Generating Reactive Programs for Graphical User Interfaces from Multi-way Dataflow Constraint Systems

Gabriel Foust
Texas A&M University, TX, USA
gfoust@cse.tamu.edu

Jaakko Järvi
Texas A&M University, TX, USA
jarvi@cse.tamu.edu

Sean Parent
Adobe Systems, Inc.
sparent@adobe.com

Abstract

For a GUI to remain responsive, it must be able to schedule lengthy tasks to be executed asynchronously. In the traditional approach to GUI implementation—writing functions to handle individual user events—asynchronous programming easily leads to defects. Ensuring that all data dependencies are respected is difficult when new events arrive while prior events are still being handled. Reactive programming techniques, gaining popularity in GUI programming, help since they make data dependencies explicit and enforce them automatically as variables' values change. However, data dependencies in GUIs usually change along with its state. Reactive programming must therefore describe a GUI as a collection of many reactive programs, whose interaction the programmer must explicitly coordinate. This paper presents a declarative approach for GUI programming that relieves the programmer from coordinating asynchronous computations. The approach is based on our prior work on "property models", where GUI state is maintained by a dataflow constraint system. A property model responds to user events by atomically constructing new data dependencies and scheduling asynchronous computations to enforce those dependencies. In essence, a property model dynamically generates a reactive program, adding to it as new events occur. The approach gives the following guarantee: *the same sequence of events produces the same results, regardless of the timing of those events*.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Domain-specific architectures

General Terms Design, Theory

Keywords Dataflow constraint systems, Graphical user interfaces, asynchronous programming

1. Introduction

For a Graphical User Interface (GUI) to remain responsive while performing lengthy tasks, e.g., image processing or remote server communication, it must support *asynchronous* execution. That is, it must be able to begin new tasks even though not all prior tasks have completed. Asynchronous execution can take the form of executing an algorithm on a separate thread, performing other work while waiting on a server response, or even using time-sharing techniques to make progress on multiple tasks at once.

GPCE'15, October 26–27, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3687-1/15/10...$15.00
http://dx.doi.org/10.1145/2814204.2814207

(a) An auto-complete text box. (b) The data dependencies.

Figure 1: An example of an auto-complete text box, and a diagram showing the data dependencies involved in its implementation.

Asynchronous execution is complicated by data dependencies between 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. The programmer must carefully guard against execution schedules that could produce incorrect results. This is not easy in the event-driven GUI programming paradigm, where data dependencies implicitly arise whenever multiple event handlers share variables.

By way of illustration, we examine 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 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 a mouse or keyboard, may select an alternate input string. Figure 1a shows an auto-complete text box being used to select a city as a travel destination.

Figure 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.

We claim these dependencies are non-trivial, and that 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 (expedia.com, orbitz.com, aa.com, united.com, hotels.com, and 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 this case, even though the K- and U- keypress events occurred before the "initiate search" event, they were handled after.

In a travel booking application the consequences of inconsistencies are not that severe. If an error manifests, it is relatively easy to detect and correct. Further, users learn to avoid errors by adapting their use, e.g., to wait for the GUI to catch up. It is not hard, however, to find more harmful problems in other widely used applications. In a recent interaction with the Blackboard system (www.blackboard.com, used by the majority of 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.

The event-driven programming model is at least partially to blame: even after the programmer understands all dependencies in a GUI, there is still the formidable task of ensuring that they are correctly enforced in all possible interleavings of different event handlers. To relieve the programmer from this task, we propose a declarative approach to GUI programming where the data dependencies are explicit and enforced by the system.

The central abstraction in our approach is what we call a *property model* that holds data used by the GUI and manages dependencies among the data using a *hierarchical multi-way dataflow constraint system* [3]. Property models are prior work of our research group [13, 14], but as presented before, their operation was defined with no regard to asynchronicity. This paper shows how property models can orchestrate asynchronous responses to user events, and guarantee both responsiveness and consistent results.

The key idea of our approach is to divide the work of the multi-way constraint system into two phases. To react to changes in the system's variables, the first phase determines which dependencies must be recalculated, and produces a *plan* for performing those computations in a consistent order. The second phase performs the computations and updates variables according to the plan. The first phase executes synchronously. The second phase is allowed to proceed asynchronously at its own pace, in any order permitted by the plan—the results are consistent and, assuming no side-effects, deterministic. After the first phase finishes, the GUI can immediately accept new events. An upper limit for the execution time of the first phase can be determined for each constraint system. In practice the execution is often instantaneous [15], 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. [2, 17, 20]). These frameworks build reactive programs from a static description of data dependencies. Data dependencies in a GUI, however, often change based on user interaction, which requires 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.

2. Background

A property model is an abstraction that automates many parts of GUI implementation. It is not tied to any particular language or GUI framework; rather, we assume it to be a library of the language, call it the *implementation language*, in which the GUI is implemented. Our proof-of-concept implementation [12] is targeted for developing web applications in JavaScript. Here we give an overview of property models and their role in GUI implementation.

2.1 Property Models

The traditional GUI programming model is organized around events which occur during the lifetime of the application: the programmer writes and registers event-handling functions for callback on appropriate events. The difficulty of programming complex GUIs with this model has long been recognized [13, 22], and it has resulted in search for alternatives. We identify two trends emerging from this search in which property models find their roots.

The first trend is the focus on the separation of concerns, which has resulted in design patterns that divide GUI applications into clearly delineated components, including *Model-View-Controller* [18], *Presentation-Model* [10], and *Model-View-ViewModel* (MVVM) [11]. The most recent of these, the MVVM pattern, divides the application into three parts. The *Model* is responsible for the "business" data and logic and is generally independent of the UI. The *View* is responsible for the presentation and translating user actions into events; it controls the behavior of the individual elements (sometimes called *widgets* or *controls*) that compose the GUI. The third component, the *View-Model*, is the application's model, or abstraction, of the View. It maintains the data used by the GUI and provides logic defining how the interface behaves as a whole. This division of responsibilities is well suited for a property model, which neatly fills the role of a View-Model.

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 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.

As a View-Model, the property model represents the current state of the GUI. Bindings ensure the View always reflects the current state of the GUI, and that changes to data of the View are translated into changes in GUI state; we refer to these changes as *edits*. Thus, the operation of a GUI based on a property model proceeds as follows. User events are translated by bindings into edits of the property model. The property model responds to the edits by producing a new GUI state. The resulting GUI state is translated back to the View by bindings.

The second trend which contributes to property models is the replacing of event-handling code with dataflow constraints. A dataflow constraint specifies how a variable's value should be calculated as a function of other variables, every time those other variables change. Many GUI frameworks today provide constructs for defining one-way dataflow constraints [17, 24, 25]. A system of one-way dataflow constraints may be viewed as a reactive program.

Property models, too, use dataflow constraints to respond to edits. Unlike in most GUI frameworks, property models' constraints can be multi-directional. They are maintained by a *hierarchical multi-way dataflow constraint system*, summarized below; Zanden provides a thorough discussion [32] of the kind of constraint systems we use with property models. A system of multi-way dataflow constraints may be viewed as a generator for a reactive program.

2.1.1 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

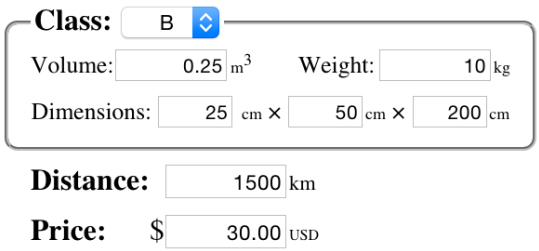

Figure 2: A hypothetical form for determining shipping prices. Note the large number of data dependencies.

rather implicitly through a set of *constraint satisfaction methods*, or simply *methods*. Each method is a function that calculates new values for some variables of the constraint using the remaining variables as inputs. A method should *enforce* the constraint. That is, after execution of the method, the constraint should be satisfied. Of the property model's variables, a method should only read its inputs, and only write to its outputs, but a method does not need to be referentially transparent.

Constraints can have any number of methods. 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 2.1.2). The second requirement, known as *method restriction* [29], ensures that a plan is found in polynomial time [31].

As an example of constraints arising in GUIs, we consider the GUI shown in Figure 2. In this hypothetical shipping application, packages are classified by weight and volume into one of several "package classes;" a package's shipping price is a function of its class and the shipping distance. Any field in the GUI may be edited; other fields are updated accordingly. The GUI thus 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 the package class directly, or choose to enter the weight and volume of the package instead, or the weight and dimensions.

First, we consider the constraint between the volume v of a package and its dimensions x, y, and z. The relation for this constraint is $v = xyz$. The constraint itself is defined by four methods, $x \leftarrow v/yz$, $y \leftarrow v/xz$, $z \leftarrow v/xy$, and $v \leftarrow xyz$, that each calculate a new value for one variable from the remaining three. We label the functions for these methods A, B, C, and D, respectively.

Next, we consider the constraint amongst volume, weight w, and class c of a package. The weight and volume determine the class of a package according to some function E, but we can also query for a representative weight and volume for a given class according to another function F (E and F are not shown). The two methods of the constraint are thus $(w, v) \leftarrow E(c)$ and $c \leftarrow F(w, v)$.

The third constraint is between the package class, shipping distance d, and shipping price p. However, this constraint is difficult to define because many price/distance combinations do not exactly match a shipping class. By adding the variable m to represent the maximum allowed price, we can define the relation like so: a package of class c can be shipped a distance d for a price p, no greater than m. This constraint can be implemented by 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, and 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 p of shipping a package of class c a distance d; this price also becomes the maximum price m.

We assume here that the binding of the *Price* text box reads from p and writes to m. In this way, the user enters the maximum price and sees the actual calculated price.

2.1.2 Hierarchical Multi-way Dataflow Constraint Systems

A *multi-way dataflow constraint system* is responsible for ensuring that 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. This means executing the methods in an order where 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*.

To satisfy all constraints in the constraint system, that is, to *solve* 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.[1]

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

2.1.3 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. Here, we discuss two of these; the Evaluation Graph [14] is not relevant to this article. We 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. The node set N is $V \cup M$. The edge set E contains an edge (v, m) if variable v is an input of method m, and an edge

[1] There are other occasional circumstances in which a constraint's priority may be altered; they are beyond the scope of this paper.

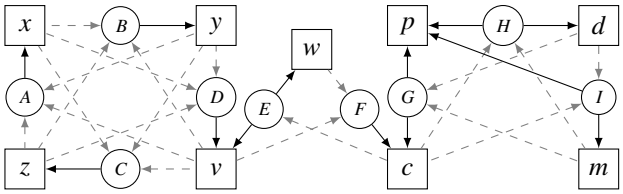

Figure 3: A constraint graph for the shipping GUI. Square nodes are variables, round nodes are methods. Dashed edges are methods' inputs, solid their outputs.

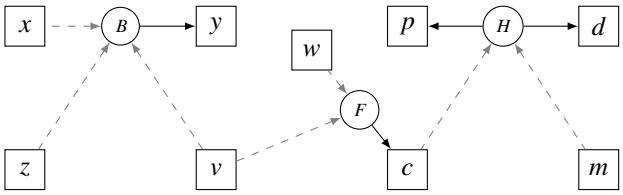

Figure 4: A solution graph based on the constraint graph of Figure 3. This solution graph shows plan $\{B, F, H\}$.

(m, v) if v is an output of 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 3. This graph reflects the three constraints we defined in Section 2.1.1; we have omitted the stay constraints. Constraints are not explicit in this graph, but because of method restriction, we may deduce them: two methods m and n are in the same constraint iff they have the same graph *neighborhood*.

Solution graph The solution graph represents the dependencies enforced by a particular plan. Specifically, given a constraint graph $G = \langle V \cup M, E \rangle$ and a plan $P \subseteq M$, a solution graph is the node-induced subgraph $S = G[V \cup P]$. By the definition of a plan, the solution graph is acyclic. Valid execution orders of a plan are all the topologically sorted sequences of the methods of P.

Figure 4 shows one possible solution graph for the shipping form. The graph represents the plan $P = \{B, F, H\}$; valid execution orders are those in which F comes before H.

3. The Core Reactive Program

Generally, reactive programs are described by a single dataflow: one sequence of computations to be repeated for every change in input. Because dataflow in a property model may change, its reactive program must be constructed dynamically. The program is constructed in layers. User events are translated to edits that the property model reacts to by scheduling new computations to resolve dependencies, resulting in a new layer of the reactive program.

At the heart of this construction process is a multi-way constraint system. This system responds to edits by computing a plan for enforcing constraints as described in Section 2.1.2, but rather than executing the plan's methods immediately, it merely schedules them. The planning and scheduling is an atomic step to ensure consistent behavior; the methods themselves, however, may be executed asynchronously. Execution of the reactive program proceeds autonomously: methods execute as their inputs become available, the View is updated to reflect new values, and older layers of the reactive program that are no longer needed are reclaimed.

3.1 Variables

A variable of a property model represents a value that changes over time. Variables' values do not change continuously; they change only in response to an edit (i.e., *set* or *touch*) of the property model: either as a direct result of the edit, or as a result of solving the constraint system in response to the edit. Once the system has been solved, all variables remain constant until the next edit.

We represent variable values using a well-known asynchronous programming construct: a promise [19]. A *promise* represents a value that may not yet be available, but will be in the future. A promise whose value is not yet available is *pending*. Once the value becomes available, the promise is *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. Subscribing to a fulfilled promise results in the callback being invoked at the next opportunity. Promises may also be *rejected* to indicate that the process intended to produce its value has failed and therefore the value will never be available. We consider rejected promises in Section 4.2.

For each variable in the system, we define a sequence of promises that we call 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 time. The last promise in the promise history, known as the *current promise*, represents the current value of the variable. To assign a new value, we add a new promise to the end of the promise history. This promise may be pending. Promises are only added to the history, never removed.

3.2 Methods

Methods define the dataflow between the promises of the reactive program. In an asynchronous property model, methods are represented by functions whose inputs and outputs are promises. When called by the constraint solver, these functions do no real work: they merely subscribe to the input promises, construct the output promises, and schedule 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 the method's function, it *schedules* the method.

It is through methods that the reactive program may achieve asynchronous execution. Methods are free to 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. 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. Which approach, if any, a method uses does not affect any other part of the reactive program.

To aid the discussion below, we define the term *method activation* for a single execution of a method: what starts with scheduling a 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 programs, 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.

3.3 Constructing the Reactive Program

As stated previously, the reactive program is generated incrementally in response to edits to 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 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 Sec-

tion 2.1.2. 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 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.

Some methods of the plan may not need to be scheduled, because executing them is known to have no effect. Given a method m in constraint c, if the last activation in c's activation history is an activation of m, and if the inputs of m remain unchanged since that activation, then we assume executing m would leave its outputs unchanged. Thus, the scheduled methods are those *not* selected in and those whose inputs have changed since the previous generation.

After all methods have been scheduled, the property model's response to an edit is complete; program control can return to the main event-loop, 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—edit events are queued if planning and scheduling for prior edits have not been completed. The worst-case execution time for planning is quadratic in the number of variables, methods, or constraints [32]. In practice planning is often instantaneous [15] and no planning is needed (the dataflow remains unchanged) when an edit is to a variable whose stay constraint is already enforced, e.g., when repeatedly editing the same variable.

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 entire reactive program.

3.4 The Reactive Program Graph

As stated above, the life span of a property model consists of a sequence of edits, each followed by an update in which the constraint system is solved. 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, containing all activations and promises generated while solving the constraint system in that generation.

By way of example, Figure 5 shows the reactive program graph of one possible sequence of edits in our shipping example. Promise nodes are labeled by the variable to which the promise is assigned and activation nodes by the methods that scheduled them. Each label is sub-scripted with a generation. 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.

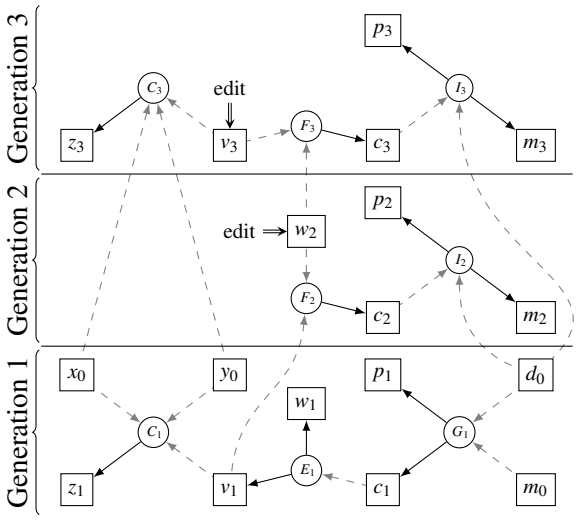

Figure 5: A reactive program graph based on the shipping form constraint system. Rectangles represent promises; circles represent method activations. Nodes from the same generation are grouped together in generations, which are ordered by time along the y-axis.

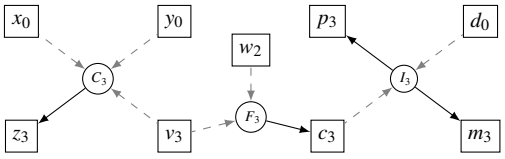

Figure 6: A "top-down" view of the reactive program graph from Figure 5, showing the most recent activation for each constraint and the most recent promise for each variable.

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 most recent promise for each variable and the most recent activation for each constraint. Such a "top-down" view of the graph found in Figure 5 may be seen in Figure 6. Once all activations are complete and all promises fulfilled, then this is the view the GUI will make visible to the user.

3.5 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 no-

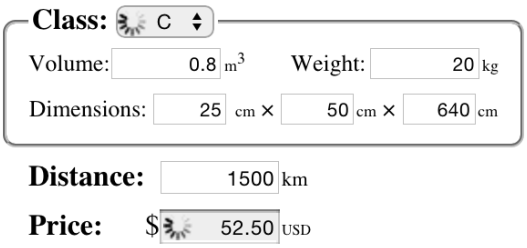

Figure 7: The shipping form after Generation 3 of Figure 5, while activation f_3 is still running.

tifications 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 the 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 is fulfilled, the variable will publish the promise's value as its own value *unless* a later promise in its promise history has already been fulfilled.

Consider again the "top-down" view of the reactive program graph shown in Figure 6. We may imagine that promises which are pending are transparent, so that for each variable we see the topmost fulfilled promise. 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 a 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 7 shows the shipping form GUI as it appears just after Generation 3 in Figure 5: 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 in part in Section 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 consistent results, no matter how quickly (or slowly) those edits occur.

3.6 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 a activation history. Here we consider when elements of the reactive program may be released and their resources collected.

We define a program element, a promise or an activation, to be *irrelevant* when it can no longer affect the GUI. The methods of a property model do not directly affect the GUI; their only external effect is to fulfill output promises. Fulfilled promises directly affect the GUI when bindings propagate their values to the view, which is when all more recent promises in the same promise history are still pending. We call such promises *visible*.

Just because a promise is not visible does not mean it is irrelevant; other promises that are visible may depend on its value. 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 irrelevant if and only if there are no paths in the reactive program graph from that element to a visible promise.

Irrelevant promises may be discarded. Activations which are irrelevant may be *unscheduled* so that they will no longer respond to the fulfillment of their input promises, and then discarded. 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 (or anywhere in) the graph as they become irrelevant. Memory usage can be expected to stay constant.

Note that even though we describe the identification of irrelevant program elements as a graph traversal task, no graph search is necessary. Our implementation uses a reference counting scheme in which promises keep track of their subscribers; when a promise looses all subscribers it becomes irrelevant and may be discarded.

4. Extending the Reactive Program

We have added several features to the core reactive program defined in Section 3. Here we describe two of them: accessing prior values in the reactive program and handling failure in method activation.

4.1 Accessing Prior Values

Each edit of a property model engenders a new generation of the reactive program in which method activations compute new values for some variables. 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. The reactive program graph gives a clear interpretation for current and prior values. Generally, a method activation is passed the current value of each of its inputs. In some methods, prior values of inputs are needed too, e.g., to calculate the difference between the previous and current generation. The prior values of output variables can be useful too.

Methods 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 current value is always represented by the current promise (the last element of the promise history); 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 constraint in our shipping form example between the volume, width, height, and length of a package. In our previous specification of this constraint, setting volume resulted in a change to just one dimension using the other two dimensions 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 $v{\leftarrow}whl$, which we previously labeled d, and a new method we label j: $(w, h, l) \leftarrow (rw', rh', rl')$ where w', h', and l' are the previous values of, respectively, w, h, and l, and where $r = \sqrt[3]{v/(w'h'l')}$.

Because no method may output to a prior value, whether or not a method uses prior values 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

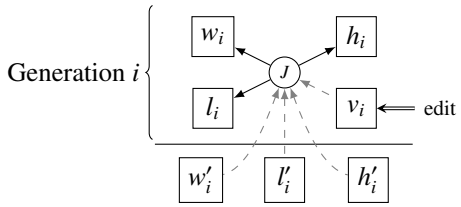

Figure 8: Program graph for a constraint using prior values.

them, representing additional data dependencies. These dependencies are identical to dependencies for current values: they prevent the method from executing until the promise is fulfilled, and provide paths by which program elements may be considered relevant. Figure 8 shows the reactive program graph for an activation of our new method, J. We do not specify which generation w'_i, h'_i, or l'_i come from; we know only that they are from a prior generation.

4.2 Failure in Method Activation

A method activation executes arbitrary user code, and thus is subject to, e.g., resource allocation failure, network failure, 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. 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. As an example, refer back to the reactive program graph in Figure 5. Let us suppose that activation G_1 fails, and consider the effect on each of the three generations of the program.

In Generation 1, G_1 fails to produce values for p_1 and c_1. Thus, p_1 and c_1 will remain pending forever, as will their dependencies, w_1, v_1, and z_1. At the end of 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) irrelevant. 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 l which are unchanged from the previous generation. Not only that, m becomes stuck in the pending state as well; 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 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 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 irrelevant, 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 the end of Generation 3, 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.

```
1  function D(x, y, z) { return x*y*z; }
2
3  function J(x, y, z, v) {
4    r = Math.cbrt(v/(x*y*z))*100;
5    return [r*x, r*y, r*z]; }
6
7  function F(v, w) {
8    p = new hd.Promise();
9    performShippingClassQuery(v, w,
10     function(c) { p.resolve(c); });
11   return p; }
```

Figure 9: The JavaScript definitions for three methods of the shipping form constraint system.

We can make the following guarantees about the behavior of a property model when method failure occurs. First, the same sequence of edits yields the same reactive program regardless of method failure. Method failure occurs after the program has been generated; its only effects are to prevent variables from being fulfilled. Second, the values of all fulfilled variables are *consistent*, meaning all constraints using only fulfilled variables are satisfied. And, third, if all methods of the property model are referentially transparent *upon success*—meaning they may fail nondeterministically, but, if they succeed, their outputs are completely determined by their inputs—then in every possible outcome of the program in which a variable is fulfilled, it has the same value.

Although forever-pending serves well as an error state for the property model, it is not very useful to users. When a method fails, a GUI should notify the user to stop waiting for a value and start taking action to correct the error. To enable error 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 require a new value before they can be used again. To help identify stale variables, 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 7.

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.

5. Implementation

Our reference implementation of property models [12] is called *HotDrink*. HotDrink is written in TypeScript, a statically typed variant of JavaScript, and compiled to a JavaScript library suitable for use in a web application. This stand-alone library allows programmers to implement a GUI's View-Model using HotDrink while still using other JavaScript libraries to implement the View.

Here we illustrate HotDrink usage by implementing the shipping form example. We begin with method implementation. We

```
1  var context = new hd.ContextBuilder()
2    .vs("d, c, m, p, v, w, x, y, z",
3        {x: 25, y: 50, z: 40 w: 10, d: 1500 })
4    .c("v, x, y, z")
5      .m("!x, !y, !z, v → x, y, z", J)
6      .m("x, y, z → v", D)
7    .c("v, w, c")
8      .m("c → v, w", E)
9      .m("v, w → c", F)
10   .c("d, p, m, c")
11     .m("d, m → c, p", G)
12     .m("c, m → d, p", H)
13     .m("d, c → m, p", I)
14   .end();
15
16 var pm = new hd.PropertyModel();
17 pm.addComponent(context);
18 pm.update();
19
20 window.addEventListener("ready", function() {
21   hd.performDeclaredBindings(
22     context, document.body); });
```

Figure 10: JavaScript that creates the property model for the shipping form example with HotDrink.

use the method definitions given in Section 2.1.1, except that we exchange the constraint $\{A, B, C, D\}$ with the constraint $\{D, J\}$ from Section 4.1 in order to illustrate the use of prior values. Figure 9 shows the implementation of three of the methods of the constraint system; the remaining four are similar. The names of the functions match the methods that they implement.

As explained in Section 3.2, each method requires a function whose inputs and outputs are promises. However, HotDrink uses a "lifting" mechanism which can convert a function over values into a function over promises. This allows methods to be written as functions whose inputs and outputs are values; for example, method D on line 1 is written as a function whose inputs and output are numbers. Line 3 shows method J which takes prior values and returns multiple outputs as an array.

Our lifting mechanism works for functions whose inputs and outputs are any mixture of values and promises. The function F on line 7, 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. We assume it does this with an Ajax call, then invokes the provided callback with the result. The callback fulfills the output promise, thus completing the method's duties.

Next, we define the property model's variables and constraints. These are created as fields in a special object called a *context*. A context serves both as a component of the property model and as a namespace, mapping names to variables and constraints. All elements of a property model can be created directly using HotDrink's API; in this example, however, we create them using a ContextBuilder factory object. This temporary object is created in line 1. Its member functions are invoked on lines 2–13. The call to end member function on line 14 finalizes the construction.

ContextBuilder's members define an embedded-DSL for creating elements of a property model. The vs function creates the variables on line 2; the parameters are the variable names and a map of initial values. Variables are initialized in the order declared, which gives the initial priority ordering of stay constraints. To create a constraint, c is first called to establish its variables, then repeated calls to m define its methods. Creating a method requires a signature to define the inputs and outputs, and the method's function.

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

Figure 11: Example HTML containing binding declarations for the shipping form example.

Once the context is created, it is added to the property model, as shown on line 17. We then call update to enforce all constraints, creating the first generation of the reactive program. Finally, we bind the property model's variables to the View. In a web application, elements of the view are not available until the Document Object Model (DOM), has been built. Thus, line 20 registers a function for callback when the DOM is ready. This function initiates binding by calling performDeclaredBindings with two arguments: a context 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 context to look up any names it encounters.

To illustrate binding specifications, Figure 11 shows excerpts of the HTML that creates 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 11. The call to value on line 2 specifies that the variable c should be bound to the value of the select box; the call to num on line 5 that the variable v should be bound to the value of the text box, and that 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 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; and the call to text on line 12 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 for this, we reuse the num binding, but bind to a construction, created by the rw function, that will read from p and write to m.

The code in Figures 9, 10, and 11, along with the method definitions and HTML that were not shown, are a complete implementation of the shipping form example using HotDrink.

6. 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 [29], Garnet [21], and Amulet [23]. These frameworks primarily focus on visual tasks such as component layout and

graphics. More recently, the Subtext [7] 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 dataflow constraints to their toolkits—frameworks such as OpenLaszlo [25], Flex [1], and JavaFX [16]. Generally these constraints are not used for maintaining relations between arbitrary variables in the program, but rather for propagating changes in data to the View—the task we refer to as *binding*. The programmer binds an expression to the View, and, as variables in that expression change, the value of the expression is recalculated and the View is updated with the result.

In between are many toolkits and frameworks providing dataflow constraints, but intended to be used in combination with some other GUI framework to provide traditional GUI functionality. Many of these, such as Knockout [17], ConstraintJS [24], and Facebook's React [27], also focus on binding. These frameworks are largely complementary to ours, and could potentially be used to propagate changes in a property model to the View. Other frameworks tend to focus more broadly on reactive dataflow within a program. For example, Microsoft's Reactive Extensions (Rx) [20] can translate changes in variables to automatic queries using LINQ. Babelsberg [9] integrates constraints with the Ruby language.

Functional Reactive Programming (FRP) [8, 33] 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 [28], Fruit [5], and Yampa [6] 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 [2] brings FRP to JavaScript, and Frappé [4] 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 above systems primarily 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 [30] 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. [13, 14] 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 [26], 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 ensuring that each variable reflects the most current known value, and that computations made irrelevant by more recent results are unscheduled.

```
var context = new hd.ContextBuilder()
  .vs("q, m, i, s")
  .c("q, m")
    .m("q → m", function(q) {
      var p = new hd.Promise();
      performQuery(q,
        function(m) { p.resolve(m); });
      return p; })
  .c("q, m, i, s")
    .m("q, m, i → s", function(q, m, i) {
      return (i >= 0) ? m[i] : q; })
  .c("q ⇒ m, i")
    .m("m, !s → i", function(m, c) {
      var i = m.indexOf(c);
      return (i >= 0) ? i : -1; })
  .cmd("inc", "!m, !i → i", function(m, i) {
    return (i < m.length-1) ? i+1 : i; })
  .cmd("dec", "!i → i", function(i) {
    return (i>0) ? i-1 : i; })
  .end();
```

Figure 12: JavaScript code that creates a property model component that defines the behavior of an auto-complete text box.

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. This would be similar to specifying that a method can execute before all of its input promises have been resolved. This prevents the lengthy calculation of a single dependency from blocking the flow of computation. We could achieve this effect in a property model, by creating two variables—one for the output of the lengthy calculation, and one for the input of the method—and then binding the first to the second; in this way, any time the first variable is modified, its value would be copied to the second.

7. Conclusion

Graphical User Interfaces are costly to develop, and difficult to implement correctly. We are used to encountering errors of various severity in almost all applications that have non-trivial GUIs. Especially tricky to get right in the still dominant event-driven programming model is the asynchronous execution of event handlers; two sequences of the same, but differently timed, events often produce different results in many GUIs, against the programmer's intent.

This paper advocates for an alternative to event-driven GUI programming. In our declarative property models approach, a GUI implementation consists of a specification of data and its dependencies in the GUI, and how that data is reflected by the GUI. From these specifications, the property model dynamically generates a reactive program tailored to any given sequence of edits. This program executes asynchronously and updates the GUI as it progresses. Despite the many possible interleavings of events and computations, the generated reactive program guarantees consistent results for every sequence of editing operations, even when some computations fail. The programmer is no longer responsible for the error-prone task of coordinating asynchronous event handlers.

We began this article with an example of inconsistent behavior exhibited by the auto-complete text box. We finish with an implementation of this GUI element as a component of a property model, shown in Figure 12. The variables q, m, i, and s represent the query parameter, menu, selected index, and input string, respectively, and the three constraints define the data dependencies shown in Figure 1b. We assume the function performQuery performs the asynchronous query to retrieve a list of potential input strings, which it

then passes as an array to the provided callback function. This array becomes the contents of variable m, which is bound to the contents of the menu below the text box.

This code uses two property model features not discussed in this article. First, the constraint declaration "q ⇒ m, i", which indicates an optional constraint whose priority in the constraint hierarchy is linked to the stay constraint for q; thus, the constraint will be enforced only when q has been edited more recently than m or i. Second, the cmd function, which defines a *command*—a computation using variables of the property model whose outputs are treated as an edit. We may use commands to schedule an atomic update of the property model. Here, the inc command increments the selected index, while the dec command decrements it. These commands may be bound, e.g., to keystrokes to enable manipulation of the menu.

This code concisely captures all the dependencies of the autocomplete text box, and it is much easier to understand and verify than the collection of event handlers that implement the same functionality in the event-driven programming model. Furthermore, this code is a reusable component: multiple instances of this context can be used in the same property model, each bound to a different autocomplete text box; it can, of course, also be reused in other GUIs.

Acknowledgments

We are grateful to Prof. Magne Haveraaen for his insightful comments on this work. The work was supported in part by NSF grant CCF-1320092.

References

[1] Adobe Flex. Free, open-source framework | Adobe Flex, Accessed June 2015. URL www.adobe.com/products/flex.html.

[2] Bacon. Bacon.js : Home, Accessed June 2015. URL baconjs.github.io.

[3] A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, and M. Woolf. Constraint hierarchies. *SIGPLAN Not.*, 22(12):48–60, Dec. 1987. doi: 10.1145/38807.38812.

[4] A. Courtney. Frappé: Functional reactive programming in Java. In *Third International Symposium on Pratical Aspects of Declarative Languages (PADL)*, March 2001.

[5] A. Courtney and C. Elliott. Genuinely functional user interfaces. *Haskell Workshop*, pages 1–29, 2001. URL webdoc.sub.gwdg.de/ebook/serien/ah/UU-CS/2001-62.pdf#page=47.

[6] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Proceedings of the ACM SIGPLAN workshop on Haskell — Haskell '03*, pages 7–18, New York, NY, USA, Aug. 2003. ACM Press. doi: 10.1145/871895.871897.

[7] J. Edwards. Subtext. *ACM SIGPLAN Notices*, 40(10):505, Oct. 2005. ISSN 03621340. doi: 10.1145/1103845.1094851.

[8] C. Elliott and P. Hudak. Functional reactive animation. *ACM SIGPLAN Notices*, 32(8):263–273, Aug. 1997. doi: 10.1145/258949.258973.

[9] T. Felgentreff, A. Borning, and R. Hirschfeld. Specifying and solving constraints on object behavior. *Journal of Object Technology*, 13(4): 1:1–38, Sept. 2014. doi: 10.5381/jot.2014.13.4.a1.

[10] M. Fowler. Presentation Model Pattern, 2004. URL martinfowler.com/eaaDev/PresentationModel.html.

[11] J. Gossman. Introduction to Model/View/ViewModel pattern for building WPF apps, October 2005. URL blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx.

[12] HotDrink. Hotdrink, 2015. URL github.com/HotDrink/hotdrink. A JavaScript library for user interface programming.

[13] J. Järvi, M. Marcus, S. Parent, J. Freeman, and J. N. Smith. Property models: from incidental algorithms to reusable components. In *GPCE'08: Proc. of the 7th Int. Conf. on Generative programming and Component Engineering*, pages 89–98, New York, NY, USA, 2008. doi: 10.1145/1449913.1449927.

[14] J. Järvi, M. Marcus, S. Parent, J. Freeman, and J. N. Smith. Algorithms for user interfaces. In *GPCE'09: Proc. of the 8th Int. Conf. on Generative programming and Component Engineering*, pages 147–156, New York, NY, USA, 2009. ACM. doi: 10.1145/1621607.1621630.

[15] J. Järvi, G. Foust, and M. Haveraaen. Specializing planners for hierarchical multi-way dataflow constraint systems. In *Proc. of the 2014 Int. Conf. on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 1–10, New York, NY, USA, 2014. ACM. doi: 10.1145/2658761.2658762.

[16] JavaFX. JavaFX — Oracle Documentation, Acessed June 2015. URL docs.oracle.com/javafx.

[17] Knockout. Knockout : Home, Accessed June 2015. URL knockoutjs.com.

[18] G. E. Krasner and S. 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.

[19] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *ACM SIGPLAN Notices*, 23(7):260–267, July 1988. doi: 10.1145/960116.54016.

[20] Microsoft Rx. Reactive Extensions, Accessed June 2015. URL msdn.microsoft.com/en-us/data/gg577609.

[21] B. Myers, D. Giuse, R. Dannenberg, B. Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85, Nov. 1990.

[22] B. A. Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proc. of the 4th annual ACM symp. on User Interface Software and Technology*, UIST '91, pages 211–220, New York, NY, USA, 1991. ACM. doi: 10.1145/120782.120805.

[23] B. A. Myers, R. G. McDaniel, R. C. Miller, A. S. Ferrency, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The Amulet environment: New models for effective user interface software development. *Software Engineering*, 23(6):347–365, 1997.

[24] S. Oney, B. Myers, and J. 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. doi: 10.1145/2380116.2380146.

[25] OpenLaszlo. OpenLaszlo | the premier platform for rich internet applications, Accessed June 2015. URL www.openlaszlo.org.

[26] J. Peterson, V. Trifonov, and A. Serjantov. Parallel functional reactive programming. In E. Pontelli and V. Santos Costa, editors, *Practical Aspects of Declarative Languages*, volume 1753 of *Lecture Notes in Computer Science*, pages 16–31. Springer Berlin Heidelberg, 2000. doi: 10.1007/3-540-46584-7_2.

[27] React. React: A JavaScript library for building user interfaces, Accessed June 2015. URL facebook.github.io/react.

[28] M. 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, Sept. 2000. ACM Press. doi: 10.1145/351240.351250.

[29] M. 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.

[30] M. Sannella and A. H. Borning. Multi-Garnet: Integrating multi-way constraints with garnet. Technical Report 92-07-01, University of Washington, Department of Computer Science and Engineering, 1992.

[31] G. Trombettoni and B. Neveu. Computational complexity of multi-way, dataflow constraint problems. In *IJCAI (1)*, pages 358–365, 1997.

[32] B. Vander Zanden. An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Trans. Program. Lang. Syst.*, 18(1):30–72, Jan. 1996. doi: 10.1145/225540.225543.

[33] Z. Wan and P. Hudak. Functional reactive programming from first principles. *ACM SIGPLAN Notices*, 35(5):242–252, May 2000. doi: 10.1145/358438.349331.