# Algorithms for User Interfaces

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

Mat Marcus
mmarcus@emarcus.org

Sean Parent
Adobe Systems Inc.
sparent@adobe.com

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

Jacob Smith
Texas A&M University
jnsmith@cse.tamu.edu

## Abstract

User interfaces for modern applications must support a rich set of interactive features. It is commonplace to find applications with dependencies between values manipulated by user interface elements, conditionally enabled controls, and script record-ability and playback against different documents. A significant fraction of the application programming effort is devoted to implementing such functionality, and the resulting code is typically not reusable.

This paper extends our "property models" approach to programming user interfaces. Property models allow a large part of the functionality of a user interface to be implemented in reusable libraries, reducing application specific code to a set of declarative rules. We describe how, as a by-product of computations that maintain the values of user interface elements, property models obtain accurate information of the currently active dependencies among those elements. This information enables further expanding the class of user interface functionality that we can encode as generic algorithms. In particular, we describe automating the decisions for the enablement of user interface widgets and activation of command widgets. Failing to disable or deactivate widgets correctly is a common source of user-interface defects, which our approach largely removes.

We report on the increased reuse, reduced defect rates, and improved user interface design turnarounds in a commercial software development effort as a result of adopting our approach.

***Categories and Subject Descriptors*** D.2.2 [*Design Tools and Techniques*]: User interfaces; D.2.13 [*Reusable Software*]: Reuse models

***General Terms*** Algorithms, Design

***Keywords*** Software reuse, Component software, User interfaces, Declarative specifications, Constraint systems

## 1. Introduction

The role of a user interface, such as a dialog window, can be summarized as supporting the user in selecting valid values for a command or function to be executed in a program. In modern applications this support may mean, for example, computing values of some user interface elements automatically when values of other elements change, storing and retrieving default values, capturing user actions into a replayable script, undo and redo functionality, disabling user interface elements when their values are irrelevant for a final result, etc. This list is long—it is no small task for programmers to implement high-quality user interfaces.

In the prevailing approach to programming graphical user interfaces (GUIs), one of many GUI frameworks [6, 17, 31] provides a selection of widgets as reusable software components, and the programmer implements a user interface as a composition of widgets by specifying the interactions between the components. The interactions are typically expressed using imperative object-oriented code placed in event handlers. Even in user interfaces with relatively simple functionality, interactions between components are often surprisingly complex. Consequently, the event-handling logic that expresses the interactions is similarly complex, often scattered to many locations in the program, and seldom reusable across user interfaces. It is thus not surprising that user interface code can account for 30–50% of applications' code [21, 25], and a disproportionately higher share of the reported defects [25].

We have previously introduced *property models* [13], an approach to explicitly model many commonalities in the behavior of a class of typical user interfaces, such as dialog windows. We showed how an algorithm for computing new values of user interface elements after changing values of other elements and an algorithm for script recording and playback can be reused across user interfaces. These algorithms are generic, parametrized by a (declaratively specified) model that represents the variables manipulated by a user interface and the functional dependencies between those variables. We suggested that where property models can be applied, the amount of code is notably reduced and software quality improves compared to using a traditional GUI framework.

This paper develops the property models approach further. We focus on how to obtain, alongside computing updated values for user interface elements, accurate information of which functional dependencies between user interface elements were active in computing those values. Besides showing how to compute this information we explain how it enables further user interface functionality to be encoded as reusable algorithms. In particular, we show how this information gives the means for algorithms for the enablement and disablement of widgets when they are not relevant to the result of a user interface, and activation and de-activation of command widgets when the current result of a user interface does not satisfy stated conditions.


GPCE '09 http://doi.acm.org/10.1145/1621607.1621630
*GPCE'09,* October 4–5, 2009, Denver, Colorado, USA.

We report on an experiment that quantifies some of the benefits of property models: the experiment shows an increase in programmer productivity and a significant reduction of defects in developing user interfaces following the adoption of our approach. The experimental results reported are obtained using the Adobe Software Technology Labs' (STLab) open-sourced *property model* library [24]. We are actively working on the system so, at times, the discussion in this paper diverges from the currently deployed property model library, reflecting instead the state of our research prototype system. This system and other artifacts related to this research are available for download [23].

To sum up the introduction, we emphasize an essential characteristic of property models, distinguishing them from contemporary GUI frameworks. Let us refer to some piece of state of a user interface element, such as its value or enablement state, as an *attribute*. As described above, in traditional GUI programming the programmer writes procedures that explicitly change the state of a set of attributes. Some GUI frameworks [20, 18, 22] allow the programmer to register functional dependencies between attributes which the framework maintains automatically. Regardless of the method of expression, the result is a network of functional dependencies between attributes and a machine that governs when and which of these functional dependencies are applied. The state of the machine the programmer can observe is the current values of the individual attributes. The state of a user interface, however, consists of more than the combined states of its individual attributes. In particular, it includes information on which functional dependency was applied to obtain the current value of each attribute. Information obtained from this "network state" is crucial for reusable algorithms for user interfaces, but it is not available in contemporary GUI frameworks. A property model is an explicit model of dependencies between parameters to a command. In particular, it maintains which dependencies are currently in effect in the model. The model does not manage individual attributes of user interface widgets, but user interface widgets can be bound to a property model's parameters: the state of the property model then provides a model for part of the network state of a user interface as well. This information is what enables reusable algorithms that implement user interface functionality. The rest of the paper expands on this observation.

## 2. Background

A large class of user interfaces perform the function of *command parameter synthesis* [13]: a user interface assists a user in producing valid parameters to some command to be executed in a program. Often these interfaces allow the user to manipulate a (larger) set of variables from which a (smaller) set of command parameters are computed. Typical examples of user interfaces performing command parameter synthesis are a "save as" dialog, an image resize dialog, toolbars and palettes that issue commands, etc. These interfaces could support more complex behavior, such as script recording and playback. Property models target this class of user interfaces. The goal is that large parts of the their functionality can be implemented as reusable algorithms, parametrized by a specification of the space of variables involved in command parameter synthesis and the dependencies among these variables.

As an example, consider a dialog for saving an image file, like the one that appears in Figure 1. It consists of a text field for entering a file name, a menu of file types, and sliders providing two different avenues for the user to configure compression when saving in a format that supports it. The slider values are tied together by some relationship expressing the trade-off between compression ratio and image quality, each on a scale from 1–100%, the details of which are irrelevant to this example.

The dialog assists the user in synthesizing the parameters for a file save command. Figure 2 shows a specification of a property

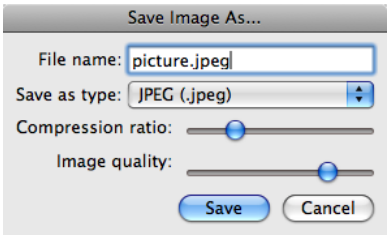


**Figure 1.** A dialog for saving an image file.

```
sheet 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 ≠ "";
}
```

**Figure 2.** A declarative specification, in the Adam language, of the property model for the dialog in Figure 1.

model for this command parameter synthesis task written in the domain specific declarative language *Adam* [24], part of Adobe's property model library. Briefly, the **interface**, **output**, and **invariant** sections declare the variables, or *properties*, of the property model. *Interface variables* can be updated by a client of the property model, e.g., as a result of a user interacting with a user interface widget. The values of *output variables* constitute the result of command parameter synthesis. The (Boolean) value of *invariant variables* indicates whether a set of variables satisfies a stated condition. The **logic** section defines the dependencies and computational rules between variables. The language for these computations is the ASL expression language, which can make calls to registered external (C++) functions, and can thus perform arbitrary actions. The task of the application programmer is to define these computations, we call them *methods*, but when and which of them are executed is controlled by the property model.

Though orthogonal to property models specifications, a complementary language, named Eve, can be used to specify the layout and presentation qualities of interface elements, as well as bindings between widgets in the user interface and values in the property model. The layout specification for the dialog in Figure 1 appears in Figure 3. We discuss the bindings established by this specification in Section 3. For further details on the Adam and Eve languages, we point the reader to their programmer documentation [2].

To demonstrate how straightforward implementing user interfaces for command parameter synthesis can be with property models, we discuss the simplified version of the "Modal Dialog Interface Kit API" [2]. This API provides a single function that brings up a dialog and manages it with the assistance of the property model

```
layout save_image_file {
  view dialog(name: "Save Image As...",
    placement: place_column,
    spacing: 6,
    child_horizontal: align_fill) {
    edit_text(bind: @file_name, name: "File name:");
    popup(bind: @file_type, name: "Save as type:", items: [
      { name: "Windows bitmap (.bmp)", value: "bmp" },
      { name: "JPEG (.jpeg)", value: "jpeg" }
    ]);
    row() {
      column(child_horizontal: align_right) {
        label(name: "Compression ratio:");
        label(name: "Image quality:");
      }
      column(horizontal: align_fill, child_horizontal: align_fill) {
        slider(bind: @compression_ratio,
            format: {first: 1, last: 100, interval: 1} );
        slider(bind: @image_quality,
            format: {first: 1, last: 100, interval: 1} );
      }
    }
    row(horizontal: align_right) {
      button(name: "Save", action: @ok, bind: @result,
        default: true);
      button(name: "Cancel", action: @cancel);
}}}
```

**Figure 3.** A declarative specification, in the Eve language, of the widget layout for the dialog in Figure 1.

library, as well as a library for automatic layout of user interface elements, and a library of user interface widget components.

```
dialog_result_t handle_dialog(
                  const dictionary_t& input_variables,
                  istream& layout_definition,
                  istream&  property_model_definition)
```

The **dictionary_t** type is an associative array data structure. The **input_variables** dictionary contains the contextual information to populate the property model's input variables. The two input streams supply the declarative layout and property model specifications: the dialog's presence and behavior (validating user input, initializing and closing the dialog, computing values shown in the dialog's widgets, disabling and enabling widgets, and recording and playing back scripts) are entirely effected by libraries, governed by the two specifications. When the dialog is dismissed, the result is returned in a **dialog_result_t** structure. This structure contains, among other things, the parameters to be supplied to a command in a program (output variables of the property model) and the action that terminated the dialog (such as "Save" or "Cancel").

## 3. Widget Enablement with Property Models

We suggest in Section 1 that the currently active functional dependencies between a property model's variables (that reflect a part of the "network state" of a user interface) is crucial input for algorithms that implement user interface functionality. We describe how this information can be used to derive a generic widget enablement algorithm in Section 5; here we describe the intuitive reasons behind widget enablement decisions, and discuss an example dialog implemented using property models that enables and disables widgets in accordance with those reasons. We note that neither the property model nor layout specifications of the dialog require any code that pertains to enablement.

In mainstream user interface programming, programmers explicitly express the conditions under which a particular widget should be disabled. This requires complex logic; in our experience user interfaces that disable widgets incorrectly, or do not implement widget enablement at all, are frequent. Programmers often follow "rules of thumb" offered by user interface guidelines for the deployment platform [3, 15, 1] to guide the decisions on when to enable and disable widgets. The advice may differ between the guides, but certain universally accepted reasons can be identified. To discuss these reasons, we distinguish between enablement of widgets that launch commands (such as command buttons) and enablement of widgets that allow users to interactively edit the widgets' value (such as text and list boxes, and radio buttons). Although these behaviors fall under the general scope of enablement, the reasons for decisions to enable or disable a widget are unrelated, and the mechanisms governing those decisions completely different. We refer to the former mechanism as *command activation*, and the latter as *widget enablement*. We discuss each of these in turn below.

### 3.1 Command activation

The result of command parameter synthesis is a set of parameters, possibly subject to preconditions, for a command. Ideally, a user interface avoids constructing command objects from parameters that do not satisfy the preconditions. A user interface for command parameter synthesis may thus be expected to provide a "latch" that controls when a command object can be constructed. The canonical form of such a latch is activating and deactivating the "OK" button of a dialog. Consider the example from Section 2. The dialog in Figure 1 for saving an image will launch a command to write a file on a disk when the "Save" button is clicked, but it may be desirable to keep this button inactive if no file name has been given. Alternatively, the button could be kept activated, but an error diagnostic is shown to the user if it is clicked. Property model engines are neutral on these matters; the behavior is a policy decision made in the architectural layers that sit on top of the property model engine.

Section 5.1 describes how a property model component computes the information that can guide activation decisions; here, we explain the reasoning with an example. The property model specification in Figure 2 contains two items noteworthy for command activation. First, the **result** variable in the **output** section holds the command arguments. Second, the **check_name** variable in the **invariant** section specifies a requirement for when the **file_name** variable is valid. To reflect the state of a property model in a user interface, widgets, like "Save" and "Cancel," are be bound to a property model's variables. Such bindings are established, for example, in an Eve layout specification using the **bind** attribute. Relevant for the discussed example, the specification in Figure 3 binds the "Save" command button to the **result** variable and the "File name" text box to the **file_name** variable.

Assuming now the "File name" text box is empty, the following reasoning by the property model and layout libraries justifies deactivating the "Save" button: (1) due to the binding established for the "File name" widget, the value of the **file_name** variable is the empty string; (2) the **check_name** invariant is not satisfied; (3) the **file_name** variable is considered to have an invalid value; (4) the output variable **result** depends on **file_name**, and it too is thus considered to have an invalid value; and (5) the "Save" command button, since it is bound to **result**, should be deactivated.

### 3.2 Widget enablement

The intuitive reason for disabling a widget is when the value of the widget cannot affect the output of command parameter synthesis. Consider again the image file save dialog. Our simplified example

offers two possible choices for the file type: BMP and JPEG. The former file type does not support lossy compression, whereas the latter does. The values of the slider widgets labeled "compression ratio" and "image quality" are involved solely in compression, and can thus be ignored (and disabled) when the chosen file type is not JPEG.

The generic mechanism for detecting when a value of a user interface element may affect the output of command parameter synthesis is the subject of Section 5.2. Here, we describe the reasoning performed by the property model library in the image file save example.

The Eve specification in Figure 3 binds the "Compression ratio" slider widget to the property model's **compression_ratio** variable, and "Image quality" widget to the **image_quality** variable. Further, the "Save as type" dropdown widget is bound to the **file_type** variable. The expression defined to compute the value of the **result** variable only uses the **compression_ratio** variable when the **file_type** variable has value **"jpeg"**. Therefore, a value other than **"jpeg"** in the "Save as type" dropdown widget's menu will not exhibit a dependency from **compression_ratio** to **result**, and thus not from **image_quality** either. This manifests as disabling the two slider widgets bound to the **compression_ratio** and **image_quality** variables.

# 4. Dependency Information in Property Models

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 repeat the point that user interface algorithms need access to information regarding which dependencies were active in arriving at the current state of the user interface. Below we describe how we can represent this network of dependencies as a concrete data structure. In particular, we summarize how the network of dependencies between variables gives rise to a constraint system, concretely represented as a *constraint graph*. We then remind how to obtain a candidate set for the currently active dependencies: this set is represented by a *solution graph*, a subgraph of the constraint graph. Finally, we define how the methods of the solution graph are evaluated to give a new valuation for the variables in the model. Evaluating the methods 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. Command activation and widget enablement decisions need to consult all of these representations. 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 $G_e$).

Our earlier work [13] described the constraint system representation of property models, including computing solution graphs based on prioritizing the variables of a property model according to how recently they have been updated by a user. Therefore, these topics are presented very briefly in Sections 4.1–4.3. We do, however, define the constraint system at hand more precisely, and describe its properties that are necessary for justifying widget enablement and command activation decisions. The operational semantics of evaluation within the solution graph and the formation of the evaluation graph are covered in depth.

## 4.1 Multi-way dataflow constraint systems

We represent the property model's network of functional dependencies as a particular kind of a *multi-way dataflow constraint system* [35]. We remind that a constraint system $S$ is a tuple $S = \langle V, C\rangle$, where $V$ is a set of variables, each having a *current value*, 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*. Executing any method $m$ in $M$ *enforces* the constraint by computing values for some subset of $R$, using another disjoint subset of $R$ as inputs, such that the relation $r$ becomes satisfied. We refer to the input and output variables of a method $m$ as $ins(m)$ and $outs(m)$, respectively. The code realizing a method is considered a "black box"—it is the programmer's responsibility to ensure that a constraint is satisfied when any of its methods is executed.

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. Each constraint can be satisfied independently by executing one of its methods. However, to ensure consistency over the whole system, methods should be chosen and executed in an order such that once a variable has been read from or written to by one method, no other method will write to it. An order of methods satisfying these conditions is called a *plan*. Depending on the problem, a plan may or may not exist, and may or may not be unique. If a plan exists for a given constraint system, we say the system is *satisfiable*; otherwise it is *over-constrained*. If more than one plan exists, the system is *under-constrained*.

We place four well-formedness conditions on the constraint systems for property models (the $\sqcup$ symbol denotes disjoint union): (WF-1) for all constraints $\langle R, r, M\rangle$ in $C$, for all methods $m$ in $M$, $ins(m) \sqcup outs(m) = R$; (WF-2) for all methods $m$ in $S$, $outs(m) \neq \emptyset$; (WF-3) for any two constraints $\langle R_1, r_1, M_1\rangle$ and $\langle R_2, r_2, M_2\rangle$ in $C$, $R_1 \neq R_2$; and (WF-4) for all constraints $\langle R, r, M\rangle$ in $C$, for any two methods $m_1, m_2$ in $M$, $outs(m_1) \not\subseteq outs(m_2)$. All of these conditions can be easily checked so that models violating the conditions can be rejected, and easily satisfied so that the conditions are no real restrictions on what programmers can express. The condition WF-1 is known as *method restriction* [29, p. 56], and it guarantees that a multi-way constraint system can be solved or found unsolvable in polynomial time [32] with respect to the number of constraints. Methods that violate WF-2 are unnecessary, as they never affect the valuation of the system's variables. WF-3 disallows two constraints that define a relation between the exact same set of variables; no system violating WF-3 (but respecting WF-1 and WF-2) is satisfiable. WF-4 rules out constraints for which there might be no criteria to favor one method over another. This condition is important to guarantee the uniqueness of plans in our constraint systems.

## 4.2 Constraint graph

A well-formed multi-way dataflow constraint system is in a one-to-one correspondence with an *oriented*, *bipartite* graph $G_c = \langle V + M, E\rangle$, with vertex sets $V$ and $M$ representing the variables and methods of the system, respectively, and $E$ the directed edges that connect each method to its input and output variables. An example constraint graph appears in Figure 4(a). Where $v, u \in V$ 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 $u$. The graph is oriented, that is, $(a, b) \in E \implies (b, a) \notin E$, because for each method $m$, $ins(m)$ and $outs(m)$ are disjoint.

The grouping of methods and variables into constraints is implicit in the representation $G_c = \langle V + M, E\rangle$. To explain, we use the notion of a *neighborhood* ($nbh$) of a vertex: in a graph $\langle V, E\rangle$, $nbh(a) = \{b \mid (a, b) \in E \text{ or } (b, a) \in E\}$. Now let $\sim_{nbh}$ be the equivalence kernel of $nbh$, defined by $m_1 \sim_{nbh} m_2 \Leftrightarrow nbh(m_1) = nbh(m_2)$. Assuming method restriction (WF-1), all method vertices of the same constraint belong to the same equivalence class in the quotient set $M/\sim_{nbh}$. That is, two methods $m_1$ and $m_2$ belong to the same constraint if $[m_1]_{nbh} = [m_2]_{nbh}$. Furthermore, by WF-2, $[m_1]_{nbh} \neq [m_2]_{nbh}$ implies that $m_1$ and $m_2$ are methods from two different constraints. In the following, when there is no fear of confusion, we omit the subscript $\cdot_{nbh}$ and just

write $[\cdot]$ and $\sim$. For instance, in Figure 4(a), the methods $m_1$ and $m_2$ are in the same neighborhood ($[m_1] = [m_2] = \{q, c\}$).

### 4.3 Solution graph

A plan for a constraint system can be explicitly represented as a subgraph of the constraint graph, called a *solution graph*. We use the notation $G[V]$ to indicate the *vertex-induced* subgraph of $G$: if $V$ is a subset of $G$'s vertex set, $G[V]$ is the graph whose vertex set is $V$ and the edge set includes all edges of $G$ with both endpoints in $V$.

**Definition 1** (Solution graph)**.** Let $G_c = \langle V + M, E\rangle$ be a constraint graph. Let $M' \subseteq M$. $G_s = G_c[V + M']$ is a *solution graph* of $G_c$ iff (1) $G_s$ is *acyclic*, (2) $\{[m] \mid m \in M'\} = M/\!\sim$, (3) $|M'| = |M/\!\sim|$, and (4) $\forall v \in V.\ \textit{in-degree}(v) \leq 1$.

The second and third conditions together establish that $M'$ contains exactly one method from each constraint, and the fourth that no two methods output to the same variable. The third condition is implied by the first and the fourth because of WF-3. The solution graph for the constraint graph in Figure 4(a) appears in Figure 4(b).

As explained in [9], to deal with the under-constrained constraint systems that arise from user interfaces we employ *constraint hierarchies* and *stay constraints* [4]. Each variable in the system is given a stay constraint, represented by a double-box 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 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, intuitively, the best system is the one that retracts the fewest and weakest constraints. The "locally-predicate-better" criterion [10] 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 to indicate that no solution can retract them. An *admissible solution* is one that enforces all constraints with the strength *must*. 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 effects 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*.

Insisting on a total order of stay constraints guarantees that the best satisfiable constraint system that has an admissible solution is unique. A stronger argument is that the solution to this best system is unique. This is the case if we start from a constraint system that satisfies the well-formedness conditions defined in Section 4.1, augment it with stay constraints for each variable (that does not have one already), and assign strengths as described above. We omit the proofs here and instead refer to an accompanying technical report [9] for a detailed account of the properties of the constraint systems of property models.

We call the unique solution graph of the best satisfiable constraint system that has an admissible solution the *most preferred solution graph*. The uniqueness of the most preferred solution graph is a crucial property for the predictability of user interfaces based on property models, and a necessary prerequisite for, e.g., an algorithm for widget enablement: if the flow of data could not be uniquely determined in each state of the user interface, there would be little hope to determine how editing particular values might affect the outputs of command parameter synthesis, or what kind of changes to the current solution graph they might cause.

As explained in [13], we employ a derivative of Zanden's Quickplan algorithm [35] 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 [9].

### 4.4 Evaluation Graph

The set of method vertices in a solution graph is a plan of a constraint system. 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. As explained above, besides the new valuation, this "execution phase" produces the evaluation graph that contains exactly the functional dependencies between variables that are active for the current priority order and variable valuation. This information is necessary for enablement and other algorithms.

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 shows one example each of a constraint, solution, and evaluation graph. The constraint graph represents all the possible functional dependencies that may be in effect during some state of the property model. The solution graph represents the functional dependencies that may be in effect for a particular priority ordering of the variables. The evaluation graph represents the functional dependencies that are in effect for a particular priority ordering and valuation of the variables. Below we focus on the execution phase of property model constraint systems that produces the evaluation graph.

As described above, a property model component maintains a valuation of variables such that certain constraints are satisfied; a priority ordering among the variables; and information on what dependencies between the variables are active. The basic requests to a property model are to assign one of its values and/or to change the priority of one of the variables. The property model component reacts to these queries by computing a new valuation of its variables. Simplifying some, these requests arise as follows. In a user interface that is a client of a property model component, some number of user interface widgets are bound to specific variables in the model and reflect their values to a user. Whenever a user edits the value of a widget, the same event handler code is executed as a response. This code requests that the property model assigns the new value to the variable to which the widget is bound, and notifies all other widgets to update themselves. The event handlers dealing with the

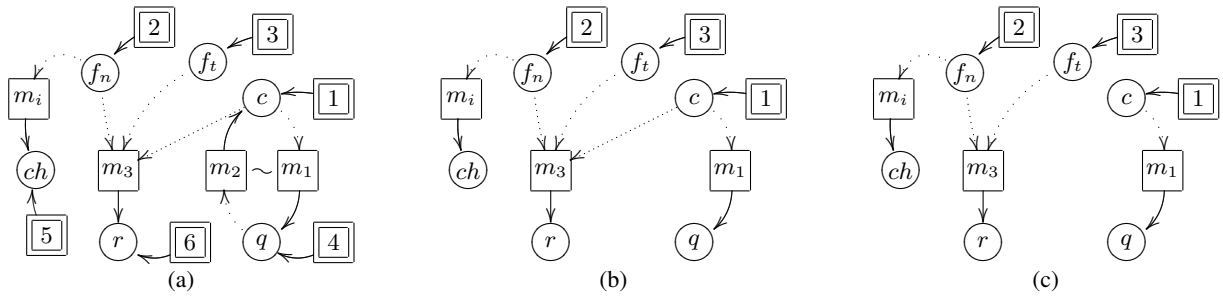


**Figure 4.** The constraint graph (a), a solution graph (b), and an evaluation graph (c) for the property model described in Figure 2. In all of the graphs, the variable $r$ is the variable **result** in Figure 2; likewise, $q$ is **quality**, $c$ **compression_ratio**, $f_n$ **file_name**, $f_t$ **file_type**, and $ch$ **check_name**. The **relate** clause in Figure 2 gives rise to the constraint consisting of the methods $m_1$ and $m_2$, and the method in the **output** section of Figure 2 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).

update notifications all perform the same task: request the (possibly new) value for a variable from the property model component and display the value in the widget. Call the above two requests set and get.

Consider a constraint graph $G_c = \langle V + M, E \rangle$. We can define the state of a property model, the *current configuration*, as a tuple $C = \langle G_s, s, \nu \rangle$, where $G_s$ is a solution graph of $G_c$, $s$ is a strength assignment (defines the priority order among the variables), and $\nu$ a valuation of variables in $V$ and edges in $E$. The valuation $\nu$ 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, $\nu$ 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, $\nu$ 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 $\nu$, except that the variable $v$ maps to *val*; the analogous notation applies for edges.

Assuming a constraint graph $G_c = \langle V + M, E \rangle$, and 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 [9]. If necessary, the solver algorithm is run to produce a new solution graph $G'_s$, otherwise $G'_s = G_s$.
3. A new valuation $\nu'$ is computed from $\nu$ 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 $\nu'$ is "threaded through" these calls, producing a new valuation $\nu''$.

The result of the above steps is a new current configuration $\langle G'_s, s', \nu'' \rangle$. As discussed above, the current evaluation graph $G'_e$ is obtained as the subgraph of $G'_s$ where the edges that $\nu''$ indicates to be irrelevant have been removed. Widgets that are notified that their values might have changed send get requests that consult $\nu''$ to obtain their new values. Other algorithms to determine attributes such as ones used to set widgets' enablement states are executed. They will query $\nu''$ and possibly all of $G_c$, $G'_s$, and $G'_e$ depending on their needs. User interface widgets are notified of possible changes in their attributes. Then, the user interface driven by the property model is in a consistent state, and the property model is ready for another change request.

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; $\nu$ for valuation functions; $\mu$ 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 $\mathit{func} \mid \nu \to t \mid \nu'$ with the following meaning: the function *func* (either eval or evalmany) is evaluated within the context of the current valuation $\nu$, which produces a new valuation $\nu'$ and the result $t$. The symbol "$\cdot$" indicates that the function has no result.

The evalmany function, defined by the rules EVALMANY and EVALMANY-EMPTY, simply invokes eval for each variable in a set in some order. Each call to eval traverses the dependencies in the solution graph upwards and evaluates all variables that are necessary for determining the value of the current variable, and then executes the method of the current variable. Along the way, the 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.

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

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

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

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

EVALMANY-EMPTY

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

EVALMANY

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

**Figure 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 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 "$\cdot$", 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. 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 $v = v_l^{\mathsf{out}}$ 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.

The correctness of the evaluation phase depends on methods never copying and saving the current valuation: the same shared state should always be used when invoking any of the callbacks of a method, and not held after the method returns. In the current property model library [24], we provide a simple expression language for defining the bodies of methods, but as explained in Section 2, calls to C++ functions are allowed. Violating the above conditions in such functions is not easy, but if the programmer so desires, possible.[1] Note also that since we allow arbitrary code to be executed in methods, we cannot offer guarantees on their termination.

[1] In a prototype implementation we have used Haskell as the language for implementing methods. Haskell's type system and monads can be easily harnessed to detect violations of the condition statically.

## 5. Enablement generically

Section 3 discussed the intuitive reasons for making decisions about widget enablement. In this section we explain how, based on the state of a property model, to determine when the conditions to enable or disable widgets exist. Widgets can act appropriately based on those conditions, and other policies. The two algorithms discussed in this section, command activation and widget enablement, are possible because of the information in the constraint, solution, and evaluation graphs; we define the algorithms in those terms. Note that when we say that a property model's variable should be enabled/activated or disabled/deactivated, we mean that a widget bound to that variable should be enabled/activated or disabled/deactivated.

### 5.1 Command activation

Recalling the intuition from Section 3.1, a widget that executes a command should be deactivated when the command's preconditions are not satisfied. In a property model, commands are bound to output variables, and preconditions are expressed using invariant variables. Thus, a command activation algorithm can determine that a command should be deactivated if the output variable to which the command is bound depends on a value upon which a failed invariant also depends.

Since the functional dependencies currently active among the property model's variables are readily available in the evaluation graph, the command activation algorithm reduces to the following graph query:

> An output variable $o$ should be deactivated if, in the evaluation graph, there exists a variable $w$ such that (1) $w$ reaches $o$, and (2) $w$ reaches a failed invariant.

### 5.2 Widget enablement

Recalling the intuition from Section 3.2, a widget should be disabled whenever its value cannot affect an output variable. There are two reasons why updating a value of some variable $v$ could affect an output variable $o$: (1) there exists a currently active functional dependency between $o$ and $v$, or (2) updating $v$ could create a functional dependency between $o$ and $v$.

Again, the current functional dependencies are represented by the evaluation graph, and the solution and constraint graphs can be analyzed to predict changes to the functional dependencies triggered by a request to update a variable. Rephrasing the rationale above in terms of the graphs, a variable $v$ could affect an output variable $o$ if (1) $v$ reaches $o$, or if (2) updating $v$ would generate a new evaluation graph in which $v$ reaches $o$.

The most precise algorithm to determine this condition is to, in turn, (1) give each variable the highest priority; (2) generate a new solution graph; (3) generate a new evaluation graph with the edges present in both the old evaluation graph and the new solution graph, and the edges present in the new solution graph that are not present in the old evaluation graph; and (4) determine if the variable reaches some output variable. See the accompanying technical report [9] for an explanation why this is as precise as possible, unless the bodies of the methods are analyzed. Unfortunately, this option is expensive to compute.

Realistically, we develop a generic, close approximation:

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

To see why this is correct, note that the solution graph is a directed acyclic graph. Each variable $v$ in the solution graph sits at the root of an *ancestor (directed acyclic) graph* consisting of every variable reachable from $v$ in the transpose of the solution graph. The *ancestors* of $v$ are the variables in its ancestor graph. For example, in the solution graph in Figure 4(b), variable $q$ has ancestors $q$ and $c$. When we speak of methods and constraints in the ancestor graph, it refers to the methods, and their constraints, in the solution graph that are connected only to members of the ancestor graph.

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

Any ancestor reachable from $v$ in the *constraint graph* is called a *reachable ancestor*. If we update $v$, then the only constraints that could have different methods selected in the new solution graph must be in $v$'s ancestor graph.[2] This implies the following: in an evaluation graph, $v$ can reach some set of variables; if we update $v$, it could reach a superset of those variables, and the difference must come from its reachable ancestors. In other words, if none of its reachable ancestors reaches an output variable, then there is no way that updating $v$ can make it reach an output variable.

## 6. Experience

Early experiences of employing property models in commercial applications are very promising. In this section we report upon the experiences at Adobe Systems, Inc. We first report on a significant reduction in defect rates and vastly increased productivity when using property models. Then we discuss further pros and cons of our approach.

Recently, an effort was carried out to rewrite user interface code for a best-selling application that formerly depended on a 32-bit-only user interface framework. Some of the user-interface code was rewritten to use a 64-bit savvy framework, while other parts were instead rewritten to use a library built around the Adam property model engine.

Four teams of roughly three engineers each, balanced in programming skill, participated in the rewrite. Three of the teams, the Adam engine (AE) teams, were tasked with rewriting code for a large number of dialogs and palettes, varying in complexity, along with the necessary visual description and supporting code to use property models. A fourth team was tasked with rewriting other dialogs, of similar complexity, to use a modern vendor-supplied object-oriented UI framework.

During the time period under study, data was collected to support a number of metrics. Here we focus on the rate at which dialogs were produced, and the number of software defects discovered.

Figure 6 compares the number of reported bugs over several months for the three Adam engine teams (AE1–AE3) and the traditional framework team (TF). We can see that it was rare for more than 2–3 defects per week to be found for a given property model team, while a conservative estimate for the defect rate for the TF team was 10–20 defects per week. That is, the teams working in the property model architecture produced defects at less than twenty percent of the rate of that of the traditional framework team. To give a sense of the importance of these results, we note that user interfaces are a notorious source of defects. In a sample of defects in a 20,000-bug database of a major Adobe desktop application, roughly half were in the interface layer that property models target.

As with any prototype system, there have been challenges in adopting property models. Property model-based programming is

[2] This nontrivial statement is justified in an accompanying technical report [9].

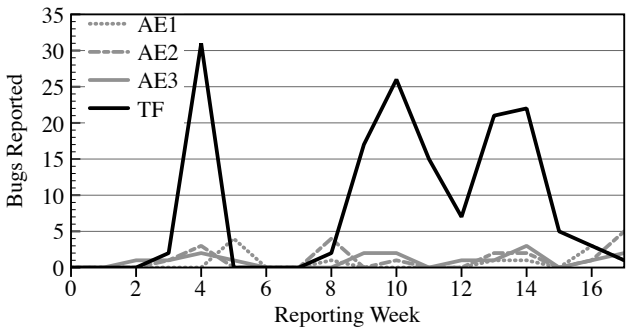


**Figure 6.** The number of reported bugs during several months for three Adam engine teams (AE1–AE3) and a traditional framework team (TF). The teams were tasked with rewriting code for a large number of dialogs of varying complexity. The teams each consisted of roughly three engineers, balanced in programming skills.

a new way of approaching user interface programming, and there is somewhat of a learning curve. The supporting tool ecosystem, though more powerful than traditional user-interface builders, is not as mature as with established frameworks. Notwithstanding these limitations, in the above study, property model programmers were substantially more productive than their counterparts on the TF team. In the time period studied, the three AE teams combined completed roughly 75 dialogs and palettes, with another 50 or so underway. The TF team completed fewer than 10 altogether. The product team did not believe they would be able to succeed in porting to 64-bits in a single release without the property model technology. As the tool set matures, in addition to the reduced complexity, we expect to see further gains.

Other difficulties faced by teams adopting property models have arisen because the existing logic in the code has actually been incomplete or inconsistent. The explicit modeling required by our approach forces the programmer to think through and correct incomplete or faulty designs.

We have also observed benefits from the UI designer perspective. User interface designers are able to play with prototypes encapsulating the user interface element relationships much earlier in the development cycle, without waiting for custom coding. Early feedback of this kind helps to avoid costly late-cycle code changes.

## 7. Related Work

Systems based on declarative specifications are common in the area of user interfaces. (Many have noticed that event handling logic for GUIs can easily degrade into a hopeless tangle of spaghetti code.) The combination of procedural and declarative program code has found success in, for example, GUI element layout. The most familiar example is perhaps HTML, CSS, and DOM combined with JavaScript. The QTk library [12] in Mozart/Oz, Glade [7], XUL [16], XAML [34], and XForms [5] serve as further examples. Some of these systems, along with rule-based systems such as Drools [27], Jess [11], and R++ [14], also support concisely specifying declarative rules for maintaining consistency across values in user interfaces. Property models are distinguished from these systems by not only providing the ability to create rules that assist in producing a valid result but also by providing an explicit model of the dependencies these rules create. Inspecting the state of the model enables generic algorithms (e.g., for widget enablement) for user interfaces.

Constraint systems have been studied extensively for use in user interfaces, mostly for automated widget layout. A large number of declarative, constraint-based GUI systems have been proposed, for example, Sketchpad [30], Amulet [20], Garnet [18], as well as ThingLab I and II, DeltaBlue, and SkyBlue [28]. A survey of model-based UI design environments for UI construction can be found in [26]. More recent active projects that support (one-way) data-flow constraints include the OpenLaszlo framework [22] for developing rich Internet applications. Constraints in these systems are mainly used for layout where the simpler one-way constraints are rather standard, e.g., in many diagram drawing tools [33]. Based on extensive experience with the Amulet system, its authors conclude that it is unlikely that constraint systems will ever be used for much other than layout [36].

Our experience indicates a more positive picture. Amulet and the related systems integrate a constraint solver into a general purpose programming language, but the state of the network of functional dependencies is hidden from the programmer, whereas it is explicitly modeled and made accessible by property models. This, we believe, is why we can benefit from the constraint system formalisms and algorithms, and apply them in an area where previously their success has been limited.

Regarding enablement logic, the Jade interactive dialog creation tool for Garnet [37], and similar work by Myers et al. [19], target the expression of enablement logic using a constraint system. Also related is work by Frank et. al [8]. These works do not, however, attempt to devise a generic enablement algorithm. Further, apart from relieving the programmer from coding explicit enablement and activation logic, our analysis clearly defines the reasons why a widget should be enabled/activated or disabled/deactivated. To improve usability, a user interface could be instrumented to show these reasons to its user.

## 8. Conclusions

Code for user interfaces, accounting for substantial portions of many applications, is typically not reusable and is a notorious source of software defects. In a large industrial code base (Adobe's major desktop application) that we studied, user interface related code comprised about 30% of all code and contained over 50% of reported defects. Furthermore, programming user interfaces is one of the more mundane tasks in software construction—and generally not high on the list of developers' favorite tasks.

Increased reuse in the domain of user interfaces can thus notably improve programmer productivity, improve software quality, and free programmers to work on more rewarding tasks. With a higher quality of user interfaces, our daily encounters with computer systems can become more productive, more pleasant, and less frustrating.

This paper describes promising results in a venture to produce reusable components in the domain of user interfaces.

We further refine property models as an abstraction for representing variables and their dependencies in command parameter synthesis tasks. In particular, we formalize the computation of values for variables in a property model. This computation reveals the currently active functional dependencies among those variables. We showed how to use this information to develop generic algorithms for command activation and widget enablement.

The experimental results obtained during the course of commercial software development within a major software company indicate that use of property models leads to significantly increased productivity, and to a dramatic reduction in the reported defect count, compared to teams relying on the services of a more traditional graphical user interface framework.

We continue to study and develop property models further, to expand the class of user interfaces that property models can support, and expand the base of use cases. Further, we believe that the domain of user interfaces, command parameter synthesis in

particular, has further algorithms to discover, to allow for more of the functionality of a high-quality user interface to be implemented in reusable libraries.

## Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Grant No. CCF-0845861.

## References

[1] Apple, Inc. *Apple Human Interface Guidelines: User Experience*. 1 Infinite Loop, Cupertino, CA 95014, June 2008.

[2] ASL. *Adobe Source Libraries*. Adobe Systems, Inc., 2005. stlab.adobe.com.

[3] C. Benson, A. Elman, S. Nickel, and C. Z. Robertson. *GNOME Human Interface Guidelines 2.2*, Mar. 2008. `http://library.gnome.org/devel/hig-book/stable/index-info.html.en`.

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

[5] J. M. Boyer, M. Dubinko, J. Leigh L. Klotz, D. Landwehr, R. Merrick, and T. V. Raman. XForms 1.0 (Third Edition), Oct. 2007. `http://www.w3.org/TR/2007/REC-xforms-20071029/`.

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

[7] E. Feldman. Create user interfaces with Glade. *Linux J.*, 2001(87):4, 2001.

[8] M. R. Frank, J. J. de Graaff, D. F. Gieskens, and J. D. Foley. Building user interfaces interactively using pre- and postconditions. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 641–642, New York, NY, USA, 1992. ACM.

[9] J. Freeman, J. Järvi, J. Smith, M. Marcus, and S. Parent. Properties of constraint systems of property models. Texas A&M University, Department of Computer Science and Engineering, Parasol Laboratory Technical Report TR09-001. `http://parasol.tamu.edu/publications/`, July 2009.

[10] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Commun. ACM*, 33(1):54–63, 1990.

[11] E. Friedman-Hill. Jess 7, Feb. 2008. `http://www.jessrules.com/jess/charlemagne.shtml`.

[12] D. Grolaux and P. V. Roy. `QTk` — an integrated model-based approach to designing executable user interfaces. In *8th Workshop on Design, Specification, and Verification of Interactive Systems (DSVIS 2001)*, Lecture Notes in Computer Science, Glasgow, Scotland, June 2001. Springer-Verlag.

[13] J. Järvi, M. Marcus, S. Parent, J. Freeman, and J. N. Smith. Property models: from incidental algorithms to reusable components. In *GPCE '08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 89–98, New York, NY, USA, 2008. ACM.

[14] D. Litman, P. F. Patel-Schneider, A. Mishra, J. Crawford, and D. Dvorak. R++: Adding path-based rules to C++. *IEEE Trans. on Knowl. and Data Eng.*, 14(3):638–658, 2002.

[15] Microsoft Corporation. *Windows Vista UX Guide: User Experience Guidelines*, 2008. `http://download.microsoft.com/download/e/1/9/e191fd8c-bce8-4dba-a9d5-2d4e3f3ec1d3/uxguide.pdf`.

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

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

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

[19] B. A. Myers and D. S. Kosbie. Reusable hierarchical command objects. In *CHI '96: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 260–267, New York, NY, USA, 1996. ACM.

[20] B. A. Myers, R. G. McDaniel, R. C. Miller, A. S. Ferrency, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The Amulet environment: New models for effective user interface software development. *Software Engineering*, 23(6):347–365, 1997. citeseer.ist.psu.edu/article/myers96amulet.html.

[21] B. A. Myers and M. B. Rosson. Survey on user interface programming. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 195–202, New York, NY, USA, 1992. ACM.

[22] OpenLaszlo project. `http://www.openlaszlo.org`.

[23] Parasol Lab, Computer Science, Texas A&M University. *Property Models Research Project's Home Page*, 2009. `http://parasol.cs.tamu.edu/groups/pttlgroup/property-models`.

[24] S. Parent. *Adobe Property Model Library*. Adobe Systems, Inc., 2005. Part of Adobe Source Libraries, `http://stlab.adobe.com`.

[25] S. Parent. A possible future for software development. Keynote talk at the Workshop of Library-Centric Software Design 2006, at OOPSLA'06, Portland, Oregon, 2006. lcsd.cs.tamu.edu/2006.

[26] P. Pinheiro da Silva. User interface declarative models and development environments: A survey. *Interactive Systems Design, Specification, and Verification*, pages 207–226, 2001.

[27] M. Proctor, M. Neale, B. McWhirter, K. Verlaenen, E. Tirelli, F. Meyer, A. Bagerman, M. Frandsen, G. D. Smet, T. Rikkola, S. Williams, B. Truit, R. Jain, C. Nagarkar, and D. Ahearn. Drools, 2008. `http://www.jboss.org/drools/`.

[28] M. Sannella. Skyblue: A multi-way local propagation constraint solver for user interface construction. In *UIST '94: Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 137–146, New York, NY, USA, 1994. ACM.

[29] M. J. Sannella. *Constraint satisfaction and debugging for interactive user interfaces*. PhD thesis, University of Washington, Seattle, WA, USA, 1994.

[30] I. E. Sutherland. Sketchpad: A man-machine graphical communication system. In *DAC '64: Proceedings of the SHARE design automation workshop*, pages 6329–6346, New York, NY, USA, 1964. ACM.

[31] Qt: A cross-platform application and UI framework. `http://www.qtsoftware.com/products`, Mar. 2009.

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

[33] M. Wybrow, K. Marriott, L. McIver, and P. J. Stuckey. Comparing usability of one-way and multi-way constraints for diagram editing. *ACM Trans. Comput.-Hum. Interact.*, 14(4):1–38, 2008.

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

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

[36] B. V. Zanden, R. Halterman, B. Myers, R. Miller, P. Szekely, D. Giuse, D. Kosbie, and R. McDaniel. Lessons learned from users experiences with spreadsheet constraints in the garnet and amulet graphical toolkits. `ftp://cs.utk.edu/pub/TechReports/2002/ut-cs-02-488.pdf`, May 2002.

[37] B. V. Zanden and B. A. Myers. Automatic, look-and-feel independent dialog creation for graphical user interfaces. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 27–34, New York, NY, USA, 1990. ACM.