstream in the solution graph from an edited variable or a newly promoted optional constraint (see Section 4.3). Another result of this adjustment is that removing a constraint from the constraint system has no effect on the current solution; that is, planning again immediately after removing a constraint results in the same plan. This property is beneficial when dealing with changes to the structure of the constraint system as discussed in Chapter 6.

4.6 Detecting Irrelevant Variables

Section 2.2.3 describes the various dataflow graphs made available by the property model. These graphs allow the creation of generic GUI algorithms, parameterized over data dependencies. This section presents one such algorithm: detecting and reporting irrelevant variables. While this basic algorithm has been presented in previous work [25], it had to be revisited and redesigned so that it works with asynchronous property models; we have also added some improvements in this second iteration.

4.6.1 Contributing and Relevant Variables

As discussed previously, methods of the constraint system should only affect the program state through their return values. However, commands may use data of the property model to perform application-specific tasks. In the shipping form example, submitting the form invokes a command that reads the shipping class and distance variables' values. We refer to variables used by commands to perform these application-specific tasks as *output variables*. The purpose of a GUI may be described as helping the user synthesize values for these output variables. By this definition, variables are only useful insofar as they can be used to calculate a value for an output variable. This insight is the basis of a reusable algorithm for automatically enabling/disabling widgets of the GUI.

Determining which variables may have been used in calculating the value of an output variable requires an analysis of the current data dependencies in the GUI. This analysis will yield more accurate results if done using the evaluation graph. Recall from Section 2.2.3 that the evaluation graph is a subgraph of the solution graph in which we remove any edge (v, m) where the activation of method m (in the generation to which the graph applies) made no use of the value of v . Thus, whereas the solution graph reveals the *potential* dependencies in one generation, the evaluation graph reveals the *actual* dependencies enforced.

Rather than creating a new graph that is nearly identical to the solution graph, we implement the evaluation graph by assigning labels to the *input edges*—edges from a vari-

able to a method—of the constraint graph. There are three possible labels for an edge. The *used* label indicates that the variable was used by the method in the generation to which the graph applies. The *unused* label indicates that the variable was not used by the method. The *unknown* label indicates that it is not currently known whether the variable was used or not because the method was not scheduled or has not finished executing. If desired, the actual evaluation graph could be constructed by making a copy of the solution graph, excluding any edges labeled *unused*. Alternatively, we may use the solution graph and simply disregard any edges labeled *unused*.

We define two classifications for variables. A *contributing* variable is one whose value was used to calculate the value of an output variable. More specifically, a variable is contributing if and only if there is a path in the evaluation graph from that variable to an output variable. Thus, whether or not a variable is contributing may be determined by simply examining the solution graph and corresponding edge labels.

A *relevant* variable is one whose value could potentially be used to calculate the value of an output variable. More specifically, a variable is relevant if and only if there is some plan of the constraint system in which there is a path from the variable to an output variable which does not use any edges labeled *unused*. Note that, by definition, a contributing variable must also be relevant.

To determine whether a non-contributing variable is relevant requires searching for a path from the variable to an output variable in the constraint graph, such that the path does not use edges labeled *unused*, nor two methods from the same constraint. Once a path to an output variable is found, we fix that path as a partial solution and solve the rest of the constraint system to ensure a plan exists that contains the path.

The above two algorithms are run after every solution to the constraint system. The results of the algorithms are made available as variable properties named *contributing* and *relevant*. The task of the property model is only to set these properties; how the properties are used is entirely up to the GUI programmer. Like the *pending* and *stale* properties, these properties may be bound to the View in order to alter its appearance. For example,

widgets may be shaded when they are non-contributing or disabled entirely when they are irrelevant.

Neither of the above two algorithms distinguish between edges labeled *used* and edges labeled *unknown*. This is based on the assumption that an edge should be treated as *used* until it is known for sure that it is *unused*. In this way we avoid, e.g., disabling widgets simply because we are unsure yet whether they are relevant.

4.6.2 Creating the Evaluation Graph

To determine the label for an edge from a variable to a method in the constraint graph, we must decide whether the activation which corresponds to the method made use of the value of the promise which corresponds to the variable. When this activation is first scheduled, this edge will be labeled *unknown*. Later, when we are able to determine if the value of the promise was used, we may change the label accordingly.

We use a simple test to decide whether the value of a promise was used: if the activation subscribed to the promise, we assume that it was used; otherwise, we assume it was unused. This approach requires the least effort on the part of the method implementer. However, two details must be taken into account. First, a method is not required to subscribe to a promise right away. A promise may be considered used as soon as a method subscribes to it; however, it may not be considered unused until the method has completed execution. To determine that a method has completed, we watch its output promises; once all output promises are fulfilled, we may consider the method finished and determine whether any of its inputs remain unused.

In principle, the execution of the methods of each generation produces a new evaluation graph. In practice, however, only the evaluation graph for the most recent generation seems useful. While it may be helpful to update the View with older variable values in order to show progress, disabling widgets based on old evaluation graphs is more likely to be frustrating than helpful. Thus, though we continue to gather more accurate information about which promises were used by activations, we ignore this information unless it affects the *current activation* (defined in Section 3.1.2).

In fact, because it is common for each successive evaluation graph to share many activations with the previous evaluation graph, we maintain a single evaluation graph that is continuously updated. By following the guidelines given above, we get the following approach.

- When a method is scheduled, first remove from the evaluation graph any method of the same constraint, along with any edges which use it. Then add to the evaluation graph the new method with its edges, labeling each input edge *unknown*.
- When a current activation subscribes to a promise, label the corresponding edge of the evaluation graph as *used*.
- When all output promises of a current activation are fulfilled, if any input edges of the corresponding method are labeled *unknown* then change them to *unused*.

Three practical issues arise in implementation. The first is that there are times when the approach of taking “subscribing” to mean “using” may be inconvenient. For this reason, we provide an alternative API by which a scheduling function can explicitly mark a promise as *used* or *unused*. If this API is used, then the property model will use the label provided over the label derived from promise subscriptions.

The second is that a promise may be used as input to multiple activations. Because a promise has only a single usage flag, it would be impossible to determine which activations used the promise and which did not. To solve this issue, when a promise is needed as input for an activation we create a duplicate promise to give the activation. This duplicate promise subscribes to the original promise so that as soon the original promise is fulfilled, the duplicate promise can be fulfilled with the same value. In this way we ensure every usage is represented by a unique promise, and therefore has a unique usage flag.

The third is that, because the label *unknown* is treated as *used*, it is possible to get “flickering”—in which the value changes back and forth—in the *contributing* or *relative* property. The situation in which this occurs results from the following three steps. First,

a method does not use one of its inputs; the corresponding edge is labeled *unused*, which causes a property to be false. Second, the method is scheduled to be executed again; the edge label is changed to *unknown*, allowing the property to become true. Third, the method executes and again does not use the input; thus, the edge label is once again set to *unused*, causing the property to return to false.

We can minimize the effects of such “flickering” by ensuring that any methods that can be executed are executed before the View is updated. For example, in the above scenario, the “flickering” property is set to false in the second step when the method is scheduled. If we were to update the View to reflect this change immediately, the corresponding widget would be enabled. However, if the scheduled method’s inputs are available, the method can be executed before updating the View. Executing the method causes the property to be set to true again. Now updating the View has no effect: the widget remains disabled. Effectively, this throttles updates to the View, preventing a rapid succession of updates. This throttling effect can be made even stronger by using a timer to wait a certain period of time—say, a fraction of a second—before updating the View.

5. OPERATIONAL SEMANTICS*

Here we provide operational semantics defining the behavior of a property model in response to edits. Our motivation for this formalism is to provide a precise description of the concepts discussed informally in Chapter 3 and Section 4.6. Because the purpose of these semantics is to clarify intent, we include certain elements which may be unnecessary in an actual implementation. For example, these semantics build an actual representation of the reactive program graph; in practice, this graph does not need to be reified into a concrete data structure.

5.1 About the Formalism

This section defines the formalism and the conventions we use in presenting it. It also explains how to interpret the evaluation rules that follow.

5.1.1 Notational Conventions

Sets are represented by capital letters, e.g., X , or by listing individual elements inside curly braces, e.g., $\{x_1, \dots, x_n\}$.

Sequences are represented by lowercase letters with a bar over them, e.g., \bar{y} , or by listing individual elements in order inside parentheses, e.g., (y_1, \dots, y_m) . Similarly, a sequence type is denoted using parentheses: the type (T) refers to a sequence where each element is type T . We use the $++$ operator to append to sequences.

$$(x_1, \dots, x_n) ++ (y_1, \dots, y_m) = (x_1, \dots, x_n, y_1, \dots, y_m)$$

The identity element for this operation is the empty sequence which is denoted “ $()$ ”.

$$\bar{x} ++ () = () ++ \bar{x} = \bar{x}$$

Elements of a product type, or *tuples*, are represented by listing the individual elements inside angle brackets, e.g., $\langle x, y, z \rangle$. A product type is denoted using angle brackets: the

*Portions of this chapter reprinted with permission from “Responsive and Consistent User Interfaces with Multi-Way Dataflow Constraint Systems” by Gabriel Foust, Jaakko Järvi, and Sean Parent. Under review for inclusion in *Computer Languages, Systems and Structures (COMLAN)*, Elsevier.

type $\langle T, U, V \rangle$ refers to a tuple in which the first element is of type T , the second of type U , and the third of type V .

Function types are written using an arrow, e.g., $T \rightarrow U$. Function application is denoted using parentheses, e.g., $f(x)$. We use the notation $[x \mapsto t]f$ to define a function which is identical to f for all inputs except x , which it maps to t .

We represents a graph as a tuple, $\langle N, E \rangle$, where N is the node set and E is the edge set. Each edge is represented as a pair, $\langle n, m \rangle$, where $n, m \in N$. We assume the following graph operations as primitives.

outs($\langle N, E \rangle, n$)

This function returns the set $\{n' \in N \mid \langle n, n' \rangle \in E\}$.

add_node($\langle N, E \rangle, n$)

This function returns the graph $\langle N \cup \{n\}, E \rangle$.

add_edges($\langle N, E \rangle, E'$)

This function returns the graph $\langle N, E \cup E' \rangle$.

topological_sort(G, N')

This function returns a sequence consisting of the nodes in N' sorted topologically according to G .

5.1.2 Symbols and Values

For this formalism, we represent variables, constraints, methods, promises, and activations as *symbols*—that is, values whose only characteristic is identity. We use the function *newsym* to generate unique symbols. We define the types **Variable**, **Constraint**, **Method**, **Promise**, and **Activation** as sets of symbols representing the elements suggested by their name.

We use a *definition function* to map these symbols to structural information indicating how they are related. We assume this function is overloaded so that it returns information specific to its argument type. Specifically, the definition function supports the following signatures.

Variable \rightarrow (Promise)

A variable symbol is mapped to its *promise history*: a sequence of promise symbols indicating the promises that have been made for this variable over the lifetime of the program, ordered from least to most recent.

Constraint \rightarrow (Activation)

A constraint symbol is mapped to its *activation history*: a sequence of activation symbols indicating the activations of methods of this constraint over the lifetime of the program, ordered from least to most recent.

Method \rightarrow \langle Constraint, Function, (Variable), (Variable) \rangle

A method symbol is mapped to a tuple of information: the constraint to which the method belongs, the scheduling function for the method (a function of the underlying language), the input variables in the order expected by the scheduling function, and the output variables in the order returned by the scheduling function.

Activation \rightarrow \langle Method, (Promise), (Promise) \rangle

An activation symbol is mapped to a tuple of information: the method invoked for the activation, the input promises in the order passed to the method's scheduling function, and the output promises in the order returned by the method's scheduling function.

Structural information does not change. We may expand the definition function as new elements are added to the system, and we may append new promises and activations to histories. However, the relationships between existing elements will not be redefined.

We use a *valuation function* to map symbols to their current values. Unlike the definition function, the value given by the valuation function may change many times over the life of the program. The valuation function is also overloaded, supporting the following signatures.

Variable \rightarrow Value

A variable symbol is mapped to a value of the underlying programming language. This is the current value of the variable, as presented by the View.

$\text{Promise} \rightarrow \langle \text{Value}, \text{StatusFlag}, \text{UsageFlag} \rangle$

A promise symbol is mapped to a tuple of information: a value of the underlying programming language (or the undefined value “.” if the promise is pending), a flag indicating the current status of the promise with respect to its value, and a flag indicating whether the promise has been used by an activation.

$\langle \text{Variable}, \text{Method} \rangle \rightarrow \text{UsageFlag}$

An input edge $\langle v, m \rangle$ of the constraint graph is mapped to a flag indicating whether the value of v was used during the most recent execution of m . This value is the edge’s label in the evaluation graph, as discussed in Section 4.6.2.

The types **StatusFlag** and **UsageFlag** are enumerated types defined below. A **StatusFlag** indicates the status of a promise: whether it is waiting on a value (**pending**) or the value has been supplied (**fulfilled**). A **UsageFlag** indicates whether the value of a promise was used by an activation (**used**), was not used (**unused**), or whether it is too early to tell (**unknown**).

$\text{StatusFlag} = \{\text{pending}, \text{fulfilled}\}$

$\text{UsageFlag} = \{\text{unknown}, \text{used}, \text{unused}\}$

5.1.3 Evaluation Environment

For this formalism, we define a property model as a collection of eight elements. We define these elements below and assign to each a meta-variable used to refer to the element in the evaluation rules.

$G_C =$ *The constraint graph* as defined in Section 2.2.3. The nodes of this graph are variable and method symbols.

$G_S =$ *The solution graph* as defined in Section 2.2.3. The nodes of this graph are variable and method symbols.

$G_R =$ *The reactive program graph* as defined in Section 3.1.4. The nodes of this graph are promise and activation symbols.

$\bar{\pi} =$ *The variable priority assignment* as defined in Section 2.2.2. This is represented as a sequence of variable symbols sorted from lowest to highest priority.

$\Lambda =$ *The modified variable set*: a set of variable symbols representing all variables modified by the current edit.

$\Gamma =$ *The definition function* as defined in Section 5.1.2.

$\Sigma =$ *The valuation function* as defined in Section 5.1.2.

$\Delta =$ *The callback set* as defined in Section 5.1.4.

A property model, then, is defined as a tuple, $\langle G_C, G_S, G_R, \bar{\pi}, \Lambda, \Gamma, \Sigma, \Delta \rangle$. Note that the evaluation graph is not represented directly in the property model; rather, it may be derived using the current solution graph and the labels provided for each edge by the valuation function.

Evaluation rules will take one of three forms. A rule describing a purely functional (i.e., side-effect free) computation will appear as below.

EXAMPLE-PURE-FUNCTION

$$\frac{\text{premises}}{\text{form} \mid G_C, G_S, G_R, \bar{\pi}, \Lambda, \Gamma, \Sigma, \Delta = \text{result}}$$

The meaning of this rule is as follows: given a program matching the specified *form* in the property model $\langle G_C, G_S, G_R, \bar{\pi}, \Lambda, \Gamma, \Sigma, \Delta \rangle$, if we can satisfy all specified *premises*, then we may reduce our program to the specified *result*. Note that, for brevity, we will list only those property model elements referred to by either the *form* or *premises*; the remaining elements are assumed to be present but irrelevant.

The second form, used for rules describing a computation which modifies the environment, will appear as below.

EXAMPLE-SIDE-EFFECT

$$\frac{\text{premises}}{\text{form} \mid G_C, G_S, G_R, \bar{\pi}, \Lambda, \Gamma, \Sigma, \Delta \rightarrow \text{result} \mid G_C', G_S', G_R', \bar{\pi}', \Lambda', \Gamma', \Sigma', \Delta'}$$

The meaning of this rule is as follows: given a program matching the specified *form* in the

property model $\langle G_C, G_S, G_R, \bar{\pi}, \Lambda, \Gamma, \Sigma, \Delta \rangle$, if we can satisfy all specified *premises*, then we may reduce our program to the specified *result* in the property model $\langle G_C', G_S', G_R', \bar{\pi}', \Lambda', \Gamma', \Sigma', \Delta' \rangle$. Effectively, this means that the evaluation of the specified *form* causes the property model element G_C to change to G_C' , the element G_S to change to G_S' , and so on. In the case where evaluation produces no value (e.g., a void function), the result is given simply as the undefined value “.”. Again, for brevity, we will list only those property model elements referred to by either the *form* or *premises*.

The third form is defined in Section 5.1.4 below.

We use the following metavariables to represent values of the specified type.

$v, w := \text{Variable}$	$p, q := \text{Promise}$
$c := \text{Constraint}$	$s := \text{StatusFlag}$
$a := \text{Activation}$	$u := \text{UsageFlag}$
$m := \text{Method}$	

We use the following metavariables to represent values and functions of the underlying programming language.

$t, o := \text{Value}$	$f, g := \text{Function}$
------------------------	---------------------------

5.1.4 The Callback Set

These semantics require the ability to specify computations to happen asynchronously at some point in the future. The expected implementation for this is registering callback functions to be executed when a promise is fulfilled. However, to avoid specifying any particular implementation of promises, callback registration, or callback execution, in this formalism we use an abstraction: the callback set. This set contains an element for every computation that should happen later. Elements of this set are defined as a disjoint-union type, implemented as a tuple in which the first field is a tag identifying the element's type. The remainder of the tuple stores the data needed by the computation, i.e., the computation's *closure*. Below are the six different tuple types which compose this disjoint-union type.

$\langle \text{var}, \text{Promise}, \text{Variable} \rangle$

Indicates that a promise has been assigned to a variable.

$\langle \text{copy}, \text{Promise}, \text{Promise} \rangle$

Indicates that the second promise is a copy of the first promise.

$\langle \text{running}, \text{Activation} \rangle$

Indicates that an activation is currently running.

$\langle \text{input}, \text{Promise}, \text{Activation} \rangle$

Indicates that a promise is an input for an activation.

$\langle \text{extern}, \text{Promise}, \text{Function} \rangle$

Indicates that some code external to the property model has registered a callback function to be executed when a promise has been fulfilled.

$\langle \text{lifted}, \text{Function}, (\text{Promise}), (\text{Promise}) \rangle$

Indicates the activation of a method implemented by lifting a function of the underlying language.

The third form for evaluation rules specifies the evaluation of an element of the callback set, as shown below.

EXAMPLE-CALLBACK

$$\frac{\Delta = \delta \cup \Delta' \quad \text{premises}}{\cdot \mid G_C, G_S, G_R, \bar{\pi}, \Lambda, \Gamma, \Sigma, \Delta \rightarrow \text{result} \mid \cdot G_C', G_S', G_R', \bar{\pi}', \Lambda', \Gamma', \Sigma', \Delta'}$$

The meaning here is as follows: at any time when the system is *not* executing a program for the property model $\langle G_C, G_S, G_R, \bar{\pi}, \Lambda, \Gamma, \Sigma, \Delta \rangle$, if the callback set Δ contains an element δ such that we can satisfy all specified *premises*, we may transition to the property model $\langle G_C', G_S', G_R', \bar{\pi}', \Lambda', \Gamma', \Sigma', \Delta' \rangle$. Notice that, in doing so, we are removing element δ from the callback set, ensuring that the computation will not occur more than once. Again, for brevity, we will list only those property model elements referred to by either the *form* or *premises*.

5.2 The Evaluation Rules

Here we present the evaluation rules, interspersed throughout the explanation of their meaning. The rules by themselves may be viewed in a continuous, uninterrupted form in Appendix A. The rules are organized into five sections: rules regarding edits to the property model, rules regarding scheduling and evaluating methods, rules regarding operations on promises, rules for callback operations, and rules for a function lifting mechanism.

5.2.1 Editing the Property Model

This first set of rules defines an API by which the external program can make edits to the variables of the property model. We define two functions for performing edits. The first is the `touch` function defined by the rule `TOUCH`. This function promotes a variable to the highest priority by removing it from its current position in the priority sequence and appending it to the end. The touched variable is also added to the modified variable set.

$$\begin{array}{c} \text{TOUCH} \\ \hline \bar{\pi} = \bar{v}_1 \uparrow (v) \uparrow \bar{v}_2 \quad \bar{\pi}' = \bar{v}_1 \uparrow \bar{v}_2 \uparrow (v) \quad \Lambda' = \Lambda \cup \{v\} \\ \hline \text{touch}(v) \mid \bar{\pi}, \Lambda \rightarrow \cdot \mid \bar{\pi}', \Lambda' \end{array}$$

The second is the `set` function which is defined by the rule `SET`. This function sets the value of a variable, as well as promotes it to the highest priority. The value to be assigned is represented as a promise. The work of adding the promise to the variable's history is delegated to an internal function, `add_promise`, which is defined in Section 5.2.3

$$\begin{array}{c} \text{SET} \\ \hline \text{touch}(v) \mid \bar{\pi}, \Lambda \rightarrow \cdot \mid \bar{\pi}', \Lambda' \quad \text{add_promise}(v, p) \mid G_R, \Gamma, \Delta \rightarrow \cdot \mid G_R', \Gamma', \Delta' \\ \hline \text{set}(p) \mid G_R, \bar{\pi}, \Lambda, \Gamma, \Sigma, \Delta \rightarrow \cdot \mid G_R', \bar{\pi}', \Lambda', \Gamma', \Sigma', \Delta' \end{array}$$

An edit is defined as one or more calls to `touch` or `set` followed by a single call to `update`, thereby making edits that affect multiple variables possible. The `update` function, defined by the rule `UPDATE`, responds to the edits by solving the constraint system: calculating a

plan and scheduling its methods.

UPDATE

$$\begin{array}{l}
G_S' = \text{plan}(G_C, \bar{\pi}, G_S, \Lambda) \quad \bar{\pi}' = \text{adjust}(\bar{\pi}, G_S') \\
M = \text{downstream_many}(\Lambda) \mid G_S', \Sigma \quad \bar{m} = \text{topological_sort}(G_S', M) \\
\text{schedule}(\bar{m}) \mid G_R, \Gamma, \Sigma, \Delta \rightarrow \cdot \mid G_R', \Gamma', \Sigma', \Delta' \quad \Lambda' = \{\} \\
\hline
\text{update}() \mid G_C, G_S, G_R, \bar{\pi}, \Lambda, \Gamma, \Sigma, \Delta \rightarrow \cdot \mid G_C, G_S', G_R', \bar{\pi}', \Lambda', \Gamma', \Sigma', \Delta'
\end{array}$$

This rule makes use of two functions defined elsewhere: *plan* and *adjust*. The function *plan* calculates and returns a plan represented as a solution graph, as discussed in Section 2.2.2. The required parameters for this function are the constraint system and the current priority order; as a concession to our implementation, which uses an incremental planning algorithm, we also pass as parameters the previous plan and the changed variable set. The function *adjust* adjusts the priority order so that it is topologically ordered with respect to the solution graph, as described in Section 4.5.

Once the plan has been calculated and the priorities adjusted, the internal function *downstream_many*, defined below, is used to collect all methods that lie downstream in the solution graph of an edited variable. These are the methods that must be scheduled. They are sorted topologically with respect to the solution graph, then scheduled using the *schedule* function, which is defined in Section 5.2.2.

The rule METHODS-DOWNSTREAM defines the function *downstream*. This internal function returns all methods downstream of a single variable. The two rules DOWNSTREAM-MANY and DOWNSTREAM-MANY-EMPTY define the internal function *downstream_many*, which simply maps *downstream* over a set of variables and returns a set containing the combined results. These two functions are mutually recursive, and together they traverse the entire downstream subgraph of their given arguments.

METHODS-DOWNSTREAM

$$\begin{array}{c}
M = \{m \in \text{outs}(G_S, v) \mid \Sigma(\langle v, m \rangle) \neq \text{unused}\} \\
V = \cup_{m \in M} \text{outs}(G_S, m) \quad M' = \text{downstream_many}(V) \mid G_S, \Sigma \\
\hline
\text{downstream}(v) \mid G_S, \Sigma = M \cup M'
\end{array}$$

DOWNSTREAM-MANY

$$\begin{array}{c}
V = \{v\} \cup V' \\
M = \text{downstream}(v) \mid G_S, \Sigma \quad M' = \text{downstream_many}(V') \mid G_S, \Sigma \\
\hline
\text{downstream_many}(V) \mid G_S, \Sigma = M \cup M'
\end{array}$$

DOWNSTREAM-MANY-EMPTY

$$\text{downstream_many}(\{\}) \mid G_S, \Sigma = \{\}$$

Note that the first premise of the METHODS-DOWNSTREAM rule specifies that the downstream traversal excludes any edges that currently have a value of `unused`. This reflects the fact that the input parameter corresponding to such an edge was not used during the last execution of the method, and therefore changes to that parameter are irrelevant.

5.2.2 Scheduling Methods

The next set of rules define the work necessary to schedule the selected methods of the plan. The rules SCHEDULE-METHODS and SCHEDULE-METHODS-EMPTY define the `schedule` function, which is the entry point for this operation; this function is called by `update`, which is defined in the previous section. This function recursively iterates over a sequence of methods that has been sorted topologically with respect to the solution graph. The rule SCHEDULE-METHODS defines the work done for each element of the sequence; the rule SCHEDULE-METHODS-EMPTY simply halts the recursion when the sequence is empty.

SCHEDULE-METHODS

$$\begin{array}{l}
\bar{m} = (m) \uparrow \bar{m}' \quad \Gamma(m) = \langle _, f, (v_1, \dots, v_j), (w_1, \dots, w_k) \rangle \\
\forall_{i=1}^j : \Gamma(v_i) = (\dots, p_i) \quad \text{duplicate}((p_1, \dots, p_j)) \mid \Sigma, \Delta \rightarrow (p'_1, \dots, p'_j) \mid \Sigma', \Delta' \\
\quad f(p'_1, \dots, p'_j) \mid \Sigma', \Delta' \rightarrow (q_1, \dots, q_k) \mid \Sigma'', \Delta'' \\
\text{add_promise_many}((w_1, \dots, w_k), (q_1, \dots, q_k)) \mid G_R, \Gamma, \Delta'' \rightarrow \cdot \mid G_R', \Gamma', \Delta^{(3)} \\
\quad \text{reset}(m) \mid \Gamma', \Sigma'' \rightarrow \cdot \mid \Gamma', \Sigma^{(3)} \\
\text{make_activation}(m, (p'_1, \dots, p'_j), (q_1, \dots, q_k)) \mid \Gamma', \Delta^{(3)} \rightarrow a \mid \Gamma'', \Delta^{(4)} \\
\quad \text{add_to_graph}(a, (p_1, \dots, p_j), (q_1, \dots, q_k)) \mid G_R' \rightarrow \cdot \mid G_R'' \\
\quad \text{schedule}(\bar{m}') \mid G_R'', \Gamma'', \Sigma^{(3)}, \Delta^{(4)} \rightarrow G_R^{(3)}, \Gamma^{(3)}, \Sigma^{(4)}, \Delta^{(5)} \\
\hline
\text{schedule}(\bar{m}) \mid G_R, \Gamma, \Sigma, \Delta \rightarrow \cdot \mid G_R^{(3)}, \Gamma^{(3)}, \Sigma^{(4)}, \Delta^{(5)}
\end{array}$$

SCHEDULE-METHODS-EMPTY

$$\text{schedule}(\cdot) \mid G_R, \Gamma, \Sigma, \Delta \rightarrow \cdot \mid G_R, \Gamma, \Sigma, \Delta$$

The scheduling of a single method involves several steps. The first step is to look up the scheduling function, input variables, and output variables for the method. Next, the most current promise for each input variable is retrieved. As described in Section 4.6, we must duplicate these promises so that each input promise has its own usage flag; this is done using the `duplicate` function, defined below. Now the scheduling function can be called, passing the duplicated input promises and capturing the returned output promises. These output promises are then added to the promise histories of their corresponding variables using the `add_promise_many` function defined in Section 5.2.3. Once this has been done, the method is scheduled; all that remains is some record-keeping steps performed by three functions defined below. The function `reset` resets all usage flags for the constraint that contains the method being scheduled. The function `make_activation` initializes a symbol with the new activation. And the function `add_to_graph` updates the reactive program graph for the new activation. Finally, we have a recursive call to continue iterating through the sequence.

The `duplicate` function is defined by the rules `DUPLICATE-PROMISES` and `DUPLICATE-`

PROMISES-EMPTY. This function recursively iterates over a sequence of promises, and makes a duplicate promise for each. Rule DUPLICATE-PROMISES defines the work done to duplicate a single promise. First, a new symbol is created for the duplicate promise and its flags are initialized to `pending` and `unknown`. Then an element is added to the callback set; this element is discussed further below. Finally, we have a recursive call to continue iterating through the sequence. The rule DUPLICATE-PROMISES-EMPTY halts recursion when the sequence is empty.

$$\begin{array}{c}
\text{DUPLICATE-PROMISES} \\
\hline
\bar{p} = (p) \dashv\vdash \bar{p}' \quad q = \text{newsym}() \quad \Sigma' = [q \mapsto \langle \cdot, \text{pending}, \text{unknown} \rangle] \Sigma \\
\Delta' = \{ \langle \text{copy}, p, q \rangle \} \cup \Delta \quad \text{duplicate}(\bar{p}') \mid \Sigma', \Delta' \rightarrow \bar{q} \mid \Sigma'', \Delta'' \\
\hline
\text{duplicate}(\bar{p}) \mid \Sigma, \Delta \rightarrow (q) \dashv\vdash \bar{q} \mid \Sigma'', \Delta'' \\
\\
\text{DUPLICATE-PROMISES-EMPTY} \\
\text{duplicate}(\langle \rangle) \mid \Sigma, \Delta \rightarrow \langle \rangle \mid \Sigma, \Delta
\end{array}$$

The rule DUPLICATE-PROMISE-FULFILLED handles the callback element added by `duplicate` once the original promise has been fulfilled. The rule specifies that the duplicate promise is to be fulfilled using the value of the original promise.

$$\begin{array}{c}
\text{DUPLICATE-PROMISE-FULFILLED} \\
\hline
\Delta = \{ \langle \text{copy}, p, q \rangle \} \cup \Delta' \quad \Sigma(p) = \langle t, \text{fulfilled}, _ \rangle \\
\Sigma(q) = \langle \cdot, \text{pending}, u \rangle \quad \Sigma' = [q \mapsto \langle t, \text{fulfilled}, u \rangle] \Sigma \\
\hline
\cdot \mid \Sigma, \Delta \rightarrow \cdot \mid \Sigma', \Delta'
\end{array}$$

The rule RESET-CONSTRAINT defines the function `reset`, which resets the usage information for the constraint containing a certain method, specified as a parameter of the function. All edges of the constraint are guaranteed to be `unknown` except for the input edges of the most recent method activated for the constraint. Thus, the function starts by looking up the constraint for the passed method, then the most recent activation of

a method of that constraint, then the method of the activation, and then the inputs of the method. Finally, the function updates the value of all input edges of the method to **unknown**.

RESET-CONSTRAINT

$$\frac{\begin{array}{l} \Gamma(m) = \langle c, _, _, _ \rangle \quad \Gamma(c) = (\dots, a) \quad \Gamma(a) = \langle m', _, _ \rangle \\ \Gamma(m') = \langle c, _, (v_1, \dots, v_j), _ \rangle \quad \Sigma' = [\forall_{i=1}^j : \langle v_i, m' \rangle \mapsto \text{unknown}] \Sigma \end{array}}{\text{reset}(m) \mid \Gamma, \Sigma \rightarrow \cdot \mid \Gamma, \Sigma'}$$

The rule MAKE-ACTIVATION defines the function `make_activation` which initializes the symbol for a new activation. In addition to recording the structural information that defines the activation, this function also adds an element to the callback set so that we can update usage information once the activation is complete; the handling of this element is defined by rule ACTIVATION-COMPLETED in Section 5.2.4.

MAKE-ACTIVATION

$$\frac{\begin{array}{l} a = \text{newsym}() \quad \Gamma' = [a \mapsto \langle m, (p_1, \dots, p_j), (q_1, \dots, q_k) \rangle] \Gamma \quad \Gamma'(m) = \langle c, _, _, _ \rangle \\ \Gamma'' = [c \mapsto \Gamma'(c) \uplus (a)] \Gamma' \quad \Delta' = \{ \langle \text{running}, a \rangle \} \cup \{ \langle \text{input}, p_i, a \rangle \mid 1 \leq i \leq j \} \cup \Delta \end{array}}{\text{make_activation}(m, (p_1, \dots, p_j), (q_1, \dots, q_k)) \mid \Gamma, \Delta \rightarrow a \mid \Gamma'', \Delta'}$$

Finally, rule ADD-TO-GRAPH defines the function `add_to_graph`. This function takes an activation with its input and output promises and adds to the reactive program graph the node and appropriate edges for the activation.

ADD-TO-GRAPH

$$\frac{\begin{array}{l} G_R' = \text{add_node}(G_R, a) \quad G_R'' = \text{add_edges}(G_R', \{ \langle p_i, a \rangle \mid 1 \leq i \leq j \}) \\ G_R^{(3)} = \text{add_edges}(G_R'', \{ \langle a, q_i \rangle \mid 1 \leq i \leq k \}) \end{array}}{\text{add_to_graph}(a, (p_1, \dots, p_j), (q_1, \dots, q_k)) \mid G_R \rightarrow \cdot \mid G_R^{(3)}}$$

5.2.3 Variables and Promises

The next set of rules defines how promises are associated with variables, and how the variables are updated when promises are fulfilled. The rule **ADD-PROMISE** defines the function `add_promise`, which appends a new promise to the end of a variable's promise history. Additionally, this function adds the promise as a node in the reactive program graph, and adds an element to the callback set so that the promise can potentially be used to update the variable when it resolves.

$$\begin{array}{c}
 \text{ADD-PROMISE} \\
 \hline
 \Gamma' = [v \mapsto \Gamma(v) \mathbin{++} (p)]\Gamma \quad G_R' = \text{add_node}(G_R, p) \quad \Delta' = \{\langle \text{var}, p, v \rangle\} \cup \Delta \\
 \hline
 \text{add_promise}(v, p) \mid G_R, \Gamma, \Delta \rightarrow \cdot \mid G_R', \Gamma', \Delta'
 \end{array}$$

Rules **ADD-PROMISE-MANY** and **ADD-PROMISE-MANY-EMPTY** define the function `add_promise_many`, that recursively iterates over two sequences, one of variables and one of promises. The function adds each promise in the promise sequence to the corresponding variable in the variable sequence using the `add_promise` function.

$$\begin{array}{c}
 \text{ADD-PROMISE-MANY} \\
 \hline
 \bar{v} = (v) \mathbin{++} \bar{v}' \quad \bar{p} = (p) \mathbin{++} \bar{p}' \quad \text{add_promise}(v, p) \mid G_R, \Gamma, \Delta \rightarrow \cdot \mid G_R', \Gamma', \Delta' \\
 \text{add_promise_many}(\bar{v}', \bar{p}') \mid G_R', \Gamma', \Delta' \rightarrow \cdot \mid G_R'', \Gamma'', \Delta'' \\
 \hline
 \text{add_promise_many}(\bar{v}, \bar{p}) \mid G_R, \Gamma, \Delta \rightarrow \cdot \mid G_R'', \Gamma'', \Delta'' \\
 \\
 \text{ADD-PROMISE-MANY-EMPTY} \\
 \text{add_promise_many}((), ()) \mid G_R, \Gamma, \Delta \rightarrow \cdot \mid G_R, \Gamma, \Delta
 \end{array}$$

The element added to the callback set by `add_promise` is handled in rule **VARIABLE-PROMISE-FULFILLED**. This rule specifies that once the promise for the variable's value has been fulfilled, the function `maybe_set_var` may be called.

VARIABLE-PROMISE-FULFILLED

$$\frac{\Delta = \{\langle \text{var}, p, v \rangle\} \cup \Delta' \quad \Sigma(p) = \langle _, \text{fulfilled}, _ \rangle \quad \text{maybe_set_var}(v, p) \mid \Gamma, \Sigma \rightarrow \cdot \mid \Gamma, \Sigma'}{\cdot \mid \Gamma, \Sigma, \Delta \rightarrow \cdot \mid \Gamma, \Sigma', \Delta'}$$

The `maybe_set_var` function is defined in rules MAYBE-SET-VAR-VISIBLE and MAYBE-SET-VAR-INVISIBLE. The former handles the case when the fulfilled promise is visible; in this case, the value of the promise becomes the value of the variable presented by the view, as defined by the valuation function. The latter handles the case when the promise is invisible; in this case, no action is taken.

MAYBE-SET-VAR-VISIBLE

$$\frac{\Gamma(v) = (p_1, \dots, p_j, \dots, p_k) \quad p_j = p \quad \forall_{i=j+1}^k : \Sigma(p_i) = \langle _, \text{pending}, _ \rangle \quad \Sigma(p) = \langle t, _, _ \rangle \quad \Sigma' = [v \mapsto t]\Sigma}{\text{maybe_set_var}(v, p) \mid \Gamma, \Sigma \rightarrow \cdot \mid \Gamma, \Sigma'}$$

MAYBE-SET-VAR-INVISIBLE

$$\frac{\Gamma(v) = (p_1, \dots, p_j, \dots, p_k) \quad p_j = p \quad \exists i : j < i \leq k \wedge \Sigma(p_i) = \langle _, \text{fulfilled}, _ \rangle}{\text{maybe_set_var}(v, p) \mid \Gamma, \Sigma \rightarrow \cdot \mid \Gamma, \Sigma}$$

5.2.4 Promise and Edge Usage

This section defines the rules which determine when promises are assigned the `used` and `unused` usage flags, and also how those flags propagate to the edges of the constraint graph. Promises are always created with the `unknown` usage flag. As described in Section 4.6, a property model will automatically deduce whether or not a promise has been used. However, it is possible that the programmer may wish to override these rules and explicitly define the usage flag of a promise.

Rules SET-USAGE and SET-USAGE-AGAIN define the `usage` function which can be used

to explicitly set the usage flag of a promise. For these semantics, we assume the usage of a promise may only be set one time. Rule SET-USAGE defines the case when the usage flag of a promise is unknown; in this case, the usage flag is assigned to the promise. Rule SET-USAGE-AGAIN defines the case when the usage flag of a promise has not yet been set; in this case the `usage` function has no effect.

$$\begin{array}{c}
\text{SET-USAGE} \\
\hline
\frac{\Sigma(p) = \langle t, s, \text{unknown} \rangle \quad \Sigma' = [p \mapsto \langle t, s, u \rangle] \Sigma}{\text{usage}(p, u) \mid \Sigma \rightarrow \cdot \mid \Sigma'}
\end{array}
\qquad
\begin{array}{c}
\text{SET-USAGE-AGAIN} \\
\hline
\frac{\Sigma(p) = \langle t, s, u' \rangle \quad u' \neq \text{unknown}}{\text{usage}(p, u) \mid \Sigma \rightarrow \cdot \mid \Sigma}
\end{array}$$

If the programmer does not explicitly set the usage flag, it will be set to `used` when the promise is subscribed to. The rules SUBSCRIBE-UNKNOWN and SUBSCRIBE-KNOWN define the `subscribe` function which implements this behavior. Rule SUBSCRIBE-UNKNOWN defines the case when the promise subscribed to has an `unknown` flag; in this case, the flag is changed to `used`. Rule SUBSCRIBE-KNOWN defines the case when the promise subscribed to has either a `used` or `unused` flag; in this case, the flag is left alone.

$$\begin{array}{c}
\text{SUBSCRIBE-UNKNOWN} \\
\hline
\frac{\Sigma(p) = \langle t, s, \text{unknown} \rangle \quad \Sigma' = [p \mapsto \langle t, s, \text{used} \rangle] \Sigma \quad \Delta' = \Delta \cup \{ \langle \text{extern}, p, f \rangle \}}{\text{subscribe}(p, f) \mid \Sigma, \Delta \rightarrow \cdot \mid \Sigma', \Delta'}
\end{array}$$

$$\begin{array}{c}
\text{SUBSCRIBE-KNOWN} \\
\hline
\frac{\Sigma(p) = \langle _, _, u \rangle \quad u \neq \text{unknown} \quad \Delta' = \Delta \cup \{ \langle \text{extern}, p, f \rangle \}}{\text{subscribe}(p, f) \mid \Sigma, \Delta \rightarrow \cdot \mid \Sigma, \Delta'}
\end{array}$$

Rules FULFILL-PROMISE and FULFILL-PROMISE-AGAIN define the `fulfill` function, used to fulfill a promise. Rule FULFILL-PROMISE defines the case when the promise to be fulfilled is pending. In this case, the promise takes the specified value and is marked as fulfilled. Note, in particular, that fulfilling a promise does not immediately invoke subscribed callbacks; these functions will be invoked by the rule SUBSCRIBED-PROMISE-

FULFILLED at some point in the future when there is a break in program execution. Rule FULFILL-PROMISE-AGAIN defines the case when the promise to be fulfilled has already been fulfilled. In this case the fulfill function has no effect.

$$\begin{array}{c}
\text{FULFILL-PROMISE} \\
\hline
\frac{\Sigma(p) = \langle _, \text{pending}, u \rangle \quad \Sigma' = [p \mapsto \langle t, \text{fulfilled}, u \rangle] \Sigma}{\text{fulfill}(p, t) \mid \Sigma \rightarrow \cdot \mid \Sigma'}
\end{array}
\qquad
\begin{array}{c}
\text{FULFILL-PROMISE-AGAIN} \\
\hline
\frac{\Sigma(p) = \langle _, \text{fulfilled}, _ \rangle}{\text{fulfill}(p, t) \mid \Sigma \rightarrow \cdot \mid \Sigma}
\end{array}$$

Rule SUBSCRIBED-PROMISE-FULFILLED specifies how callback functions are invoked. We assume the callback function invoked does not make any structural changes to the property model; thus, the definition function is not affected. However, the callback function may create and schedule promises; thus, the valuation function and callback function may be affected.

$$\begin{array}{c}
\text{SUBSCRIBED-PROMISE-FULFILLED} \\
\hline
\frac{\Delta = \{\langle \text{extern}, p, f \rangle\} \cup \Delta' \quad \Sigma(p) = \langle t, \text{fulfilled}, _ \rangle \quad f(t) \mid \Sigma, \Delta' \rightarrow \cdot \mid \Sigma', \Delta''}{\cdot \mid \Sigma, \Delta \rightarrow \cdot \mid \Sigma', \Delta''}
\end{array}$$

Rule ACTIVATION-COMPLETED handles the running callback element added when an activation is created. This rule may be invoked once all output promises of the activation have been fulfilled. Once the outputs are fulfilled, the method is considered to be complete; therefore, any promises whose usage flag is still unknown are updated to indicate they were unused.

$$\begin{array}{c}
\text{ACTIVATION-COMPLETED} \\
\hline
\frac{\begin{array}{l} \Delta = \{\langle \text{running}, a \rangle\} \cup \Delta' \\ \Gamma(a) = \langle m, (p_1, \dots, p_j), (q_1, \dots, q_k) \rangle \quad \forall_{i=1}^k : \Sigma(q_i) = \langle _, \text{fulfilled}, _ \rangle \\ \Sigma' = [\forall_{i=1}^j : \Sigma(p_i) = \langle t, s, \text{unknown} \rangle \implies p_j \mapsto \langle t, s, \text{unused} \rangle] \Sigma \end{array}}{\cdot \mid \Gamma, \Sigma, \Delta \rightarrow \cdot \mid \Gamma, \Sigma', \Delta'}
\end{array}$$

When the usage flag of an input promise is set, that flag should be copied to the

corresponding input edge *unless* there is a more recent activation for a method of the same constraint. Rule INPUT-USAGE-KNOWN handles an `input` callback element created for an input promise when an activation is created. Once the usage flag of the promise has been set to something other than `unknown`, the function `maybe_set_edge` is called.

INPUT-USAGE-KNOWN

$$\begin{array}{c} \Delta = \{\langle \text{input}, p, a \rangle\} \cup \Delta' \\ \hline \Sigma(p) = \langle _, _, u \rangle \quad u \neq \text{unknown} \quad \text{maybe_set_edge}(a, p) \mid \Gamma, \Sigma \rightarrow \cdot \mid \Gamma, \Sigma' \\ \hline \cdot \mid \Gamma, \Sigma, \Delta \rightarrow \cdot \mid \Gamma, \Sigma', \Delta' \end{array}$$

Rules MAYBE-SET-EDGE-CURRENT and MAYBE-SET-EDGE-OLD define the function `maybe_set_edge`. Rule MAYBE-SET-EDGE-CURRENT handles the case where the promise whose usage flag has been set is an input promise for the most recent activation of some constraint. In this case, the usage flag is assigned to the edge for this input, thereby altering the evaluation graph—since the evaluation graph is derived from the application of labels to the constraint graph (see Section 4.6.2). Rule MAYBE-SET-EDGE-OLD handles the case where there is an activation more recent than the one which uses the promise; in this case, no action is taken.

MAYBE-SET-EDGE-CURRENT

$$\begin{array}{c} \Gamma(a) = \langle m, \bar{p}, _ \rangle \quad \Gamma(m) = \langle c, _, (v_1, \dots, v_k), _ \rangle \quad \Gamma(c) = (\dots, a) \\ \bar{p} = (p_1, \dots, p_j, \dots, p_k) \quad p_j = p \quad \Sigma(p) = \langle _, _, u \rangle \quad \Sigma' = [\langle v_j, m \rangle \mapsto u] \Sigma \\ \hline \text{maybe_set_edge}(a, p) \mid \Gamma, \Sigma \rightarrow \cdot \mid \Gamma, \Sigma' \end{array}$$

MAYBE-SET-EDGE-OLD

$$\begin{array}{c} \Gamma(a) = \langle m, _, _ \rangle \quad \Gamma(m) = \langle c, _, _, _ \rangle \quad \Gamma(c) = (\dots, a') \quad a \neq a' \\ \hline \text{maybe_set_edge}(a, p) \mid \Gamma, \Sigma \rightarrow \cdot \mid \Gamma, \Sigma \end{array}$$

5.2.5 Lifting Functions

As discussed in Section 3.1.2, one approach for creating scheduling functions is to lift a function over values to create a function over promises. Here we give semantics for such a lifting mechanism.

Rule LIFT-FUNCTION defines the lift function. This function takes as parameters a function over values g , the number of outputs k returned by g , and some number j of input promises. The function first sets the usage flags of all input promises to **used**. It then creates and initializes k output promises. Finally, it adds an element to the callback set. The return value of the function is the output promises created. Note that by binding the first two of these parameters, we create a function which takes and returns promises; this binding of the first two parameters represents the act of “lifting” the function g .

LIFT-FUNCTION

$$\begin{array}{c}
\Sigma' = [\forall_{i=1}^j : \Sigma(p_i) = \langle t, s, \text{unknown} \rangle \implies p_i \mapsto \langle t, s, \text{used} \rangle] \Sigma \\
\forall_{i=1}^k : q_i = \text{newsym}() \quad \Sigma'' = [\forall_{i=1}^k : q_i \mapsto \langle \cdot, \text{pending}, \text{unknown} \rangle] \Sigma' \\
\Delta' = \{ \langle \text{lifted}, g, (p_1, \dots, p_j), (q_1, \dots, q_k) \rangle \} \cup \Delta \\
\hline
\text{lift}(g, k, p_1, \dots, p_j) \mid \Sigma, \Delta \rightarrow (q_1, \dots, q_k) \mid \Sigma'', \Delta'
\end{array}$$

The **lifted** callback element is handled by the rule LIFTED-INPUTS-READY. This rule may be invoked any time after all input promises have been fulfilled. The rule retrieves the value of all input promises, then invokes the lifted function g . The return values of g are then used to fulfill the output promises.

LIFTED-INPUTS-READY

$$\begin{array}{c}
\Delta = \{ \langle \text{lifted}, g, (p_1, \dots, p_j), (q_1, \dots, q_k) \rangle \} \cup \Delta' \\
\forall_{i=1}^j : \Sigma(p_i) = \langle t_i, \text{fulfilled}, _ \rangle \quad g(t_1, \dots, t_j) = (o_1, \dots, o_k) \\
\Sigma' = [\forall_{i=1}^k : \Sigma(q_i) = \langle _, \text{pending}, u_i \rangle \implies q_i \mapsto \langle o_i, \text{fulfilled}, u_i \rangle] \Sigma \\
\hline
\cdot \mid \Sigma, \Delta \rightarrow \cdot \mid \Sigma', \Delta'
\end{array}$$

5.3 Summary

These rules give a precise specification of the reaction of the property model to external events. These external events may be organized into three categories: the invocation of API functions by data bindings (`touch`, `set`, and `update`), the invocation of API functions by methods (`usage`, `subscribe`, `fulfill`, and `lift`), and the fulfilling of promises by methods. These rules reflect the non-determinism which naturally arises from an implementation using promises. Nevertheless, the application of these rules in any order allowed by their premises will always result in the same valuation of the property model's variables.

6. COMPONENTS AND DYNAMIC ELEMENTS

It is well established that software systems designed from reusable components tend to be more robust and less costly to develop than their hand-crafted counterparts [4, 16, 37]. To support a component-based approach to system design, a property model must be a reusable software artifact. It should be possible to build libraries of property model fragments corresponding to prevalent GUI elements or behaviors, and then to build complete GUIs by composing such fragments. To this end, we introduce the notion of a property model component.

There is an additional motivation for property model components. GUIs commonly support user actions that add or remove data, thereby adding or removing data dependencies as well. One common example is a table in a GUI that allows the user to add or remove rows. Data dependencies may exist, not only between data belonging to the same row, but between data belonging to different rows as well. As rows are added to or removed from the table, constraints may be added, removed, or modified to reflect the changes. Our goal is to make it easy to implement such operations as the addition or removal of property model components.

Components serve as containers for elements of the property model: variables, constraints, etc. Some of these elements are created as the component is created and destroyed as the component is destroyed. We refer to these elements as *static members* of the component. However, elements can also be defined in terms of the relationships between components. These elements may be automatically created or destroyed as the relationships between components change. We refer to these elements as *dynamic members* of the component.

As an example, consider the GUI shown in Figure 6.1. This GUI presents a schedule of events arranged as a table, with each row representing a single event. Each event contains a title, the time at which it begins, the time at which it ends, and its duration in minutes.

When considering the property model for this GUI, a natural design is to define each event as a separate component. To add a new event to the schedule, a new row is added to the table in the View and, at the same time, a new event component is created and added to the property model.

	<u>Title</u>	<u>Begin</u>	<u>End</u>	<u>Duration</u>	
1	<input type="text" value="Do something"/>	<input type="text" value="8:30"/>	<input type="text" value="9:00"/>	<input type="text" value="30"/> min	[X]
2	<input type="text" value="Do something else"/>	<input type="text" value="9:00"/>	<input type="text" value="10:00"/>	<input type="text" value="60"/> min	[X]
3	<input type="text" value="Do nothing"/>	<input type="text" value="10:00"/>	<input type="text" value="11:30"/>	<input type="text" value="90"/> min	[X]
	<input type="button" value="Add Event"/>				

Figure 6.1: A hypothetical scheduling application. Each row represents a single event. Constraints exist between elements of the same row as well as elements of different rows.

Each event component contains variables to store the data associated with the event: the variable t for title, b for begin time, e for end time, and d for duration. The component also contains a constraint to enforce the relationship between the begin time, end time, and duration, i.e., $b + d = e$. These variables and this constraint are static members of the component: they are created with the component and exist as long as the component itself does.

For every component except the last, an additional constraint is needed between the end time of the event and the begin time of the next event. We can express this constraint generically as the relation $e_i \leq b_{i+1}$, i.e., the end time of event i must be less than or equal to the begin time of event $i + 1$. This generic constraint defines a pattern that can be instantiated for each event. Furthermore, this pattern can be used to update constraints as rows are added to, removed from, and rearranged in the table.

This chapter presents a component mechanism that serves to group property model elements into components and gives a precise definition for how dynamic members arise as components are connected to one another.

6.1 Components and Composition

From the perspective of a property model, a component is simply a collection of property model elements grouped together as a single entity, allowing them to be added to and removed from the property model as a group. We define a *property model element* as one of the following:

- a *variable*, representing a storage location,
- a *constraint* in the constraint system,
- a *command* in the property model,
- a *touch dependency* between two constraints,
- an *output* designation for a variable, or
- a property model component.

A component *owns* the elements of which it is composed; thus, it is responsible for creating and destroying them. Static members are created as the component is created and destroyed as it is destroyed; dynamic members are created or destroyed as relationships between components change. It follows that each property model element belongs to exactly one component. An element may not be removed from, nor outlive the component it belongs to.

We assume some mechanism for referencing property model elements from outside the component, such as pointers. We refer to these as *references*. Property model elements, too, may contain references to other elements. For example, a constraint may contain references to the variables used by the constraint. Note, however, that references do not indicate ownership. Thus, a constraint does not own the variables for which it has references.

A property model *manages* a set of property model elements as described in Chapters 3 and 4: solving the system of constraints, scheduling the dataflow functions, etc. However, we define a property model, not as a collection of elements, but as a collection of components. The elements managed by the property model are exactly those contained by the components which define it. It follows that elements are not added directly to or removed

directly from a property model, but rather indirectly by adding or removing components.

A component may contain a nested component. As with other component members, a component owns its nested components, and is responsible for creating and destroying them. When a component is added to a property model, we recursively add all nested components to the property model as well.

A component may contain two additional types of members. These member types are different in that they are not considered to be property model elements; the property model does not manage them or interact with them directly. The first type is *templates*, which serve as patterns for dynamic elements. The second is *reference variables*, used to hold references to other property model elements. Together, templates and reference variables are the basis for dynamic elements, discussed in Section 6.2.

A component uses reference variables to identify and keep track of related components and/or elements. For example, components in a list may store a reference to the next element in the list; components in a tree may store references to child components. Composition of property model components consists not only in grouping components together, but also assigning references to components or elements as appropriate to signify the correct relationships between the components.

Reference variables are the only component elements that may be modified destructively: a reference variable may be modified to refer to some other element or, in the case of the null reference, no element at all. All other component members are assumed to remain unchanged for their entire lifespans. This includes dynamic elements; changes to relationships between components may result in the destruction of existing elements and the creation of new ones, but never modifications of existing ones.

Though a component owns all reference variables it contains, it does *not* own the elements referred to. Thus, it does not create or destroy the elements, and adding the component to a property model does *not* add the elements. It is the programmer's job to ensure that an element inside the property model does not refer to an element outside the property model.

HotDrink provides support for creating and composing reusable property model components. Section 3.5 introduced HotDrink, and described how an individual component may be created using an embedded DSL provided by the `ComponentBuilder` object. The same sequence of commands that can be used to create a component can also be used to create a component *specification*. A specification is a data structure which serves as a pattern for creating a new component. Multiple components can be created from the same specification, allowing reuse.

```

1 var EventSpec = new hd.ComponentBuilder()
2   .vs("t, b, e, d", {d: 10})
3   .r("n")
4   .c("b, e, d")
5     .m("b, d → e", hd.sum)
6     .m("e, d → b", hd.diff)
7     .m("e, b → d", hd.diff)
8   .spec();
9
10 var evt1 = new hd.Component(EventSpec, {b: 0});
11 var evt2 = new hd.Component(EventSpec);
12 var evt3 = new hd.Component(EventSpec);
13
14 var pm = new hd.PropertyModel();
15 pm.addComponent(evt1);
16 pm.addComponent(evt2);
17 pm.addComponent(evt3);
18
19 evt1.n = evt2;
20 evt2.n = evt3;

```

Figure 6.2: Creating and composing components in HotDrink.

Figure 6.2 shows the code needed to define and compose event components for the scheduling application of Figure 6.1. The construction of the specification begins on line 1 with the creation of a `ComponentBuilder` object. Line 2 defines the four variables `t`, `b`, `e`, and `d` for this component, providing a default value for the variable `d` alone. Line 3 defines a reference variable, `n`, to hold a reference to the next event in the list. Line 4 defines

a constraint between `b`, `e`, and `d`; that constraint has three methods, given on lines 5–7. Finally, line 8 retrieves the component specification; had this line instead been a call to the `component` function, as in Figure 3.11, we would have retrieved an actual component rather than a component specification.

Lines 10–12 show the creation of three components using the specification. As an alternative, we could have defined a subtype of `hd.Component` which was automatically constructed according to this specification. Note that the construction of `evt1` on line 10 includes an initial value for the variable `b`. The three components are then added to the property model on lines 15–17. This is the first step in composing them: group them together in the same property model. The second step is to assign references; this is done in lines 19–20. In this example, the assigning of references has no effect other than to link the components together. The next section examines how we can generate dynamic elements in response to this act of composition.

6.2 Templates and Dynamic Elements

A *template* is a pattern for creating a dynamic element. We can create templates for any property model element which contains a reference to another element—specifically, constraints, commands, touch dependencies, and output designations. A template contains all required information for constructing one such element, except for one or more of the required references. Missing references are represented by a marker called a *path*. A path describes a potential property model element using one or more reference variables.

As an example, consider the table of events used by the scheduling application of Figure 6.1. As mentioned at the beginning of this chapter, we may represent this table by creating a component for each event containing variables t , b , e , and d . Each component also contains a static constraint, call it C , for the relation $b = e + d$.

In order to create a constraint between the end time e of one event and the begin time b of the next, we must first add to each component a reference variable, n , to hold a reference to the next event. Now we can define a constraint template T using a reference to the variable e and the path $n.b$, where $n.b$ describes the b variable of the component

referred to by n . The constraint represented by this template enforces the relation $e \leq n.b$.

Figure 6.3 gives an illustration of such a component. The dashed rectangle represents the component itself. It contains variables t , b , e , and d , represented as rectangles, and also constraint C , represented as a circle. We use edges to connect node C with each variable used in the constraint. The reference variable n is represented with a triangle; for now, we assume this variable holds a null reference, indicated by the grounded arrow. The template T is drawn as a constraint in gray; we may think of this as a *potential* constraint. Notice the node for T shares an edge with the variable e , and also a node indicated by the path $n.b$. The node $n.b$ is drawn outside of the component because it describes a variable not belonging to the component.

When all paths in a template describe valid elements, the template may be *instantiated* by creating a new element of the type corresponding to the template, replacing all paths with references to the elements they describe. In the case of template T , when the reference variable n is assigned a reference to another event component, T may be instantiated as a constraint between the variable e and the variable $n.b$.

Figure 6.4 gives an illustration of three event components in a list. Notice the reference variables in the first and second components contain references to the next component in the list. Therefore, in these two components the path $n.b$ describes a valid variable, and the template T may be instantiated into a constraint. This is indicated in the figure by drawing the template in black, and replacing the node $n.b$ with the variable described by

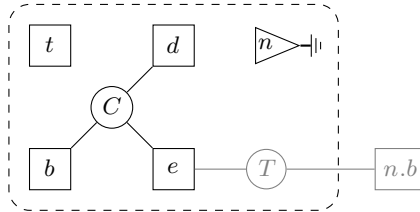


Figure 6.3: Representation of a component containing a template. Variables and paths are squares; constraints and templates are circles; reference variables are triangles. Variables and constraints are black; paths and templates are gray. The dashed rectangle indicates which elements belong to the component.

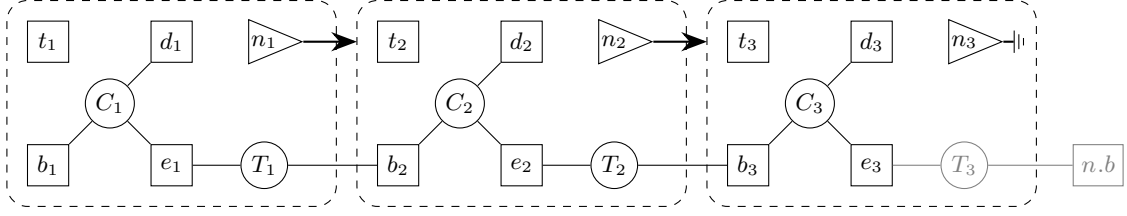


Figure 6.4: Representation of multiple components containing instantiated templates. Templates T_1 and T_2 have been instantiated to create a constraint between successive events.

that path—i.e., the b variable of the component referred to by n . In the third component, however, the template remains uninstantiated, indicated by the gray color.

Because reference variables may change, the element described by a path in a template may change as well. If this happens, all instantiations of that element are automatically destroyed; then new instantiations may be constructed as appropriate.

Figure 6.5 gives an example of this process by illustrating the insertion of a new component in a list of components. Figure 6.5a shows the original list of three components. This list is identical to the one in Figure 6.4 except that we have omitted n and t from the figure for simplicity. As soon as the reference to from the first to the second component is modified, the instantiation of template T_1 is destroyed. This may be seen in Figure 6.5b; template T_1 is gray, indicating that it is no longer instantiated. Figure 6.5c shows the list with the new element positioned for insertion, but with none of the reference variables yet assigned. Once they are assigned, the templates T_1 and T_4 may be instantiated to create the necessary constraints.

It is important to note that constraint T_1 in Figure 6.5a and constraint T_1 in Figure 6.5d are two different constraints. We have labeled them both T_1 because they are both instantiations of template T_1 ; however, the property model sees them as two completely separate constraints. This is in keeping with our semantics that constraint definitions are not altered.

In HotDrink, the syntax for creating a template is identical to the syntax for creating

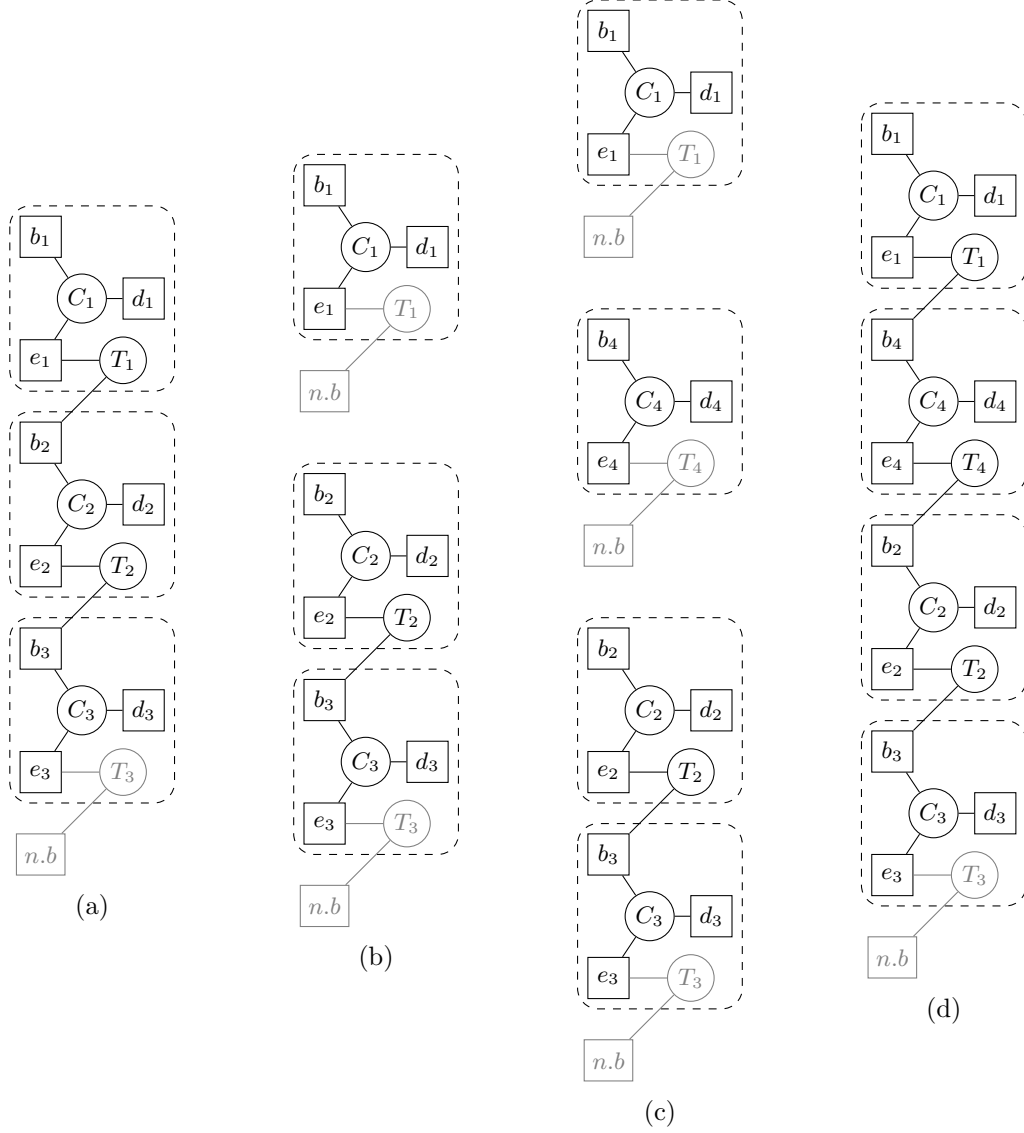


Figure 6.5: Constraints modified by inserting a component into a list of components. The original list is shown in (a) with constraints T_1 and T_2 generated by that list. Breaking the list at the insertion point results in the deletion of constraint T_1 , as shown in (b). The new element is positioned in (c); once references are connected, constraints T_1 and T_4 are created, as shown in (d).

```

1 var EventSpec = new hd.ComponentBuilder()
2   .vs("t, b, e, d", {d: 10})
3   .r("n")
4   .c("b, e, d")
5   .m("b, d → e", hd.sum)
6   .m("e, d → b", hd.diff)
7   .m("e, b → d", hd.diff)
8   .c("e, n.b")
9   .m("!e, n.b → e", hd.min)
10  .m("e, !n.b → n.b", hd.max)
11  .spec();

```

Figure 6.6: Creating a constraint template in HotDrink.

a property model element. HotDrink decides whether it should create a template based on whether any of the paths involved make use of reference variables. Figure 6.6 shows an extended version of the event component from Figure 6.2, this time with a template for the constraint between adjacent events. The template is defined on line 8. The only indication that it is a template is that it contains the path `n.b` where `n` is a reference variable. Every time a reference is assigned to `n`, all existing instantiations of this template are destroyed and, if the assigned reference identifies a valid event component, a new instantiation is created.

6.3 Templates as Signals

This section provides a detailed account of how templates may be instantiated to produce dynamic elements. For the sake of clarity, this process is defined in a purely-functional notation for types inspired by Haskell. The purpose in using this notation is to communicate intent, not to suggest implementation.

6.3.1 Signals

We begin by adopting a concept of Functional Reactive Programming (FRP) [12, 52]: a *signal* is a value which can change over time.¹ The use of signals makes it possible to

¹The term *behavior* is sometimes used for this concept as well. The terms *behavior* and *signal* are frequently interchangeable; however, *behavior* is more common when referring to implementation as a function, whereas *signal* is more common when referring to implementation as a sequence of values.

abstract away changes to values over time and focus instead on the computations that occur as values change. A signal may be represented as a function of time; the value of a signal s at time t is $s(t)$. Thus, we may define the type of a signal as follows.

type Signal $T = \text{Time} \rightarrow T$

This functional representation makes it easy to define new signals from old ones. For example, given two signals a and b , we may define a new signal whose value is the sum of a and b by the function $\lambda t.a(t) + b(t)$. In practice, however, signals are rarely implemented as functions like this due to efficiency concerns. In FRP, signals are often assumed to be discrete functions, and are represented as sequences of time-value pairs. It is also possible to adopt a *push-based* approach following the *Observer* pattern. In this approach signals publish a notification any time their values change. This approach is discussed further in Section 6.3.6.

An example of a signal in the context of property models is a reference variable of a component. We would like to think of this variable not as a pointer, but as a property model element that may change over the life of the program. Additionally, because a reference variable may contain the null reference, it is possible that it may represent no value. In keeping with Haskell, we use the type **Maybe** T to represent either a value of type T or no value at all. Thus, we may define the type of a reference variable as follows.

type ReferenceVariable $T = \text{Signal } (\text{Maybe } T)$

6.3.2 Labels

We assume each member of a property model component is given a label that may be used to refer to it. We adopt the standard dot-notation for applying a label to a component: if e is a component, then $e.l$ identifies the member l in e . We require a label to have some run-time representation. In practice, this may take the form of an offset that can be combined with the component address, or a string that can be mapped to an element, perhaps using reflection. Here we simply use **Label** T as the type for a label

identifying an entity of type T . Thus, if $e.l$ denotes a variable, its type is **Label Variable**.

A label may be applied to a component at run-time. We generalize this with the function **applyL** which returns a signal whose value is the result of applying a label to a component. A label may identify a reference variable, so the type of the signal must be **Signal (Maybe T)**. If the label identifies a member other than a reference, then **applyL** simply returns the constant signal whose value is always that member. This avoids the unnecessary distinction between members contained directly by the component and members referred to by a reference variable. The type of **applyL** is as follows.

applyL :: Component \rightarrow Label $T \rightarrow$ Signal (Maybe T)

In practice, the type of a label may or may not be known. In a statically-checked environment, we may enforce restrictions on the type of labels. In a dynamically-checked environment, we may have to wait until run-time to ensure that a label produces the correct type. We abstract away this detail by assuming that if at any time the application of a label does not produce a value of the correct type, then at that time the signal returns no value.

6.3.3 Paths

We define a path to be a non-empty sequence of labels. When writing paths, we use dots to separate the labels—e.g., $n.b$ is a path consisting of two labels: n and b . A path is applied to a component by applying each label in turn. For example, applying the path $n.b$ to the component c , written $c.n.b$, requires first applying n to c , then b to the result of the first application. This implies that each label in a path, with the possible exception of the last, should identify a component for the next label to be applied to. Therefore, we may represent a path as a (possibly empty) list of component labels, followed by one additional label of any type.

type Path $T = ([\text{Label Component}], \text{Label } T)$

For example, if our representation of a label is a string, then the path e is represented as

([], "e") and the path *n.b* as (["n"], "b").

Just as with labels, we must be able to apply a path to a component at run-time. This we generalize with the function `applyP`. This function returns a signal whose value is the final result obtained after applying each label in turn. If at any time one of the applications fails to produce a value (due to a null reference or type error), then at that time the signal returns no value. The type of `applyP` is as follows .

`applyP :: Component → Path T → Signal (Maybe T)`

6.3.4 Templates

We define a template as a property model element in which all references to other elements are represented by paths. This is a generalization; in reality, a template may contain both references and paths. However, restricting templates to contain only paths simplifies the representation. We simply treat a reference as a constant path, that is, a path that always describes the same variable.

To give an example of a template, we must first define a property model element. Let us represent a constraint as a list of variables and a list of methods. Assume that methods do not reference variables directly, but rather through the constraint's variable list, e.g., using an index. This gives the following type.

type Constraint = ([Variable], [Method])

In this case, a constraint template would have the same representation, with the exception that variables would be replaced with paths to variables. Thus, the type for a constraint template is as follows.

type ConstraintTemplate = ([Path Variable], [Method])

Just as a label or path can be applied to a component, so too can a template be applied to a component. This requires replacing each path in the template with the result of applying the path to the component. This application process fails to produce a result if

the application of any path fails to produce a result. Again, we represent this application with a signal, given by the function `applyT`.

`applyT :: Component → ConstraintTemplate → Signal (Maybe Constraint)`

In a similar manner, we may define templates for commands, touch dependencies, and outputs, along with application functions for each.

We do not consider it an error condition when the application of a path fails to produce a result; the template is merely not instantiated and produces no value. However, it is possible that all paths produce a value, but that the instantiation would still result in an invalid element. In such a case, we say that instantiation has failed. In a dynamically typed environment, an instantiation may fail if a path produces a value of the wrong type. Even if all values are of the correct type, there are still possibilities of error due to *aliasing*, i.e., when multiple paths refer to the same element. Although we do not consider aliasing itself to be an error, it is likely to produce invalid elements, such as a constraint that outputs to the same variable twice. A failed instantiation produces no value. It is, however, likely an indication of programmer error, and it may be beneficial to produce a warning message.

6.3.5 Resulting Elements

Finally, we may define components themselves. Here, we take `Element` to be the union of all property model element types and `Template` to be the union of all template types. We may define a component as consisting of static elements, templates, and references, like so:

type `Element` = `Variable` | `Constraint` | `Command` | `TouchDependency` |
`Output` | `Component`
type `Template` = `ConstraintTemplate` | `CommandTemplate` |
`TouchDependencyTemplate` | `OutputTemplate`
type `Component` = (`Set Element`, `Set Template`, `Set Reference`)

As mentioned previously, property models view a component as a set of property model elements. This set may change as dynamic elements are created or destroyed; therefore

we represent it as a signal, given by a function `elements`. The value of this signal at any given time is the set containing all static elements of the component, together with the instantiations produced by all templates of the component at that time, excluding the templates that cannot be instantiated.

`elements :: Component → Signal (Set Element)`

This signal represents the entire interaction between a property model and one of its components. It defines, at any given time, the property model elements contained by the component. To place this in the context of property model execution, recall from Chapter 3 that the property model responds to edits by solving a constraint system. The constraint system solved is defined by the variables and constraints contained by the property model, which, in turn, is defined by the members of the components that compose the property model. Thus, it is as if, after every edit, the property model queries the `elements` signal for each of its components, takes the union of the results, then solves the constraint system defined by those elements.

6.3.6 Implementation

There are two general approaches to signal implementation. The first is a *pull-based* approach in which a signal must be queried for its value. This is the approach used throughout this section (Section 6.3) by defining a signal as a function. Using this approach requires, at the beginning of every update, the property model to query each component to retrieve the elements of that component. Querying the component results in a query the value of each of its template, which causes templates to query the value of their paths, and so on.

The second is a *push-based* approach in which the signal publishes a notification every time its value changes. In this approach, we assume that the set of property model elements remains unchanged until notification is given otherwise. When the programmer modifies a reference variable, that variable gives notification to any paths which use it. This results in the paths giving notification to any templates using them, the templates to give notification

to their components, and so on.

This push-based approach requires slightly more coordination, since previous signal values must be remembered and altered according to notifications—though reactive programming techniques can lighten this burden considerably. The advantage of this approach is that, when changes occur, only the affected elements must be recalculated. This can considerably lighten the work involved for small changes to the property model. This is significant, as small changes to the property model are relatively much more common than large changes. For example, adding an event to the scheduling application requires only the addition of three variables and two constraints; the rest of the property model can be left unchanged.

6.4 Dynamic Elements Using Arrays

Sequences or lists arise commonly in GUIs—for example, options in a selection or rows in a table. Section 6.2 discussed how reference variables allow dynamic constraints to be defined between adjacent components in a sequence by giving each component a reference to the next component in the sequence. However, this linked-list implementation is not always ideal. Just as a linked list makes random access difficult, it also makes it difficult to define dynamic elements with constraints that extend further than one row and the next.

When random access is required, the preferred representation for a sequence is usually an array, allowing elements to be selected by an index. Use of arrays in property model components allows dynamic elements in a sequence to be expressed more naturally and can support a wider variety of templates. For example, returning to the scheduling application example of Figure 6.1, if we define the property model for the table as an array s (for “schedule”) in which each array element is a component containing variables b , e , and d , we may express the relation between table rows using standard array notation as $s[i].e \leq s[i+1].b$. The interpretation for this relation is, for every integer i such that i and $i+1$ are valid indices of s , the end time of component $s[i]$ is less-than-or-equal-to the begin time of component $s[i+1]$. As another example, we could guarantee no more than four events in an hour using a constraint for the relation $s[i].b + 60 \leq s[i+4].b$. We could even require

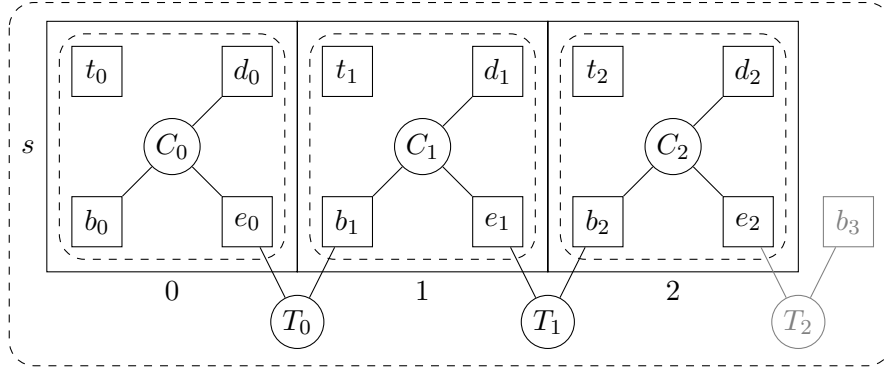


Figure 6.7: Representation of a constraint containing an array and an array template. Variables and paths are squares; constraints and templates are circles; property model components are dashed rectangles. The array s is a sequence of squares representing the individual elements of the array; below each square is the corresponding array index. Paths and templates are gray.

a half-hour between even-numbered and odd-numbered events using a constraint for the relation $s[2i].e + 30 \leq s[2i + 1].b$.

Figure 6.7 shows a representation of a component containing an array. The outermost dashed rectangle represents a component containing an array s of event components, as well as a constraint template T for the relation $s[i].e \leq s[i + 1].b$. The array has a length of three, allowing T to be instantiated twice: once for $i = 0$ and once for $i = 1$, resulting in constraint T_0 and T_1 respectively. Because the template T belongs to the outermost component, so too do all instantiations of T . We may contrast this with Figure 6.4 in which each component had its own template with, potentially, its own instantiation.

We allow for the possibility that arrays may change size over the life of the program. If this happens, new instantiations should be created or old ones deleted as appropriate. In the figure, node T_3 represents a potential constraint that will be instantiated should the array be extended so that 3 is a valid index.

Figure 6.8 contains JavaScript code that creates the property model of Figure 6.7. Lines 1–7 define the specification for a component type for events; this is the type of each member of the array. Lines 9–14 define the outermost component. The array s is defined on

```

1 var EventSpec = new hd.ComponentBuilder()
2   .vs("t, b, e, d", {d: 10})
3   .c("b, e, d")
4   .m("b, d → e", hd.sum)
5   .m("e, d → b", hd.diff)
6   .m("e, b → d", hd.diff)
7   .spec();
8
9 var model = new hd.ComponentBuilder()
10  .n("s", hd.arrayOf(EventSpec))
11  .c("s[i].e, s[i+1].b")
12    .m("s[i].e, !s[i+1].b → s[i+1].b", hd.max)
13    .m("!s[i].e, s[i+1].b → s[i].e", hd.min)
14  .component();
15
16 model.s.expand( 3 );
17 model.s[0].b.set( 0 );
18
19 var pm = new hd.PropertyModel();
20 pm.addComponent(model);
21 pm.update();

```

Figure 6.8: Creating and composing components in HotDrink.

line 10 as a nested component—i.e., a component contained by the outermost component. The arguments to the `nested` function are a name for and constructor to create the nested component. The function `hd.arrayOf` is a helper function which creates a constructor for an array component; the argument is the element type of the array. The template T is defined on line 11. As before, the only indication that this is a template is that it references elements of the array `s`. Before the array is added to the property model, we dynamically adjust its size by calling the `expand` function on line 16. This function extends the length of the array by the amount given as a parameter. We also set the begin time `b` of the first event on line 17.

In this code listing we create an array of size three to match the picture in Figure 6.7. However, array components in HotDrink may be dynamically resized. As the size of the array changes, new event components are created or destroyed, as are instantiations of the constraint template. These modifications to the property model are handled automatically

in response to the change in array size.

6.5 Array Components as Signals

This section provides a detailed account of how array templates may be instantiated to produce dynamic elements. We do this by revising the function types given in Section 6.3 to take array components into account. As in Section 6.3, this is done to communicate intent, not to suggest implementation.

6.5.1 Array Components

We define a special property model component type called an *array component* in which elements are labeled by numbers rather than identifiers. An array component represents, but is distinct from, an array of the implementation language. Array components are treated as a subtype of ordinary components. Thus, an array component representing an array of property model elements owns the elements it contains, and adding the array component to the property model recursively adds all of its elements to the property model. We may also define an array component to represent an array of reference variables. As with all property model components, the array component would *not* own any elements referred to by reference variables.

We define the type of an array component as a signal whose value is an array of the implementation language, as shown below. Because the value of a signal may change over time, the array component may represent many different arrays; alternatively, we may interpret this as a single array which may be modified—both by changing the size of the array, as well as by modifying elements if their type supports such operations (e.g., reference variables.) To accommodate the possibility of null references, the type of each element in the array is `Maybe T`.

$$\text{asArray} :: \text{ArrayComponent } T \rightarrow \text{Signal } (\text{Array } (\text{Maybe } T))$$

6.5.2 Indexing Expressions

The addition of array components means that paths may include, not only labels, but also *indexing expressions*. An indexing expression is a mathematical expression containing no more than one free variable, known as the *index variable*. We use the standard notation of writing indexing expressions inside square brackets. For example, the path $a[i + 1]$ consists of the label a followed by the indexing expression $i + 1$ with index variable i . In its most general form, an indexing expression is simply a bijection between two sets of integers. This bijection may be represented by a function and its inverse, called the *indexing function* and the *inverse indexing function*, respectively. For example, the indexing expression $i + 1$ represents the indexing function $\lambda n.n + 1$ and inverse indexing function $\lambda n.n - 1$. We give both of these functions the type $\text{Int} \rightarrow \text{Maybe Int}$ to accommodate the fact that not every integer may map forward or in reverse. For example, the indexing expression $i/2$ will not produce a value for odd integers.

Although array components represent one-dimensional arrays, we can simulate multi-dimensional arrays through the use of arrays-of-arrays. For this reason, it may be desirable to permit paths to contain indexing expressions which use different index variables. For example, given a table t , we might wish to create a constraint between consecutive elements of the same column—e.g., $t[i][j]$ and $t[i][j + 1]$. In theory, the names of these index variables do not matter. In practice, it is helpful to define the names of index variables ahead of time to make them easy to identify. In HotDrink, the names i , j , and k are initially recognized as index variables, though this is configurable. For purposes of this discussion, we assume n index variables named i_1, \dots, i_n . We distinguish between indexing expressions which use different index variables by assigning them different types, as shown below. Here we define types supporting expressions of two index variables, i_1 and i_2 . This may be extended to any number of index variables as desired.

```

type Index0 = Int
type Index1 = (Int  $\rightarrow$  Maybe Int, Int  $\rightarrow$  Maybe Int)
type Index2 = (Int  $\rightarrow$  Maybe Int, Int  $\rightarrow$  Maybe Int)

```

Here, the type `Index0` represents a *constant indexing expression*—that is, an expression without an index variable, e.g., as in the path `a[3]`. The type `Index1` represents an indexing expression using index variable i_1 , and `Index2` represents an indexing expression using index variable i_2 .

Just as a label may be applied to an ordinary component, so too may an indexing expression be applied to an array component, provided we know the value of the expression’s index variable. We represent this application with a set of functions, `applyAn`, where $n = 0, 1, 2, \dots$. Because the value of an array at a given index may change over time, the return type of this function is a signal. The value of this signal is obtained by applying the indexing function to the value of the index variable, then using the result as an index into the component’s array. For example, if the indexing expression is $i_1 + 1$ and the value of index variable i_1 is 4, then the signal would result in the element of the array with index 5. The signal has no value if the indexing function itself returns no value, if the indexing function returns an out-of-bounds index, or if the corresponding element of the array has no value (e.g., a null reference). Note that no integer is required to apply a constant indexing expression since it has no index variable.

```

applyA0 :: ArrayComponent T → Index0 → Signal (Maybe T)
applyA1 :: ArrayComponent T → Index1 → Int → Signal (Maybe T)
applyA2 :: ArrayComponent T → Index2 → Int → Signal (Maybe T)

```

Now let us consider the case where the value of the index variable is unknown. Calling `applyA1` or `applyA2` with only an array component and an indexing expression yields a function of type `Int → Signal (Maybe T)`; that is, it yields a function mapping a value of an index variable to a value of the array. In theory, this function could be used to iterate over all possible values of the given indexing expression. In practice, however, such a function is a poor data structure for iteration since most integers will map to no value. Remembering that a signal is simply a function, switching the order of the parameters gives us `Signal (Int → Maybe T)`. We may then replace the function type `Int → Maybe T` with the type for a map data structure, `Map Int (Maybe T)`; such a data structure is much

more practical for iteration. We assume such a map would contain only integers which map to a value. For this reason, there is no longer any need for the `Maybe` type. This gives the type `Signal (Map Int T)`. We define alternative versions of the `applyAn` functions, named `queryAn`, for use when only the first two parameters are known. Note that we need not define `queryA0` since type `Index0` represents a constant indexing expression.

```
queryA1 :: ArrayComponent → Index1 T → Signal (Map Int T)
queryA2 :: ArrayComponent → Index2 T → Signal (Map Int T)
```

6.5.3 Paths

Just as we classify indexing expressions by the index variable used, we may similarly classify paths according to the index variables they contain. We say an *n-dimensional* path contains indexing expressions for *n* distinct index variables. (Note that this is different from saying it contains *n* indexing expressions because a path could contain two indexing expressions of the same variable.) In this discussion, we will use the simplifying assumption that, if a path uses *n* index variables, it will be the first *n* index variables, i.e., i_1 through i_n . Removing this assumption complicates the implementation slightly, but does not alter the basic theory.

We define a distinct type for each different possible dimension of a path. As noted, a path may now contain, not only labels, but also indexing expressions of different index variables. To generalize, we refer to each individual element of a path as a *leg*. We define a leg type for each dimension as a union of the allowable types for that dimension. We then define a path as a non-empty sequence of legs of the appropriate type.

```
type Leg0 T = Label T | Index0 T
type Leg1 T = Label T | Index0 T | Index1 T
type Leg2 T = Label T | Index0 T | Index1 T | Index2 T

type Path0 T = ([Leg0 Component], Leg0 T)
type Path1 T = ([Leg1 Component], Leg1 T)
type Path2 T = ([Leg2 Component], Leg2 T)
```

The type **Path0** represents a 0-dimensional path—i.e., a path containing only labels and constant indexing expressions. The type **Path1** represents a 1-dimensional path; thus, it may also contain indexing expressions using index variable i_1 . The type **Path2** represents a 2-dimensional path; thus, it may also contain indexing expressions using index variables i_1 and i_2 . These typing rules are ambiguous: type **Path0** is actually a subtype of **Path1**, **Path1** of **Path2**, etc. We resolve this ambiguity by saying a path should be given the smallest dimension possible.

Applying a path to a component is similar to applying an indexing expression. If values are known for all index variables, applying path results in either a single value or no value. Determining the value requires applying each leg of the path in turn. Labels are applied using **applyL**; indexing expressions of type **Index0** are applied using **applyA0**; indexing expressions of type **Index n** (for $n = 1, 2, \dots$) are applied using **applyAn** with the value of the n^{th} index variable. Because of our assumption that a path always uses the first n index variables, we may use an n -tuple to represent an assignment of values to index variables, e.g., (i_1, i_2, \dots) .

```

applyP0 :: Component → Path0 T → Signal (Maybe T)
applyP1 :: Component → Path1 T → Int → Signal (Maybe T)
applyP2 :: Component → Path2 T → (Int, Int) → Signal (Maybe T)

```

Just as with the **applyAn** functions, calling an **applyP n** function with only two parameters yields a function mapping index variable values to path values: $\text{Int} \rightarrow \text{Signal (Maybe T)}$ in the case of one-dimensional paths, and $(\text{Int}, \text{Int}) \rightarrow \text{Signal (Maybe T)}$ in the case of two-dimensional paths. In general, an n -dimensional path will produce a mapping from n -tuples to values. And, as with the **applyAn** functions, we define alternative versions of the **applyP n** functions, named **queryP n** , that provide data structures more suitable for iteration.

```

queryP0 :: Component → Path0 T → Signal (Map () T)
queryP1 :: Component → Path1 T → Signal (Map Int T)
queryP2 :: Component → Path2 T → Signal (Map (Int, Int) T)

```

Note that the function **queryP0** is essentially identical to **applyP0**: we have simply

replaced the type `Maybe T` with the type `Map () T`—a map which can contain at most one entry. In this way, we provide a consistent interface: each function `queryP n` returns a map from n -tuple to value.

6.5.4 Templates

Templates are classified by dimension in a manner similar to paths. An n -dimensional template uses paths of n or fewer dimensions. Below we give the types for constraint templates of dimension 0, 1, and 2. As with paths, we resolve the ambiguity in these types by saying a template should be given the smallest dimension possible.

```

type PathLE0 T = Path0 T
type PathLE1 T = Path0 T | Path1 T
type PathLE2 T = Path0 T | Path1 T | Path2 T

type ConstraintTemplate0 = ([PathLE0 Variable], [Method])
type ConstraintTemplate1 = ([PathLE1 Variable], [Method])
type ConstraintTemplate2 = ([PathLE2 Variable], [Method])

```

As with indexing expressions and paths, so with templates can application take two forms. If values are known for all index variables, application generates at most one instantiation. To apply the template, we apply each path to the component in turn. An n -dimensional path must be applied with `applyP n` and the values of the first n index variables. If all applications give valid results, we may create an instantiation, otherwise not.

```

applyT0 :: Component → ConstraintTemplate0 → Signal (Maybe Constraint)
applyT1 :: Component → ConstraintTemplate1 → Int → Signal (Maybe Constraint)
applyT2 :: Component → ConstraintTemplate2 → (Int, Int) → Signal (Maybe Constraint)

```

If values are not known for all index variables, application generates a mapping from index variable values to instantiations.

```

queryT0 :: Component → ConstraintTemplate0 → Signal (Map () Constraint)
queryT1 :: Component → ConstraintTemplate1 → Signal (Map Int Constraint)
queryT2 :: Component → ConstraintTemplate2 → Signal (Map (Int, Int) Constraint)

```

6.5.5 Resulting Elements

For an n -dimensional template, the function `queryT n` yields a map from n -tuple to property model element. The values in this map are all possible instances of the template. All of these are considered dynamic members of the component and must be included in the set of elements returned by the `elements` signal of Section 6.3.5. As far as the property model is concerned, the keys of this map are of no consequence. However, the keys do uniquely identify each instantiation, which is important for an incremental implementation, as we shall discuss in the next section. Note that templates of all dimensions except zero have the potential to generate multiple elements; 0-dimensional templates still generate at most a single element. All other elements in a component remain the same.

6.5.6 Implementation

As before implementation can follow a pull-based or push-based approach. And, as before, the push-based can help to minimize the work required for small changes. We give here two notes regarding the push-based approach.

First, in order to allow incremental updates of the property model we may replace a signal of type, e.g., `Signal (Map Int T)` with a signal of type `(Int, Maybe T)` representing changes made to the map. In this way, when a single element of an array changes (e.g., when adding a new element to the end), paths using that element may report a single change, and templates using those paths may update a single instantiation.²

Second, there may be times when a change results in a large number of elements “switching” keys. For example, suppose in our scheduling example that the user moves the last event in the array to the front of the array. This change causes the first event to become the second event, the second event to become third, and so on. Effectively, this modifies every element of the array, thereby forcing the template for the relation $s[i].e \leq s[i + 1].b$ to rebuild all instantiations. Yet, if we were to compare the set of constraints previously

²A template may still need to update multiple instantiations if the path that is affected has a smaller dimension than the template itself. If a template has dimension n , then a path of dimension $m < n$ may be used in multiple instantiations.

generated by the template to the set of constraints generated after the change, we would find only two differences: a constraint removed and a constraint added. Because the component only cares about the constraints contained by the template, and not the keys that map to them, performing such a comparison of the sets can minimize the number of changes reported to the property model. This is the approach taken in HotDrink.

6.6 Signature Variations

As described in Chapter 3, dataflow functions in the property model—i.e., methods and operations—are represented by a scheduling function whose parameters and return values are promises for the inputs and outputs of the dataflow function. When invoking a scheduling function, the parameters must be passed in a particular order; we similarly assume that scheduling a function returns multiple promises in a particular order. In Chapter 5, we represented the inputs and outputs of a dataflow function as sequences of variables; these sequences also served to indicate the order in which promises are passed to and returned from the scheduling function. We refer to these two variable sequences together as the *signature* of the function, or individually as the *input signature* and *output signature*, respectively.

For purposes of this discussion, we will write input and output signatures as a list of variables in parentheses. For example, we write an input (or output) signature consisting of, in order, the variables a , b , and c as (a, b, c) . We write a full signature by separating inputs and outputs with an arrow. For example, $(a, b) \rightarrow (x, y)$ indicates the inputs are, in order, a and b , and the outputs are, in order, x and y .

Making the input and output signature of a scheduling function distinct from the input and output sets of a dataflow function allows us some additional flexibility. For example, the input signature of a method need not include every input variable. This could be useful if a variable is not needed, but must be included as an input to satisfy *method restriction* (see Section 2.2.1). Another example of added flexibility is an input signature that contains the same variable twice, thereby allowing the value of one variable to be passed as the argument for two separate parameters.

This section examines two types of signature variations that enable a wider range of dynamic elements to be created: signatures containing constant values, and signatures containing nested signatures.

6.6.1 Constants and Partial Instantiation

We allow an input signature to contain constant values—e.g., numeric or string constants. When calling the dataflow function, these constant values are passed to the corresponding parameter. For example, an input signature of $(a, 3, b)$ indicates that the scheduling function is to be called with three parameters: the value of the variable a , the number 3, and the value of the variable b . Variable values are always passed as promises; for consistency, constant values are converted to promises as well: a new promise is created and immediately fulfilled using the constant value. In this way, scheduling functions need not know whether a particular parameter comes from a variable or a constant. This promotes reusability by allowing general functions to be used in specific situations by fixing one or more of their parameters. Note that an output signature cannot contain a constant value since a constant cannot be written to.

We extend our definition of a component to include constant values of the implementation language. Constants may be embedded directly in the component or they may be referred to by a reference variable. We assume constants are never modified; however, a reference variable may be modified to refer to a different value. Because constants may be included in components, a path in a template may describe a constant value. If a template for a dataflow function has an input path that describes a constant, we may instantiate the template with the constant in the function’s signature. If a template for a dataflow function has an output path that refers to a constant, that template cannot be instantiated. This may be considered an instantiation failure, similar to the failures discussed in Section 6.3.4.

In some cases a constraint template may be instantiated even if the instantiation of one or more of its methods fails. As long as at least one method can be instantiated, the constraint can still be enforced; there are simply fewer methods for doing so. We call

this a *partial instantiation* of a constraint template. As an example, consider a constraint template for the relation $a = b + c$ with three methods: $a \leftarrow b + c$, $b \leftarrow a - c$, and $c \leftarrow a - b$. Suppose that this template is applied to a component in which a and b refer to variables, but c refers to the constant value 1. This makes the method $c \leftarrow a - b$ invalid; however, this template may be partially instantiated resulting in a constraint over the variables a and b with the methods $a \leftarrow b + 1$ and $b \leftarrow a - 1$; in other words, this constraint enforces the relation $a = b + 1$.

Continuing the example, suppose we now apply the same constraint template to a component in which b and c refer to the same variable. This invalidates both methods $b \leftarrow a - c$ and $c \leftarrow a - b$ as they would use the same variable as input and output. However, the remaining method may still be instantiated as, effectively, $a \leftarrow b + b$. Thus, this constraint can be partially instantiated to enforce the relation $a = 2b$.

```

1 var constraintSpec = new hd.ComponentBuilder()
2   .c("a", "b", "c")
3   .m("a", "b → c", hd.sum)
4   .m("c", "a → b", hd.diff)
5   .m("c", "b → a", hd.diff)
6   .spec();
7
8 var dataSpec = new hd.ComponentBuilder()
9   .vs("a", "b", {a: 0})
10  .const("c", 1)
11  .spec();
12
13 var model = new hd.Component(dataSpec, constraintSpec);

```

Figure 6.9: Partial instantiation of a template in HotDrink.

Our embedded DSL in HotDrink does not support constant values in signatures.³ However, we do allow the inclusion of constant values in property model components, thereby

³Our reason for this design decision is to avoid confusion about which JavaScript expressions are, and which are not, supported by our DSL.

allowing constants to be introduced into signatures indirectly. We also support the partial instantiation of constraint templates. Figure 6.9 shows an example of this.

The example begins with two component specifications. The first, beginning on line 1, defines a constraint for the relation $\mathbf{a} = \mathbf{b} + \mathbf{c}$. Note that this specification uses the paths \mathbf{a} , \mathbf{b} , and \mathbf{c} without defining them; it is intended that these paths be defined by an additional specification. Such a specification may be found on line 8. This specification defines \mathbf{a} and \mathbf{b} as variables, and defines \mathbf{c} as the constant value 1.

These two specifications are combined into a single component on line 13. This may be viewed as a form of multiple inheritance; the constructed component contains all elements described by both specifications. In the resulting component, the constraint template is partially instantiated as a constraint for the relation $\mathbf{a} = \mathbf{b} + 1$.

6.6.2 Nested Signatures and Array Slices

We further expand our definition of input and output signatures to include, not only variables and (in the case of input signatures) constants, but also *nested signatures*—that is, sequences of variables and constants nested within the signature. A nested input signature contains inputs that are to be passed together as an array rather than as individual parameters. For example, the input signature $((a, b, c), d)$ indicates the scheduling function should be called with two parameters: first an array containing promises for, in order, a , b , and c , and second a promise for d . Similarly, a nested output signature contains outputs that will be returned from the scheduling function as an array rather than as individual return values. For example, the output signature $(w, (x, y, z))$ indicates that the scheduling function will return two values: a promise for w and an array containing, in order, promises for x , y , and z .

One obvious use for nested signatures is to support multiple outputs for programming languages which only allow a single return value for functions. The use of nested signatures allows multiple outputs to be returned as an array. (In fact, this was our convention even before we introduced nested signatures; the use of nested signatures simply makes this explicit.) For example, the output signature $((x, y, z))$, indicates the scheduling function's

one return value will be an array containing, in order, promises for x , y , and z .

Another use for nested signatures comes from dataflow functions which need to access every element in an array component. For example, suppose we have an array component a which represents an array of variables, and we wish to define a constraint with a single method that calculates the sum of the variables in a and writes it to a variable t (for “total”). Using nested signatures, we can create an array parameter for this method which mirrors the structure of a . If, for example, a has a length of three, then we may write the signature of our method as $((a[0], a[1], a[2])) \rightarrow (t)$. This signature specifies that our scheduling function takes one parameter: an array containing, in order, promises for the variables of a . Recall that signatures are distinct from input and output sets; as far as the constraint system is concerned, this is simply a method with three input variables and one output variables.

The method that sums the variables of a can easily be made to work for any size of a by iterating over the array passed to it. In order to make a signature that similarly works for all sizes of a , we introduce the notion of an array *slice*. Broadly speaking, a slice is an indexing expression that refers to multiple elements of an array. For now, we consider only the slice which refers to all elements of an array, which we write with the indexing expression “ $*$ ”. Thus, in our example, we may write the method signature as $(a[*]) \rightarrow (t)$. We may think of a path containing an array slice as being dynamically expanded to an array of paths: one for each index of the array component which has a value. If a has a length of three, our method signature will be expanded to $((a[0], a[1], a[2])) \rightarrow (t)$, as desired.

We assume array components can be dynamically resized. This implies that if a method signature includes a path containing an array slice, the expanded version of that signature will change as the size of the array component changes. Therefore, only templates may include paths containing array slices, and they must be re-instantiated after every change to the size of the array. Continuing with our example, if a is expanded to have a length of four, the constraint with method $((a[0], a[1], a[2])) \rightarrow (t)$ must be destroyed and replaced

with a constraint with method $((a[0], a[1], a[2], a[3])) \rightarrow (t)$. This ensures that no matter how the size of a changes, the variable t will always contain the sum of the variables of a .

To implement this change, we must redefine a path so that it describes, not a single value, but a list of values. Returning to the type definitions of Section 6.5, applying a path should yield signal whose value is of type `Maybe [T]`. For most paths, this value will be either a list containing a single element or no value. For paths containing an array slice, however, the size of the list may vary. The resulting types of `applyP n` and `queryP n` as follows.

```

applyP0 :: Component → Path0 T → Signal (Maybe [T])
applyP1 :: Component → Path1 T → Int → Signal (Maybe [T])
applyP2 :: Component → Path2 T → (Int, Int) → Signal (Maybe [T])

queryP0 :: Component → Path0 T → Signal (Map () [T])
queryP1 :: Component → Path1 T → Signal (Map Int [T])
queryP2 :: Component → Path2 T → Signal (Map (Int, Int) [T])

```

All other types remain the same as in Section 6.5. Note that changes to the size of the array result in changes to the signal produced by a path; templates respond to this change by updating their instantiations.

Figure 6.10 shows an example of array slices in `HotDrink`. This code is a continuation of the scheduling example of Figure 6.1 in which we have added a variable `t` and a constraint ensuring `t` will always equal the sum of the duration `d` of each event. This constraint is written as a specification which is intended to be added to the `model` component of Figure 6.8. In particular, it refers to the variable `s`, which is defined in Figure 6.8 as an array of event components.

The variable `t` is declared on line 2. This is followed by a constraint template for the constraint between `t` and `d`. Line 3 defines the variables of the constraint as `s[*].d` and `t`. The path `s[*].d` will be expanded for every element of `s`—i.e. `s[0].d`, `s[1].d`, etc. Line 4 defines a method which reads all duration variables, calculates their sum, and returns it to be stored in `t`. Line 8 defines a method which uses `t` to update the duration variables. It does


```

1 var extendedModelSpec = new hd.ComponentBuilder()
2   .v("t")
3   .c("s[*].d, t")
4   .m("s[*].d → t", function(d) {
5     for (var t = 0, i = 0; i < d.length; ++i) {
6       t += d[i]; }
7     return t; } )
8   .m("!s[*].d, !t, t → s[*].d", function(d, t1, t2) {
9     for (var i = 0; i < d.length; ++i) {
10      d[i] += (t2 - t1)/d.length; }
11     return d; } } )
12   .spec();

```

Figure 6.10: A template with an array slice in HotDrink.

this using the prior value of t and the prior value of all duration variables: the difference between the sums is distributed evenly among all duration variables. This is, perhaps, not an ideal implementation for this method as it may result in negative or fractional duration values. However, it communicates the possibilities introduced by nested signatures.

6.7 Proper Placement of Property Model Modifications

Section 6.1 describes how the elements managed by a property model are determined by the composition of one or more components. This set of elements does not change unless the composition changes—that is, unless components are added to or removed from the property model, or reference variables are changed, thereby altering the dynamic elements generated from templates. We refer to these changes as structural modifications of the property model. Chapter 3 describes how the property model restricts variable access to methods and operations, thereby ensuring consistent behavior. In this section we consider what restrictions are necessary on structural modifications of the property model to ensure this consistency is maintained.

To begin, let us observe that dataflow functions (methods and operations) are unsuitable places for property model modification. For one thing, allowing methods to modify the property model would mean that solving the constraint system could result in modifying the constraint system—thus, it immediately needs to be solved again! Perhaps more

importantly, since dataflow functions run asynchronously, we would have no confidence as to when any such structural modifications would actually take effect. This would make our reactive program non-deterministic.

Modifications affect the property model by redefining what variables exist and what constraints must be enforced. It follows that, once modifications are made, the property model cannot be assumed to be in a consistent state until the constraint system has been solved. Additionally, if a modification adds variables to the property model, it may need to initialize those variables, which implies an operation. Taking this all together, we conclude that structural modifications should take place in a command, as defined in Section 3.2. Commands may be followed by an operation, allowing a chance for initialization, and are always followed by solving the constraint system. This results in a state transition diagram as shown in Figure 6.11.

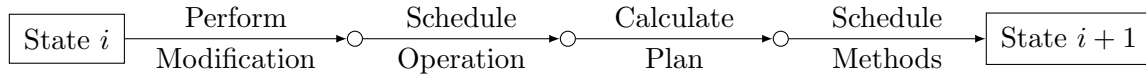


Figure 6.11: GUI state transition, this time including structural modifications of the property model.

Either the first or second step in this transition may be omitted, but not both. Thus, a transition consists of either a modification, an operation, or a modification followed by an operation. In any case, the transition ends with solving the constraint system: calculating a plan and scheduling methods. Note that a modification is *not* a dataflow function. In particular, it may not read the value of any variables. It may, however, examine structure—e.g., check the value of reference variables and modify them as desired. If a modification needs to use the value of a variable, then the modification cannot be performed until the value of the variable is known.

For example, consider a GUI that responds to input by performing a database query

and displaying the results in a table. When the input is given, a method or command may immediately initiate the query; it may not, however, immediately issue the modification to create new rows in the table if it does not know how many records it will retrieve. Only once the query has completed can we perform the modification to create rows in the table, and the operation to initialize them with the results. Users will be unable to edit or interact with these new rows while waiting for the query to complete.

7. STATIC ANALYSIS: SPECIALIZING PLANNERS*

In a GUI controlled by a property model, the property model’s constraint system must be solved after every user interaction. No further user events may be handled until a plan has been calculated and the methods of the plan scheduled. Though hierarchical multi-way dataflow constraint systems can be planned efficiently (in the worst-case quadratic time in the number of constraints, and often linear time) [51], minimizing the time of planning is important to ensure responsiveness. The performance requirements can be even more stringent: for some user interface behaviors, such as enabling and disabling widgets automatically, a planner may need to be executed several times for each interaction [25]. Further, planning is but one part of responding to a user event and thus the quicker it can be done the better.

As discussed in Section 2.2.2, the inputs to the planning algorithm are a specification of the constraint system and the current constraint hierarchy. For many GUIs the constraint system remains unchanged, or changes little, over many planning tasks, while the constraint hierarchy changes often. It may therefore be beneficial to generate a planning algorithm that is specialized for a particular constraint system, instead of using the same general purpose algorithm to plan all systems.

This chapter presents a specialization scheme for generating planners for *hierarchical multi-way dataflow constraint systems* [6]. The generated planners are able to execute in linear time in the number of variables in the system. The planners are DFAs, guaranteed to take no more transition steps than there are variables. The performance experiments reported confirm substantial speedups over a general purpose planner, and that for constraint systems of the size and complexity that can arise in practical user interfaces for which the general purpose planners may not be instantaneous, the specialized planners

*This chapter is reprinted with permission from “Specializing Planners for Hierarchical Multi-way Dataflow Constraint Systems” by Jaakko Järvi, Gabriel Foust, and Magne Haveraaen. *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, pp. 1–10, Copyright 2014 by ACM. <http://dx.doi.org/10.1145/2775053.2658762>

are.

7.1 Supplementary Background Material

This section provides additional background material not contained in Chapter 2. The work described here is not a contribution of this dissertation. It is presented here to serve as a basis for the specialization scheme that follows.

7.1.1 Planning Algorithms

Many algorithms have been developed for solving dataflow constraint systems (for example, see [51, 47, 18]), differing, among other things, on the strategy of finding a plan. Planning algorithms based on the propagation of the degrees of freedom scheme [49, 51] identify *free variables* to avoid backtracking. A variable that belongs to exactly one constraint is free. We also call a method free if all its outputs are free. The significance of a free method is that it can enforce a constraint without restricting which methods can be used to enforce other constraints in the system. Hence, a plan can be found by repeating the following steps: (1) find a constraint with a free method, (2) include the method in the resulting plan, and (3) remove the constraint from consideration. The process either succeeds to remove all constraints, in which case a plan has been found, or fails with no free methods remaining, in which case there are no plans. With some reasonable assumptions about the structure of the system the complexity is linear in the number of variables, methods, and constraints [51].

Planning algorithms for constraint hierarchies iterate the above simple planning algorithm for candidate constraint sets, attempting to identify the strongest constraint set that has a plan. The basic strategy is to add constraints one by one, from strongest to the weakest starting from the empty system, and run the simple planning algorithm after every addition. Constraints that make the system unsatisfiable are discarded, ones that keep it satisfiable are kept [51]. There are variations of this strategy, but the worst case is that the simple planner must be run for each constraint, leading to overall complexity of $O(n^2)$ for planners for hierarchical systems, where n is any of the number of variables,

constraints, or methods.

7.1.2 Constraint Systems as a Monoid

Prior work by Järvi et al. [23] explains how a multi-way dataflow constraint system can be viewed as a commutative monoid. This view is useful in two ways. First, it makes many properties of dataflow constraint systems obvious. For example, planning becomes nothing more than composing all of the system’s constraints with the monoid’s binary operation. Second, it unifies the notions of a constraint and constraint system. Any set of constraints can be composed with the monoid’s binary operator into a single constraint. The plans for a system of constraints are then the methods of the composition of those constraints. This makes it straightforward to examine all possible plans for the purpose of generating specialized planners.

Section 2.2.1 gives an informal definition of multi-way dataflow constraints. Formally, we may define such a constraint as a tuple $\langle R, r, M \rangle$, where R is a set of variables, r is an n -ary relation ($n = |R|$) among variables in R , and M is a set of constraint satisfaction methods. In this representation, r is the relation of the constraint: the constraint is satisfied if the values of the variables in R satisfy r .

We may also formalize the method requirements given in Section 2.2.1 as a set of well-formedness conditions for constraints. Let $ins(m)$ and $outs(m)$ refer to, respectively, the input and output variables of a method m . Then a constraint $\langle R, r, M \rangle$ is well-formed if for all methods $n, m \in M$:

- $outs(n) \not\subseteq outs(m)$ and
- $\{ins(m), outs(m)\}$ is a partition of R .

Furthermore, for any two constraints $\langle R_1, r_1, M_1 \rangle$ and $\langle R_2, r_2, M_2 \rangle$ in the same system, $R_1 \neq R_2$,

In the monoid representation, a constraint continues to be defined as $\langle R, r, M \rangle$, where M is a set of methods, but the notion of a method is generalized. In the graph representation of a constraint given in Section 2.2.3, a method consists of a single method vertex, connected

to a set of variable vertices with incoming edges for input variables and outgoing edges for output variables. In the monoid view, we take this *method graph* as the representation of a method, and generalize it from the “one method vertex connected to variable vertices” structure to a (bipartite) DAG that can contain several method vertices. To be a valid method graph, the in-degree of every variable vertex is at most one. For purposes of executing a method, this internal graph structure of the method is irrelevant; a method can be viewed as a function from its input to output variables. For purposes of correctly composing constraints, the graph structure must be retained.

The composition of constraints $\langle R_1, r_1, M_1 \rangle$ and $\langle R_2, r_2, M_2 \rangle$ is a union over their variables, a conjunction over their relations, and the set of those (non-disjoint) graph unions¹ of all pairs of method graphs, one from each constraint, that are method graphs themselves (we use the symbol $+$ for this “pairwise graph unions followed by a filtering” operation): $\langle R_1 \cup R_2, r_1 \wedge r_2, M_1 + M_2 \rangle$. Every method graph of a constraint contains all of the constraint’s variable vertices, and has the same number of method vertices as all the other method graphs.

Below we refer to constraints that are not a composition of other constraints as *primitive*. Similarly, methods of primitive constraints are called primitive.

Since in a constraint $\langle R, r, M \rangle$, M uniquely determines R and r is not used in planning, we can, for the purposes of planning, think of a constraint to be merely a set of method graphs. The constraint system monoid is thus as follows:

- The carrier set is all sets of method graphs.
- The binary operation forms a Cartesian product of the two sets of method graphs, computes the graph union of each pair of this product, and discards those graphs that are not valid method graphs. Formally, let $A = \{a_1, a_2, \dots, a_m\}$ and $B = \{b_1, b_2, \dots, b_n\}$ be sets of method graphs. Then the monoid operation is defined as

¹ The graph union operation $\langle V_1, E_1 \rangle \cup \langle V_2, E_2 \rangle$ is defined in the canonical way as the union of vertices and edges: $\langle V_1 \cup V_2, E_1 \cup E_2 \rangle$. When composing constraints, only variable vertices are shared, never method vertices or edges.

$$A + B = \{c \mid a \in A, b \in B, c = a \cup b, c \text{ a method graph}\}.$$

- The identity element is the singleton set whose sole element is the null graph.

The binary operation is both associative and commutative because of the associativity and commutativity of graph union [5, §1.4]. The monoid has an absorber, the empty set.

If an element of the monoid is generated (by repeated application of the monoid operation) from primitive constraints that satisfy the well-formedness conditions from Section 7.1.2, it too will satisfy those conditions. The condition that $\{ins(m), outs(m)\}$ is a partition of a constraint's variables, though, requires a clarification: variables that are sources in the method graph are considered inputs, all other variables outputs.

Order your custom picture frame below.

Width:	<input type="text" value="60"/>	cm	Perimeter:	<input type="text" value="200"/>	cm
Height:	<input type="text" value="40"/>	cm	Area:	<input type="text" value="2400"/>	cm ²
Cost:	\$64.00 USD				
					<input type="button" value="Place Order"/>

Figure 7.1: A user interface for ordering customized picture frames. The cost is determined as a function of perimeter (for the frame itself) and area (for the glass.)

For purposes of illustration, let us consider a GUI used to order custom picture frames, as shown in Figure 7.1. Here, the price of the frame is a function of the material costs, which are in turn based on the perimeter (for the frame) and area (for the glass) of the frame. These values may be edited directly, or they may be derived from the dimensions of the frame. Figure 7.2 shows the constraint graph describing the constraints between these four values. In order to keep this and subsequent related figures less cluttered, the constraint graph does not contain the one-way constraint between area, perimeter, and cost.

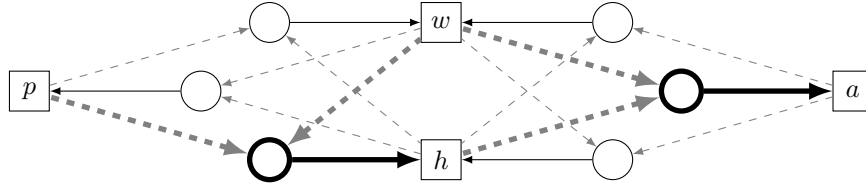


Figure 7.2: Constraints arising from the picture frame GUI.

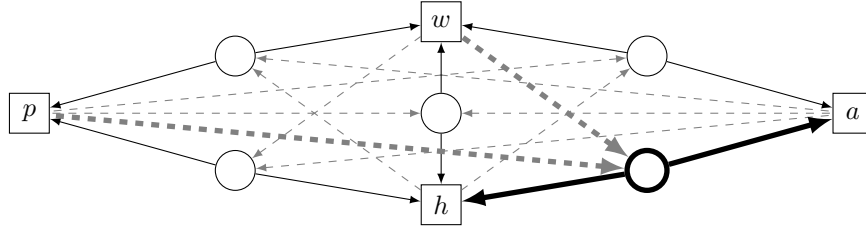


Figure 7.3: Constraint graph resulting from composing the constraints in Figure 7.2.

Figure 7.3 shows the two constraints from Figure 7.2 composed into a single constraint. The circular nodes represent method graphs; the internal graph structures of the methods are not shown. Of the nine graph unions between two method graphs, one from each operand constraint, the two method graphs $p \rightarrow \cdot \langle \overset{w}{\underset{h}{\rightarrow}} \cdot a$ and $p \rightarrow \cdot \langle \overset{w}{\underset{h}{\leftarrow}} \cdot a$ must be discarded as cyclic and the two method graphs $p \leftarrow \cdot \langle \overset{w}{\underset{h}{\rightarrow}} \cdot a$ and $p \leftarrow \cdot \langle \overset{w}{\underset{h}{\leftarrow}} \cdot a$ because a variable vertex has an in-degree larger than one. This leaves five method graphs representing the five possible plans of the original constraint system graph: $p \rightarrow \cdot \langle \overset{w}{\underset{h}{\rightarrow}} \cdot a$, $p \rightarrow \cdot \langle \overset{w}{\underset{h}{\leftarrow}} \cdot a$, $p \leftarrow \cdot \langle \overset{w}{\underset{h}{\rightarrow}} \cdot a$, $p \leftarrow \cdot \langle \overset{w}{\underset{h}{\leftarrow}} \cdot a$, and $p \leftarrow \cdot \langle \overset{w}{\underset{h}{\leftarrow}} \cdot a$. These are the graphs of the five methods in Figure 7.3. The method shown as bold corresponds to the selected plan shown in Figure 7.2.

7.1.3 Planning of Constraint Hierarchies as Monoid Composition

The iterative planning of a hierarchical constraint system has a simple interpretation in terms of the monoid:

foldl ($\lambda a b. \text{if } a + b = z \text{ then } a \text{ else } a + b$) e S

S is a sequence of the system's constraints in the order of decreasing strength (highest

priority first), z the absorber element \emptyset , e the identity element $\{\emptyset\}$, and **foldl** is as defined, e.g., in Haskell. This expression composes all constraints of \mathbf{S} into a single constraint, excluding those which would cause the composition to go to z . The result is a single constraint with just one method: the unique plan.

Instead of starting the fold from the identity constraint, in practice we first compose all mandatory constraints into a single constraint, and then attempt to compose this constraint with the individual stay constraints in order of decreasing priority.

Composing a stay constraint with another constraint, call it C , is an even simpler operation than composing two arbitrary constraints; it is effectively a filtering of methods of C . Let S_x be a stay constraint of variable x . Some method graphs in C might contain an edge whose target is x , some not. The former will be filtered out, since S_x 's only method graph, that must be used in each graph union, contains an edge whose target is x . $C + S_x$ will thus contain a subset of the method graphs of C , each augmented with one additional method vertex and an edge from that vertex to the variable vertex x . In our specialization scheme we use stay constraints purely to prune the set of plans. We thus restrict each method in $C + S_x$ to only include vertices and edges in C . We use the notation $(C + S_x)|_C$ for this operation.

In the above fold operation, if adding a stay constraint to a constraint results in the empty constraint (the absorber), the stay constraint cannot be enforced, and the value of the corresponding variable will not be preserved. In this case the original constraint is kept. Because the stay constraints are added in order of decreasing priority, a variable of higher priority is always preserved over one of lower priority, if possible. The interpretation of the (restricted) sum $(C + s_1 + \dots + s_n)|_C$ is the set of all plans that will preserve the variables of stay constraints s_1, \dots, s_n . These sums will directly correspond to the states of the planning state machines, to be discussed in Section 7.3, that our specialization scheme generates.

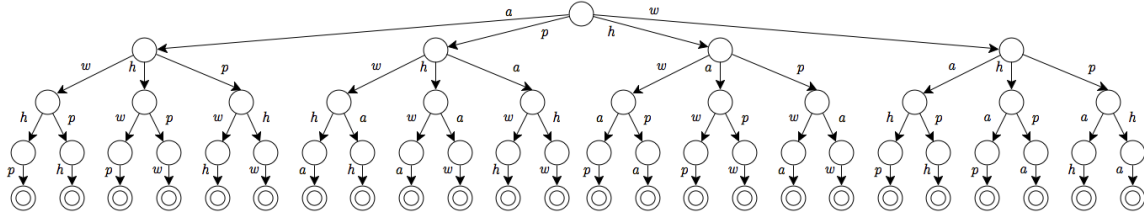


Figure 7.4: A planner for the four-variable system of Figure 7.2, implemented as a decision tree.

7.2 Properties of Constraint Compositions

The number of plans of a multi-way dataflow constraint system can be large. Assume a constraint system of n variables. The upper bound for the number of methods is then $\binom{n}{\lfloor n/2 \rfloor}$, which can be seen as follows. Taking advantage of the monoid representation, we can view the system as a single constraint. As the inputs and outputs of each method partitions the constraints' variables, and no two methods can have the same partitioning, a method can be identified with the set consisting of its output variables (or, equivalently, its input variables). The well-formedness conditions on constraints guarantee that the output variables of a method are not a subset of the output variables of another method. The number of methods in a constraint is thus limited by the maximum number of different subsets of variables, where none is a subset of the other. This maximum is $\binom{n}{\lfloor n/2 \rfloor}$, obtained when all subsets have $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil$ elements. This is a rather large number. In practice the number of plans can be expected to be much smaller, so that it is feasible to generate code for a state machine that encompasses all those possible plans.

There are several reasons why we can expect the number of plans to remain quite reasonable in practice. In typical user interfaces, the most common constraint is a one-way constraint, where the constraint's relation is a function, and subsequently it needs only one method. Composing a one-way constraint with another constraint that has, say, m , methods produces a constraint that has at most m methods.

Two-way constraints are also somewhat common. Composing a two-way constraint with an existing m -method constraint, such that the constraints share at least one variable, produces a constraint that has at most $m + 1$ methods.

We have not conducted empirical studies to quantify the distribution of constraints with different number of methods, but constraints with a large number of methods are rare. Even if a constraint involves many variables, often each method has just one output. In such a case, the number of methods in a constraint is at most n (i.e., the number of variables), instead of $\binom{n}{\lfloor n/2 \rfloor}$.

Since the constraint monoid is commutative, it makes no difference in which order constraints are composed. The order, however, can drastically impact the size of the intermediate constraints, that is, the number of methods in those constraints. In particular, if two constraints C_a and C_b share no variables, and C_a has m_a and C_b has m_b methods, then their composition $C_a + C_b$ has $m_a \times m_b$ methods. The more variables they share, the more the resulting method set gets pruned as many graph unions will be cyclic or contain variable nodes with more than one incoming edge, violating the conditions for method graphs.

7.3 Specializing a Constraint System

The monoid representation of a constraint provides a foundation for a specialization scheme in which the relatively complex hierarchical planning algorithm discussed in Section 7.1.1 (a nested iteration that at the outer level runs a simple planner algorithm for each stay constraint, which at the inner level iterates over all constraints, “peeling off” constraints that have a free method until a plan is found) can be specialized into a simple automaton that takes no more transitions than there are stay constraints in the system.

7.3.1 Planner as a DFA

The specialized planner is a DFA whose input is the sequence of all variables ordered from the highest priority to the lowest. Each state in the machine corresponds to a set of possible plans, i.e., a constraint. The start state of the state machine corresponds to the constraint that represents all possible plans. Each transition corresponds to an attempt to add a stay constraint to the constraint of the current state. The accepting states are constraints with only one method, which is the resulting plan.

If the size of the state machine were of no concern, we could use a decision tree where the nodes at the i th level of the tree encode a decision based on which variable is in the i th position of the priority order. In such a tree, each path from the root to a leaf represents one priority order. As a priority order uniquely determines a plan, we can associate a plan with every leaf node of the decision tree. For example, the decision tree for the four-variable constraint system in Figure 7.2 (and 7.3) is shown in Figure 7.4. Each leaf corresponds to a unique priority ordering, and would be associated with whatever plan this ordering induces.

As a large number of states in the “full-blown” decision tree are equivalent, the same decision logic can be implemented with a much smaller DFA. The DFA corresponding to the decision tree in Figure 7.4 is shown in Figure 7.2. The DFA has only ten distinct states, and it reaches an accepting state in at most three steps; once the machine reaches an accepting state, it stays in the same state, so the DFA can be implemented to ignore any remaining input. As discussed in Section 7.2, the number of plans is usually significantly smaller than the number of priority orders; here there are five distinct plans for the 24 different priority orders.

There are several sources of redundancy that the DFA generation takes advantage of in keeping the number of states low:

1. A transition that corresponds to an attempt to add an unenforceable stay con-

straint can loop back to the same state.

2. A unique plan is often determined by a (short) prefix of the input sequence.
3. Two different compositions of constraints can produce the same result, and thus equivalent states. Some of the equalities follow from the monoid's properties, others only occur for some graph structures. Different equalities are detected by the DFA generation algorithm by different means. Two special cases are:
 - (a) All sub-paths that consist of the same variables in different order lead to equivalent states. This is implied by the commutativity of the constraint composition operation.
 - (b) For some constraints, a variable is preserved by all methods. A transition that adds a stay constraint for such a variable can loop back to the same state.

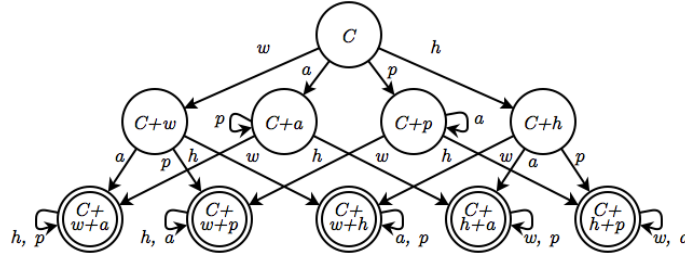


Figure 7.5: A DFA equivalent to the decision tree of Figure 7.4. The sums at state labels are assumed to be restricted to C .

7.3.2 Generating the DFA

The DFA to be generated maps a priority order to a plan. The generation has to determine which plan results from which priority order, and thus analyze every possible priority order—or realize that certain priority orders need not be analyzed.

The strategy is to traverse the decision tree of all priority orders in a depth-first order, recognizing equivalent states along the way, so that no redundant states are generated, and no unnecessary paths are traversed.

The starting point of the generation is the constraint formed as a composition of all mandatory constraints, call it C , using the composition operation defined in Section 7.1. This representation expresses the entire constraint system as a single constraint, whose methods are the possible plans of the system. Stay constraints are then composed one by one into the constraint, every time checking whether an equivalent state has already been generated. Most of the equivalent states are detected prior to having to traverse downward in the tree, so that entire sub-trees can be pruned. These cases take advantage of the algebraic properties (commutativity) of the composition operation. Some states can only be recognized to be equivalent after fully constructing both states. These are the cases where composing two different sets of stay constraints with C happens to produce the same constraint.

To be able to recognize equivalent states based on the commutativity property, each state stores the labels, i.e., variable identifiers, occurring within the path from the initial state to the current state. *Self-loops*, transitions that loop back to their initial state, are not considered. A hash table, we call it the *path map*, that maps these sets of labels (using a suitable canonical representation for sets) to states enables quickly checking for the existence of equivalent states. Additional checking for equivalent states is based on the equivalence of transition tables of each state—for this, the transition tables must have been already fully generated.

Algorithm 7.1 summarizes this process. This algorithm generates a new state for a given constraint C or reuses an existing state in the DFA. If a new state needs to be generated, it first recurses to each transition out of the generated state. When started from the initial state, the algorithm generates the entire DFA.

The parameters of the algorithm are V_s , the set of variables seen in the path to

Input: V_s , the set of variables seen in the path to current state
Input: V_u , the set of variables not yet seen in the path to current state
Input: C_c , the constraint represented by current state
Global: $Paths$, a map from variable sets (representing paths) to states
Global: $States$, a map from transition tables to states
Result: the current state

```

1 Function DFASSTATE( $V_s, V_u, C_c$ ):
2   if  $|C_c| == 1$  then
3     return accept( $C_c$ )
4    $t \leftarrow$  an empty transition table
5   foreach  $v \in V_u$  do
6     if  $Paths[V_s \cup \{v\}]$  defined then
7        $t[v] \leftarrow Paths[V_s \cup \{v\}]$ 
8     else
9        $C'_c \leftarrow (C_c + \text{stay}(v))|_{C_c}$ 
10      if  $|C'_c| > 0 \wedge |C'_c| < |C_c|$  then
11         $t[v] \leftarrow \text{DFASSTATE}(V_s \cup \{v\}, V_u \setminus \{v\}, C'_c)$ 
12  if  $States[t]$  defined then
13     $s \leftarrow States[t]$ 
14  else
15     $s \leftarrow$  a new unique state
16     $s.\text{transition\_table} \leftarrow t$ 
17     $States[t] \leftarrow s$ 
18   $Paths[V_s] \leftarrow s$ 
19  return  $s$ 

```

Algorithm 7.1: Calculate DFA state for a given path

the current state; V_u , the unprocessed input (also a set of variables); and C_c the current constraint that this state represents. The algorithm accesses and modifies two lookup tables that are global variables: *Paths* is the path map and *States* the mapping from transition tables to DFA states. We assume the transition tables are in a form that enables fast look-up. The algorithm returns a state, either one of the existing ones, or a newly constructed.

Initially the algorithm is invoked as $\text{DFASTATE}(\{\}, V, C)$, where $\{\}$ is the empty set of already processed variables, V the set of all variables in the system, and C the composition of all mandatory constraints. The global variables *Paths* and *States* are assumed to be empty lookup tables.

First, the algorithm checks whether the current state represents a solution (a constraint with exactly one method). If so, an accepting state is returned. We assume that the accept function will either create a new state, or return an existing one if the plan represented by C_c has already been encountered.

If an accepting state has not been generated, the algorithm proceeds to generate transitions for all variables v that have not yet been seen. If a state for a path that is a different permutation of the current path V_s extended with v has already been generated, the transition can reuse that state (relying on the commutativity property), and the algorithm does not need to recurse. Otherwise the current constraint C_c is composed with v 's stay constraint to produce a new constraint C'_c . If the stay constraint of v can be enforced (C'_c has at least one method, i.e., it is not the absorber element) and if it is not enforced in all methods in C_c , then the algorithm will recurse to determine a new state (which can end up being newly constructed or reused).

Once all variables in V_u have been analyzed, the transition table is complete, and can be looked up in *States*. If an entry exists, that existing state can be reused. If not, a new state will be created and t is assigned as its transition table (the `.transition_table` notation accesses a state's transition table). The new state is added

to *States* prior to returning it. Whether a state was generated or reused, the path to it must be added to *Paths*.

There are no explicit checks for exhausting the variable set V_u . This is because all priority orderings uniquely determine a plan, and thus the algorithm is guaranteed to have generated an accepting state by the time it has exhausted its input, likely sooner.

A DFA generated by the DFAS_{TATE} algorithm is not yet a correctly functioning planner as the algorithm does not add self-loops. We assume every state has a self-loop for all possible inputs that do not have a transition out of the state. For example, the state $C + a$ in Figure 7.5 has such a self-loop for the input p .

The only transitions in accepting states are the self-loops that keep the state machine in the same accepting state for all remaining inputs. As an optimization, the generated DFA returns a plan when it first enters an accepting state, ignoring any remaining input.

The algorithm guarantees that the generated DFA is minimal—all states are reachable and distinguishable. That all states are reachable is trivial. That all states are distinguishable requires justification. By construction, each state corresponds to a set of methods. In the following discussion, we use these two descriptions interchangeably. Each final state is a singleton set. Every state reaches exactly the final states corresponding to the methods of that state (because of the constraints' well-formedness conditions, see Section 7.1.2, every method of a state is necessarily included in at least one of the state's direct successors; also no transition adds methods). Thus, two states that contain different methods are distinguishable by any input leading to a final state corresponding to a method contained by one but not the other. What remains is to show that no two states contain the same methods. The condition on line 10 guarantees that a transition is always to a state that has fewer methods, based on which an inductive argument can be constructed to

show that the transition table generated for a given set of methods is unique. Per the condition on line 12, the algorithm never creates a new state if a state with an equivalent transition table exists.

7.4 Experiments

To assess the improvements in speed obtainable using planner specialization, we performed multiple experiments using our *HotDrink* implementation. As a reminder, this property model implementation is written in TypeScript and compiled to JavaScript to be executed in web browsers. As a JavaScript library, it is also suitable for use in command-line applications using a suitable JavaScript engine, such as Node.js [39].

7.4.1 Generator Implementation

We used the ideas presented in this chapter to implement a tool that can generate specialized planners for use with our library. The intent is that the tool is used offline to read in a constraint system specification and generate code for a planner specialized for the particular constraint system. That code may then be included in an application, to be used by our UI library in lieu of the default general-purpose planner.

The generator is also written in TypeScript so as to take advantage of our existing code base. The generator expects constraint system specifications to be in the format used with the *HotDrink* library, using its embedded domain-specific language for defining constraints. A constraint system is created from this specification, then converted into a DFA using the algorithm described in Section 7.3.

Internally, the generator represents a DFA simply as a collection of transition tables, one for each state. It would be possible to simply serialize this data structure (e.g. using JavaScript Object Notation) so that it could be inserted into and used by some application. However, we generate executable JavaScript code: one function

for each state, with a switch statement to implement transitions.

Figure 7.6 shows an excerpt of the code generated for the planner DFA for the system in Figure 7.2. For clarity, we show all variables and functions as if they were defined globally. In reality, the only function exposed is the `dfa` function found at the end of the listing; all others are hidden from direct access.

Each DFA state gives rise to one function. The global variable `input` is the sequence of constraint system variables in priority order and `i` the current index into the sequence. The state functions collectively iterate through the input sequence; each state function advances the iteration, and uses a switch statement to choose the next state to transition to. If a corresponding case is not found, the machine stays in the same state. This is how the self-loops discussed in Section 19 are implemented.

Function `n4` is the function corresponding to the state labeled C in Figure 7.5. This state makes transitions on inputs of 0, 1, 2, and 3, which correspond to the variables w , h , a , and p , respectively.

Function `n3` corresponds to the state $C+p$. It has two transitions leading to accept states; all other transitions are implicit self-loops. An accept state is represented as an array containing the indices of the methods to be selected. Thus, an accept state is simply a plan. In this example, 4, 8, and 9 are the indices for $w \xrightarrow{h} a$, $p \xrightarrow{h} w$, and $p \xrightarrow{h} w$, respectively. States `n0`, `n1`, and `n2` are very similar to `n3`, and have thus been omitted for brevity.

Each state, rather than calling the next state function directly, performs a transition by returning the function corresponding to the new state. The main function `dfa` then invokes the function. An accepting state returns a non-function, which stops the machine. JavaScript virtual machines do not typically recognize tail calls, so this mechanism (sometimes called *trampolining*) ensures that the state machine runs in constant space.

```

1  var input, i;
2
3  function n0() { ... }
4
5  function n1() { ... }
6
7  function n2() { ... }
8
9  function n3() {
10     while (true) {
11         switch (stays[i--]) {
12             case 0: return [4, 8];
13             case 1: return [4, 9];
14         }
15     }
16 }
17
18 function n4() {
19     while (true) {
20         switch (input[i--]) {
21             case 0: return n0;
22             case 1: return n1;
23             case 2: return n2;
24             case 3: return n3;
25         }
26     }
27 }
28
29 function dfa( varlds ) {
30     input = varlds; i = varlds.length - 1;
31     var f = n4;
32     do { f = f(); } while (typeof f === 'function');
33     return f;
34 }

```

Figure 7.6: Code generated for a DFA.

7.4.2 Methodology

We compared the running times of several specialized planners with those of our library’s default planner. This general-purpose planner is based on Zanden’s QuickPlan [51] algorithm—an incremental version of the degrees-of-freedom algorithm which reuses the previous solution whenever possible. In cases where the correct plan contains many of the same methods as the previous plan, incremental planning can be very fast. The worst-case time complexity, though, remains quadratic in the number of variables.

Our experiments used fifteen different constraint systems, selected to cover a wide variety of different constraint graph structures. Some of the systems came directly from small user interfaces, others were generated programmatically by repeating a pattern to generate large constraint systems. We describe the test systems in more detail below.

The tests updated randomly selected variables of the system; the time of finding a plan for each update was measured. An update to a variable promotes the corresponding stay constraint to the highest priority, thus altering the priority order.

The running times of both QuickPlan and the specialized planners were generally only a fraction of a second. Timers in JavaScript measure wall-clock time, not CPU time. This introduces the risk that sudden ill-timed actions performed by the operating system may interfere with accurate timing. To counteract this, we ran each test case seven times, recording the individual times of each planning operation, and used the median time as our measurement. The random sequence of variable updates was hard-coded into each test case, guaranteeing that the same sequence of planning operations occurred every time we ran the test case.

There are three kinds of constraints that most of the constraint systems in the test cases are composed of, and are also common in constraint systems modeling user interfaces. We name them to help concisely describe the test cases: a *one-way*₂

constraint consists of two variables and a single method with one input and one output, graphically $\square \rightarrow \square$; a *two-way*₂ constraint consists of two variables and two methods, both with one input and one output, graphically $\square \rightleftarrows \square$; and a *three-way*₃ constraint consists of three variables and three methods, each with two inputs and one output, graphically $\square \rightleftarrows \begin{array}{c} \square \\ \square \end{array}$.

The fifteen different constraint systems that we tested with can be categorized as follows:

- *hotel*, *dimensions*, and *frame* are constraint systems powering user interfaces, *hotel* based on a form for reserving a hotel room, *dimensions* on a dialog for specifying dimensions of an image, and *frame* is the example system from Figure 7.2. These are small systems, so their planning times are very short.
- *single*, *solid*, *dotted*, and *dashed* are systems where the number of input variables dominates that of output variables in their plans. In planning such systems, composing a stay constraint will succeed more often than fail. As explained in Section 7.3, successful compositions follow a transition to a new state in the DFA planner. The state machines in this group thus tend to have a larger number of states than those of systems with the opposite characteristic.

The detailed description of the four tests in this group are: *single* is one large constraint consisting of 10 variables and 10 methods, each method writing to a single variable; *Solid* is a chain of five three-way₃ constraints, where two adjacent constraints share one variable; *dotted* is a chain of five three-way₃ constraints, intercalated by altogether four one-way₂ constraints, each two adjacent constraints sharing one variable; and *dashed* is chain of three groups of two three-way₃ constraints, intercalated by one-way₂ constraints, each two adjacent constraints sharing one variable.

- *chain-n* and *ladder-n* are templates of systems where the number of outputs

dominates that of output variables in their plans. Each template was instantiated with $n = 10, 20, 50$, and 100 . The *chain- n* tests consist a chain of n variables, each adjacent pair of variables connected with a two-way₂ constraint. The *ladder- n* tests consist of n variables and $n - 2$ three-way₃ constraints, the constraints arranged in a “ladder” sequence, such that each adjacent pair of constraints shares two variables. The structure of the *ladder-4* system is the one shown in Figure 7.2.

We ran the experiments on several platforms, covering different browser implementations and types of machines:

- Node.js JavaScript runtime on MacBook Pro (2.7 GHz Intel Core i7, 8GB of memory),
- Firefox desktop web browser on the same 2013 MacBook Pro,
- Chrome mobile web browser on a Google Nexus 10 tablet, and
- Safari mobile web browser on an iPhone 5S smart phone.

As mentioned in Section 7.4.1, the intended application of the specialized planners are to serve as a drop-in replacement for the planner in our existing library UI programming library. We thus matched the interface of the specialized planner to the interface of the existing planner. In particular, the existing planner operates directly on a graph data structure which is then used, not only to determine which methods were selected, but also to determine dependency information required by other parts of the library

To accurately measure the performance benefit in the context of this real-world use scenario, we recorded the time of the *entire* specialized planner—including the time it spent building the graph data structure. This method ensures fairness, but it also somewhat masks the performance benefit of the DFA, as it adds overhead

which is not strictly necessary for the DFA algorithm. In most cases the time of constructing the graph data structure is much more expensive than running the planner itself. The graph construction could be avoided by redesigning the parts of the library that need the dependency information, but we have not done that.

To compare just the DFA planner with the QuickPlan-based planner, we performed additional test runs using the Node.js runtime, not including the graph construction time. These tests were executed only on Node.js; it was the only JavaScript runtime that provided a timer with a high enough resolution—the DFAs generally run in less than ten microseconds.

7.4.3 Results

Tables 7.1 and 7.2 show, respectively, the average and maximum of the measured planning times. For each test, the time for the QuickPlan-based planner and the combined time for the specialized planner followed by constructing the solution graph are reported. For Node.js, the running time of the specialized planner without graph construction is also reported. To give a sense of the achieved improvement, the running times for the specialized planners are also given relative to those of the QuickPlan-based planner.

In almost all tests, the specialized planner is significantly faster than the QuickPlan-based planner. The difference is expectably most pronounced in the *chain-n* and *ladder-n* tests as their DFA planners determine the best plan based on the first few highest priority variables.

One test case, *single*, shows a performance decrease for the specialized planner. This rather pathological case consists of exactly one constraint, so there is little to plan. Further, that constraint is such that the DFA always makes the maximum number of transitions to decide the best method, and is in that sense the worst-case scenario for the specialized planner. Regardless, the DFA part runs in a fraction of the time of the QuickPlan-based planner, but when the graph construction is included

	Node.js			Firefox		Mobile Chrome		Mobile Safari	
	QP	Spec	DFA	QP	Spec	QP	Spec	QP	Spec
hotel	0.34	0.12 (36%)	0.0063 (1.85%)	0.57	0.19 (34%)	7.68	2.49 (32%)	1.22	0.45 (37%)
dimensions	0.25	0.14 (55%)	0.0071 (2.85%)	0.50	0.24 (48%)	5.51	2.93 (53%)	0.98	0.57 (58%)
frame	0.31	0.09 (29%)	0.0057 (1.86%)	0.59	0.18 (31%)	7.18	1.94 (27%)	1.17	0.19 (16%)
single	0.09	0.17 (194%)	0.0110 (12.65%)	0.18	0.34 (190%)	1.84	3.89 (211%)	0.35	0.93 (266%)
solid	0.43	0.19 (45%)	0.0096 (2.25%)	0.70	0.36 (52%)	8.60	4.92 (57%)	1.54	0.97 (63%)
dotted	0.54	0.29 (54%)	0.0093 (1.74%)	0.81	0.45 (55%)	9.10	6.88 (76%)	2.01	0.98 (49%)
dashed	0.36	0.29 (78%)	0.0111 (3.03%)	0.61	0.46 (75%)	7.25	6.09 (84%)	1.38	1.10 (80%)
chain-10	1.71	0.18 (10%)	0.0047 (0.28%)	2.39	0.28 (12%)	25.49	3.93 (15%)	6.28	0.66 (11%)
chain-20	6.26	0.32 (5%)	0.0067 (0.11%)	6.29	0.39 (6%)	46.86	7.71 (16%)	19.84	1.02 (5%)
chain-50	23.15	0.77 (3%)	0.0066 (0.03%)	23.79	0.97 (4%)	152.20	15.69 (10%)	57.16	2.75 (5%)
chain-100	81.65	1.57 (2%)	0.0083 (0.01%)	90.08	1.94 (2%)	476.77	17.73 (4%)	165.90	5.53 (3%)
ladder-10	2.20	0.19 (8%)	0.0065 (0.29%)	2.63	0.24 (9%)	25.60	4.90 (19%)	7.54	0.80 (11%)
ladder-20	7.46	0.33 (4%)	0.0068 (0.09%)	8.70	0.49 (6%)	58.10	9.59 (16%)	26.18	1.40 (5%)
ladder-50	32.94	0.78 (2%)	0.0075 (0.02%)	38.53	1.26 (3%)	213.32	16.03 (8%)	73.37	3.32 (5%)
ladder-100	113.77	1.48 (1%)	0.0084 (0.01%)	137.26	2.72 (2%)	654.48	23.32 (4%)	221.82	6.17 (3%)

Table 7.1: Average planning times for QuickPlan planner (QP), specialized planners (Spec), and planner DFAs (DFA). All times are in milliseconds. Times relative to QP for Spec and DFA are in parentheses.

	Node.js			Firefox		Mobile Chrome		Mobile Safari	
	QP	Spec	DFA	QP	Spec	QP	Spec	QP	Spec
hotel	1.29	0.21 (16%)	0.0112 (0.87%)	1.66	0.24 (14%)	27.10	3.69 (14%)	4.00	1.00 (25%)
dimensions	1.03	0.19 (18%)	0.0123 (1.20%)	1.79	0.30 (17%)	25.21	4.20 (17%)	4.00	1.00 (25%)
frame	0.89	0.11 (12%)	0.0064 (0.72%)	1.65	0.24 (15%)	22.45	2.64 (12%)	4.00	1.00 (25%)
single	1.13	0.22 (20%)	0.0132 (1.17%)	2.17	0.46 (21%)	25.52	4.97 (19%)	6.00	2.00 (33%)
solid	1.72	0.27 (16%)	0.0122 (0.71%)	2.45	0.48 (20%)	30.33	8.10 (27%)	7.00	1.00 (14%)
dotted	3.40	0.37 (11%)	0.0116 (0.34%)	5.15	0.57 (11%)	45.01	8.64 (19%)	16.00	2.00 (13%)
dashed	2.13	0.41 (19%)	0.0133 (0.63%)	3.37	0.60 (18%)	30.33	7.64 (25%)	9.00	2.00 (22%)
chain-10	3.36	0.24 (7%)	0.0055 (0.16%)	3.37	0.35 (10%)	46.01	4.35 (9%)	10.00	1.00 (10%)
chain-20	8.74	0.41 (5%)	0.0080 (0.09%)	8.84	0.46 (5%)	80.12	9.71 (12%)	28.00	2.00 (7%)
chain-50	32.29	1.03 (3%)	0.0075 (0.02%)	33.41	1.09 (3%)	217.21	20.55 (9%)	82.00	5.00 (6%)
chain-100	118.00	1.90 (2%)	0.0097 (0.01%)	146.13	2.31 (2%)	649.01	30.46 (5%)	232.00	7.00 (3%)
ladder-10	4.15	0.24 (6%)	0.0083 (0.20%)	4.58	0.28 (6%)	53.47	6.06 (11%)	14.00	1.00 (7%)
ladder-20	11.60	0.52 (4%)	0.0076 (0.07%)	12.46	0.53 (4%)	89.82	17.25 (19%)	37.00	2.00 (5%)
ladder-50	48.84	1.04 (2%)	0.0084 (0.02%)	53.93	1.51 (3%)	306.86	21.02 (7%)	103.00	4.00 (4%)
ladder-100	170.56	1.85 (1%)	0.0096 (0.01%)	239.32	3.24 (1%)	936.98	51.55 (6%)	313.00	7.00 (2%)

Table 7.2: Maximum planning times for QuickPlan planner (QP), specialized planners (Spec), and planner DFAs (DFA). All times are in milliseconds. Times relative to QP for Spec and DFA are in parentheses.

Node.js				Firefox		Mobile Chrome	
	QP	Spec	DFA	QP	Spec	QP	Spec
hotel	0.37 (109%)	0.02 (15%)	0.0008 (13%)	0.58 (103%)	0.01 (4%)	8.94 (116%)	0.59 (24%)
dimensions	0.30 (119%)	0.02 (12%)	0.0009 (12%)	0.58 (115%)	0.01 (5%)	7.17 (130%)	0.57 (19%)
frame	0.31 (100%)	0.01 (10%)	0.0003 (5%)	0.56 (95%)	0.03 (14%)	7.48 (104%)	0.23 (12%)
single	0.27 (311%)	0.02 (11%)	0.0005 (5%)	0.51 (281%)	0.04 (11%)	6.16 (334%)	0.44 (11%)
solid	0.43 (101%)	0.03 (14%)	0.0007 (7%)	0.68 (97%)	0.05 (14%)	8.80 (102%)	1.17 (24%)
dotted	0.77 (143%)	0.03 (9%)	0.0007 (7%)	1.05 (130%)	0.04 (9%)	12.02 (132%)	1.45 (21%)
dashed	0.47 (130%)	0.04 (13%)	0.0007 (6%)	0.74 (121%)	0.05 (11%)	8.87 (122%)	0.73 (12%)
chain-10	0.69 (41%)	0.02 (13%)	0.0003 (6%)	0.81 (34%)	0.01 (4%)	11.90 (47%)	0.46 (12%)
chain-20	1.72 (28%)	0.03 (10%)	0.0003 (4%)	1.67 (27%)	0.02 (5%)	14.28 (30%)	1.20 (15%)
chain-50	4.18 (18%)	0.08 (10%)	0.0004 (6%)	4.42 (19%)	0.03 (3%)	29.28 (19%)	4.14 (26%)
chain-100	19.21 (24%)	0.16 (10%)	0.0007 (8%)	22.16 (25%)	0.11 (6%)	90.03 (19%)	4.24 (24%)
ladder-10	1.31 (59%)	0.02 (11%)	0.0004 (6%)	1.53 (58%)	0.01 (4%)	16.22 (63%)	0.52 (11%)
ladder-20	2.75 (37%)	0.05 (15%)	0.0004 (5%)	3.14 (36%)	0.01 (2%)	21.48 (37%)	1.42 (15%)
ladder-50	8.27 (25%)	0.07 (9%)	0.0004 (5%)	9.58 (25%)	0.07 (5%)	52.34 (25%)	3.58 (22%)
ladder-100	30.13 (26%)	0.10 (7%)	0.0007 (8%)	40.03 (29%)	0.25 (9%)	162.95 (25%)	8.35 (36%)

Table 7.3: Standard deviation and relative standard deviation of running times for QuickPlan planner (QP), specialized planners (Spec), and planner DFAs (DFA).

to the former, their combined running time is notably more than the running time of the latter.

We also observe that the performance difference is more significant when looking at the maximum times. The running time of the QuickPlan-based planner fluctuates significantly more than the running time of the specialized planners. This can also be seen from the standard deviations of both types of planners, shown in Table 7.3. The timer for mobile Safari had insufficient resolution for our test setup, so the standard deviations were highly skewed, and not reported. We still report the averages and maximum times, as they at least give an indication of the performance.

Our tests do not give a precise picture of the growth of the algorithms' running times with the size of the systems. The *chain-n* and *ladder-n* tests, however, repeat the same test for four different sizes. They give some indication of a higher than linear growth for the QuickPlan-based planner, but not so for the specialized planners.

It should be noted that all of the tests, even with the slower general purpose planner, were solved in a fraction of a second; the worst-case running time of the slowest test was about one second on an Android tablet. Even though planning is already quick—for small systems it may be hard to imagine that QuickPlan would not be sufficiently fast—the reported performance gains are still significant.

First, the reaction to many tasks in user interfaces should be instantaneous. In order for the user to feel that the system is reacting instantaneously, the interface needs to respond to input in approximately 0.1 seconds or less [38, Ch. 5]. In the larger test cases, the QuickPlan-based planner is already on the wrong side of the 0.1 second guideline. Taking into account that planning is not the only task that must take place before the system is ready to respond (the methods in the constraint system need to be executed, parts of the UI re-rendered reflecting any changed values, to name a few), reducing the planning time even in the smaller systems is beneficial.

Second, QuickPlan's performance varies more than the specialized planners' per-

formance. For guarantees on instantaneousness, one needs to worry about the maximum running times, not merely the average. Variation in planning time could in some settings also lead to intermittent unresponsiveness.

Third, several interface behaviors call for tentative planning. For example, *pinning* [17] attempts to preserve a particular value by keeping its stay constraint at the highest priority. More than one variable may be pinned but not all combination of variables are valid for pinning. Therefore, given a set of variables already pinned, an application must determine which additional variables can and which cannot be pinned. This requires attempting to find a plan which preserves all pinned variables plus the additional variable—thereby calling the planner once for each variable that is not pinned. Another algorithm that can benefit from tentative planning is deciding when widgets bound to a variable should be enabled and when disabled [25]. For all but the smallest constraint systems, such algorithms quickly become infeasible if based on QuickPlan. Note that repeated runs of the planner for the purposes of pinning do not need the dependency graph: the relevant running time of a specialized solver is just the DFA execution time, without the overhead of constructing a solution graph.

Finally, we address the topic of the size of the generated DFA planners. All of our test cases were quite manageable, though a few generated DFAs with a moderate size: *dashed*, 2200 states in 27 seconds; *single*, 1013 states in 87 seconds; *solid*, 812 states in 4.5 seconds; *dotted*, 519 states in 1.6 seconds; and *ladder-100*, 101 states in 19 seconds. All other planners had fewer than hundred states, and they were generated in few seconds, or in a fraction of a second. The code for generating the DFA was not carefully optimized, but the data structures that it uses were selected so that the generator is not unnecessarily “pessimized”.

For a large and complex constraint system, a specialized planner can be prohibitively large. As a specialized planner can be a drop-in replacement for a general

planner, the decision to specialize or not can thus be made case by case, with ease.

8. CONCLUSION AND FUTURE WORK

It has become more and more common for GUIs to manifest rich and complex behaviors. These behaviors often require performing multiple interrelated tasks simultaneously, and modifying data dependencies to reflect the current state of the GUI. As GUIs grow in complexity, they become harder and harder to manage with traditional event-driven programming techniques. As a result, GUIs are difficult to implement correctly, difficult to debug, and commonly exhibit unpredictable behavior.

There are many factors contributing to erroneous or inconsistent behavior in GUIs. One substantial factor is the difficulty to orchestrate asynchronous computations so that they are not interleaved in ways that might lead to breaking the data dependencies that should be maintained in the GUI. Failure to prevent such bad interleavings results in GUI behavior that may be sensitive to slight changes in the timing of events. Event-driven programming places the entire burden of both identifying and enforcing data dependencies on the programmer.

A second contributing factor is the difficulty in adjusting to changing dependencies as data is added to and removed from the GUI, and as relationships between pieces of data are modified, e.g., when rearranging rows in a table. Again, event-driven programming places on the programmer the entire burden of changing the calculation of data dependencies to match the current shape of the data in the GUI. Failure to adapt to changes may result in dropped or erroneous dependency calculations. Combining this complex dependency management with the orchestration of asynchronous computations only increases the difficulty of both.

A third contributing factor is the inability to reuse GUI behavior across multiple applications. In event-driven programming, the only constant factor among all GUIs is events, yet the interpretation of those events varies greatly from one GUI to

another. This lends itself to an ad-hoc programming style in which the majority of the code implementing GUI behavior applies only to a single GUI.

GUI programming with property models is a new approach which relieves the programmer from the burden of dependency management. The programmer's responsibility is only to identify the possible data dependencies. It is the property model which enforces these dependencies and adapts as changes are made to the arrangement of data within the program. Furthermore, a property model defines a standard representation of dependencies within a program. Not only does this make it possible to create generic, reusable components, it also allows the possibility of generic algorithms parameterized over a specification of data dependencies, such as our algorithm for disabling irrelevant widgets.

There is still work to be done towards a complete programming model based on property models. One area we have identified for future work is formalizing a more principled model for structural changes to a property model. Our goal is that defining a property model would include defining structured ways in which it may be modified so that changes may be enacted automatically. In this way, the programmer is relieved of the burden of ensuring structural changes are enacted correctly.

A second area we have identified for future work is the bindings between the View to the View Model. Open questions include how the View may be automatically manipulated to match changes in the property model, and how commands may be delayed while still ensuring consistency in GUI behavior.

One final area for future work is the validation of variable values. Supporting validation of the output of dataflow functions promotes component-based property models by lowering the requirements for composing two components. We can also provide dataflow-sensitive validation. This allows validation to occur as close to the user's input as possible.

The work described in this dissertation defines a new programming model for

GUIs. In this model a run-time system assumes the responsibility of orchestrating asynchronous computations and responding to structural changes in a GUI. GUIs implemented in the model are guaranteed to be responsive and consistent. The model promotes reusable software components and algorithms that can capture fragments of GUI behavior. This new programming model lays a foundation for a future where rich, complex GUI behavior may be predictably implemented correctly.

REFERENCES

- [1] Adobe Flex — free, open-source framework, Accessed June, 2015. <http://www.adobe.com/products/flex.html>.
- [2] Apple Inc. *Cocoa Application Tutorial*, October 2007. <http://developer.apple.com/documentation/Cocoa/Conceptual/ObjCTutorial/>.
- [3] Bacon.js : Home, Accessed June, 2015. <http://baconjs.github.io/>.
- [4] 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.
- [5] John A. Bondy and Uppaluri S. R. Murty. *Graph Theory*. Springer, New York, NY, 2008.
- [6] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. *SIGPLAN Not.*, 22(12):48–60, December 1987.
- [7] 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.
- [8] Antony Courtney. Frappé: Functional reactive programming in Java. In *Third International Symposium on Practical Aspects of Declarative Languages (PADL)*, March 2001.
- [9] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. *Haskell Workshop*, pages 1–29, 2001.

- [10] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the ACM SIGPLAN workshop on Haskell — Haskell '03*, pages 7–18, New York, NY, USA, August 2003. ACM Press.
- [11] Jonathan Edwards. Subtext. *ACM SIGPLAN Notices*, 40(10):505, October 2005.
- [12] Conal Elliott and Paul Hudak. Functional reactive animation. *ACM SIGPLAN Notices*, 32(8):263–273, August 1997.
- [13] Ember.js - a framework for creating ambitious web applications., Accessed August, 2015. <http://emberjs.com/>.
- [14] Tim Felgentreff, Alan Borning, and Robert Hirschfeld. Specifying and solving constraints on object behavior. *Journal of Object Technology*, 13(4):1:1–38, September 2014.
- [15] Martin Fowler. Presentation Model Pattern, July 2004. <http://martinfowler.com/eaaDev/PresentationModel.html>.
- [16] 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.
- [17] 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.
- [18] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Commun. ACM*, 33(1):54–63, 1990.
- [19] John Gossman. Introduction to Model/View/ViewModel pattern for building WPF apps, October 2005.

- [20] John Gossman. Introduction to Model/View/ViewModel pattern for building WPF apps, October 2005. <http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx>.
- [21] Hotdrink: A javascript library for user interface programming, Accessed October, 2015. <http://github.com/HotDrink/hotdrink>.
- [22] Jaakko Järvi, Gabriel Foust, and Magne Haveræen. Specializing planners for hierarchical multi-way dataflow constraint systems. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 1–10, New York, NY, USA, 2014. ACM.
- [23] 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.
- [24] 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.
- [25] 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.
- [26] JavaFX — Oracle Documentation, Accessed June, 2015. <http://docs.oracle.com/javafx>.
- [27] Knockout : Home, Accessed June, 2015. <http://knockoutjs.com/>.

- [28] 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.
- [29] 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.
- [30] Barbara Liskov and Liuba Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *ACM SIGPLAN Notices*, 23(7):260–267, July 1988.
- [31] 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.
- [32] Reactive Extensions, Accessed June, 2015. <http://msdn.microsoft.com/en-us/data/gg577609>.
- [33] Brad A. Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proceedings of the 4th annual ACM symposium on User Interface Software and Technology*, UIST '91, pages 211–220, New York, NY, USA, 1991. ACM.
- [34] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander. Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85, November 1990.
- [35] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and

- Patrick Doane. The Amulet environment: New models for effective user interface software development. *Software Engineering*, 23(6):347–365, 1997. citeseer.ist.psu.edu/article/myers96amulet.html.
- [36] 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.
- [37] 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.
- [38] Jakob Nielsen. *Usability engineering*. AP Professional, Cambridge, MA, 1993.
- [39] Node.js, Accessed October, 2015. Node.js is a JavaScript runtime built on Chrome’s V8 JavaScript engine.
- [40] Stephen Oney, Brad Myers, and Joel Brandt. ConstraintJS: programming interactive behaviors for the web by integrating constraints and states. In *Proceedings of the 25th annual ACM symposium on User Interface Software and Technology*, UIST '12, pages 229–238, New York, NY, USA, 2012. ACM.
- [41] OpenLaszlo — the premier platform for rich internet applications, Accessed June, 2015. <http://www.openlaszlo.org/>.
- [42] 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.
- [43] John Peterson, Valery Trifonov, and Andrei Serjantov. Parallel functional reactive programming. In Enrico Pontelli and Vtor Santos Costa, editors, *Practical Aspects of Declarative Languages*, volume 1753 of *Lecture Notes in Computer Science*, pages 16–31. Springer Berlin Heidelberg, 2000.

- [44] Mike Potel. MVP: Model-view-presenter: The Taligent programming model for C++ and Java, 1996.
- [45] Meurig Sage. FranTk — a declarative GUI language for Haskell. In *Proc. of the fifth ACM SIGPLAN Int. Conf. on Functional programming — ICFP '00*, volume 35, pages 106–117, New York, NY, USA, September 2000. ACM Press.
- [46] 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.
- [47] 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.
- [48] Michael Sannella and Alan Hamilton Borning. Multi-Garnet: Integrating multi-way constraints with garnet. Technical Report 92-07-01, University of Washington, Department of Computer Science and Engineering, 1992.
- [49] Ivan E. Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, Massachusetts Institute of Technology, Lincoln Lab, 1963. Also published as technical report UCAM-CL-TR-574 of the University of Cambridge, UK, Computer Laboratory.
- [50] Gilles Trombettoni and Bertrand Neveu. Computational complexity of multi-way, dataflow constraint problems. In *IJCAI (1)*, pages 358–365, 1997.
- [51] Brad Vander Zanden. An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Trans. Program. Lang. Syst.*, 18(1):30–72, January 1996.

- [52] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. *ACM SIGPLAN Notices*, 35(5):242–252, May 2000.

APPENDIX A

LISTING OF OPERATIONAL SEMANTICS

This appendix lists the full operational semantics of property models. These semantics are explained in detail in Chapter 5. This appendix simply makes it easier to view the entire semantics at once without interruptions.

A.1 Overloaded Signatures

The following signatures are supported by the definition function.

Variable \rightarrow (Promise)

A variable is mapped to its *promise history*.

Constraint \rightarrow (Activation)

A constraint is mapped to its *activation history*.

Method \rightarrow \langle Constraint, Function, (Variable), (Variable) \rangle

A method is mapped to the constraint to which it belongs, its scheduling function, its input variables, and its output variables.

Activation \rightarrow \langle Method, (Promise), (Promise) \rangle

An activation is mapped to its method, the input promises, and the output promises.

The following signatures are supported by the valuation function.

Variable \rightarrow Value

A variable is mapped to its value.

Promise \rightarrow \langle Value, StatusFlag, UsedFlag \rangle

A promise is mapped to its value, its status, and its usage.

\langle Variable, Method $\rangle \rightarrow$ UsedFlag

An input edge of the constraint graph is mapped to its usage.

The enumerated types **StatusFlag** and **UsedFlag** are defined as follows.

StatusFlag = {pending, fulfilled}

UsedFlag = {unknown, used, unused}

A.2 Evaluation Environment

The following elements define a property model. Each is listed with the meta-variables used to represent them.

G_C = The constraint graph.

G_S = The solution graph.

G_R = The reactive program graph.

$\bar{\pi}$ = The variable priority assignment, sorted from lowest to highest priority.

Λ = The modified variable set.

Γ = The definition function.

Σ = The valuation function.

Δ = The callback set.

The following are other meta-variables appearing in the evaluation rules.

v, w := Variable

p, q := Promise

c := Constraint

s := StatusFlag

a := Activation

u := UsedFlag

m := Method

t, o := Value

f, g := Function

A.3 Editing the Property Model

TOUCH

$$\frac{\bar{\pi} = \bar{v}_1 \dot{+} (v) \dot{+} \bar{v}_2 \quad \bar{\pi}' = \bar{v}_1 \dot{+} \bar{v}_2 \dot{+} (v) \quad \Lambda' = \Lambda \cup \{v\}}{\text{touch}(v) \mid \bar{\pi}, \Lambda \rightarrow \cdot \mid \bar{\pi}', \Lambda'}$$

SET

$$\frac{\text{touch}(v) \mid \bar{\pi}, \Lambda \rightarrow \cdot \mid \bar{\pi}', \Lambda' \quad \text{add_promise}(v, p) \mid G_R, \Gamma, \Delta \rightarrow \cdot \mid G_R', \Gamma', \Delta'}{\text{set}(p) \mid G_R, \bar{\pi}, \Lambda, \Gamma, \Sigma, \Delta \rightarrow \cdot \mid G_R', \bar{\pi}', \Lambda', \Gamma', \Sigma', \Delta'}$$

UPDATE

$$\begin{array}{l} G_S' = \text{plan}(G_C, \bar{\pi}, G_S, \Lambda) \quad \bar{\pi}' = \text{adjust}(\bar{\pi}, G_S') \\ M = \text{downstream_many}(\Lambda) \mid G_S', \Sigma \quad \bar{m} = \text{topological_sort}(G_S', M) \\ \text{schedule}(\bar{m}) \mid G_R, \Gamma, \Sigma, \Delta \rightarrow \cdot \mid G_R', \Gamma', \Sigma', \Delta' \quad \Lambda' = \{\} \\ \hline \text{update}() \mid G_C, G_S, G_R, \bar{\pi}, \Lambda, \Gamma, \Sigma, \Delta \rightarrow \cdot \mid G_C, G_S', G_R', \bar{\pi}', \Lambda', \Gamma', \Sigma', \Delta' \end{array}$$

METHODS-DOWNSTREAM

$$\begin{array}{l} M = \{m \in \text{outs}(G_S, v) \mid \Sigma(\langle v, m \rangle) \neq \text{unused}\} \\ V = \cup_{m \in M} \text{outs}(G_S, m) \quad M' = \text{downstream_many}(V) \mid G_S, \Sigma \\ \hline \text{downstream}(v) \mid G_S, \Sigma = M \cup M' \end{array}$$

DOWNSTREAM-MANY

$$\begin{array}{l} V = \{v\} \cup V' \\ M = \text{downstream}(v) \mid G_S, \Sigma \\ M' = \text{downstream_many}(V') \mid G_S, \Sigma \\ \hline \text{downstream_many}(V) \mid G_S, \Sigma = M \cup M' \end{array}$$

DOWNSTREAM-MANY-EMPTY

$$\text{downstream_many}(\{\}) \mid G_S, \Sigma = \{\}$$

A.4 Scheduling Methods

SCHEDULE-METHODS

$$\begin{array}{l}
\bar{m} = (m) \dashv\vdash \bar{m}' \quad \Gamma(m) = \langle _, f, (v_1, \dots, v_j), (w_1, \dots, w_k) \rangle \\
\forall_{i=1}^j : \Gamma(v_i) = (\dots, p_i) \quad \text{duplicate}((p_1, \dots, p_j)) \mid \Sigma, \Delta \rightarrow (p'_1, \dots, p'_j) \mid \Sigma', \Delta' \\
\quad f(p'_1, \dots, p'_j) \mid \Sigma', \Delta' \rightarrow (q_1, \dots, q_k) \mid \Sigma'', \Delta'' \\
\text{add_promise_many}((w_1, \dots, w_k), (q_1, \dots, q_k)) \mid G_R, \Gamma, \Delta'' \rightarrow \cdot \mid G_R', \Gamma', \Delta^{(3)} \\
\quad \text{reset}(m) \mid \Gamma', \Sigma'' \rightarrow \cdot \mid \Gamma', \Sigma^{(3)} \\
\text{make_activation}(m, (p'_1, \dots, p'_j), (q_1, \dots, q_k)) \mid \Gamma', \Delta^{(3)} \rightarrow a \mid \Gamma'', \Delta^{(4)} \\
\quad \text{add_to_graph}(a, (p_1, \dots, p_j), (q_1, \dots, q_k)) \mid G_R' \rightarrow \cdot \mid G_R'' \\
\quad \text{schedule}(\bar{m}') \mid G_R'', \Gamma'', \Sigma^{(3)}, \Delta^{(4)} \rightarrow G_R^{(3)}, \Gamma^{(3)}, \Sigma^{(4)}, \Delta^{(5)} \\
\hline
\text{schedule}(\bar{m}) \mid G_R, \Gamma, \Sigma, \Delta \rightarrow \cdot \mid G_R^{(3)}, \Gamma^{(3)}, \Sigma^{(4)}, \Delta^{(5)}
\end{array}$$

SCHEDULE-METHODS-EMPTY

$$\text{schedule}() \mid G_R, \Gamma, \Sigma, \Delta \rightarrow \cdot \mid G_R, \Gamma, \Sigma, \Delta$$

DUPLICATE-PROMISES

$$\begin{array}{l}
\bar{p} = (p) \dashv\vdash \bar{p}' \quad q = \text{newsym}() \quad \Sigma' = [q \mapsto \langle \cdot, \text{pending}, \text{unknown} \rangle] \Sigma \\
\Delta' = \{ \langle \text{copy}, p, q \rangle \} \cup \Delta \quad \text{duplicate}(\bar{p}') \mid \Sigma', \Delta' \rightarrow \bar{q} \mid \Sigma'', \Delta'' \\
\hline
\text{duplicate}(\bar{p}) \mid \Sigma, \Delta \rightarrow (q) \dashv\vdash \bar{q} \mid \Sigma'', \Delta''
\end{array}$$

DUPLICATE-PROMISES-EMPTY

$$\text{duplicate}() \mid \Sigma, \Delta \rightarrow () \mid \Sigma, \Delta$$

DUPLICATE-PROMISE-FULFILLED

$$\begin{array}{l} \Delta = \{\langle \text{copy}, p, q \rangle\} \cup \Delta' \quad \Sigma(p) = \langle t, \text{fulfilled}, _ \rangle \\ \Sigma(q) = \langle \cdot, \text{pending}, u \rangle \quad \Sigma' = [q \mapsto \langle t, \text{fulfilled}, u \rangle] \Sigma \\ \hline \cdot \mid \Sigma, \Delta \rightarrow \cdot \mid \Sigma', \Delta' \end{array}$$

RESET-CONSTRAINT

$$\begin{array}{l} \Gamma(m) = \langle c, _, _, _ \rangle \quad \Gamma(c) = (\dots, a) \quad \Gamma(a) = \langle m', _, _ \rangle \\ \Gamma(m') = \langle c, _, (v_1, \dots, v_j), _ \rangle \quad \Sigma' = [\forall_{i=1}^j : \langle v_i, m' \rangle \mapsto \text{unknown}] \Sigma \\ \hline \text{reset}(m) \mid \Gamma, \Sigma \rightarrow \cdot \mid \Gamma, \Sigma' \end{array}$$

MAKE-ACTIVATION

$$\begin{array}{l} a = \text{newsym}() \quad \Gamma' = [a \mapsto \langle m, (p_1, \dots, p_j), (q_1, \dots, q_k) \rangle] \Gamma \\ \Gamma'(m) = \langle c, _, _, _ \rangle \quad \Gamma'' = [c \mapsto \Gamma'(c) \uparrow (a)] \Gamma' \\ \Delta' = \{\langle \text{running}, a \rangle\} \cup \{\langle \text{input}, p_i, a \rangle \mid 1 \leq i \leq j\} \cup \Delta \\ \hline \text{make_activation}(m, (p_1, \dots, p_j), (q_1, \dots, q_k)) \mid \Gamma, \Delta \rightarrow a \mid \Gamma'', \Delta' \end{array}$$

ADD-TO-GRAPH

$$\begin{array}{l} G_R' = \text{add_node}(G_R, a) \quad G_R'' = \text{add_edges}(G_R', \{\langle p_i, a \rangle \mid 1 \leq i \leq j\}) \\ G_R^{(3)} = \text{add_edges}(G_R'', \{\langle a, q_i \rangle \mid 1 \leq i \leq k\}) \\ \hline \text{add_to_graph}(a, (p_1, \dots, p_j), (q_1, \dots, q_k)) \mid G_R \rightarrow \cdot \mid G_R^{(3)} \end{array}$$

A.5 Variables and Promises

ADD-PROMISE

$$\frac{\Gamma' = [v \mapsto \Gamma(v) \mathbin{++} (p)]\Gamma \quad G_R' = \text{add_node}(G_R, p) \quad \Delta' = \{\langle \text{var}, p, v \rangle\} \cup \Delta}{\text{add_promise}(v, p) \mid G_R, \Gamma, \Delta \rightarrow \cdot \mid G_R', \Gamma', \Delta'}$$

ADD-PROMISE-MANY

$$\frac{\bar{v} = (v) \mathbin{++} \bar{v}' \quad \bar{p} = (p) \mathbin{++} \bar{p}' \quad \text{add_promise}(v, p) \mid G_R, \Gamma, \Delta \rightarrow \cdot \mid G_R', \Gamma', \Delta' \quad \text{add_promise_many}(\bar{v}', \bar{p}') \mid G_R', \Gamma', \Delta' \rightarrow \cdot \mid G_R'', \Gamma'', \Delta''}{\text{add_promise_many}(\bar{v}, \bar{p}) \mid G_R, \Gamma, \Delta \rightarrow \cdot \mid G_R'', \Gamma'', \Delta''}$$

ADD-PROMISE-MANY-EMPTY

$$\text{add_promise_many}((), ()) \mid G_R, \Gamma, \Delta \rightarrow \cdot \mid G_R, \Gamma, \Delta$$

VARIABLE-PROMISE-FULFILLED

$$\frac{\Delta = \{\langle \text{var}, p, v \rangle\} \cup \Delta' \quad \Sigma(p) = \langle _, \text{fulfilled}, _ \rangle \quad \text{maybe_set_var}(v, p) \mid \Gamma, \Sigma \rightarrow \cdot \mid \Gamma, \Sigma'}{\cdot \mid \Gamma, \Sigma, \Delta \rightarrow \cdot \mid \Gamma, \Sigma', \Delta'}$$

MAYBE-SET-VAR-VISIBLE

$$\frac{\Gamma(v) = (p_1, \dots, p_j, \dots, p_k) \quad p_j = p \quad \forall_{i=j+1}^k : \Sigma(p_i) = \langle _, \text{pending}, _ \rangle \quad \Sigma(p) = \langle t, _, _ \rangle \quad \Sigma' = [v \mapsto t]\Sigma}{\text{maybe_set_var}(v, p) \mid \Gamma, \Sigma \rightarrow \cdot \mid \Gamma, \Sigma'}$$

MAYBE-SET-VAR-INVISIBLE

$$\frac{\Gamma(v) = (p_1, \dots, p_j, \dots, p_k) \quad p_j = p \quad \exists i : j < i \leq k \wedge \Sigma(p_i) = \langle _, \text{fulfilled}, _ \rangle}{\text{maybe_set_var}(v, p) \mid \Gamma, \Sigma \rightarrow \cdot \mid \Gamma, \Sigma}$$

A.6 Promise and Edge Usage

SET-USAGE

$$\frac{\Sigma(p) = \langle t, s, \text{unknown} \rangle \quad \Sigma' = [p \mapsto \langle t, s, u \rangle] \Sigma}{\text{usage}(p, u) \mid \Sigma \rightarrow \cdot \mid \Sigma'}$$

SET-USAGE-AGAIN

$$\frac{\Sigma(p) = \langle t, s, u' \rangle \quad u' \neq \text{unknown}}{\text{usage}(p, u) \mid \Sigma \rightarrow \cdot \mid \Sigma}$$

SUBSCRIBE-UNKNOWN

$$\frac{\Sigma(p) = \langle t, s, \text{unknown} \rangle \quad \Sigma' = [p \mapsto \langle t, s, \text{used} \rangle] \Sigma \quad \Delta' = \Delta \cup \{ \langle \text{extern}, p, f \rangle \}}{\text{subscribe}(p, f) \mid \Sigma, \Delta \rightarrow \cdot \mid \Sigma', \Delta'}$$

SUBSCRIBE-KNOWN

$$\frac{\Sigma(p) = \langle _, _, u \rangle \quad u \neq \text{unknown} \quad \Delta' = \Delta \cup \{ \langle \text{extern}, p, f \rangle \}}{\text{subscribe}(p, f) \mid \Sigma, \Delta \rightarrow \cdot \mid \Sigma, \Delta'}$$

FULL-FILL-PROMISE

$$\frac{\Sigma(p) = \langle _, \text{pending}, u \rangle \quad \Sigma' = [p \mapsto \langle t, \text{fulfilled}, u \rangle] \Sigma}{\text{fulfill}(p, t) \mid \Sigma \rightarrow \cdot \mid \Sigma'}$$

FULL-FILL-PROMISE-AGAIN

$$\frac{\Sigma(p) = \langle _, \text{fulfilled}, _ \rangle}{\text{fulfill}(p, t) \mid \Sigma \rightarrow \cdot \mid \Sigma}$$

SUBSCRIBED-PROMISE-FULFILLED

$$\frac{\Delta = \{ \langle \text{extern}, p, f \rangle \} \cup \Delta' \quad \Sigma(p) = \langle t, \text{fulfilled}, _ \rangle \quad f(t) \mid \Sigma, \Delta' \rightarrow \cdot \mid \Sigma', \Delta''}{\cdot \mid \Sigma, \Delta \rightarrow \cdot \mid \Sigma', \Delta''}$$

ACTIVATION-COMPLETED

$$\begin{array}{c}
\Delta = \{\langle \text{running}, a \rangle\} \cup \Delta' \\
\Gamma(a) = \langle m, (p_1, \dots, p_j), (q_1, \dots, q_k) \rangle \quad \forall_{i=1}^k : \Sigma(q_i) = \langle _, \text{fulfilled}, _ \rangle \\
\Sigma' = [\forall_{i=1}^j : \Sigma(p_i) = \langle t, s, \text{unknown} \rangle \implies p_j \mapsto \langle t, s, \text{unused} \rangle] \Sigma \\
\hline
\cdot \mid \Gamma, \Sigma, \Delta \rightarrow \cdot \mid \Gamma, \Sigma', \Delta'
\end{array}$$

INPUT-USAGE-KNOWN

$$\begin{array}{c}
\Delta = \{\langle \text{input}, p, a \rangle\} \cup \Delta' \\
\Sigma(p) = \langle _, _, u \rangle \quad u \neq \text{unknown} \quad \text{maybe_set_edge}(a, p) \mid \Gamma, \Sigma \rightarrow \cdot \mid \Gamma, \Sigma' \\
\hline
\cdot \mid \Gamma, \Sigma, \Delta \rightarrow \cdot \mid \Gamma, \Sigma', \Delta'
\end{array}$$

MAYBE-SET-EDGE-CURRENT

$$\begin{array}{c}
\Gamma(a) = \langle m, \bar{p}, _ \rangle \quad \Gamma(m) = \langle c, _, (v_1, \dots, v_k), _ \rangle \quad \Gamma(c) = (\dots, a) \\
\bar{p} = (p_1, \dots, p_j, \dots, p_k) \quad p_j = p \quad \Sigma(p) = \langle _, _, u \rangle \quad \Sigma' = [\langle v_j, m \rangle \mapsto u] \Sigma \\
\hline
\text{maybe_set_edge}(a, p) \mid \Gamma, \Sigma \rightarrow \cdot \mid \Gamma, \Sigma'
\end{array}$$

MAYBE-SET-EDGE-OLD

$$\begin{array}{c}
\Gamma(a) = \langle m, _, _ \rangle \quad \Gamma(m) = \langle c, _, _, _ \rangle \quad \Gamma(c) = (\dots, a') \quad a \neq a' \\
\hline
\text{maybe_set_edge}(a, p) \mid \Gamma, \Sigma \rightarrow \cdot \mid \Gamma, \Sigma
\end{array}$$

A.7 Lifting Functions

LIFT-FUNCTION

$$\begin{array}{c}
\Sigma' = [\forall_{i=1}^j : \Sigma(p_i) = \langle t, s, \text{unknown} \rangle \implies p_i \mapsto \langle t, s, \text{used} \rangle] \Sigma \\
\forall_{i=1}^k : q_i = \text{newsym}() \quad \Sigma'' = [\forall_{i=1}^k : q_i \mapsto \langle \cdot, \text{pending}, \text{unknown} \rangle] \Sigma' \\
\Delta' = \{ \langle \text{lifted}, g, (p_1, \dots, p_j), (q_1, \dots, q_k) \rangle \} \cup \Delta \\
\hline
\text{lift}(g, k, p_1, \dots, p_j) \mid \Sigma, \Delta \rightarrow (q_1, \dots, q_k) \mid \Sigma'', \Delta'
\end{array}$$

LIFTED-INPUTS-READY

$$\begin{array}{c}
\Delta = \{ \langle \text{lifted}, g, (p_1, \dots, p_j), (q_1, \dots, q_k) \rangle \} \cup \Delta' \\
\forall_{i=1}^j : \Sigma(p_i) = \langle t_i, \text{fulfilled}, _ \rangle \quad g(t_1, \dots, t_j) = (o_1, \dots, o_k) \\
\Sigma' = [\forall_{i=1}^k : \Sigma(q_i) = \langle _, \text{pending}, u_i \rangle \implies q_i \mapsto \langle o_i, \text{fulfilled}, u_i \rangle] \Sigma \\
\hline
\cdot \mid \Sigma, \Delta \rightarrow \cdot \mid \Sigma', \Delta'
\end{array}$$