Helping Programmers Help Users

John Freeman
Texas A&M University
jfreeman@cse.tamu.edu

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

Wonseok Kim
Texas A&M University
guruwons@cse.tamu.edu

Mat Marcus
Canyonlands Software Design
mmarcus@emarcus.org

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

Abstract

User interfaces exhibit a wide range of features that are designed to assist users. Interaction with one widget may trigger value changes, disabling, or other behaviors in other widgets. Such automatic behavior may be confusing or disruptive to users. Research literature on user interfaces offers a number of solutions, including interface features for explaining or controlling these behaviors. To help programmers help users, the implementation costs of these features need to be much lower. Ideally, they could be generated for "free." This paper shows how several help and control mechanisms can be implemented as algorithms and reused across interfaces, making the cost of their adoption negligible. Specifically, we describe generic help mechanisms for visualizing data flow and explaining command deactivation, and a mechanism for controlling the flow of data. A reusable implementation of these features is enabled by our property model framework, where the data manipulated through a user interface is modeled as a constraint system.

Categories and Subject Descriptors H.2.2 [*Software Engineering*]: Design Tools and Techniques—user interfaces

General Terms Algorithms

Keywords user interfaces, software reuse, constraint systems, software architecture

1. Introduction

The dull, run-of-the-mill user interfaces—dialogs, forms, and such—do not get much attention from the software research community, but they collectively require a lot of attention from the programmer community. User interfaces abound, and they are laborious to develop and difficult to get correct. As an attempt to reduce the cost of constructing user interfaces, we have introduced *property models*, a declarative approach to programming user interfaces [8, 9]. The long term goal of this work is to reach a point where most (maybe all) of the functionality that we have come to expect from a high quality user interface would come from reusable algorithms or components in a software library, parametrized by a specification of the data manipulated by the user interface. In particular, we have described reusable implementations for the propagation of values between user interface elements, the enablement and disablement of user interface widgets, and the activation and deactivation of widgets that launch commands.

This paper describes our work to direct these advances to the improvement of user interfaces. One purpose of a user interface is to provide the user with an easily interpreted view of a conceptual model for the internal states of the application and the interface itself. To the extent that the interface fails to do this, there exists a *gulf of evaluation* [7]. The gulf of evaluation exacerbates the cognitive effort required to understand and use an application, and can lead to user frustration.

This paper shows that with the power of components, generativity, and reuse we can go beyond merely implementing existing behavior more economically. If a user interface behavior can be successfully packaged into a reusable component, then we should explore more functionality for assisting users and closing the gulf of evaluation. We should aim for more consistent user interfaces with less surprising behavior, more explanations of why a user interface behaves the way it does, and more abilities to change the behavior of a user interface "on the fly" to better serve users' goals. In sum, we should aim for more features that *help* users in their interactions with an interface.

This paper describes several generic realizations of help and convenience features that could be provided as standard features of dialogs and forms. In particular, we focus on (1) visualizing how data flows in a user interface, (2) providing help messages for commands that are deactivated, and (3) providing the user with means to control the direction of the flow of data. We emphasize that the main contributions of the paper are the algorithms and the software architecture that enable implementing these features in a reusable manner, applicable to a large class of user interfaces with negligible programming effort. The realizations of these algorithms build on the property models approach, in which the data that a user interface manipulates and the dependencies within this data are modeled explicitly as a constraint system. Reusable user interface algorithms are thus algorithms that inspect and manipulate this constraint system.

We are at an early stage in our effort. To not overstate our contribution, we note that we have not conducted user studies, and we have not applied the proposed tools and algorithms to a large collection of user interfaces drawn from existing software. The computer-human interaction (CHI) research community, however, has devised many help and support features for user interfaces and

GPCE'11, October 22–23, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0689-8/11/10...$10.00

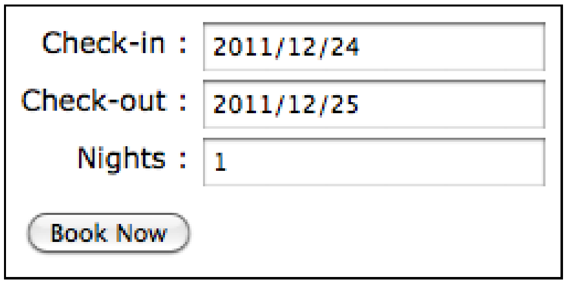


Figure 1. A dialog for reserving a hotel room.

argued for their usefulness [5, 13, 14]. We believe in these results, and wish to employ the science of software libraries, components, and generative programming to make implementing such features affordable, so that they become ubiquitous, liberated from their lonely existence in a handful of research systems.

We are working on a library that implements the property models approach in JavaScript, so that it can be run entirely within a web browser. The library is under active development, and its core is already publicly available [4].

2. Background

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

1. The property model is essentially a constraint system: a set of variables and a set of relations that should hold true for those variables. We provide a declarative domain-specific language for specifying property models.
2. A layout is just that: a set of widgets provided by a graphical user interface (GUI) library and their positions. We provide a simple declarative language for specifying layouts, but a layout could just as well be created through calls to a GUI library API or by using a GUI design tool.
3. A binding connects one or more widgets with one or more variables in the property model, and can be one of two kinds: a *view binding*, where a widget is set to display the current value of a variable, or a *control binding*, where user interactions with a widget are translated to requests to change the value of a variable. Often a widget is bound to a variable via both types of bindings; this would be the case, for example, when a textbox that a user can edit is bound to a variable. In our current system, the specifications of bindings are embedded in the layout specification.

To demonstrate how these specifications give rise to a user interface, consider a simple dialog for reserving a hotel room, like the one that appears in Figure 1. The property model for this dialog is shown in Figure 2 and the layout and bindings in Figure 3. The core of these specification languages is described in [9], and we elaborate on extensions in the rest of the paper.

Launching a user interface merely requires passing the above three specifications to a function provided by the property models library, with a call that, omitting some details, looks roughly like this:

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

```
model {
  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("Check-in date should not be in the past.")
    not_in_the_past <== check_checkin(checkin);
    @description("The Check-in date must come before
      the Check-out date.")
    at_least_one_night <== nights > 0;
  }
  output: {
    result <== { checkin: checkin, checkout: checkout };
  }
}
```

Figure 2. The model specification for the dialog in Figure 1. The expressions have access to JavaScript functions available in the runtime environment, e.g., those for manipulating dates.

```
layout {
  text (label : "Check-in", value : checkin);
  text (label : "Check-out", value : checkout);
  text (label : "Nights", value : nights);
  errors ();
  commandButton (label : "Book Now", value : result);
}
```

Figure 3. The layout and bindings specifications for the dialog in Figure 1.

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

We draw attention to the complete absence of event handling logic. All of it is delegated to algorithms in a software library. We have explained how we can generically support value propagation, script recording and playback, widget enablement and disablement, and command activation features [8, 9]. Our prior publications also contain a more introductory explanation of the property models approach. Here, we briefly summarize the approach, so that we can explain how the help features proposed in this paper can be implemented as reusable algorithms.

2.1 Property Model Constraint System

We represent property models as *hierarchical multi-way dataflow constraint systems* [16]. As mentioned above, a constraint system consists of variables and constraints. Abstractly, constraints represent relations among subsets of those variables; *solving* a constraint system means finding a valuation for the system's variables so that all relations in the system are satisfied. A hierarchical constraint system is one where constraints may have different strengths. If not all constraints can be satisfied, partial solutions to hierarchical constraint systems prefer to ignore weaker constraints in favor of enforcing stronger ones.

Formally, a multi-way dataflow constraint system is a tuple $\langle V, C\rangle$, where V is a set of variables and C is a set of constraints. Each constraint in C is a tuple $\langle R, r, M_C\rangle$, where $R \subseteq V$, r is some n-ary relation among variables in R ($n = |R|$), and M_C 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*. Each method in M_C computes values for some subset of R using another subset of R as inputs. In the constraint systems of property models, all variables of R appear as an input or output of each method in M_C, and no two methods in M_C may share the same set of outputs. The programmer must ensure that executing any method of M_C will *enforce* the constraint, that is, satisfy the relation r.

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. One such valuation is obtained if exactly one method from each constraint in C is executed, such that no variable is assigned a value by two methods; and the methods are executed in an order where no variable is assigned a value after another method has already used it as an input. A way to obtain such a valuation is thus characterized by a sequence of methods, often called a *plan*.

A multi-way dataflow constraint system can be represented as an *oriented*, *bipartite* graph $G_c = \langle V + M, E\rangle$. The vertex sets V and M correspond, respectively, to the variables and methods of the system, and the edges in E connect each method to its input and output variables: if $v, u \in V$ and $m \in M$, then 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 the graph of a constraint system formed this way the *constraint graph*.

A plan can also be represented as a 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.

A third graph that is relevant for our purposes is the *evaluation graph*. It is obtained by observing which of its input variables a method needs in order to compute its result—methods use "by-name" parameter passing. If in computing the value of the method m, the value of one of its inputs v is needed, then the edge (v, m) of the solution graph is said to be *relevant*. Otherwise the edge is not relevant. Evaluating every method in a solution graph induces a partition of edges into relevant and not relevant. 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, $G_e = \langle V + M, E_V + E_r\rangle$.

The algorithm that guides how a user interface responds to user interaction is essentially that of maintaining the property model in a state where all constraints are satisfied. When a user changes the value of a user interface element, such as a textbox, and that element is bound to one of a property model's variables, a request to change the value of the variable is sent to the model. When the variable's value changes, some constraints may be no longer satisfied. The system is brought back to a satisfied state by computing a new solution graph, followed by evaluating the constraint satisfaction methods in this graph in a topological order. New values of changed variables are reflected back to the user interface elements bound to those variables.

It is usually the case that the constraint system of a property model is underconstrained. That is, many different solution graphs exist, and each solution graph corresponds to a different direction of flowing data in a user interface. For example, in the dialog for selecting dates for a hotel stay, data could flow from the start date and end date fields to the field showing the number of nights; or it could flow from the start date and the number of nights fields to the field showing the end date.

To select among the many possible plans, we assign a "priority" to each variable, based on the order of updates to the property model's variables. Every time a variable is updated by a client, it is assigned the highest priority among all variables. Priorities are totally ordered. The flow that tends to follow the principle of least surprise for the user is the one that preserves as many of the values of the variables that have been updated most recently, that is, variables with high priorities. We can express an ordering between plans precisely in terms of solution graphs: a solution graph A is better than B if, of those variables that are sources[1] in exactly one of the graphs, the variable with the highest priority is a source in A. This definition induces a total ordering on solution graphs; the best solution graph according to this order is the one selected.

To compute the solution graph, we map the priorities of the variables to *stay constraints* in the underlying constraint system. When enforced, a stay constraint keeps a variable's value unchanged. We give these constraints strengths that respect the priority order of the variables they were derived from. With this mapping, we can apply Zanden's Quickplan algorithm [16] to produce a new solution graph.

The three graphs described above—the constraint graph, solution graph, and evaluation graph—have the following interpretations: the constraint graph represents the dependencies that could be in effect at some point during interaction with a user interface; the solution graph represents the dependencies that could be in effect with the current order of updates to the variables of a property model, for some values of those variables; and the evaluation graph represents the dependencies that are currently in effect for the current order of updates to the variables, for their current values.

Besides the algorithm outlined above for propagating values within user interfaces, other algorithms guide aspects of a user interface's functionality in our system. These algorithms enable and disable widgets and activate and deactivate commands. All these functionalities are based on queries to the three graphs described above. The information that enables the reusable implementations of the help functionalities described in this paper are similarly derived from those three graphs.

3. Dataflow Visualization

There are often dependencies among different values displayed in a user interface. When a value changes, the interface may try to satisfy certain relations in order to preserve the consistency of the data that is displayed. It will react to user edits by changing the values of other widgets, yielding a dataflow. In many cases, values may need to be propagated in different, even opposite directions, depending on what has changed and when. In determining the dataflow, user interfaces should strive for complying with the "principle of least surprise." As explained in Section 2, this means, for example, that the values that the user just finished editing should not be overwritten.

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

Consider a simple dialog for image scaling, such as what appears in Figure 4. It provides two ways of editing the image dimensions—either absolutely, in pixels, or relatively, in

[1] In graph terminology, a vertex with no incoming edges is called a *source*.

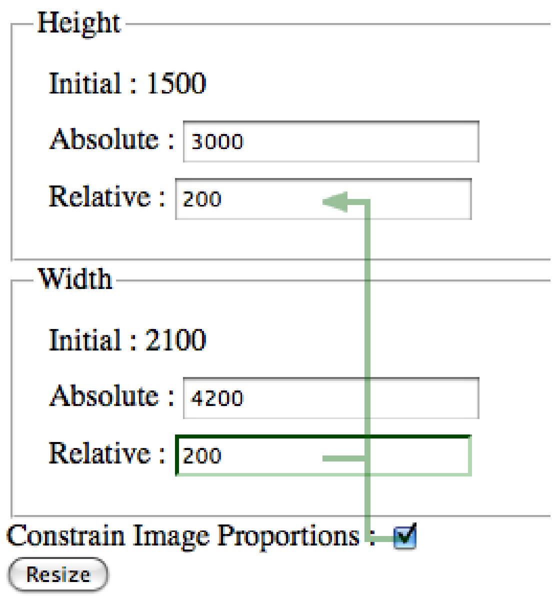


Figure 4. A dialog for resizing an image. The arrow is a visualization of a functional dependency in the dialog's property model.

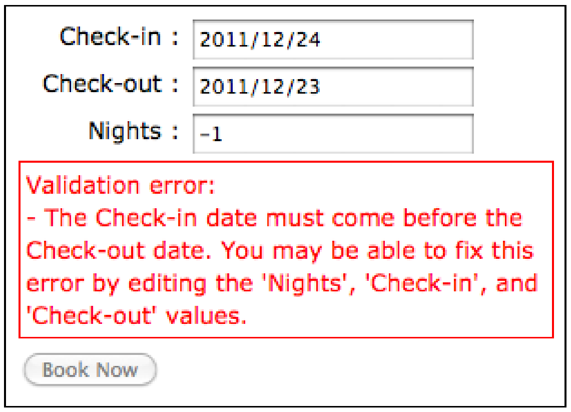
percentages—with the option to preserve the ratio between the image's height and width. The values of all four textboxes and the checkbox are tied together in a complex, multi-way relationship. If the user edits one of the numeric values, the other three could change. To understand this behavior, a user will need to know which values currently affect which others.

In the context of property models, this information is contained in the evaluation graph: a variable's value is affected by its ancestors and affects its descendants. What is needed is an effective means—such as visualization—of communicating this information to the user.

In our implementation, we animated the flow of data among values in the model. For each method in the evaluation graph, an arrow is drawn connecting the widgets bound to its inputs and outputs. (Consequently, this means the arrow might have multiple heads or multiple tails or both.) To prevent confusion, an arrow is drawn only if the outputs had changed as a result of the most recent evaluation. These arrows are displayed, one at a time, according to the order of the methods' evaluation. To avoid interrupting casual use of the interface, the dataflow illustration is triggered only upon the explicit request of the user, indicated by clicking a button.

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

We realize that our first attempt is rudimentary, and after seeing it in action, we identified some potential improvements. In one way, the animation seemed to play too quickly. The time an individual relationship was displayed seemed too short for digesting the information the visualization was trying to convey. In another way, the animation seemed to play too slowly. If we wanted to focus on one particular relationship, which was often the case, then we would have to wait for the animation to get to that point, and then wait for it to complete before we could see it again. The animation could have conformed better to how we wanted to consume the relationship information—typically we were interested in just one or two relationships out of the whole graph.



Validation error:
- The Check-in date must come before the Check-out date. You may be able to fix this error by editing the 'Nights', 'Check-in', and 'Check-out' values.

Figure 5. A dialog for reserving a hotel room, with a disabled command widget and accompanying help text.

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

Finally, a more sophisticated layout algorithm could be used to produce more visually appealing arrows.

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

4. Explaining Command Availability

Whenever a user interface command is unavailable in the current context, most rich user interfaces will prohibit its execution by deactivating the corresponding command widget that invokes it. Visually, deactivation can take several forms, including "graying out" or hiding the widget. Human interface guidelines for several popular platforms recommend this behavior and alternatively describe widgets that are not enabled as "dimmed" [1], "disabled" [3, 11], or "insensitive" [6].

We have previously published an algorithm for property models that can automatically determine when a command widget should be deactivated according to the above guidelines [9]. A command becomes unavailable only in certain contexts, and to describe those contexts is to describe preconditions for the command. In a property model, the preconditions for a command are defined by the programmer with the help of *invariants*. Each of these special variables hold the result of a boolean expression that is true when the precondition is satisfied and false otherwise. Returning to our earlier example, a command that reserves a hotel room may require a positive number of nights, as in Figure 5. This precondition is expressed in the `at_least_one_night` invariant in the property model in Figure 2.

Continuing, a command widget launches some command in the program, and takes the command's parameters from the variable to which it is bound. In our example dialog, the specification in

Figure 3 binds the Book Now widget to the result variable. The value of this variable is derived from the two variables checkin and checkout, as can be seen from the model specification in Figure 2.

When an invariant variable evaluates to false, i.e., when a precondition fails, then the variables that contributed to that violation are marked, or "blamed," for being responsible. If any blamed variable is also used to evaluate a command's parameters, i.e., if in the evaluation graph a failed precondition and a command share a common dependency, then the command is deactivated.

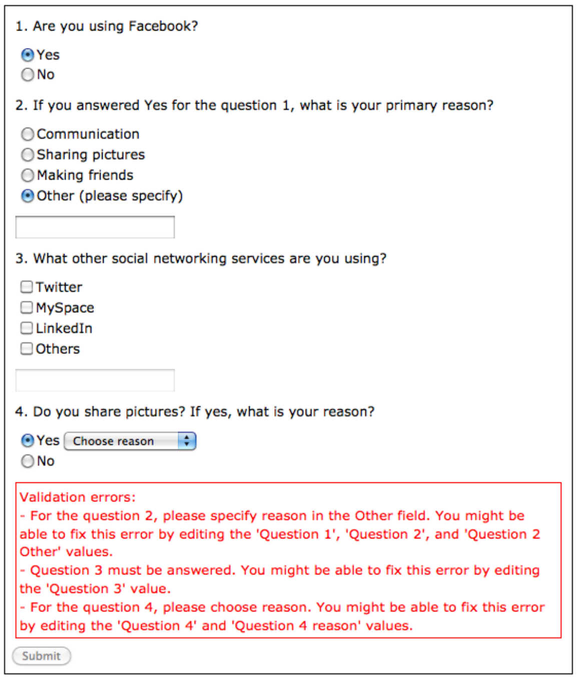
From the perspective of the user, a couple of issues surround the behavior described above. First, a user may not understand why a particular command widget is deactivated. As stated above, programmers can express these reasons through invariants in the property model. However, since they appear only in the user interface code, the user may not know what they are and, consequently, have a limited understanding why a command widget is deactivated. A recent experience of one of the authors illustrates this case. In trying to change a password to a web system, an error message kept repeating simply that the new password entered did not satisfy the requirements of a valid password. The requirements were mostly revealed through trial and error. We can reasonably expect that a large portion of the users of that system will not succeed in the same task on the first, second, or even third try.

To alleviate the above problem, we can automatically generate help text that describes the reasons why a command widget is deactivated. If the programmer adds a natural language description of the precondition to the invariant variable, then we can present it to the user as an explanation for a deactivated command widget. Examples of such annotations are the @description strings in the property model specification of Figure 2. The explanation could be included with other contextual information, such as the dataflow visualization described in Section 3.

The second problem is that even after reading an explanation, a user may not know the actions necessary to re-activate a command widget. Using the property model, we can find which variables are responsible (to varying degrees) for the failed precondition. We can then direct the user to the widgets bound to those variables and expect that they can deduce how to interact with them to satisfy the precondition. In our hotel reservation example from Figure 5, we inform the user that changing any of the check-in, check-out, or nights values may be able to resolve the error. To determine the responsible variables, we look at the invariant representing the failed precondition and take its ancestors in the evaluation graph, i.e., the variables that contributed to its false value. We find the interactive widgets that are bound to those variables, then reference their labels in the help text.

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

The dependencies exhibited in the hotel dialog are simple, and a user likely does not have trouble finding ways to satisfy the command's preconditions. We use that example to explain the mechanism of generating help texts, rather than to advocate its usefulness. In larger dialogs and forms, dependencies easily get more complicated, and thus accurate help text are of greater importance. Figure 6 shows a (still relatively simple) survey form with more dependencies, implemented with our system. Many preconditions are


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 reason
No

Validation errors:
- For the question 2, please specify reason in the Other field. You might be able to fix this error by editing the 'Question 1', 'Question 2', and 'Question 2 Other' values.
- Question 3 must be answered. You might be able to fix this error by editing the 'Question 3' value.
- For the question 4, please choose reason. You might be able to fix this error by editing the 'Question 4' and 'Question 4 reason' values.

Submit

Figure 6. A user survey form demonstrating preconditions alongside more complex value relationships.

violated, and the generated help messages accurately identify the variables responsible for each violation.

It is important to note that the generation of help text is completely orthogonal to and works in harmony with the dataflow visualization described in Section 3. After marking values that are violating the precondition, the user can see the network of relationships among them to better determine a root cause.

5. Pinning

As is evident by now, a rich user interface may automatically, without consent from the user, change values in order to enforce relationships among them. Even though reasonable heuristics are applied, the behavior may be unexpected to the user. In our running example, the hotel reservation dialog from Figure 1, consider this sequence of user actions:

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

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

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

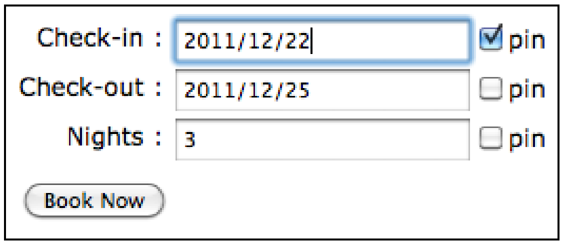


Figure 7. A dialog for reserving a hotel room, with a pinned value.

user interface should thus provide means to the user to control the preferred dataflow.

We provide a feature that allows the user to "protect," or *pin*, certain values as he moves along. Pinning does not prevent the user from further editing the value. It simply guarantees that the system will not automatically change the pinned value as a reaction to the user changing some other value. Figure 7 shows how pinning is offered in our system for the familiar hotel reservation example.

Implementing pinning translates naturally to property models. As we explained in Section 2, a stay constraint keeps a variable's value unchanged. The desired effect is thus attained if the stay constraint of the pinned variable is promoted to the same strength as that of the programmer-defined constraints. This guarantees that the stay constraint will be enforced in all solution graphs, and thus the pinned variable is not overwritten by any method.

Pinning is not without complications, though. Pinning a variable expands the set of constraints that must be enforced in each plan by adding the variable's stay constraint. If enough variables are pinned, the property model could become overconstrained, leaving it unsolvable. To prevent this, we can disable the pinning option for widgets bound to variables that, if pinned, would result in an overconstrained system. Identifying a variable as "pinnable" is straightforward: a single run of the constraint solver suffices to determine if the system can still be solved after a particular variable is pinned. Furthermore, after pinning a variable, it may be that some other variables are *derived* in all possible plans. Any user edits to such values will be overwritten. To prevent confusion, widgets bound to such variables should be disabled. We can identify these variables as well: after removing from constraints any methods that write to pinned variables, any variables written by a method in a constraint with no other methods will be derived in all solutions.

6. Role of Property Models in Implementation

We emphasize that the visual presentations of the three features discussed above are divorced from their underlying implementations. User interface developers are free to integrate them with existing interfaces however they see fit. For example, instead of using checkboxes labeled "pin", which may suggest to some users that pinning will "lock in" a value, protecting it even from user edits, a more user-friendly presentation could be to use a star-shaped checkbox, suggesting that pinning simply indicates high-priority, "favorite" values. Developers can experiment to find which visualization methods work best. Our intent is merely to provide the non-trivial framework to support them.

7. Related work

Features related to the three help mechanisms we have presented have been incorporated into various systems developed by the CHI community. We discuss a few representatives below. We are not aware of other work, however, where the main focus is on a generic mechanism that supports implementing a large selection of such features alongside each other.

UIDE [5, 14] and HUMANOID [12] use preconditions on commands to disable widgets and to generate helpful explanations. UIDE also uses postconditions to explain how to enable a command widget [2, 13, 15]. The authors acknowledge that their system is not prepared to handle situations with complex, multi-way dependencies among actions and widgets; such dependencies are supported in an interface built on property models. Unlike UIDE, our generated help text does not attempt to provide a precise sequence of interactions for re-activating a command widget (in some cases, there may be many sequences to choose from), and we have not investigated its (in)feasibility. Further, we try not to burden the programmer with specifying postconditions for user interactions. We believe that the information available in a property model is enough to provide sufficient help.

The Heracles system supported pinning of its controls [10]. In Heracles, not all constraints are enforced after each user edit, and multi-way constraints are not supported. Thus, some of the issues facing pinning in property models are avoided. Heracles also opts to pin a variable automatically upon user edit, whereas we pin a variable upon explicit request only. The authors of Heracles, like us, cite potentially unclear dataflow and overwritten values as motivations for pinning.

8. Conclusion

This paper explains how three different user interface features, aimed at improving the user's experience when interacting with a computer, can be implemented generically—to provide any or all of the features within some user interface requires no code specific to the user interface. The adoption cost of these features is thus very small.

To emphasize the importance of the low cost of adoption, we concentrate on one of the three features. In Section 4 we describe the automatic generation of helpful explanations for why a command widget in a user interface is deactivated. We gave an example where a user had to, through trial and error, "fight" a system in accomplishing a password change task that would have been trivial had the system given adequate error diagnostics. The account holders of that in-house system are counted in thousands, and each has to go through the procedure periodically. Hence, collectively the time wasted by the users likely far surpasses the time it would have taken for the developer to program that help feature. Still the economics of programming did not make it worthwhile to do so. That particular help feature likely was not key during specifying the requirements or defining the deliverables, and at development time, the programmer(s) may have had other feature requests with higher priorities. On the other hand, we as users shrug off small frustrations with computer systems, once we find a way around them, and do not think twice about it. It is of no user's interest to spend much time to find out how one could get a small defect fixed or missing feature added.

We want to cast blame on neither the user nor the programmer—the incentives to take action to rectify the state of affairs is too small for both. The property models approach scales the incentives to cover a significantly larger class of user interfaces.

References

[1] Apple. Apple human interface guidelines. `http://developer.apple.com/library/mac/documentation/UserExperience/Conceptual/AppleHIGuidelines/index.html`, May 2011.

[2] J. J. de Graaff, P. Sukaviriya, and C. van der Mast. Automatic generation of context-sensitive textual help. Technical Report GIT-GVU-93-11, Georgia Institute of Technology, Apr. 1993.

[3] Eclipse. User interface guidelines. `http://wiki.eclipse.org/User_Interface_Guidelines`, May 2011.

[4] J. Freeman, W. Kim, and J. Järvi. Hotdrink. URL `http://code.google.com/p/hotdrink/`.

[5] D. F. Gieskens and J. D. Foley. Controlling user interface objects through pre- and postconditions. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '92, pages 189–194, New York, NY, USA, 1992. ACM. ISBN 0-89791-513-5.

[6] GNOME. GNOME human interface guidelines. `http://developer.gnome.org/hig-book/stable/`, May 2011.

[7] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. *Human-Computer Interaction*, 1:311–338, December 1985. ISSN 0737-0024.

[8] J. Järvi, M. Marcus, S. Parent, J. Freeman, and 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, pages 89–98, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-267-2.

[9] J. Järvi, M. Marcus, S. Parent, J. Freeman, and J. N. Smith. Algorithms for user interfaces. In *Proceedings of the eighth international conference on Generative programming and component engineering*, GPCE '09, pages 147–156, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-494-2.

[10] C. A. Knoblock, S. Minton, J. L. Ambite, M. Muslea, J. Oh, and M. Frank. Mixed-initiative, multi-source information assistants. In *Proceedings of the 10th international conference on World Wide Web*, WWW '01, pages 697–707, New York, NY, USA, 2001. ACM. ISBN 1-58113-348-0.

[11] Microsoft. Windows user experience interaction guidelines. `http://msdn2.microsoft.com/en-us/library/aa511258.aspx`, May 2011.

[12] R. Moriyon, P. Szekely, and 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, pages 225–231, New York, NY, USA, 1994. ACM. ISBN 0-89791-650-6.

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

[14] P. Sukaviriya and 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, pages 152–166, New York, NY, USA, 1990. ACM. ISBN 0-89791-410-4.

[15] P. N. Sukaviriya, J. Muthukumarasamy, A. Spaans, and 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, pages 44–52, New York, NY, USA, 1994. ACM. ISBN 0-89791-733-2.

[16] B. V. Zanden. An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Trans. Program. Lang. Syst.*, 18 (1):30–72, 1996.