TOWARDS LOW-COST FEATURE-RICH WEB USER INTERFACES

A Thesis

by

WONSEOK KIM

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2011

Major Subject: Computer Science

TOWARDS LOW-COST FEATURE-RICH WEB USER INTERFACES

A Thesis

by

WONSEOK KIM

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2011

Major Subject: Computer Science

ABSTRACT

Towards Low-Cost Feature-Rich Web User Interfaces. (December 2011)

Wonseok Kim, B.S., Seoul National University

Chair of Advisory Committee: Dr. Jaakko Järvi

Web-based user interfaces are used widely. They are replacing conventional desktop-based user interfaces in many domains and are emerging as front-ends for online businesses. The technologies for web user interfaces have advanced considerably to support high-quality user interfaces. However, the usability of web interfaces continues to be an issue. We still encounter web forms where basic interactive features are missing or work unexpectedly. User interface is a costly and error-prone area of software construction. This is particularly true for web user interfaces. They are typically implemented with fewer reusable components on programmers' toolboxes than conventional user interfaces built using user interface frameworks such as Windows Forms, Cocoa, and Qt. Consequently, web interface programmers tend to struggle with low productivity, or low quality and high defect rates. This thesis focuses on *property models*, a declarative approach to programming user interfaces. In this approach, common user interface behaviors are automatically derived from the specifications of the data manipulated by user interfaces. The approach aims to reuse user interface algorithms that are common across interfaces and allow the programmers to focus on application-specific concerns. This thesis work is a part of project "hotdrink," a JavaScript implementation of the property model system, which has the goal of providing the benefits of property models for web interfaces. This thesis builds on previous work on property models, and adds to it three reusable help and convenience features, which can be especially useful for web forms. In particular, this thesis describes the generic mechanisms of the following user interface features: (1) validating data coming from a user and presenting useful messages that help the user to fix errors, (2) controlling the flow of data through “pinning,” and (3) canceling the user’s previous actions through undoing. The main contributions of the thesis are the mechanisms and the software architecture that enable implementing these behaviors in a reusable manner. This thesis also presents several examples to illustrate the benefits of the proposed mechanisms.

To my lovely wife: Suja Kim

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Jaakko Järvi. In addition to providing his insightful idea for my research, he has given me thorough reviews of this thesis and valuable feedback for improving this thesis. He has also been very understanding and patient with my struggles along the way. Without his patient guidance this thesis would not be possible.

I would also like to thank Dr. Frank Shipman and Dr. Byung-Jun Yoon for giving me valuable comments as my committee members.

I take this opportunity to thank John Freeman and Xiaolong Tang. I built my knowledge on my research topic with help from John. The discussions with him helped me form the technical foundation of my research and proceed to the goal of this thesis. Xiaolong also gave me technical directions when I was implementing the parsers for the languages used in the research.

Finally, I would like to gratefully acknowledge my wife, Suja Kim, for her support and sacrifice. I would not have been able to start pursuing a Master's degree and successfully finishing it without her continuous encouragement and support. I feel blessed to have the most amazing spouse of my life. Also, I thank my son, Sunghoon. He has been a source of joy that has kept me going through tough times.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Web-based user interfaces are in wide use. They are replacing conventional desktop-based user interfaces in many domains and are emerging as front-ends for online businesses. The technologies for web user interfaces (WUI) have advanced considerably to support high-quality user interfaces despite the limitations stemming from their document-centric orientation on layouts and event handling. However, the usability of a web interface continues to be an issue [1]. We still encounter web forms where basic interactive features, such as validating inputs and canceling the last action, are missing or work unexpectedly.

User interface is a costly and error-prone area of software construction [2]. Software reuse is one way to address the issue as reuse has become recognized as a key in many areas to improve productivity [3]. Although user interface programmers usually reuse widgets, such as textboxes, buttons, and dialogs, they hardly reuse algorithms that implement certain user interface behaviors which handle events and manipulate data and visual components accordingly. Those algorithms are usually scattered across event handlers, and they are too specialized for specific user interfaces to be reused. The complexity of them is often high and the size of them is not ignorable. It is, thus, not surprising that event handling code has been reported to account for 30–50% of applications' code [2, 4] and generate a high portion of all known defects [4]. Understanding the ad-hoc event handling code also imposes a significant maintenance burden, up to 50–90% of maintenance cost [3].

Though the above cited studies were conducted in the context of constructing conventional user interfaces, the observations apply to web-based user interfaces as well, possibly even more. Conventional user interfaces, typically built using graphical user interface

The journal model is Science of Computer Programming.

(GUI) frameworks such as Windows Forms, Cocoa, and Qt [5, 6, 7], have a large number of reusable components in their toolboxes with numerous resources helping the programmers to implement common user interface features. Web interface programmers, on the other hand, have relatively fewer reusable components at their disposal. Thus, more effort is required to implement even common user interface features. Adding them repetitively across user interfaces with little reuse is also laborious. Consequently, web interface programmers tend to struggle with low productivity, or low quality and high defect rates.

For example, even a seemingly simple interactive web form, such as a user registration form, may require complex program logic to be expressed in event handlers. Features expected of a modern high-quality user interface might include validating the length of the suggested password as the user is typing it in, reporting precise reasons of user errors and suggesting possible fixes, automatically updating some parts of the address when a zip code is entered, disabling irrelevant widgets depending on the entered country, and activating a submit button only if data in the user interface widgets satisfies a set of validity criteria.

Web programmers have to struggle with the complexity to encode these functionalities within event handling code, usually in JavaScript. Although similar user registration forms with these functionalities are quite commonly used, web programmers seldom reuse the code of these behaviors across web sites. The logic for the behaviors can be formed into some algorithm, but it is not generic enough to be reused.

To reduce the effort to build high-quality feature-rich user interfaces, Järvi et al. have introduced *property models*, a declarative approach to programming user interfaces [8, 9]. In this approach, common user interface behaviors are derived from the specifications of the data manipulated by user interfaces. The approach aims to reuse user interface algorithms that are common across interfaces and allow the programmers to focus more on application-specific concerns. The authors have shown how ad-hoc event handling code can be replaced with library routines that codify user interface features as reusable algorithms. In particular, they have demonstrated reusable mechanisms for propagating values between user interface elements, recording the user's intentions, enabling and disabling of user interface widgets, and activating and deactivating of widgets that launch commands.

This thesis builds on the above work on property models and adds to it three reusable help and convenience features, which can be especially useful for web forms. This thesis work is a part of project "hotdrink" [10], a JavaScript implementation of the property model system, which has the goal of providing the benefits of property models for web interfaces. In particular, this thesis describes the generic mechanisms of the following user interface features:

1. validating data coming from a user and generating useful messages that help the user to fix errors;
2. controlling the flow of data through "pinning"; and
3. canceling the user's previous actions through "undoing."

This thesis expands the portfolio of user interface algorithms that can be packaged into reusable software components using the property models approach. We conjecture that providing the behaviors at little cost will increase the richness and usability of web user interfaces.

The main contributions of the thesis are the mechanisms and the software architecture that enable implementing these behaviors in a reusable manner, applicable to a large number of user interfaces. These behaviors build on the property models approach, in which the data that a user interface manipulates and the relationships between the data are modeled as a dataflow constraint system. The underlying constraint system is the key for realizing the reusable behaviors.

This thesis is organized as follows: Chapter II explains the background of the property models approach. Chapter III gives the details about validating user input and generating useful error diagnostic messages based on the state of the property model constraint system. Chapter IV describes the generic mechanism of pinning that can be used for controlling the flow of data in a web user interface. Chapter V presents the generic mechanism of undo in dialogs and forms. Chapter VI evaluates the work presented in the thesis. Chapter VII concludes the thesis and summarizes its contributions.

CHAPTER II

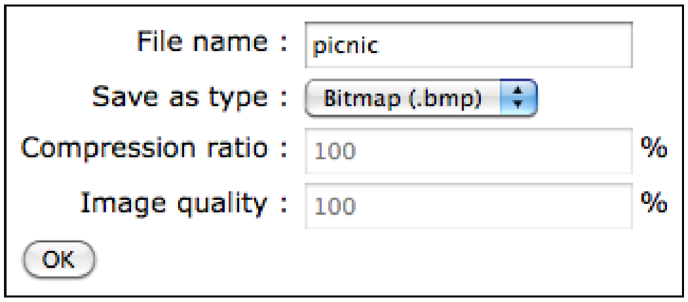
BACKGROUND

This chapter presents the relevant background of property models, paraphrased from the presentation in the earlier papers on property models [8, 9]. We first give the "programmer's view" to the property models approach, describing what specifications need to be written and how they are used to give rise to a user interface. Next, we explain the *property model constraint system*, which is the foundation that enables generic user interface algorithms. Finally, we briefly discuss some reusable user interface algorithms presented in the earlier papers [8, 9], and related work.

A. Property model systems: a programmer's view

In the property models system, a user interface and its complete behavior is derived from three specifications: the model of the data manipulated through the user interface, the visual elements and their layout, and the connections between the visual elements and the data. We call these three specifications the *property model*, *layout*, and *bindings*, respectively.

- A property model is essentially a constraint system with a set of variables and a set of relations that exist between variables. We provide a declarative domain-specific language (DSL) for specifying property models.
- A layout is a set of visual user interface elements, i.e., widgets, provided by a graphical user interface (GUI) library and their positions. We provide a simple declarative language for specifying layouts, but a layout can be expressed in any other layout language, such as HTML, or through library APIs.
- Bindings connect one or more widgets with one or more variables in the property



File name : picnic

Save as type : Bitmap (.bmp)

Compression ratio : 100 %

Image quality : 100 %

Fig. 1. A web form for saving an image file.

model. In our current system, the specifications of bindings are embedded in the layout specification.

Consider a simple web form for saving an image file, as shown in Figure 1. It consists of a text field for entering a file name, a menu of file types, and two interrelated text fields for the user to configure compression. The two compression-related text fields are tied together by some relationship expressing the trade-off between compression ratio and image quality, each on a scale from 1–100%. The property model for this form is shown in Figure 2, and the layout and bindings are in Figure 3.

The specification of a property model is written in the domain specific declarative language *Adam* [10, 11]. The `interface`, `logic`, `output`, and `invariant` sections, shown in Figure 2, declare the variables, or *properties*, and relations among variables of the property model. *Interface variables* in the `interface` section are usually bound to widgets and manipulated by a user. The `logic` section defines the dependencies and computational rules between variables. These computations, we call them *methods*, are expressed in what we call the *expression language*. JavaScript calls are valid subexpression of the expression language. Thus, full expression power of JavaScript is available. The task of the application programmer is to define the computations, but when and which of them are executed is controlled by the property model system. *Output variables* in the `output` section represent

```
model save_image_file {
  interface: {
    file_name : ””;
    file_type : ”bmp”;
    compression_ratio : 100;
    image_quality : 100;
  }
  logic: {
    relate {
      compression_ratio <== 100 − 4 ∗ (100 − image_quality);
      image_quality <== 100 − (100 − compression_ratio) / 4;
    }
  }
  output: {
    result <== (file_type == ”jpeg”) ?
      { type: file_type, name: file_name, ratio: compression_ratio } :
      { type: file_type, name: file_name };
  }
  invariant: {
    check_name <== file_name != ””;
  }
}
```

Fig. 2. The property model specification for the form in Figure 1. A property model definition consists of four *sections*— interface, logic, output, and invariant sections. The interface section declares interface variables with initial values. Interface variables are bound to widgets, and thus they can be manipulated by a user. The logic section defines one constraint representing a relation between compression_ratio and image_quality. Each method in the constraint gives one possible dataflow satisfying the constraint. Here, data can flow either from image_quality to compression_ratio, or compression_ratio to image_quality. The output section has an output variable, result, which has a conditional expression. The output variable collects compression_ratio value only if the file_type is “jpeg.” The invariant section defines conditions that should hold true for all valuations of the variables; a false invariant is an indication that the model is in an invalid state.

```
 1  view {
 2    text (label : ”File name”, value : file_name);
 3    dropdown (
 4      label : ”Save as type”,
 5      items : [
 6        { name : ”Bitmap (.bmp)”, value : ”bmp” },
 7        { name : ”JPEG (.jpeg)”, value : ”jpeg” }
 8      ],
 9      value : file_type
10    );
11    number (
12      label : ”Compression ratio”,
13      units : ”%”,
14      value : compression_ratio
15    );
16    number (
17      label : ”Image quality”,
18      units : ”%”,
19      value : image_quality
20    );
21    commandButton (label : ”OK”, value : result);
22  }
```

Fig. 3. The layout and bindings specifications for the form in Figure 1. The view contains widget specifications, such as text and dropdown. Widgets that the specifications represent will be displayed on the screen in the order their specifications appear in the view. Each widget specification defines the attributes of its widget using name/value pairs. The text widget specification in line 2 defines the “file name” text field and its label. The value attribute in this definition designates a variable of the model specification, here file_name, to which the text field is bound. The dropdown widget specification in lines 3–10 defines the drop-down menu for the “file type.” The items for the menu are supplied as a list in the items attribute. Each item has a name for its descriptive text and a value that will be passed to the bound variable of the drop-down widget when the item is selected. The number widget specification in lines 11–15 defines the “compression ratio” text field. The units attribute defines the text to be displayed on the right of the text field. Line 21 defines the “OK” button. The button is bound to the result variable of the model specification, so when the button is clicked, the value of result will be supplied to the on-click event handler of the button.

the result of the form. They contain the values to be supplied to a command of the form. The (Boolean) value of an *invariant variable* in the invariant section indicates whether a set of variables satisfies a stated condition.

In addition, a complementary language, named *Eve*, can be used to specify the layout and presentation qualities of interface elements, as well as bindings between these elements and variables in the property model. The layout specification for the form in Figure 1 appears in Figure 3. Further details on the Adam and Eve languages are given in their documentation [10, 12].

To launch a user interface, the programmer needs to pass the property model and layout specifications to a JavaScript open_dialog function provided by the property models library. Omitting some details, such a call looks roughly like this:

```
open_dialog(model, layout, initial_values);
```

Initial values in our system are dictionaries of labeled values, and they are used to initialize the variables in the property model.

B. Property model constraint system

A property model encapsulates the values of a set of variables manipulated by a user interface, defines the functional dependencies among them, and manages the application of those dependencies. We can represent the network of dependencies as a *hierarchical multi-way dataflow constraint system* [13]. More concretely, in the property model constraint system, a property model is represented as a graph. The vertices of this *constraint graph* are the variables and methods of a property model and the edges are input or output connections between variable and method vertices. For a particular state of a user interface, we compute a candidate set for the currently active dependencies: this set is represented by a *solution graph*, a subgraph of the constraint graph. Evaluating the methods of the solution graph gives a valuation for the variables in the model. The evaluation yields, as a by-product, a third graph—the *evaluation graph*—that represents the currently active functional dependencies, and is a subgraph of the solution graph. When discussing these graphs, we use the subscripts c, s, and e to denote whether a graph is a constraint, solution, or evaluation graph (e.g., G_c, G_s, and G_e).

The earlier papers on property models [8, 9] described the constraint system representation of property models, including computing solution graphs and evaluation graphs. We summarize the topics here.

1. Multi-way dataflow constraint systems

A multi-way dataflow constraint system is formally represented as a tuple $S = \langle V, C\rangle$, where V is a set of variables and C a set of constraints. Each constraint in C is a tuple $\langle R, r, M\rangle$, where $R \subseteq V$; r is some n-ary relation between variables in R, where $n = |R|$; and M is a set of *constraint satisfaction methods*, or just *methods*. If the values of variables in R satisfy r, we say that the constraint is *satisfied*. A method m in M computes values for some subset of R using another subset of R as inputs, and it satisfies the relation r, i.e., *enforces* the constraint. It is the programmer's responsibility to ensure that the constraint is enforced no matter which of the methods is evaluated.

The constraint satisfaction problem for a constraint system $S = \langle V, C\rangle$ is to find a valuation of the variables in V such that each constraint in C is satisfied. We can find such a valuation by executing exactly one method from each constraint, such that a variable is not written by more than one method, and the methods are executed in an order where a variable is assigned a value before another method uses it as an input. An order of methods satisfying these conditions is called a *plan*.

2. Constraint graph

A multi-way dataflow constraint system can be represented as an *oriented*, *bipartite* graph, the *constraint graph*, $G_c = \langle V + M, E \rangle$, where vertex sets V and M represent the variables and methods of the system, respectively, and E the directed edges that connect each method to its input and output variables. If $v, u \in V$ and $m \in M$, the edge (v, m) indicates that the variable v is an input of the method m, and (m, u) that m outputs to the variable u. We call this type of graph the *constraint graph*. An example of a constraint graph from our image saving model in Figure 2 appears in Figure 4(a).

3. Solution graph

A plan for a constraint system can be explicitly represented as a subgraph of the constraint graph, called a *solution graph*. Let $G_c = \langle V + M, E \rangle$ be a constraint graph and $M' \subseteq M$ the set of methods in the plan. The solution graph of the plan is $G_s = G_c[V + M']$, the vertex-induced subgraph of G_c. A solution graph is acyclic and the in-degree of all variable nodes is at most one. A plan corresponds to a topological ordering of the method nodes of a solution graph. The solution graph for the constraint graph in Figure 4(a) appears in Figure 4(b)

If more than one solution graph exists, we call the constraint system *under-constrained*. To deal with the under-constrained constraint systems that arise from user interfaces we employ *constraint hierarchies* and *stay constraints* [14]. Each variable in the system is given a stay constraint, represented by a rectangle with double frames in Figure 4. A stay constraint consists of a single method (*stay method*) with one output and no inputs—it is thus a constant function. Every time the valuation of a variable changes, the stay method of that variable's stay constraint is constructed anew, so that the constant function has the current value of the variable. Thus, executing a stay method of a variable keeps the variable

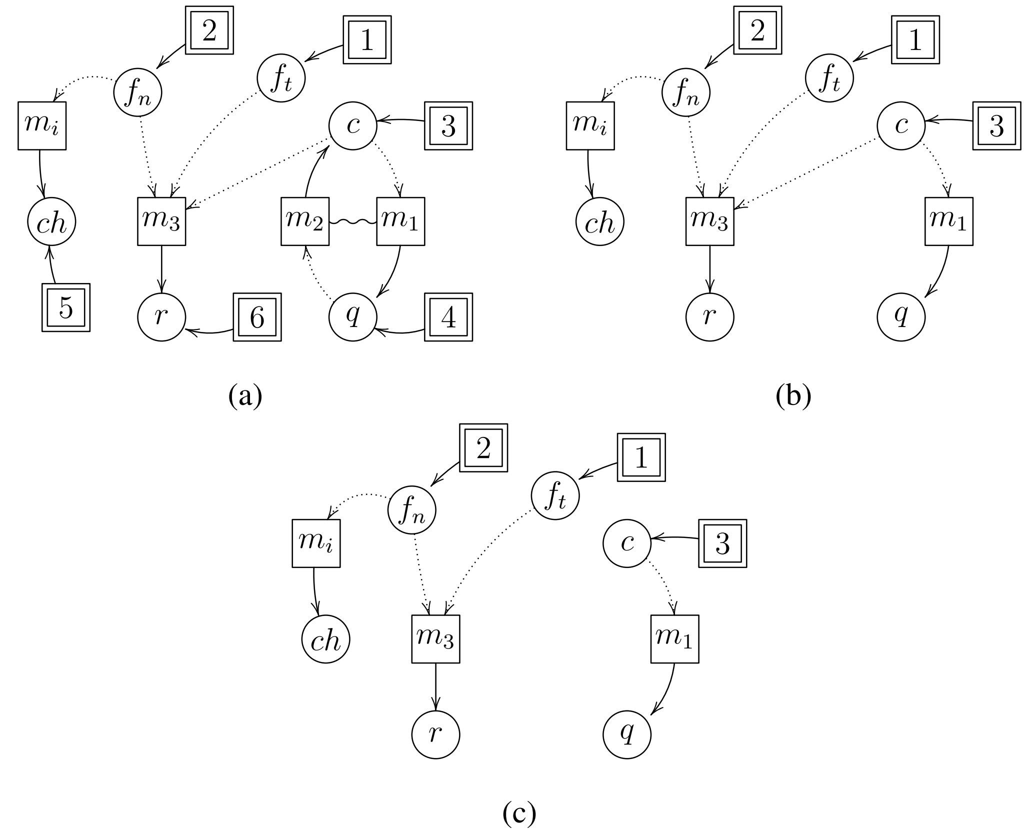


Fig. 4. The constraint graph (a), a solution graph (b), and an evaluation graph (c) for the property model described in Figure 2. In all of the graphs, the variable r is the variable result, q quality, c compression_ratio, f_n file_name, f_t file_type, and ch check_name. The relate clause in the model gives rise to the constraint consisting of the methods m_1 and m_2, and the method in the output section of the model gives rise to the constraint consisting of the method m_3. The constraint graph (a) contains all stay methods (rectangles with double frames), all user-defined methods (rectangles with single frames), and all variables (circles). The stay methods 4, 5, 6, and method m_2 and the edges connected to them are not part of the solution graph (b). The evaluation of method m_3 does not consider the variable c, thus the edge from c to m_3 is *irrelevant*: this edge is not included in the evaluation graph (c).

unchanged.

Since not all stay constraints and user-defined constraints can be satisfied simultaneously, the solution of this over-constrained system is defined as the solution to the "best" satisfiable constraint system that retracts some of the constraints. To decide which constraints to retract, each constraint is assigned a *strength* and the best system is the one that retracts the fewest and weakest constraints. The "locally-predicate-better" criterion [15] defines this precisely: if one solution enforces a constraint that the other does not, and every stronger constraint is either retracted in both solutions or enforced in both solutions, then the former solution is locally-predicate-better than the latter.

We assign the highest strength, we call it *must*, to the user-defined constraints, so that no solution will retract them. We arrange all stay constraints to be weaker than the *must* constraints. Strengths of stay constraints are totally ordered. The ordering is determined by the editing history of user interface elements bound to the variables: stay constraints of the variables bound to the most recently-edited widgets will be strongest, indicating that the values of those variables (and thus the values of the user interface widgets bound to the them) should be preserved. This heuristic approximates the "least surprising" behavior for user interfaces based on property models. We refer to the ordering between variables that a property model maintains as the *priority order*.

A total order of stay constraints guarantees that the solution to the best satisfiable constraint system is unique [16], if there is an admissible solution. We call the unique solution graph of the best system the *most preferred solution graph*.

As explained in [8, 9], we employ a derivative of Zanden's Quickplan algorithm [13] to find the most preferred solution graph for a particular strength assignment. Adapting Zanden's analysis of Quickplan, it can be shown that the worst-case time complexity for finding the most preferred solution graph is $O(n^2)$, where n is the number of constraints in the system [16].

4. Evaluation graph

Executing the methods in the plan according to a topological ordering of the vertices of the solution graph will give the system's variables a valuation that satisfies all its constraints. This "execution phase" produces an evaluation graph that contains exactly the functional dependencies between variables that are active for the current priority order and variable valuation. The evaluation graph may differ from the solution graph because a method may not need all of its inputs in computing its result. We pass input variables to a method by name: to obtain a value of one of its input variables, a method has to explicitly ask for it. Only if a method m asks for the value of its input variable v during execution of the method is the edge (v, m) included in the evaluation graph. We call these edges *relevant* and say that the variable v is relevant to the method m. Assuming a solution graph $G_s = \langle V + M, E_V + E_M \rangle$, where E_V are the edges whose target vertex is in V, and E_M the edges whose target vertex is in M, the evaluation graph G_e is the subgraph of G_s induced by the edges $E_V + E_r$ where $E_r \subseteq E_M$ are the relevant edges. That is, $G_e = \langle V + M, E_V + E_r \rangle$. Figure 4(c) shows an example of an evaluation graph for the solution graph in Figure 4(b).

5. Evaluation

The basic requests to a property model are requesting the value of a variable (get), assigning a new value to a variable (set), and changing the priority of one of the variables. The request for the priority change of a variable is not made explicitly for a property model, but it is made implicitly when assigning a value to the variable. The property model component processes a set query by computing a new valuation of its variables.

Consider a constraint graph $G_c = \langle V + M, E \rangle$, where V and M represent the variables and methods of the system, and E the directed edges. We can define the state of a property model, the *current configuration*, as a tuple $C = \langle G_s, s, \nu \rangle$. G_s is a solution graph of G_c, s is a strength assignment (defines the priority order among the variables), and ν a valuation of variables in V and edges in E. The valuation ν maps a variable to the tuple $\langle t, c, h \rangle$, where t is the current value of the variable; c a flag indicating whether the value of the variable is "computed," i.e., up-to-date; and h a flag indicating whether the value was changed from a previous evaluation round. For clarity, instead of Boolean values, c can have values uncomputed and computed, and h the values unchanged and changed. Further, ν maps all edges $e = (v, m) \in E$ to one of the values relevant or irrelevant. The former signifies that when the code of the method m was executed, the value of the variable v was requested, the latter that it was not. Consequently, ν is overloaded so that the expressions $\nu(v)$ and $\nu(e)$ are both valid. We use the notation $[v \mapsto \mathit{val}]\nu$ for the valuation function identical to ν, except that the variable v maps to *val*; the analogous notation applies for edges.

Assuming a current configuration $C = \langle G_s, s, \nu \rangle$, an invocation of set to assign a new value, say t, for some variable v has the following effect on C:

1. A new strength assignment s' is computed from s, such that the stay constraint of v will become the strongest of the stay constraints, and the relative order of other stay constraints remains the same. Thus, variable v is given the highest priority.

2. Some changes to the strength assignment are such that the most preferred solution graph is known to remain the same. In particular, the solution graph will not change if v is a source in the current solution graph [16]. If necessary, the solver algorithm is run to produce a new solution graph G'_s, otherwise $G'_s = G_s$.

3. A new valuation ν' is computed from ν as follows: the value of v is set to t; the computed-flag of every variable is set to uncomputed; the changed-flag is set to changed for v and to unchanged for all the other variables; and the relevancy-flag

is set to relevant for all edges (v', m') from variables to methods, such that (v', m') is not an edge in G_s but is an edge in G'_s. A method is not executed if it can be seen that its relevant inputs have not changed. The above treatment of relevancy-flags makes sure that all new methods of G'_s that were not included in G_s will be executed during evaluation.

4. The eval function, shown in Figure 5 and described in detail below, is applied to each variable in V. A call to eval may change the valuation, so the current valuation ν' is "threaded through" these calls, producing a new valuation ν''.

The result of the above steps is a new current configuration $\langle G'_s, s', \nu'' \rangle$. As discussed in the previous section, the current evaluation graph G'_e is obtained as the subgraph of G'_s where the edges that ν'' indicates to be irrelevant have been removed. Widgets that are notified that their values might have changed send get requests that consult ν'' to obtain their new values.

In the definition of the eval function we use the following metavariables, using primes, subscripts, and superscripts as appropriate: G_s for solution graphs; V for sets of variables; u and v for variables; m for methods ($\cdot$ is a special value for m that indicates "no method"); t for values of variables; h for values of the changed-flag; ν for valuation functions; μ for mapping a method to the code of the method, and to two sequences of variables indicating the input and output variables of the method; and f for the code of a method. We use the underscore symbol "_" as a variable that binds to anything, similarly to how it is used in, say, Haskell or ML.

Figure 5 defines the function that evaluates a new value of a variable. We use the notation *func* $\mid \nu \to t \mid \nu'$ with the following meaning: the function *func* (either eval or evalmany) is evaluated within the context of the current valuation ν, which produces a new valuation ν' and the result t. The symbol "$\cdot$" indicates that the function has no result.

EVAL-COMPUTED

$$\frac{\nu(v) = \langle t, \mathsf{computed}, \_\rangle \qquad m \neq \cdot}{\mathsf{eval}(v, m, G_s) \mid \nu \to t \mid [(v, m) \mapsto \mathsf{relevant}]\nu}$$

EVAL-COMPUTEDNOMETHOD

$$\frac{\nu(v) = \langle t, \mathsf{computed}, \_\rangle}{\mathsf{eval}(v, \cdot, G_s) \mid \nu \to t \mid \nu}$$

EVAL-INPUTS

$$\frac{\begin{array}{c}\nu(v) = \langle \_, \mathsf{uncomputed}, \_\rangle \qquad \{m'\} = ins_{G_s}(v) \qquad V^{\mathsf{in}} = ins_{G_s}(m') \\ v' \in V^{\mathsf{in}} \qquad \nu(v') = \langle \_, \mathsf{uncomputed}, \_\rangle \qquad \mathsf{evalmany}(V^{\mathsf{in}}, G_s) \mid \nu \to \cdot \mid \nu' \qquad \mathsf{eval}(v, m, G_s) \mid \nu' \to t \mid \nu''\end{array}}{\mathsf{eval}(v, m, G_s) \mid \nu \to t \mid \nu''}$$

EVAL-UNCHANGED

$$\frac{\begin{array}{c}\nu(v) = \langle \_, \mathsf{uncomputed}, \_\rangle \\ \{m'\} = ins_{G_s}(v) \qquad \{v_1^{\mathsf{in}}, \ldots, v_n^{\mathsf{in}}\} = ins_{G_s}(m') \qquad \nu(v_1^{\mathsf{in}}) = \langle \_, \mathsf{computed}, \_\rangle \ \cdots \ \nu(v_n^{\mathsf{in}}) = \langle \_, \mathsf{computed}, \_\rangle \\ \nu((v_1^{\mathsf{in}}, m')) = \mathsf{relevant} \implies \nu(v_1^{\mathsf{in}}) = \langle \_, \_, \mathsf{unchanged}\rangle \ \cdots \ \nu((v_n^{\mathsf{in}}, m')) = \mathsf{relevant} \implies \nu(v_n^{\mathsf{in}}) = \langle \_, \_, \mathsf{unchanged}\rangle \\ \{v_1^{\mathsf{out}}, \ldots, v_l^{\mathsf{out}}, \ldots, v_k^{\mathsf{out}}\} = outs_{G_s}(m') \qquad v = v_l^{\mathsf{out}} \qquad \nu(v_1^{\mathsf{out}}) = \langle t_1, \_, h_1\rangle \ \cdots \ \nu(v_k^{\mathsf{out}}) = \langle t_k, \_, h_k\rangle \\ \nu' = [v_1^{\mathsf{out}} \mapsto \langle t_1, \mathsf{computed}, h_1\rangle, \ldots, v_k^{\mathsf{out}} \mapsto \langle t_k, \mathsf{computed}, h_k\rangle]\nu \qquad \mathsf{eval}(v, m, G_s) \mid \nu' \to t' \mid \nu''\end{array}}{\mathsf{eval}(v, m, G_s) \mid \nu \to t' \mid \nu''}$$

EVAL-CHANGED

$$\frac{\begin{array}{c}\nu(v) = \langle \_, \mathsf{uncomputed}, \_\rangle \qquad \{m'\} = ins_{G_s}(v) \qquad \{v_1^{\mathsf{in}}, \ldots, v_l^{\mathsf{in}}, \ldots, v_n^{\mathsf{in}}\} = ins_{G_s}(m') \\ \nu(v_1^{\mathsf{in}}) = \langle \_, \mathsf{computed}, \_\rangle \ \cdots \ \nu(v_n^{\mathsf{in}}) = \langle \_, \mathsf{computed}, \_\rangle \qquad \nu((v_l^{\mathsf{in}}, m')) = \mathsf{relevant} \qquad \nu(v_l^{\mathsf{in}}) = \langle \_, \_, \mathsf{changed}\rangle \\ \nu' = [(v_1^{\mathsf{in}}, m') \mapsto \mathsf{irrelevant}, \ldots, (v_n^{\mathsf{in}}, m') \mapsto \mathsf{irrelevant}]\nu \qquad \mu(m') = \langle f, (u_1^{\mathsf{in}}, \ldots, u_n^{\mathsf{in}}), (u_1^{\mathsf{out}}, \ldots, u_k^{\mathsf{out}})\rangle \\ f(\nu', \lambda\nu.(\mathsf{eval}(u_1^{\mathsf{in}}, m', G_s) \mid \nu), \ldots, \lambda\nu.(\mathsf{eval}(u_n^{\mathsf{in}}, m', G_s) \mid \nu)) \to (t_1', \ldots, t_k') \mid \nu'' \\ \nu^{(3)} = [u_1^{\mathsf{out}} \mapsto \langle t_1', \mathsf{computed}, \mathsf{changed}\rangle, \ldots, u_k^{\mathsf{out}} \mapsto \langle t_k', \mathsf{computed}, \mathsf{changed}\rangle]\nu'' \qquad \mathsf{eval}(v, m, G_s) \mid \nu^{(3)} \to t' \mid \nu^{(4)}\end{array}}{\mathsf{eval}(v, m, G_s) \mid \nu \to t' \mid \nu^{(4)}}$$

EVALMANY-EMPTY

$$\mathsf{evalmany}(\emptyset, G_s) \mid \nu \to \cdot \mid \nu$$

EVALMANY

$$\frac{\mathsf{eval}(v, \cdot, G_s) \mid \nu \to \cdot \mid \nu' \qquad \mathsf{evalmany}(V', G_s) \mid \nu' \to \cdot \mid \nu''}{\mathsf{evalmany}(\{v\} \sqcup V', G_s) \mid \nu \to \cdot \mid \nu''}$$

Fig. 5. The evaluation rules for obtaining a value of a variable in a property model and effecting the consequent state changes to the variable and edge valuation. We overload the ins_{G_s} and $outs_{G_s}$ functions for both methods and variables, so that they return the sets of incoming and outgoing vertices in G_s.

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

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

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

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

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

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

EVAL-CHANGED matches when all the input variables of m' are computed and at least one input variable of m' has changed. If this is the case, the code of m' needs to be executed. This entails (1) retrieving the code f of m', (2) constructing a callback function for each input variable of f that will obtain the value of the input variable by another call to eval, (3) passing the callbacks to f, and (4) executing f. If the code of a method invokes any of its callbacks, a new call to eval results, where the method requesting the value of a variable is m'. This call will eventually reach EVAL-COMPUTED, which will set the corresponding relevancy-flag of the edge to m', and thus establish one piece of the evaluation graph. Once the method f returns, the values in the tuple it returns are written to the output variables of m' (v is one of them) and the computed and changed-flags of these output variables are set as well.

C. User interface algorithms based on the property model constraint system

The earlier papers [8, 9] on property models introduced three user interface algorithms derived from the property model constraint system. They are *value propagation algorithm*, *command activation algorithm*, and *widget enablement algorithm*.

The value propagation algorithm is naturally achieved through the property model constraint system. As described in earlier sections, solving the constraint system and evaluating the plan find an admissible valuation for the constraint system. This process takes care of propagating values according to functional dependencies, resulting in the value propagation algorithm.

The command activation algorithm deactivates a command when the command's preconditions, which are expressed as invariants, are not satisfied. If a command is bound to an output variable that is reachable from an ancestor of a false invariant in the evaluation graph, the command button is deactivated to prevent using the "poisoned" output variable.

The widget enablement algorithm disables widgets that do not affect any output variable. Disabling widgets that are not relevant to the output prevents a possible confusion of a user by eliminating irrelevant choices. The algorithm checks if there exists a currently active dependency between an interface variable bound to a widget, and some output variable, or if it is possible that a editing the value of the widget could create such a dependency.

The constraint system underneath makes the dependencies among data explicit, and packages them in a well-defined data structure, for which algorithms can be written. We track three kinds of dependencies: those that could be active at some point in the life of a user interface, those that are active for current editing order, and those that are active for current editing order and values. Each of these three sets of dependencies can be represented as a graph. All of these graph representations are necessary for the algorithms described in this section, as well as the new algorithms presented in this thesis.

D. Related work

Constraint systems have been extensively applied to user interfaces, mostly for automated widget layout. A large number of declarative, constraint-based GUI systems have been proposed. Sketchpad [17], Amulet [18], Garnet [19], as well as ThingLab I and II, DeltaBlue, and SkyBlue [20] serve as examples. Regarding web user interfaces, some prior work [21, 22] presents *constraint-based style sheets* which extends cascading style sheets (CSS) to employ constraints for more flexible layout. More recently, XUL [23], XAML [24], Flex [25], and OpenLaszlo [26] support (one-way) dataflow constraints for the layout specifications of web user interfaces.

What distinguishes the property model approach from other earlier uses of constraint systems for user interface programming is that constraints are utilized to give programmers a method to explicitly model functional dependencies among data; this explicit model serves as a basis for generic algorithms for user interface behaviors.

CHAPTER III

VALIDATION OF USER INPUT

A. Introduction

To support users well, a user interface must provide good error messages. However, programming error handling for a user interface is usually laborious, and, perhaps even worse, the resulting code is not reusable. One reason for this is that error handling logic is typically mingled within all other, often widget-specific, event handling code. In particular, the validation of user input, which is a commonly required behavior in dialogs and web forms, increases the complexity of event handling logic. User interface programmers may not extend enough effort to implement the validation logic correctly for all possible error scenarios, and, similarly, they may not invest enough on producing helpful error diagnostics for the user. As a result, we often encounter web interfaces with missing or erroneous support for diagnosing errors.

Property models allow programmers to write validation conditions with error messages in a declarative manner, and the property models system takes care of applying those rules when appropriate, and acting appropriately when the conditions are not satisfied. In this approach, the programmers can focus on the validation conditions without being distracted by event handling code around them. Furthermore, property model systems contain additional analytic information on errors, which can be relayed to the user to help in recovering from an invalid state that resulted from an erroneous input.

B. Validation and error reporting with property models

As explained in [9], property model systems can automatically deactivate a command widget when certain preconditions to the command, launched by the command widget, do not hold. The preconditions for a command are defined by the programmer as invariants. As explained in Section A of Chapter II, invariants are Boolean expressions that encode validation conditions in property models. They can represent either simple validation conditions that check individual interface variables, or complex validation conditions concerned with derived values or relationships between variables.

The basic mechanism that the property model system uses to handle invariants is as follows. When an invariant evaluates to false, the variables that contribute to that violation are marked, or "blamed," for being responsible. If any blamed variable contributes to the command's parameters (i.e., the evaluation graph contains a path from any blamed variable to the output variable bound to the command) the command is deactivated. However, in a high quality user interface, more than deactivating a command, say, by "graying out," is needed. Without being offered a proper reason, a user may not understand why a command button is deactivated. Figure 6 shows a case where a user may be left puzzled over why the "Register" command is grayed out.

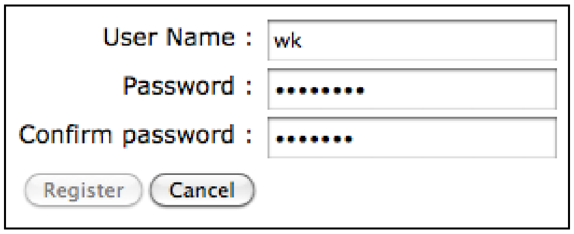
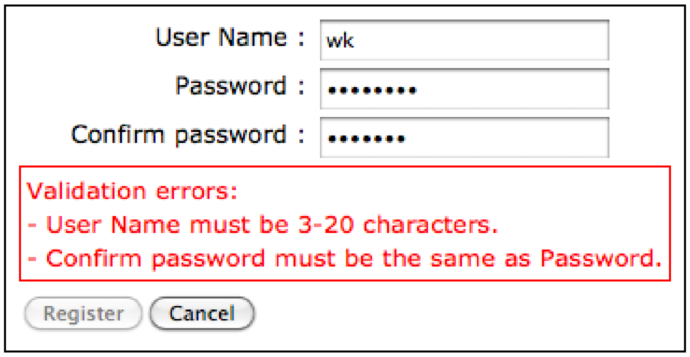

User Name :	wk
Password :	••••••••
Confirm password :	•••••••

Fig. 6. A user registration form that deactivated its command button. The "Register" command button is grayed out because the "user name" does not meet the minimum length and the "confirm password" is different from "password." However, the user may have no idea why the command button is disabled.

We can overcome the limitation by exploiting property models further, i.e., having them include error descriptions that identify the failed invariants in an easily understandable form. The key idea is that programmers provide slightly structured texts to describe invariants, and the property model system takes care of constructing informative error messages that pinpoint the reasons accurately, based on the state of the constraint, solution, and evaluation graphs. More specifically, we allow an invariant to be annotated with a @description annotation that contains a natural language description of the invariant. For example: the following invariant shows the description attached to it.

```
@description(”Confirm password must be the same as Password.”)
password_match <== password1 == password2;
```

As explained above, a failed invariant may lead to deactivating command buttons. To inform the user of the reasons for these actions, the system collects the descriptions of all failed invariants. These descriptions can then be shown to the user, as illustrated in Figure 7.



User Name :	wk
Password :	••••••••
Confirm password :	•••••••

Validation errors:
- User Name must be 3-20 characters.
- Confirm password must be the same as Password.

Fig. 7. A user registration form that displays error messages for invalid inputs.

The description can include references to variables. Prior to displaying the description, each reference is translated to the label of the widget that the referenced variable is bound to. This helps avoiding hard-coding the labels of referenced widgets. Hard-coding the labels may be problematic since they may need to be altered later as the appearance of the form changes.

A variable name is referred to with the syntax of ${variable}. Consider again the description above:

```
    @description(”${password2} must be the same as ${password1}.”)
    password_match <== password1 == password2;
```

When displaying the description to the user, a layout module retrieves the bindings between variables and widgets, and substitutes the labels for the variable names. Figure 8 shows the layout specification that has the labels and bindings of the widgets in Figure 7. The layout module can pick up the corresponding labels for the referred variables from the specification. The full property model specification of the user registration form, including validation conditions, is given in Figure 9.

```
view {
  text (label : ”User Name”, value : username);
  password (label : ”Password”, value : password1);
  password (label : ”Confirm password”, value : password2);
  errors ();
  row {
    commandButton (label : ”Register”, value : result);
    commandButton (label : ”Cancel”);
  }
}
```

Fig. 8. The layout and bindings specifications for the form in Figure 7. The widget that is manipulated by a user can have a label attribute for the label for the widget. The value attribute of a widget declares a binding to a variable of the property model. Based on the bindings between widgets and variables, the layout module can find the corresponding “labels” of the widgets that variables are bound to. The errors widget, which is added in the middle, is a message pane for displaying error messages. Our prototype is using the widget to display all validation error messages in one place.

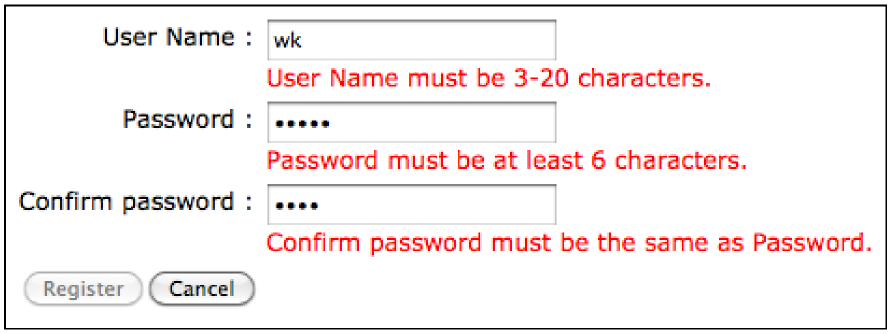
Our prototype displays all error messages in a separate area, as shown in Figure 7. This is one among many possible choices. In practice, web user interfaces employ various presentations for displaying validation error messages, as shown in the showcases for web-form validation [27]. For example, user-registration forms might display error messages

```
model register {
  interface : {
    username;
    password1;
    password2;
  }

  output : {
    result <== {username: username, password: password1};
  }

  invariant : {
    @description("${username} is required.")
    username_required <== NotEmpty(username);

    @description("${password1} is required.")
    password_required <== NotEmpty(password1);

    @description("${username} must be 3-20 characters.")
    username_size <== Size(username, 3, 20);

    @description("${username} must start with alphabet
      and only consists of alphabet and digit.")
    username_pattern <== Pattern(username, "^[a-zA-Z][a-zA-Z0-9]*$");

    @description("${password1} must be at least 6 characters.")
    password_size <== Size(password1, 6);

    @description("${password2} must be the same as ${password1}.")
    password_match <== password1 == password2;
  }
}
```

Fig. 9. The property model specification with invariant descriptions for the form in Figure 6. In the invariant section, the invariants are boolean expressions written using the property model expression language, which can contain calls to JavaScript functions. Here, for example, we make calls to the JavaScript functions such as NotEmpty, Size, and Pattern; these functions provide primitive validations.

below or beside the widgets that have invalid values, or highlight them with red outlines.

The presentation of error messages is orthogonal to the validation of preconditions. The generic validation mechanism of property models described in this section is the same regardless of how the user interface presents error messages. The validation mechanism provides the validation results, a collection of error messages indexed by invariants, to the layout module, and it is the role of the layout module to determine how to present them. To demonstrate the orthogonality of the presentation of error messages, Figure 10 shows another presentation, implemented in our prototype. The experiment with our prototype demonstrates that adopting different visualizations for displaying error messages does not affect the validation mechanism presented in this section.



User Name :	wk
	User Name must be 3-20 characters.
Password :	•••••
	Password must be at least 6 characters.
Confirm password :	••••
	Confirm password must be the same as Password.

Fig. 10. A user registration form that displays each error message below the widget it pertains to.

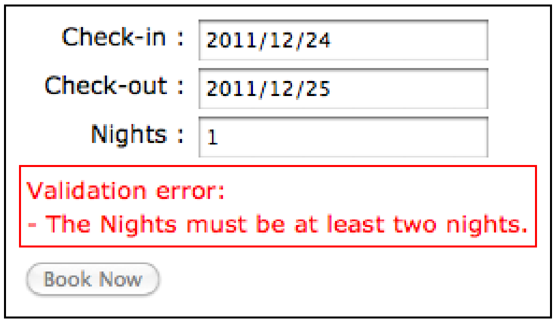
C. Pinpointing the source of an error using property model constraint system

When the validation condition of an invariant has only variables that contain the values directly entered from a user, e.g., the `username` variable of the registration form in Figure 9, the description of the invariant directly identifies the variables, and their values, to blame, in case of a failed invariant. Sometimes, however, an invariant may fail due to variables that the invariant does not refer to directly; other variables that the user alters can contribute to the invariant according to the functional dependencies in the property model. In such a case, the provided error message does not give precise information that pinpoints exactly the values that contribute to the error. The information about the values can be only obtained from the currently active dataflow, which changes dynamically.

For example, the hotel-booking form in Figure 11 displays a validation error complaining about the "nights" value. The invariant of the failed validation, which is from the model in Figure 12, is as follows:

```
@description("The ${nights} must be at least two nights.")
at_least_two_nights <== nights >= 2;
```

Although the invariant fails when the "nights" value is not at least two, the failure may be due to the "check-in" or "check-out" value that the user just entered (from which the "nights" value was computed by the system). In this case, the error message that blames the "nights" value may not be enough for the user to know the cause.


Check-in :	2011/12/24
Check-out :	2011/12/25
Nights :	1

Validation error:
- The Nights must be at least two nights.

Fig. 11. A hotel-booking form with an error message complaining about "Nights" value.

Property model systems can provide information about which variables are *responsible* for a failed invariant. *Responsible variables* are the variables that may affect the invariant according to the current dataflow. This information can help the user to figure out the reason for the error and how to fix it. Figure 13 shows an example how this information could be presented to the user. Hovering the mouse over an error message shows a pop-up

```
model hotel {
  interface: {
   checkin : today();
   nights: 1;
   checkout;
  }

  logic: {
    relate {
      checkout <== add_days(checkin, nights);
      checkin <== remove_days(checkout, nights);
      nights <== day_difference(checkin, checkout);
    }
  }

  invariant: {
    @description(”The ${nights} must be at least two nights.”)
    at_least_two_nights <== nights >= 2;
  }

  output: {
    result <== {checkin: checkin, checkout: checkout};
  }
}
```

Fig. 12. The property model specification for the form in Figure 11.

hint that directs the user to the widgets bound to the responsible variables.

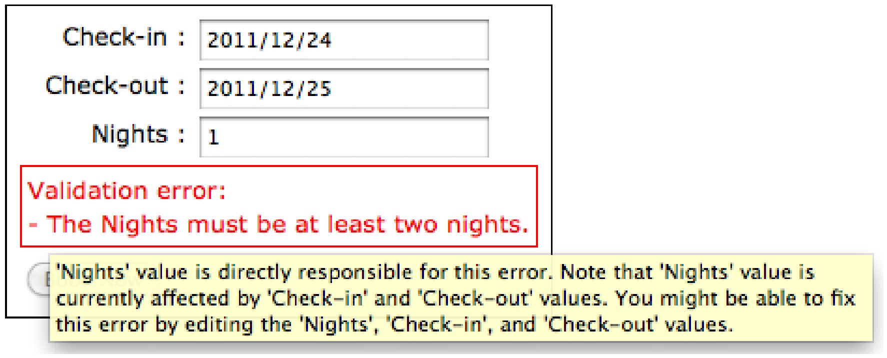

Check-in : 2011/12/24
Check-out : 2011/12/25
Nights : 1
Validation error:
- The Nights must be at least two nights.
'Nights' value is directly responsible for this error. Note that 'Nights' value is currently affected by 'Check-in' and 'Check-out' values. You might be able to fix this error by editing the 'Nights', 'Check-in', and 'Check-out' values.

Fig. 13. A hotel-booking form that shows a generated pop-up hint indicating the responsible values for the error.

Variables can be responsible to a failed invariant in three different ways: *directly responsible variables*, *contributing variables*, and *possibly responsible variables*. These notions can be useful for users as well.

First, *directly responsible variables* are the interface variables that directly appear in the invariant condition. They directly contribute to the value of the invariant. We only consider interface variables that are bound to widgets; it would be meaningless to identify variables that are not visible to the user. In our hotel-booking form's property model, shown in Figure 12, the invariant condition is checking the nights variable, so this variable is directly responsible if the invariant fails. The set of directly responsible variables can be statically determined by looking at the constraint graph. For each invariant, we analyze its method node. The interface variable nodes that are input to the method node are the set of directly responsible variables. For example, the first sentence of the pop-up hint in Figure 13, saying "'Nights' value is directly responsible for this error," presents the directly responsible variable to the user.

Second, *contributing variables* are the interface variables that are used to evaluate the invariant, excluding the directly responsible variables of the invariant. Contributing variables exist if the values of directly responsible variables are derived from the values of other variables according to the currently active dataflow. For example, in the hotel-booking form, the nights value is derived from the checkin and checkout values, if the user edited them more recently than the nights value. In this case, the checkin and checkout variables are the contributing variables to the nights value.

Contributing variables are dynamically determined from the evaluation graph. For each invariant, they are the set of ancestor nodes of the invariant node, excluding the set of the nodes of the directly responsible variables. Figure 13 shows an example that displays the contributing variables in the pop-up hint, saying “Note that ‘Nights’ value is currently affected by ‘Check-in’ and ‘Check-out’ values,” in the second sentence.

Finally, *possibly responsible variables* are the interface variables that may be used to evaluate the invariant. These variables include all interface variables that may affect the validation result; these include directly responsible variables and all the possible set of contributing variables. For example, in the hotel-booking form, the nights, checkin, and checkout variables can possibly contribute to the invariant. Thus, these variables are possibly responsible variables for the invariant. To determine these variables, we look at the constraint graph which has all the possible dataflows. For each invariant, they are the set of interface variable nodes that reach the invariant node. These variables give the user the candidates that can be edited to fix the error. The third sentence of the pop-up hint in Figure 13, saying “You might be able to fix this error by editing the ‘Nights’, ‘Check-in’, and ‘Check-out’ values,” presents the possibly responsible variables to help the user to fix the error.

D. Related work

Preconditions were first used for user interface design by [28] as a part of a formal specification, but they were not used to control widgets or generate help text at runtime. User Interface Design Environment (UIDE) used preconditions to control the enabling of individual menu items [29]. The initial UIDE only allowed a set of predefined expressions on its preconditions. Daniel F. Gieskens and James D. Foley later extended the UIDE to allow arbitrary predicates to control the visibility and enabling of arbitrary widget [30]. Their system supported only one-way dependencies between pieces of data. Other work [31, 32, 33, 34] that extended UIDE used postconditions to generate help text indicating how to enable a disabled widget. Property model systems do not provide the sequence of actions to enable a disabled widget like the earlier work, but they enumerate a list of widgets that can possibly affect the disabled widget.

Bigwig project [35] provides a high-level DSL for building interactive web interfaces. It employs a domain-specific sublanguage, called *PowerForms* [36], for form field validation. PowerForms allows a validation condition to check complex interdependencies between form fields. Our system and PowerForms both provide a client-side validation of web forms by generating JavaScript code from DSLs. PowerForms needs to compile the validation conditions on the server-side, whereas our system works as a client-side library. Another difference is that our system supports a flexible expression language for validation conditions where calls to external JavaScript functions are possible.

E. Conclusion

A considerable effort in web programming is spent on validating user input, that is, checking whether the data supplied by the user is valid, and when it is not, producing error messages to help the user fix the data. Although validation of user input is a common behavior across web user interfaces, it is still not easy for web programmers to reuse the behavior. This chapter explores a generic validation mechanism based on property models. We extend the original property model system to be able to report validation errors when a command button is disabled due to failed validation conditions. In addition, our system generates additional analytic information on validation errors, which includes responsible values that may affect the failed validation conditions.

We have implemented the validation mechanism within hotdrink. Our prototype demonstrates the benefits of reusing the validation behavior. With the prototype, web programmers do not need to add a custom event handling code for validating user data, reporting the errors of the data, or constructing help messages in case of errors.

CHAPTER IV

PINNING

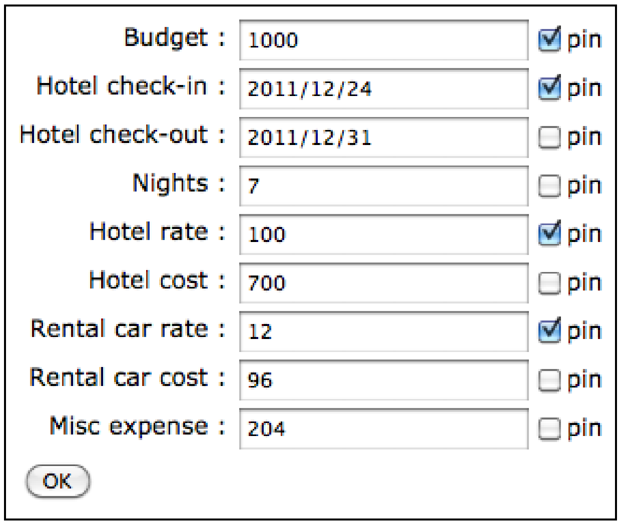
A. Introduction

Property models allow implementing an interactive user interface that supports complex "multi-way" dependencies. A property model system automatically coordinates the flow of data when a user interacts with the interface. The system may change some variables' values to enforce constraints. Selecting a particular dataflow is determined through a heuristic: try to keep the more recent changes from being overwritten. This is a reasonable heuristic and it works well often, but sometimes the system behavior may look unexpected to the user if it does not match the user's intention in a particular situation. In such a case, the user may struggle to figure out the system's behavior.

Pinning provides a user with means to control the preferred dataflow. It allows the user to "lock in" certain values, guaranteeing that the system does not override the values while it maintains the relationships between values. It does not prevent the user from further editing the pinned values; it only affects the flow of data.

Pinning can be offered as a reusable user interface feature. The property model constraint system provides a way to fix a specific variable by only admitting dataflows where the variable's stay constraint is enforced. In this chapter, we introduce a generic pinning mechanism based on the property model constraint system.

Pinning is useful for the user interfaces that have to support multi-way dataflow constraints. Those interfaces can employ the pinning mechanism to provide the control feature. Consider a budget planning form, shown in Figure 14, which provides a flexible interface with several multi-way constraints. Thanks to these multi-way constraints, a user can experiment easily: editing any value will lead to other related values being automatically



Budget :	1000	✓ pin
Hotel check-in :	2011/12/24	✓ pin
Hotel check-out :	2011/12/31	☐ pin
Nights :	7	☐ pin
Hotel rate :	100	✓ pin
Hotel cost :	700	☐ pin
Rental car rate :	12	✓ pin
Rental car cost :	96	☐ pin
Misc expense :	204	☐ pin

Fig. 14. A trip budget planning form with a pinning feature. Clicking on the pin checkbox next to each textbox widget will pin or unpin the value of the widget. The checkbox remains checked if the value is pinned. The interface may reject pinning if the system is not able to enforce the pinning. In this instance, “budget,” “hotel check--in,” “hotel rate,” and “rental car rate” are pinned. Thus, if the user modifies the “nights” value, “hotel check-out” value will be newly computed according to the constraint (1), and “hotel cost” value, according to the constraint (2), “rental car cost” value, according to the constraint (3), and “misc expense” value, according to the constraint (4), will be newly computed as well. These constraints are defined in Figure 15.

```
(1) [nights] = day_diff([hotel check−in], [hotel check−out])
(2) [hotel cost] = [nights] ∗ [hotel rate]
(3) [rental car cost] = [nights] ∗ [rental car rate]
(4) [budget] = [hotel cost] + [rental car cost] + [misc expense]
```

Fig. 15. Multi-way dataflow constraints defined for the data in Figure 14. In this pseudo-code, each line defines an equality relation between the variables which are expressed using square brackets. In the first relation, day_diff is a function that computes the number of days between two dates.

computed to satisfy the constraints shown in Figure 15. When the user updates a value, there may be different dataflows, that is, solution graphs that could be used to satisfy the relations in the constraint system. Without pinning, the property model system chooses a dataflow solely based on the editing history. When there are complex relationships, as is the case here, the default heuristic can lead to surprising results and the user may not be able to predict the dataflow and which values will be overwritten. In this kind of a situation, pinning lets the user control which values should be fixed and which allowed to change by the property model system. Figure 14 shows an example situation where pinning is used to force a particular dataflow.

B. Pinning mechanism

Pinning can be realized as a generic mechanism based on a property model constraint system. In the constraint system, a solution graph provides a candidate dataflow. As explained in Section 3 of Chapter II, the solution graph is determined by the strength assignment of stay constraints. Each enforced stay constraints of a variable in the graph keeps the variable unchanged. Thus, pinning a variable naturally translates to requiring that the variable's stay constraint will be enforced—guaranteeing that the pinned variable is not overwritten by any method.

To make the system always enforce the stay constraints of pinned variables, we need a slightly modified strength assignment scheme. We assign the highest strength, *must*, to all the stay constraints of pinned variables, and we assign lower strengths to the other stay constraints than the must constraints. The stay constraints of variables that are not pinned are ordered according the editing history—the stay constraint of a more recently edited variable is stronger. Even after taking pinning into consideration, the resulting order of all stay constraints is still a total order. Therefore, the most preferred solution graph is guaranteed to be unique [16]. This is important, as other generic algorithms for user interface behaviors, such as a widget enablement algorithm [9], rely on the uniqueness for their correctness.

Adding “must” stay constraints may lead to an over-constrained system that does not have an admissible solution anymore. Thus, pinning should fail if adding a must stay constraint for the pinned variable makes the constraint system unsolvable.

Clearing the pinned status of a variable is done through *unpinning*. Unpinning a pinned variable is achieved through demoting the stay constraint of the variable to a regular (*non-must*) stay constraint. The stay constraint of the just-unpinned variable becomes strongest among unpinned variables. With this heuristic, the stay constraint will still remain enforced. The important consequence is that unpinning a variable does not change the current dataflow. Therefore, when unpinning a variable, the user will not be surprised by a sudden change of dataflow.

We present the pinning mechanism more formally in terms of a state transition. The current configuration, the state of a property model, was described in Section 5 of Chapter II. We add a set of pinned variables to the state, redefining the current configuration to be a tuple $C = \langle G_s, s, p, \nu \rangle$, where G_s is a solution graph, s a strength assignment, p a set of pinned variables, and ν a valuation. We define two requests, pinning and unpinning, to the property model as pin and unpin, respectively.

Assuming a set of variables V and a current configuration $C = \langle G_s, s, p, \nu \rangle$, an invocation of pin to pin a variable v, where $v \notin p$, has the following effect on C:

1. A new pinned set p' is computed, such that $p' \leftarrow p \cup \{v\}$.

2. A strength assignment s' is computed from s, such that the stay constraint of v has a “must” strength, and the relative order of other stay constraints remains the same.

3. A solution graph G'_s is produced by the solver algorithm. In some cases, the change to

the strength assignment leaves the most preferred solution graph the same as G_s [16]. If v is a source in the current solution graph G_s, the solution graph will not change, resulting in $G'_s = G_s$. If the solver algorithm cannot find a solution, pinning fails and the system restores the pinned set from p and the strength assignment from s; and does not proceed further.

4. A new valuation ν' is computed from ν as follows: the computed-flag of every variable is set to uncomputed; the changed-flag of every variable is set to unchanged; and the relevancy-flag is set to relevant for all edges (v', m') from variables to methods, such that (v', m') is not an edge in G'_s but is an edge in G''_s.

5. The eval function, shown in Figure 5, is applied to each variable in V. A call to eval may change the valuation, so the valuation v' produces a new valuation v''.

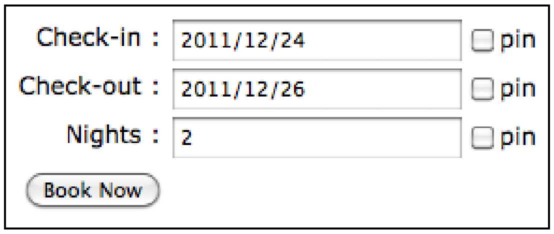
The result of the above procedure is a new current configuration $C' = \langle G'_s, s', p', \nu'' \rangle$. The evaluation (in the step 5) may produce the values of variables identical to the previous state C. However, the evaluation must still run to update the relevancy-flags of variables and to obtain an up-to-date evaluation graph. The correctness of some existing algorithms for user interface behaviors, such as a widget enablement algorithm, relies on the evaluation graph.

Another request to the property model is unpinning. An invocation of unpin to unpin a variable v, where $v \in p$, has the following effect on $C = \langle G_s, s, p, \nu \rangle$:

1. A new pinned set p' is computed, such that $p' \leftarrow p - \{v\}$.

2. A strength assignment s' is computed from s, such that the stay constraint of v becomes the strongest among the non-must stay constraints, and the relative order of other stay constraints remains the same.

The result of the above procedure is a new current configuration $C' = \langle G_s, s', p', \nu \rangle$.



Check-in :	2011/12/24	☐ pin
Check-out :	2011/12/26	☐ pin
Nights :	2	☐ pin

Fig. 16. A hotel-booking form with a pinning feature.

Pinning also affects the set operation, which is a request to assign a new value to a variable and/or change the priority of the variable (the original procedure of set is described in Section 5 of Chapter II). The first step of set has to employ the modified strength assignment scheme that we used for pinning as follows:

1. A strength assignment s' is computed from s, such that the stay constraint of v becomes the strongest among the non-must stay constraints, and the relative order of other stay constraints remains the same. That is, variable v is given the highest priority among the variables that are not in the pinned set p.

C. Example scenario involving pinning

To illustrate the pinning mechanism, we present a simple hotel-booking form, shown in Figure 16. The form has only one user-defined constraint as specified in Figure 17. Consider the following sequence of events: the user enters a "check-in" date, and then enters a "check-out" date; the system automatically updates the "nights" value computed from those two values according to the user-defined constraint in Figure 17. If the user increases the "nights" value thereafter, the "check-in" value will be overwritten by the value computed from the "nights" and "check-out" values. The reason for this is that the system tries to keep values from the most-recent edits, "check-out" value in the above case, overwriting the value of a variable with a lower priority, i.e., "check-in" value. Figure 18(a) shows the

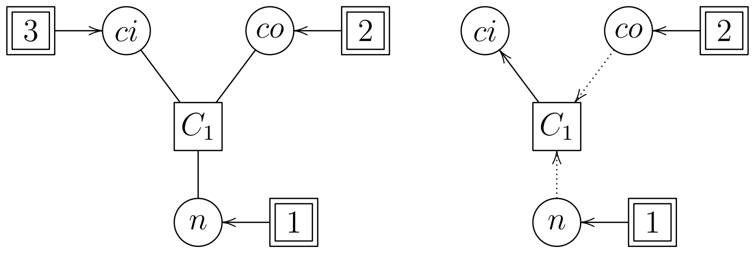
```
model hotel {
  interface: {
   checkin : today();
   nights: 2;
   checkout;
  }

  logic: {
    relate {
      checkout <== add_days(checkin, nights);
      checkin <== remove_days(checkout, nights);
      nights <== day_difference(checkin, checkout);
    }
  }

  output: {
    result <== {checkin: checkin, checkout: checkout};
  }
}
```

Fig. 17. The model specification for the form in Figure 16. One user-defined constraint among checkin, checkout, and nights variables is defined in the model.

constraint graph of the property model with a strength assignment matching this scenario. The solution graph, shown in Figure 18(b), yields the data flow where the “check-in” value is derived from other values.

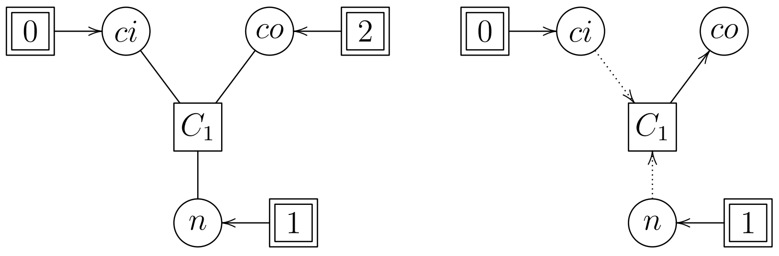
In this scenario, let us assume that the user wants to keep the “check-in” date unchanged when altering the “nights” value. If the user pins the “check-in” value, the system will promote the stay constraint of the “check-in” variable to a must constraint, resulting in a strength assignment shown in Figure 19(a). The new data flow, shown in Figure 19(b), now derives the “check-out” value when the user updates the “nights” value, keeping the original “check-in” value unchanged.

Now if the user unpins the “check-in” value that was pinned earlier, the stay constraint



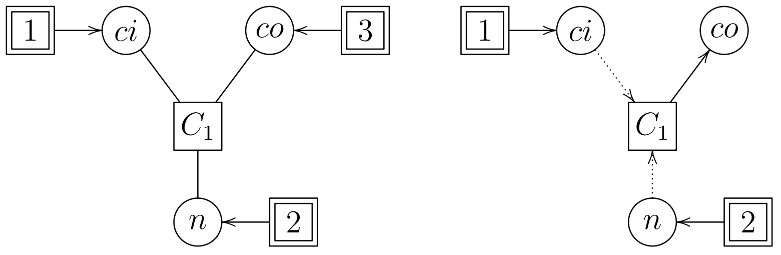
(a) constraint graph (b) solution graph

Fig. 18. The constraint graph and solution graph when the variables are edited in the order: (1) checkin, (2) checkout, and (3) nights. The labels ci, co, and n stand for the checkin, checkout, and nights variables, respectively. C_1 represents the user-defined constraint, defined in Figure 17.



(a) constraint graph (b) solution graph

Fig. 19. The constraint graph and solution graph after pinning ci.



(a) constraint graph (b) solution graph

Fig. 20. The constraint graph and solution graph after unpinning ci.

of the variable will no longer be a must constraint. As we explained earlier, we give the strongest strength to the constraint among unpinned variables, as shown in Figure 20(a). This will result in the solution graph in Figure 20(b), which is the same as the previous solution graph in Figure 19(b). This strength assignment makes the system preserve the previous dataflow, giving the least surprise to the user.

D. Related work

The Heracles system [37] shares the motivation of pinning with us, citing potentially unclear dataflows and inadvertently overwritten values. It provides checkboxes for locking and unlocking certain values, which are equivalent to pinning and unpinning in our system, respectively. Heracles supports only one-way dataflow constraints, and unpinning works differently from our system. In Heracles, unpinning a value triggers recomputation that can result in a new dataflow, overwriting the just-unpinned value, whereas our system keeps the existing dataflow to not confuse the user.

Although Heracles included a pinning feature, the authors of the system did not provide the details of how they implemented pinning and whether pinning was implemented in a reusable manner. The main contribution of our approach is providing the generic pinning mechanism.

E. Conclusion and future work

Currently, a pinning facility does not frequently appear in many web user interfaces. One reason for this is that it is expensive to add this, seemingly simple, feature. This chapter explores a generic pinning mechanism which can be incorporated into web user interfaces with almost no cost. The property model constraint system enables keeping a variable unchanged simply by rejecting solutions that do not enforce the stay constraint of that variable. We provide a pinning algorithm by modifying the original strength assignment scheme of the property model system.

We have implemented the pinning mechanism within hotdrink. Our prototype demonstrates that a pinning facility can be easily incorporated into the user interfaces that need to support multi-way dependencies. We expect that our work lowers the entry barrier of a pinning feature.

For future work it would be interesting to find a way to prevent pinning failure. As mentioned earlier, after enough variables are pinned, pinning a variable can fail if the property model constraint system becomes over-constrained. In this case, it is better to prevent the possible failure by making the pin option unavailable for the variable. Pinning such a variable would indeed be senseless. A variable that cannot be pinned is always *derived* in all possible plans. Any new value that is put into the variable will be overwritten. To prevent the confusion, the widget should be disabled (read-only).

Checking if a variable is “pinnable” is possible through a single run of the constraint solver. However, the system needs to check every unpinned variable whenever the set of pinned variables changes, i.e., when pinning or unpinning is requested. This repetitive computation may affect the performance of a user interface. Experiments would be needed to assess its feasibility.

Another related but orthogonal area of research is the presentation of the pinning/unpinning feature to the user. For the sake of simplicity, our prototype is using checkboxes to provide a pinning facility, but a large number of checkboxes may make the user interface look cluttered. It might be better to provide pin/unpin commands in the context menu of the widget, and make the border of a pinned widget highlighted when it is pinned. Finding better pinning interfaces would require user studies as well.

CHAPTER V

UNDO

A. Introduction

A 1976 research report by Lance A. Miller and John C. Thomas noted that “It would be quite useful to permit users to ‘take back’ at least the immediately preceding command (by issuing some special *undo* command)” [38]. Undo enables a user to cancel his/her previous actions. This facility is especially important for graphical user interfaces because it is so easy for a user to click a wrong widget that leads to an undesired effect, e.g., resetting an important data by clicking a wrong checkbox. The undo feature allows the user to revert the system state with ease when he/she recognizes that a wrong action has been made. Undo also encourages explorative learning, which is especially important on the web where users encounter new interfaces frequently.

Although desktop user interfaces usually include an undo feature, currently web-based user interfaces rarely support it. Like other convenience interface features, implementing undo imposes an additional burden to programmers.

This chapter introduces an undo mechanism that reverts the state of a property model to its previous state. Restoring the previous state includes recovering the previous values of variables, solution graph, strength assignment, valuation, and others which are the elements of a property model system state. The recovery process also has to coordinate with other behaviors such as enabling/disabling widgets to maintain the whole system in a consistent state after undoing. In this chapter, we take a *partial checkpoint* strategy [39]: store a part of the system state, enough to be able to reverse the effect of the immediately preceding command.

B. Background

Researchers have explored several undo models for many years [39, 40, 41]. Linear model is the simplest model and it is extensively used in practice. In the linear model, *a linear history* is a chain of commands where commands can be undone one by one in a sequential order, or undone commands redone, similarly one command at a time.

Although the linear model can provide long undo-histories, it does not allow undoing arbitrary commands without undoing all commands that follow them. In contrast, non-linear models allow the user to undo arbitrary actions in isolation. There are several types of non-linear models, such as the *script model* [40] and the *selective undo model* [39]. The history structure of non-linear models, however, can become very complicated; a user may struggle understanding the history unless a good visualization of the history is provided.

Conceptually a linear undo model consists of two lists of commands: the undo and redo lists, as shown in Figure 21. When a command that changes the state of the system is executed, it is placed as the last element of the undo list. If the undo list is not empty, an invocation of undo extracts the last command from the undo list and reverses its effect—and places the command as the first element of the redo-list. Whenever the redo list is not empty, one can redo the previously undone command. An invocation of redo extracts the first element of the redo list, executes the command, and appends the command to the end of the undo list. In addition to the commands, the undo and redo lists need to store enough state of the system that the effects of commands can be reversed or redone.

C. Undoing set command

When a user interface is implemented using the property models approach, the entire state of the UI is stored in the property model. Implementing a generic undo behavior thus requires us to maintain the undo and redo lists, where the commands are change requests to

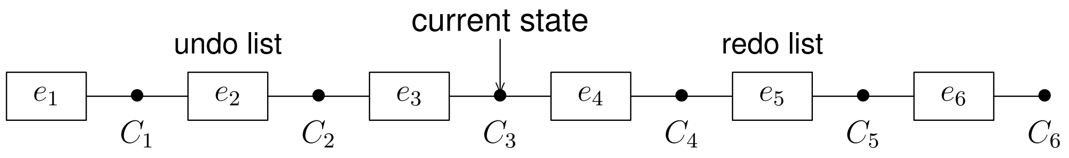


Fig. 21. A linear undo/redo history of commands. $e_1, e_2, \cdots, e_6$ are commands (or events), and $C_1, C_2, \cdots, C_6$ are the system states. In this case, the current state is C_3 with undo list $= [e_1, e_2, e_3]$ and redo list $= [e_4, e_5, e_6]$.

the property model and where the state consists of the property model's variables' values, the editing history of the variables, the current solution graph, and the current evaluation graph.

In the property model system, a basic modification command is set, which assigns a new value to a variable. pin and unpin are also modification commands that are undoable. In this thesis we focus on undoing a set command.

As explained in Section 5 of Chapter II, the state of a property model, the current configuration, is defined as a tuple $C = \langle G_s, s, \nu \rangle$, where G_s is a solution graph, s a strength assignment, and ν a valuation. The set command may change the strength assignment s as well as the solution graph G_s. The command will also produce a new valuation and a new evaluation graph, which is obtained with the relevancy information in the valuation.

Undo list could store all elements in the configuration (*full checkpoint strategy* [40]). However, a lot of information is computed from other information, so alternatively we can store only the information that is not derived (partial checkpoint strategy [40]). We take the latter approach to reduce storage overhead. In this approach, reverting is a bit more complex than just restoring a snapshot.

In this section, we present an undo mechanism, which involves re-running evaluation. The undo command we implement can be seen as a set command that reverses the effect of the immediately preceding command. It performs a process that is similar to the process for set. The process for set can be summarized: (1) set a new strength assignment s', (2) com-

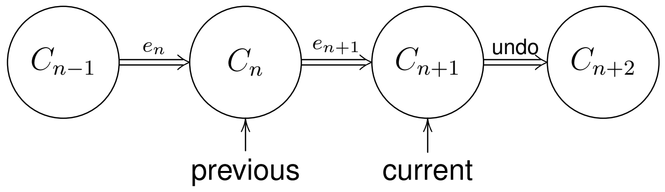


Fig. 22. A series of configuration transitions of a property model, where C_n is the previous configuration, C_{n+1} is the current configuration after e_{n+1} command, and C_{n+2} is the configuration after undoing the preceding command.

pute a new solution graph G'_s if necessary, (3) compute a new valuation ν' by setting a new value and initializing computed-flags, changed-flags, and relevancy-flags (*pre-evaluation stage*), and (4) finally run evaluation functions, resulting in a new configuration (the details are described in Section 5 of Chapter II).

Figure 22 illustrates the working of our undo command. Assuming the system is at the state C_{n+1} where the previous state was C_n, and the previous command was set, undoing the set command invokes the following steps: (1) restore the strength assignment of C_n, (2) restore the solution graph of C_n, (3) restore the valuation of the pre-evaluation stage that existed right before the system state becomes C_n, and (4) run evaluation functions to resurrect the valuation of C_n. The result of the procedure is a new configuration C_{n+2} which is identical to C_n except for the changed-flags of variables (we will describe the difference between the two configurations in more detail below). For this, a modification command needs to save certain elements of the previous configuration such as a strength assignment and the values of variables before making changes.

More specifically, a set command appends an *event object*, which represents the command, to the history. The event object contains data that represent the change and that are necessary to undo the command. The event object for set includes the following: the target variable to set a value, the previous strength assignment (of C_n), and the list of variable and value pairs that will be used to resurrect the valuation of the previous configuration C_n. The latter list includes the values collected at the pre-evaluation stage during the transition from C_{n-1} to C_n. The values enable us to restore the valuation of the pre-evaluation stage and reach to the valuation of C_n after running evaluation.

We present the undo mechanism more formally in terms of a state transition. Assume the following things: the current configuration is C_{n+1}, the previous configuration is C_n, and the previous configuration of C_n is C_{n-1}; the valuation at the pre-evaluation stage that existed during the transition from C_{n-1} to C_n is ν'_{n-1}; the strength assignment, solution graph, and valuation of C_n are s, G_s, and ν_n, respectively; the strength assignment, solution graph, and valuation of C_{n+1} are s', G'_s, and ν_{n+1}, respectively; the previous command is set that sets t' to a variable v; and the event object of the command contains v, s, and rv which is the list of variable and value pairs from ν'_{n-1}. Invoking undo has the following effect on the current configuration C_{n+1}:

1. A strength assignment s'' is computed, such that $s'' \leftarrow s$. This restores the strength assignment of C_n.

2. A solution graph G''_s is computed, such that $G''_s = G_s$. Although we did not save G_s, we can reconstruct G_s. If $s = s'$ or v is a source in G'_s, the solution graph G'_s is the same as G_s. In this case, we can use G'_s. Otherwise, G'_s may be different from G_s. In this case, the solver algorithm is run to produce a new solution graph G''_s, where $G''_s = G_s$ (because the most preferred solution graph is unique [16]).

3. A new valuation ν'_{n+1} is computed from ν_{n+1} as follows: for each variable and value pair (k, u) in rv, if $u \neq \nu_{n+1}(k)$, k is set to u, an old value, and the changed-flag of k is set to changed; the changed-flag is set to unchanged for all the other variables; the computed-flag of every variable is set to uncomputed; and the relevancy-flag is set to relevant for all edges (v', m') from variables to methods, such that (v', m') is not an

edge in G'_s but is an edge in G''_s.

4. (Evaluation round) The eval function, shown in Figure 5, is applied to each variable in V. A call to eval may change the valuation, producing a new valuation ν_{n+2}.

The result of this transition is a new current configuration $C_{n+2} = \langle G''_s, s'', \nu_{n+2} \rangle$.

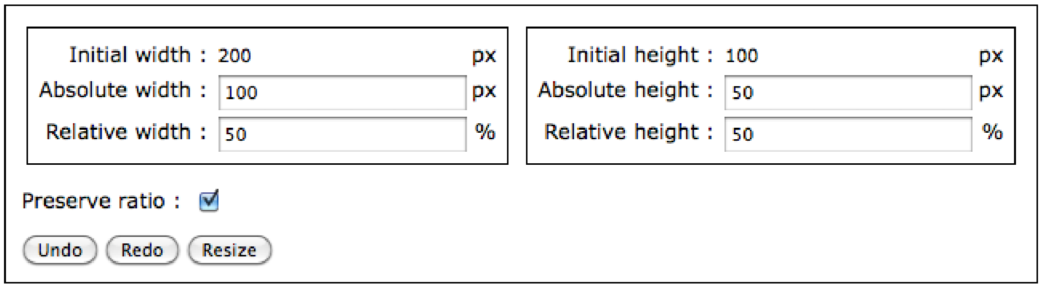
If we assume that the methods executed during the evaluation round are "regular" functions [42], i.e., functions that always produce the same outputs for the same inputs, C_{n+2} is identical to C_n except the changed-flags in the valuation. In the step 1 and 2 of the above procedure, the strength assignment and solution graph are reverted such that $s'' = s$ and $G''_s = G_s$. In the step 3, the valuation ν'_{n+1} is computed such that the values of variables in the valuation are identical to the ones in ν'_{n-1}, the valuation of the pre-evaluation stage during the transition from C_{n-1} to C_n. This is realized by overwriting the values with the ones in rv, which holds the very values of variables of ν'_{n-1}. The evaluation in the step 4 will make all the variables have the values identical to the values of variables of C_n if the executed methods are regular. The computed-flags of variables do not matter since they are all set to uncomputed before evaluation, and they all become computed after evaluation. The changed-flags of variables in C_{n+2}, however, may be different from those of C_n because the variables that are set to changed in ν'_{n+1} may be different from those in ν'_{n-1} (step 3), and the difference may result in different changed-flags after evaluation.

In our mechanism, we do not recover the changed-flags of variables. This is intentional to guarantee the correctness of existing implementations of user interface behaviors. One of the implementations is that for the widget enablement algorithm. Whenever the state of a property model changes, the implementation is run with the information on the changed variables. Therefore, the configuration that is produced by undo needs to reflect the actual changes of variables in their changed-flags.

We implemented the undo mechanism within hotdrink. The implementation fits well into the existing framework. The addition of undo had no impact on the implementation of other generic user interface behaviors.

D. Example scenario involving undoing

Consider an image resize web form, shown in Figure 23. This example originally appeared in an earlier property models paper [8]. Four text fields are editable: *absolute width*, *absolute height*, *relative width*, and *relative height*. The interface allows the user to either set the image's absolute sizes or scale it in proportion to the initial width and height. The initial sizes are visible in the interface, but not editable. In addition, the form contains a checkbox, *preserve proportions*, that sets or clears a flag that ensures that the height and width retain the same aspect ratio as that between the initial height and width. The variables that comprise the interface and the relationships between the variables are defined as the property model in Figure 24.



Initial width :	200	px	Initial height :	100	px
Absolute width :	100	px	Absolute height :	50	px
Relative width :	50	%	Relative height :	50	%

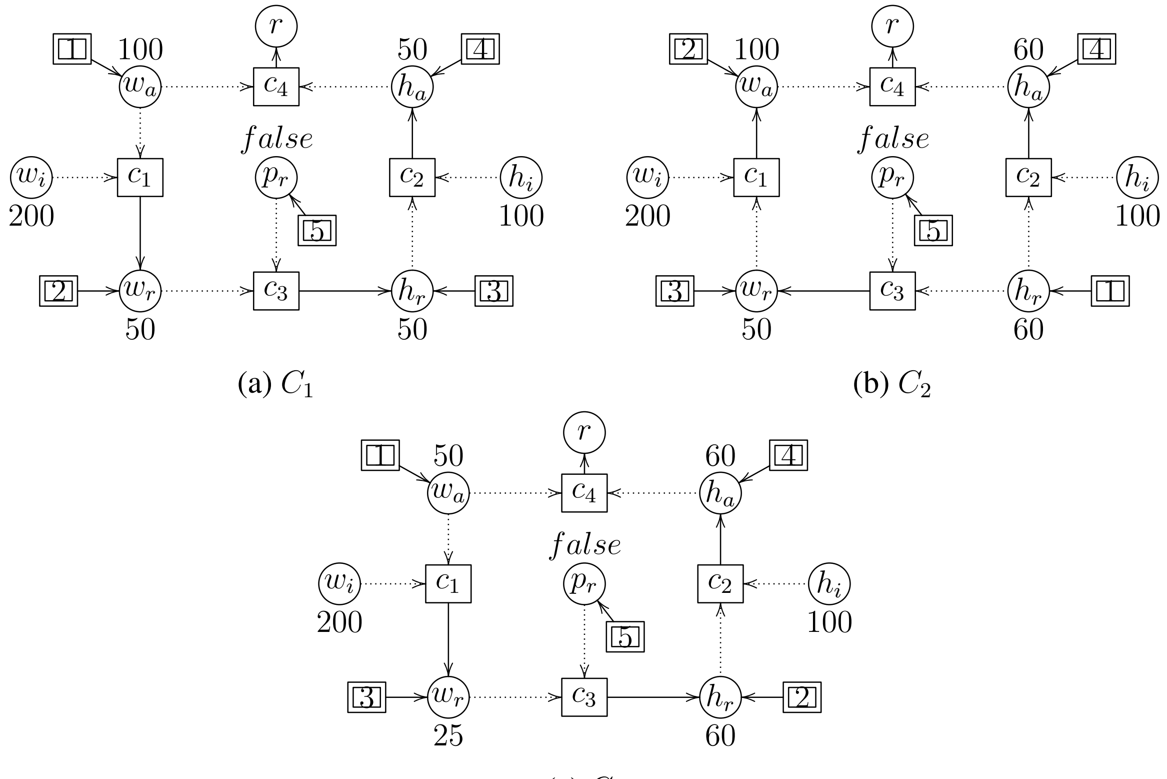
Preserve ratio : ✓

Fig. 23. An image resize form with undo/redo features. The user can use the undo button to revert his or her change on a text field or a checkbox. The undo action can be canceled by using the redo button.

Assuming that the system is at some state C_1, which is shown in Figure 25(a), consider the following scenario: the user (1) sets 60 to the "relative height," then (2) sets 50 to the "absolute width," and (3) undoes the last set command. The first set command results in the configuration C_2, which is shown in Figure 25(b), and the second set command produces

```
model image_resize {
  input: {
    initial_height : 5 * 300;
    initial_width : 7 * 300;
  }

  interface: {
    absolute_height : initial_height;
    absolute_width;
    relative_height;
    relative_width;
    preserve_ratio : true;
  }

  logic: {
    relate {
      absolute_width <== relative_width * initial_width / 100;
      relative_width <== absolute_width * 100 / initial_width;
    }
    relate {
      absolute_height <== relative_height * initial_height / 100;
      relative_height <== absolute_height * 100 / initial_height;
    }
    relate {
      relative_width <== (preserve_ratio) ? (relative_height) : (relative_width);
      relative_height <== (preserve_ratio) ? (relative_width) : (relative_height);
    }
  }

  output: {
    result <== { height: absolute_height, width: absolute_width };
  }
}
```

Fig. 24. The property model specification for the web form in Figure 23. Three constraints are defined by the user: c_1 is between absolute_width, relative_width, and initial_width; c_2 between absolute_height, relative_height, and initial_height; and c_3 between relative_width, relative_height, and preserve_ratio.



(a) C_1 (b) C_2

(c) C_3

Fig. 25. The solution graph, strength assignment, and values of the configuration C_1, C_2, and C_3, where r is the variable result in Figure 24; likewise, w_a is absolute_width, w_i initial_width, w_r relative_width, h_a absolute_height, h_r relative_height, and p_r preserve_ratio. c_1, c_2, and c_3 are user-defined constraints defined in the property model in Figure 24. c_4 is a single-method constraint generated by the system to compute the result value. The double-framed rectangles are stay constraints with strength values. The values of variables are displayed either above or below each variable. The system goes from C_1 to C_2 when the user sets 60 to h_r. At C_2 the changed-flags of h_r, h_a, and r are changed, whereas the changed-flags of the other variables are unchanged. Next, if the user set 50 to w_a, the system goes to another state C_3, where the changed-flags of w_a, w_r, and r become changed, and the changed-flags of the other variables are unchanged.

the configuration C_3 as shown in Figure 25(c).

For recovery, the event object of the second set command that resulted in C_3 contains the previous priority order, $[h_r, w_a, w_r, h_a, p_r, \cdots]$, and the values of variables, $[(h_r, 60), (w_r, 50), (p_r, false)]$, which are collected at the pre-evaluation stage of the first set command that resulted in C_2,

In this case, at the pre-evaluation stage, we collected only the values of h_r, w_r, and p_r variables that may be referred during evaluation. The values of the other variables are derived during evaluation (the solution graph indicates which variables, such as w_a, h_a, and r, will be derived), or constant (input variables such as w_i and h_i), so we do not need to keep their values.

Undoing the second set command will go through the following steps:

1. Restore the priority order of C_2: $p \leftarrow [h_r, w_a, w_r, h_a, p_r, \cdots]$.

2. Run the solver algorithm to get the previous solution graph of C_2.

3. Restore the valuation of pre-evaluation stage: $\nu(w_r) \leftarrow \langle 50, \_, \text{changed}\rangle$. The values of h_r and p_r are the same as the ones of C_3, thus, those variables are not modified.

4. Run eval function for each variable, resulting in a new valuation.

The resulting configuration is C_4 where the values of variables are the same as Figure 25(b), and the changed-flags of w_r, w_a, and r are changed. C_4 has the same priority order, solution graph, values of variables, and relevancy-flags as C_2. The changed-flags of variables are, though, different from C_2.

E. Related work

Systems that support undo/redo are common in desktop applications. It is not hard to find the recovery feature in desktop productivity tools such as word processors and spreadsheets. They are also commonly found in design applications like Adobe Photoshop [43]. A design-level mechanism called *Memento* pattern [44] provides an implementation-independent mechanism for undo in application development. We discuss a few more generic frameworks that help implementing the undo facility below.

GINA [45] is an application framework and a class library to facilitate the construction of interactive graphical user interfaces. It provides an object-oriented framework for implementing linear undo models and *selective undo* models (undoing arbitrary commands) using command objects. This early system enables application programmers to add the undo facility by implementing undo/redo functions for each command object they provide. GINA also provides the common features such as undo/redo menu entries and a history scroller, which enables the user to go back and forth in the history. However, GINA does not reduce the programmer's burden to program the reverse operations for common actions.

Amulet [18], a user interface development environment for C++, provides an undo mechanism similar to that of GINA. It utilizes command objects like GINA, but its selective undo mechanism adds the ability to repeat previous commands and to undo selections and scrolling. Like our system, Amulet supports multi-way dataflow constraint systems and the actions of their basic widgets are undoable without any extra code.

Undo facility is not very common in web-based user interfaces. In certain web applications, a "back" button, which is provided by most web browsers, often fulfills the need for going back to the previous state, or canceling the last action [46]. However, even in feature-rich web applications, the back button does not always play the role of undo. Under certain circumstances, web forms provide a "reset" button. However, the reset button does not satisfy what people expect from undo since it can only empty or initialize all fields in the form.

An early web-based ontology system [47], which enables collaboration on the web, demonstrates the usefulness of undo/redo capabilities on the web environment. The authors of the system argue that the undo/redo features allow the user to repair any mistakes, so the features improve the reliability and trustworthiness that are essential to the multi-user shared systems on the web. However, the authors do not provide any detail about implementing the undo/redo on the web environment.

Tecnicia's web-based MIS systems [48] support undo, motivated by the added value of the feature—undo provides a suitable tool to easily correct human errors. The authors of the systems argue that undo is a basic feature of *recovery-oriented computing* (ROC), which aims to increase system's dependability by reducing repair and recovery time. Like our system, their undo mechanism is based on a partial checkpoint strategy, re-updating the data using a normal update process. The main difference is that the MIS systems provide a persistent undo function, which can recover data that users have changed or deleted in past sessions. To this ends, the MIS systems have implemented the undo function on their database servers, duplicating data upon modification.

Cappuccino, a web application framework using Objective-J, supports undo/redo [49]. This framework provides a programming model for application programmers to support undo for some actions on web applications. Cappuccino only provides programming interfaces with which the programmers have to implement the reverse operations by themselves.

F. Conclusion and future work

Human errors are pegged as the root cause of roughly 20–50% of system errors [50]. Therefore, interactive user interfaces should provide a facility for users to recover from their errors. Although a lot of desktop user interfaces provide an undo command to go back to a point before things went wrong, few user interfaces on the web provide a similar convenience feature.

This chapter explores an undo mechanism which can be incorporated into web user interfaces based on property models. We take a partial checkpoint and re-run strategy to provide an efficient linear undo mechanism. To add undo/redo facilities to the existing property models system, we abstract a user action as an event object which contains the data for recovery. When undo is initiated, we restore the partial state with the values of variables and re-run the evaluation round to produce a complete previous state. This strategy enables us to add an undo facility without a major change on the existing system.

We have implemented the undo prototype as part of our hotdrink project. Our prototype demonstrates that an undo facility can be reused across several user interfaces without having to write any user interface specific code.

Our work extends property models with a reusable recovery feature. Experience with this feature suggests several avenues for future research. We currently do not offer any way to inspect the undo history, but a good visualization feature would likely be helpful for users. In addition, we have not considered pinning behavior in this chapter. Undoing pinning or unpinning operations can be done in a mechanism similar to undoing a set operation. The mechanism will have to recover the strength assignment as well as the pinned set.

CHAPTER VI

EVALUATION

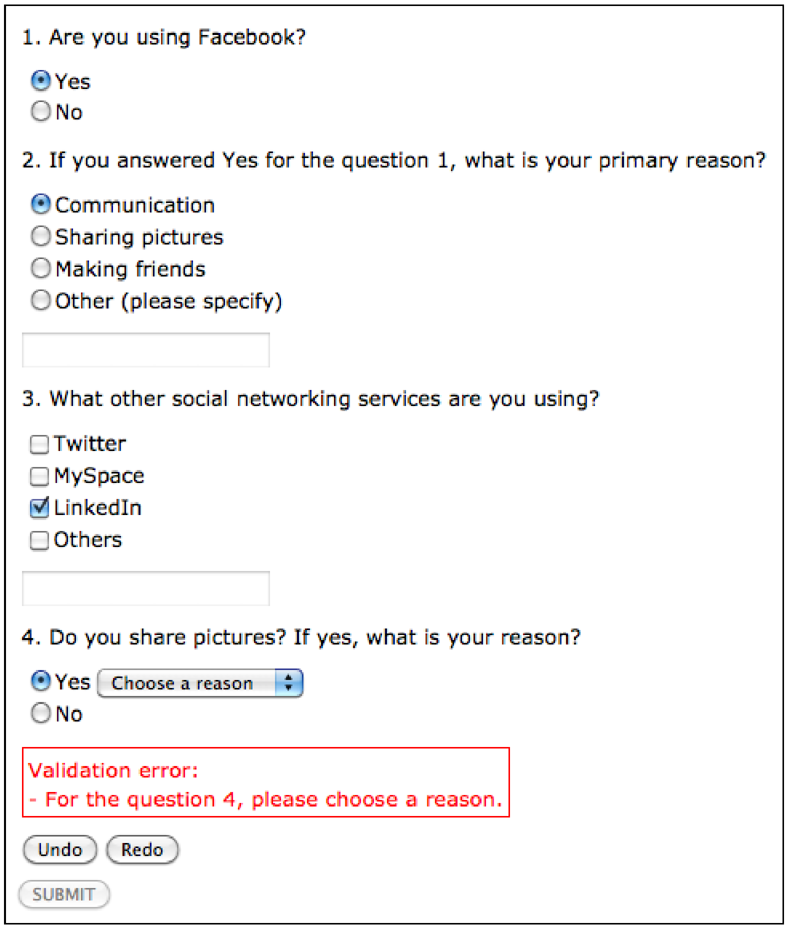
This thesis proposes three reusable user interface behaviors. We have not conducted user studies for their usefulness, but we rely on our knowledge and experience with user interfaces in advocating them as beneficial to the user. Further, there are studies in the computer-human interaction (CHI) research community that support our view [33, 30, 32]. Thus, this evaluation section is not about assessing the usefulness of the user interface behaviors, but rather about assessing their implementation effort. The main point of the thesis is that the implementation of these behaviors should be practically free. In this chapter, we provide small experiments that quantify the savings in implementation effort.

We have implemented the prototype of the proposed mechanisms for three help and convenience behaviors, and we have applied it in constructing several user interfaces, web forms and dialogs in particular. The results have been positive, demonstrating increased productivity and reusability.

The property models approach is reported to significantly reduce defect rates and vastly increase productivity of programmers when applied to implementing user interfaces for a desktop application [9]. Although we have not conducted the same evaluation for web-based user interfaces, we believe that the results apply to web user interfaces. Our small-scale evaluation serves as supporting evidence.

To evaluate our approach in terms of productivity, we constructed two web forms, which support the behaviors presented in this thesis, using two alternative approaches—one using our prototype and another using a popular JavaScript library, jQuery [51]—and compared them in terms of their code sizes and programming experiences.

The web form we first constructed for evaluation is an online survey form shown in Figure 26. The survey form automatically enables and disables choices of answers de-


1. Are you using Facebook?

[✓ Yes]
No

2. If you answered Yes for the question 1, what is your primary reason?

[✓ Communication]
Sharing pictures
Making friends
Other (please specify)

3. What other social networking services are you using?

☐ Twitter
☐ MySpace
☑ LinkedIn
☐ Others

4. Do you share pictures? If yes, what is your reason?

[✓ Yes] Choose a reason
No

Validation error:
- For the question 4, please choose a reason.

Fig. 26. An online survey form with widget enablement, validation and undo behaviors.

	hotdrink	jQuery
Widget enablement	6	33
Validation	19	69
Undo	0	41
Total code	55	261

Fig. 27. The number of code lines in two implementations, one using "hotdrink," the other using "jQuery," of the user interface shown in Figure 26. We show the line counts of code related to three different features: widget enablement, validation, and undo. In the hotdrink implementation, the code counted for widget enablement and undo are the `logic` and `invariant` sections of the property model definition. The row labeled "total code" represents all code except for the layout code (Eve specification and HTML code).

pending on the answers to related questions (widget enablement behavior). In particular, it supports the two behaviors discussed in this thesis: (1) validation and generating error messages; and (2) undo and redo.

The three behaviors can be easily provided when using property models. A programmer simply has to define the property model, which consists of variables, each representing a piece of data, the relationships between the variables, and validation conditions with error message templates. In the implementation that used jQuery, the declarative model specification is not needed. Instead, the programmer writes event handlers that react to the update events of widgets, enable and disable widgets, validate the values of widgets, generate error messages, and undo/redo user actions.

Figure 27 shows the number of code lines in two implementation versions. The implementation that used jQuery required more than five times as many code lines as the implementation that used property models. Most of the code (45 lines out of 55) in the property model version is to define the property model. In the property model version, the lines counted for each behavior are to define the relationships between variables and the invariants in the property model. No event handling code is needed to provide widget enablement, validation, and undo behaviors. They are enabled simply by "turning them on" in the layout specification. In the jQuery version, the lines counted for each behavior are event handling code that is required to provide each behavior.

The result shows that the jQuery version, which was implemented in a more traditional way using event handlers, required the programmer to put more programming effort in terms of code lines. In addition to being much longer, the code in the event handler implementation was clearly not reusable in other web forms.

The second web form we constructed is a trip budget planner shown in Figure 14 in Chapter IV. The trip budget planner form takes advantage of multi-way dataflow constraints, and it thus has to support value propagation and support for pinning is desirable. These are the behaviors we implemented in both versions of the planner form. Again, in the property model version, these behaviors are simply enabled by turning them on, whereas in the jQuery version, the behaviors are encoded as event handlers.

Figure 28 shows the line counts of the two versions. We use line counts as an indication of how much effort goes into implementing a user interface, and thus as a measure of programmer productivity. Again, we observe that the jQuery version is considerably longer—reassuring us that the traditional way gives lower productivity than our approach. We further note that the jQuery version does not implement all the dependencies that the property model version implemented. The reason for this is that adding code for satisfying all the multi-way dependencies was too much complicated. In the property model version, the multi-way constraint solver takes care of the multi-way dependencies.

We have so far applied our approach mostly to web forms and dialogs. We have not yet applied it to large, complex real-world user interfaces. While a more thorough evaluation remains as future work, we nevertheless interpret the results of the experiments reported in

	hotdrink	jQuery
Value propagation	17	168
Pinning	0	
Total code	87	276

Fig. 28. The number of code lines in two implementations, one using "hotdrink," the other using "jQuery," of the user interface shown in Figure 14. We show the line counts of code related to two different features: value propagation and pinning. In the hotdrink version, the code counted for value propagation is the `logic` section of the property model definition. In the jQuery version, the code for value propagation and pinning behaviors was mingled, so we counted them together. The row labeled "total code" represents all code except for the layout code (Eve specification and HTML code).

this chapter as very positive.

CHAPTER VII

CONCLUSION

Entry to implementing web-based user interfaces is relatively low. HTML and JavaScript tutorials abound, and it is easy to get some web user interface implementations up and running quickly. The vast amount of web-sites with forms of all kinds is the proof of this. Many of those forms also serve as a proof that the entry level programming capabilities only go so far. Low-quality interfaces are almost a norm, rather than an exception. Implementing feature-rich user interfaces, supporting, e.g., value propagation, pinning, undo and redo, and guiding the user with precise error messages—all this with no or very few defects—is difficult and laborious.

This thesis builds on the promising approach of property models, which make common user interface features reusable components of a software library, considerably lessening the task of implementing a feature-rich user interface, and implementing it correctly. In particular, this thesis describes three help and convenience features that can be generically implemented based on property model systems. The behaviors of validating user data, generating precise, helpful error diagnostics, and allowing users to undo actions can significantly improve the usability of web forms. Pinning gives users an explicit control of dataflow—a feature that is particularly useful in user interfaces that represent pieces of data with complex dependencies.

This thesis expands the range of user interface behaviors that can be packaged into reusable software components and algorithms, and in doing so practically removes the implementation cost of adding support for those behaviors in new user interfaces. In the future, these results have the potential of improving the user experience of countless web applications.

REFERENCES

[1] K. Scoresby, Win consumers with better usability, e-Business Advisor, June 2000. `http://www.scoresby.com/resources/article.php?a=0`.

[2] B. A. Myers, M. B. Rosson, Survey on user interface programming, in: CHI '92: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, New York, NY, USA, 1992, pp. 195–202.

[3] T. A. Standish, An essay on software reuse, IEEE Transactions on Software Engineering 10 (1984) 494–497.

[4] S. Parent, A possible future for software development, Keynote talk at the Workshop of Library-Centric Software Design 2006, at OOPSLA'06, Portland, OR, USA, 2006.

[5] Microsoft, System.Windows.Forms, 2009. `http://msdn.microsoft.com/en-us/library/system.windows.forms.aspx`.

[6] Apple, Apple developer connection: Cocoa, 2009. `http://developer.apple.com/cocoa/`.

[7] Nokia, Qt: A cross-platform application and UI framework, 2009. `http://qt.nokia.com/`.

[8] J. Järvi, M. Marcus, S. Parent, J. Freeman, J. N. Smith, Property models: From incidental algorithms to reusable components, in: Proceedings of the 7th International Conference on Generative Programming and Component Engineering, GPCE '08, ACM, New York, NY, USA, 2008, pp. 89–98.

[9] J. Järvi, M. Marcus, S. Parent, J. Freeman, J. Smith, Algorithms for user interfaces, in: Proceedings of the 8th International Conference on Generative Programming and Component Engineering, GPCE '09, ACM, New York, NY, USA, 2009, pp. 147–156.

[10] J. Freeman, W. Kim, J. Järvi, Hotdrink: Library for building user interfaces on the web, 2011. `http://code.google.com/p/hotdrink/`.

[11] S. Parent, Adobe Property Model Library, 2005. `http://stlab.adobe.com`.

[12] Adobe Systems, Adobe Source Libraries, 2005. `http://stlab.adobe.com`.

[13] B. V. Zanden, An incremental algorithm for satisfying hierarchies of multiway dataflow constraints, ACM Transactions on Programming Languages and Systems 18 (1996) 30–72.

[14] A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, M. Woolf, Constraint hierarchies, SIGPLAN Notices 22 (1987) 48–60.

[15] B. N. Freeman-Benson, J. Maloney, A. Borning, An incremental constraint solver, Communications of the ACM 33 (1990) 54–63.

[16] J. Freeman, J. Järvi, J. Smith, M. Marcus, S. Parent, Properties of constraint systems of property models, Technical Report TR09-001, Parasol Laboratory, Department of Computer Science and Engineering, Texas A&M University, 2009. `http://parasol.cs.tamu.edu/groups/pttlgroup/property-models/pm-tr-1.pdf`.

[17] I. E. Sutherland, Sketchpad: A man-machine graphical communication system, in: DAC '64: Proceedings of the SHARE Design Automation Workshop, ACM, New York, NY, USA, 1964, pp. 6329–6346.

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

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

[20] 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, ACM, New York, NY, USA, 1994, pp. 137–146.

[21] A. Borning, R. K.-H. Lin, K. Marriott, Constraint-based document layout for the web, Multimedia Systems 8 (2000) 177–189.

[22] G. J. Badros, A. Borning, K. Marriott, P. Stuckey, Constraint cascading style sheets for the web, in: Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology, UIST '99, ACM, New York, NY, USA, 1999, pp. 73–82.

[23] Mozilla Foundation, XML user interface language (XUL) 1.0, 2006. `http://www.mozilla.org/projects/xul/xul.html`.

[24] Microsoft, XAML: Extensible application markup language, Microsoft Developer Network (MSDN), 2008. `http://msdn.microsoft.com/en-us/library/ms747122.aspx`.

[25] Adobe Systems, Flex, 2004. `http://www.adobe.com/products/flex.html`.

[26] OpenLaszlo project, 2005. `http://www.openlaszlo.org`.

[27] C. Watson, Web form error message design showcase - examples, 2008. `http://www.smileycat.com/miaow/archives/001411.php`.

[28] M. Green, The design of graphical user interfaces, Technical Report CSRI-170, Computer Systems Research Institute, University of Toronto, 1985.

[29] J. Foley, W. C. Kim, S. Kovacevic, K. Murray, Defining interfaces at a high level of abstraction, Software, IEEE 6 (1989) 25 –32.

[30] D. F. Gieskens, J. D. Foley, Controlling user interface objects through pre- and post-conditions, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '92, ACM, New York, NY, USA, 1992, pp. 189–194.

[31] R. Moriyon, P. Szekely, R. Neches, Automatic generation of help from interface design models, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Celebrating interdependence, CHI '94, ACM, New York, NY, USA, 1994, pp. 225–231.

[32] P. Sukaviriya, J. J. de Graaff, Automatic generation of context-sensitive "show and tell" help, Technical Report GIT-GVU-92-18, Georgia Institute of Technology, 1992.

[33] P. Sukaviriya, J. D. Foley, Coupling a UI framework with automatic generation of context-sensitive animated help, in: Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology, UIST '90, ACM, New York, NY, USA, 1990, pp. 152–166.

[34] P. N. Sukaviriya, J. Muthukumarasamy, A. Spaans, H. J. J. de Graaff, Automatic generation of textual, audio, and animated help in UIDE: The User Interface Design,

in: Proceedings of the Workshop on Advanced Visual Interfaces, AVI ’94, ACM, New York, NY, USA, 1994, pp. 44–52.

[35] C. Brabrand, A. Møller, M. I. Schwartzbach, The bigwig project, ACM Transactions on Internet Technology 2 (2002) 79–114.

[36] C. Brabrand, A. Møller, M. Ricky, M. I. Schwartzbach, PowerForms: Declarative client-side form field validation, World Wide Web 3 (2000) 205–214.

[37] C. A. Knoblock, S. Minton, J. L. Ambite, M. Muslea, J. Oh, M. Frank, Mixed-initiative, multi-source information assistants, in: Proceedings of the 10th International Conference on World Wide Web, WWW ’01, ACM, New York, NY, USA, 2001, pp. 697–707.

[38] L. A. Miller, J. C. Thomas, Behavioral issues in the use of interactive systems, in: IBM Germany Scientific Symposium Series, 1976, pp. 193–216.

[39] T. Berlage, A selective undo mechanism for graphical user interfaces based on command objects, ACM Transactions on Computer-Human Interaction 1 (1994) 269–294.

[40] J. E. Archer, Jr., R. Conway, F. B. Schneider, User recovery and reversal in interactive systems, ACM Transactions on Programming Languages and Systems 6 (1984) 1–19.

[41] Yiya, Yang, Undo support models, International Journal of Man-Machine Studies 28 (1988) 457 – 481.

[42] A. Stepanov, P. McJones, Elements of Programming, 1st Edition, Addison-Wesley Professional, 2009.

[43] Adobe Systems, Adobe Photoshop, 1990. `http://www.adobe.com/products/photoshop.html`.

[44] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Professional Computing Series, Addison-Wesley Longman Publishing Co., Inc., 1995.

[45] M. Spenke, C. Beilken, An overview of GINA—the generic interactive application, in: Proceedings of the Workshop on User Interface Management Systems and Environments on User Interface Management and Design, UIMS, Springer-Verlag New York, Inc., New York, NY, USA, 1991, pp. 273–293.

[46] Google, Gmail, 2004. `http://mail.google.com/`.

[47] J. Rice, A. Farquhar, P. Piernot, T. Gruber, Using the web instead of a window system, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Common Ground, CHI '96, ACM, New York, NY, USA, 1996, pp. 103–110.

[48] N. Serrano, F. Alonso, J. Sarriegi, J. Santos, I. Ciordia, A new undo function for web-based management information systems, Internet Computing, IEEE 9 (2005) 38–44.

[49] 280 North, Cappuccino project, 2008. `http://cappuccino.org`.

[50] D. Oppenheimer, A. Ganapathi, D. A. Patterson, Why do internet services fail, and what can be done about it?, in: Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4, USITS'03, USENIX Association, Berkeley, CA, USA, 2003, pp. 1–1.

[51] J. Resig, jQuery, 2006. `http://www.jquery.com/`.

VITA

Wonseok Kim received his Bachelor of Science degree in Computer Science and Engineering from Seoul National University in 2005. He entered the Computer Science program at Texas A&M University in September 2009, and received his Master of Science degree in December 2011. His research interests include programming languages and web-based user interfaces.

Wonseok can be reached at Parasol Lab, Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843-3112. His email address is guruwons@gmail.com.