Property Models

From Incidental Algorithms to Reusable Components

Jaakko Järvi
Texas A&M University
College Station, TX, U.S.A
jarvi@cs.tamu.edu

Mat Marcus
Adobe Systems, Inc.
Seattle, WA, U.S.A
mmarcus@adobe.com

Sean Parent
Adobe Systems, Inc.
San Jose, CA, U.S.A
sparent@adobe.com

John Freeman
Texas A&M University
College Station, TX, U.S.A
jfreeman@cs.tamu.edu

Jacob N. Smith
Texas A&M University
College Station, TX, U.S.A
jnsmith@cs.tamu.edu

Abstract

A user interface, such as a dialog, assists a user in synthesising a set of values, typically parameters for a command object. Code for "command parameter synthesis" is usually application-specific and non-reusable, consisting of validation logic in event handlers and code that controls how values of user interface elements change in response to a user's actions, etc. These software artifacts are *incidental*—they are not explicitly designed and their implementation emerges from a composition of locally defined behaviors.

This article presents *property models* to capture explicitly the algorithms, validation, and interaction rules, arising from command parameter synthesis. A user interface's behavior can be derived from a declarative property model specification, with the assistance of a component akin to a constraint solver. This allows multiple interfaces, both human and programmatic, to reuse a single model along with associated validation logic and widget activation logic.

The proposed technology is deployed in large commercial software application suites. Where we have applied property models, we have measured significant reductions in source-code size with equivalent or increased functionality; additional levels of reuse are apparent, both within single applications, and across product lines; and applications are able to provide more uniform access to functionality. There is potential for wide adoption: by our measurements command parameter synthesis comprises roughly one third of the code and notably more of the defects in desktop applications.

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

General Terms Algorithms, Design

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

GPCE'08, October 19–23, 2008, Nashville, Tennessee, USA.

1. Introduction

Software systems utilizing reusable components tend to be more robust and less costly than their hand-crafted counterparts (Basili et al. 1996; Frakes and Succi 2001; Nazareth and Rothenberger 2004). Indeed, the software industry has been successful in capturing often needed functionality into reusable generic components, witnessed by the wide availability of software libraries in all mainstream programming languages and the ubiquitous use of components from those libraries. There are, however, domains commonly encountered in mainstream day-to-day programming in which reuse remains modest—and in which the industry continues to struggle with low quality, high defect rates, and low productivity.

As the scale of software increases, software development relies more on reusable components—at the same time, there is an increase in the amount of code that composes and relates components. Often such code is not explicitly designed, and it is rarely reusable. In larger collections of components, networks of relationships between components arise. We refer to such networks as *incidental data structures*—data structures that emerge out of compositions of components and have neither an explicit encoding in the program nor an explicit run-time representation accessible to the rest of the program. Consequently, such data structures cannot be operated on by generic, reusable algorithms. Instead, they are manipulated with *incidental algorithms*, similarly emerging from the combined behavior of locally defined actions, and with no explicit encoding in the program. We believe that a large reuse potential exists within incidental algorithms and data structures.

In this paper we describe some of the architectural challenges in creating reusable libraries for rich user interfaces. We identify the communication and relationships between different elements of user interfaces as an architectural domain where incidental data structures and algorithms are prevalent; we refer to this domain as *command parameter synthesis*. We demonstrate a dramatic increase in re-usability of user interface code if the incidental structures of command parameter synthesis are modeled explicitly. To represent these explicit models, we present a new implementation mechanism, *property models*.

Command parameter synthesis assists a client in selecting and validating parameters for some command to be executed in the program. This is a common task in interactive applications—or in any application with a non-trivial, human or programmatic, interface. Typical examples of user interfaces requiring command parame-

ter synthesis are a print dialog for selecting parameters that guide how a document should be printed, and an image scaling script for selecting parameters that determine how an image size should change. In today's industrial programming practices, code for command parameter synthesis is distributed throughout script processing code and event handlers for user interface elements. In simple cases, this code consists of "validators" for individual elements, but often the desired behavior is more complex: changing a value of one element can trigger changes in other elements. Seemingly simple examples often contain complex and cyclic dependencies between elements, manifesting in large amounts of non-reusable event handling code tightly dependent on the vendor's application framework.

As its main contribution, this paper describes explicit property models, algorithms, and the supporting component architecture that can replace the incidental algorithms and data structures arising in command parameter synthesis code. Preliminary results from deploying the technology in an industrial setting exhibit significant reductions in defects and code size for command parameter synthesis. We also discuss how the proposed approach provides more consistent functionality in and across applications, opens the door for new functionality, aids automated reasoning, and decreases the gap between user interface design and implementation.

The approach described in this paper is used in the Adobe Software Technology Labs (STLab) open-sourced property model library (Parent 2005). The Parasol Lab at Texas A&M University and Adobe's STLab collaborate on a number of projects, including property model research and prototype development. At times the discussion in this paper diverges from the documentation of the currently deployed property model library, reflecting instead the state of our research prototype system, available for download (Parasol 2008). We point out key differences when relevant.

Finally, as motivation for focusing on command parameter synthesis, we note that a significant portion of the code of desktop applications deals with user interfaces. A survey from 1992 (Myers and Rosson 1992) reported this portion as almost 50%. A recent analysis of a large industrial code base indicated that approximately one third of the code, and more than half of the reported defects, were in code that coordinates event handling logic, widgets, and other Graphical User Interface (GUI) components (Parent 2006)—essentially in code for command parameter synthesis. Industrial programming practices seem to be accepting this state of affairs. A quote from the developer documentation of a widely used GUI framework is revealing, given that it is not even expected that the code of command parameter synthesis (here lumped as part of the "controller" of the Model-View-Controller architecture (Burbeck 1987)) could be generic or reusable: "*Since what a controller does is very specific to an application, it is generally not reusable even though it often comprises much of an application's code.*" (App 2007, §1)

1.1 Command parameter synthesis

Figure 1 situates the command parameter synthesis code in an application that follows the *Command* pattern (Gamma et al. 1995, §5). Changes to a document are effected through command objects that encapsulate a command with its associated parameters; command parameter synthesis code constructs these parameters, and can record which variables contributed to computing the values of the parameters. In a typical situation, such code is woven throughout user interface and script components that assist the user in producing the parameters. Command parameter synthesis also depends on the current state of the document, possible default values, and stored preferences. Conversely, the values of variables produced during command parameter synthesis can be used as future preferences or initial values. They can also be recorded as a script for

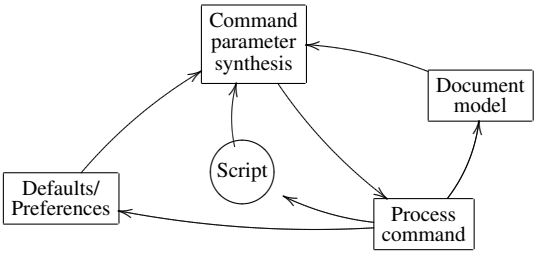


Figure 1. Relation of command parameter synthesis to Command Pattern.

repeating the command in the future (for a different document) and thus, instead of or in addition to a human user, command parameter synthesis code may receive input from an explicitly written or recorded script.

To be more concrete, we consider a dialog box for displaying and controlling variables contributing to parameter synthesis. For example, in an image editing application, an image-resize command might be written to expect horizontal and vertical dimensions of the new image in pixels. A dialog box can be more flexible, offering the user the choice of whether to set the width and height in pixels or in percents, relative to their initial values. It might further assist the user by providing a constrain-proportions flag which, when set, arranges matters so that if the user doubles the width of the image the height will automatically double as well. As another example, an invalid value in any of the fields might cause an "OK" button to be deactivated. Ultimately, of all the values involved, the underlying command must be constructed with absolute height and width parameters. We might go so far as to claim that the role of a user interface in an application is to assist the user in producing valid parameters for a command object.

The above kind of rules form the essence of the behavior of a dialog window, yet in an application utilizing a typical modern object-oriented GUI framework, such rules have no explicit representation; they are dispersed throughout event handling functions. Such event handling code is an example of an incidental algorithm. Furthermore, the state upon which the incidental algorithm operates is often distributed, and replicated, across individual components, giving rise to an incidental data structure.

To improve this situation we explicitly model the space of variables in command parameter synthesis. Programmers are accustomed to working with explicit models of documents using the Model-View-Controller (MVC) pattern. We choose to also apply the essence of MVC to the command parameter synthesis process. The values of the variables on which a command's parameters might depend, for example the width and height variables in the above example, together with their interdependencies and invariants, comprise the (property) model. The widgets in a dialog can play the role of both controller (upon user interaction) and view (through the widget's visual characteristics). Unfortunately, many widget libraries lack a clean separation of view and controller and, worse yet for our purposes, they tend to operate on state in the widgets themselves. These issues present some barriers to the explicit modelling approach, and we typically find that a widget set must be (lightly) wrapped (see Section 4.1) to cleanly support MVC for command parameter synthesis.

A single application may contain numerous commands. Furthermore, each command parameter synthesis for a given command may occur via several different avenues: dialog boxes, palettes, scripting interfaces, and so forth. Realizing the property model as an explicit software component thus provides reuse opportunities in multiple dimensions: within a single application over multiple

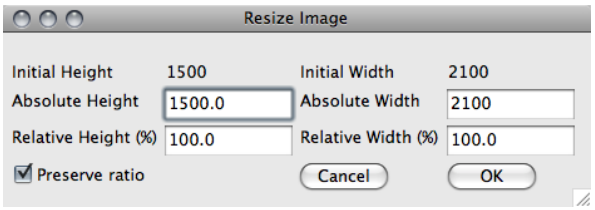


Figure 2. A dialog for resizing a graphical object.

interfaces, across multiple commands, and across multiple applications. In particular, we explore architectures that capture commonality using a property modelling component, together with auxiliary subsystems such as small declarative languages for concisely expressing the unique characteristics of a particular property model.

1.2 Image size — a closer look

Figure 2 shows an example dialog for resizing a graphical object. Four widgets maintain data that can be edited: *absolute width*, *absolute height*, *relative width*, and *relative height*. The absolute values are in some integral units, such as pixels, and the unit of relative width and height is percent. The intent is to allow the user to either set the object's size to some absolute width and height or to scale it in proportion to the *initial* width and height that were in effect when the dialog was launched. The initial width and height are also shown in the dialog but cannot be altered. Furthermore, the dialog has a check box that sets or clears a flag that ensures that the height and width retain the same aspect ratio as that between the initial height and width. The user of the dialog thus has several ways to produce the inputs needed by the image-resize command object, which are the width and height of the image in pixels.

Figure 3 demonstrates the complexity of the incidental data structure arising from this dialog. The dependencies in the figure were derived from the event handling code in the implementation of the dialog within an object-oriented GUI framework. Our goal is to express these complex relations with a property model, and realize the model as an explicit data structure that exists independently of the dialog box, but to which the above dialog box, or any other user interface, scripting system, or other tool can attach non-intrusively.

We support property models via a software library that provides a *sheet* data structure that stores the values of variables, connections between variables, and references to functions that perform computations between variables, as well as algorithms that maintain invariants, or stated relations between variables, when the values of some of the variables are perturbed.

Rather than using a library API directly, one of the ways a programmer can describe a property model is using a domain specific declarative language. The specification language for Adobe's property model library is known as *Adam* (Parent 2005). For example, the specifics of the property model for synthesizing parameters to the image-resize command are captured in the specification shown in Figure 4. Briefly, the **input**, **interface**, and **output** sections declare the variables, or *properties*, of the property model, and the **relate** sections define the dependencies and computational rules between variables. We return to the details of this specification in Section 2.2.

Regarding code reuse, we note that, for example, all of the validation logic in the event handlers, dialog initialization and closing code, code for disabling and enabling user interface elements, and script recording and play back, can be derived from a property model specification, saving the programmer from writing significant amounts of non-reusable application and dialog specific code.

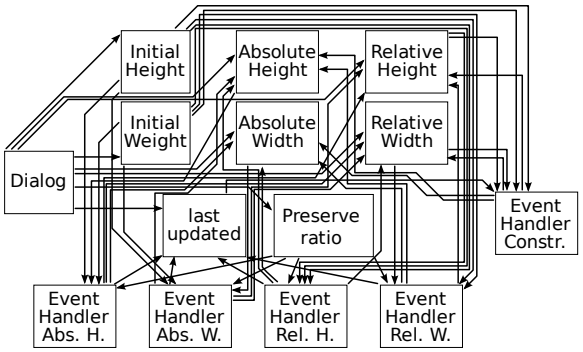


Figure 3. The incidental data structure formed as the dependencies between the UI components and event handlers in an object-oriented implementation of the user interface logic of dialog shown in Figure 2. The edges correspond to the relations "event handler writes to a value" or "event handler reads a value."

```
sheet image_resize
{
 input:
   initial_width  : 5 * 300;
   initial_height  : 7 * 300;
 interface:
   preserve_ratio  : true;
   absolute_width   : initial_width;
   absolute_height  : initial_height;
   relative_width;
   relative_height;
 logic:
   relate {
     absolute_width  <== relative_width * initial_width / 100;
     relative_width  <== absolute_width * 100 / initial_width;
   }
   relate {
     absolute_height  <== relative_height * initial_height / 100;
     relative_height  <== absolute_height * 100 / initial_height;
   }
   when (preserve_ratio) relate {
     relative_width  <== relative_height;
     relative_height  <== relative_width;
   }
 output:
   result <== { height: absolute_height, width: absolute_width };
}
```

Figure 4. A declarative specification of the model for parameter synthesis for an image-resize command. The syntax is that of the Adam language of Adobe's property model library.

2. Explicit property models for command parameter synthesis

The dialog in Figure 2 supports the user in synthesizing parameters for the image-resize command object by implementing a multitude of mappings between values displayed in different user interface elements, such as the one that computes the absolute width from the relative and initial widths. The control logic of the dialog orchestrates when and which of these mappings should be applied, e.g., a given map might automatically change the width to match the original image's proportions when the height changes. In our proposed architecture, these mappings are captured in a property model.

A property model, like a spreadsheet, maintains relationships and invariants across variables. But, unlike a spreadsheet, values (cells) in a property model might be able to play the role of source value or derived value according to which mappings (formulas) are currently in effect. When requested to change the value of a variable, the property model calculates which mappings are to be in effect, based on which variable is requested to be changed and which are the source and derived variables in the current state of the model. All such logic is entirely contained in the property model—the role of a view, e.g., a set of widgets in a dialog box, is limited to responding to signals that the model has been updated by refreshing the display appropriately. Similarly, a controller's task is simply to generate a request to the property model to modify the value of a single variable.

We have found that constraint systems, in particular the formulations of *multi-way dataflow constraint systems* (Zanden 1996), are useful for representing property models for command parameter synthesis. These systems define relations between variables. Each relation has a set of methods such that each can satisfy the relation by assigning new values to some of its variables.

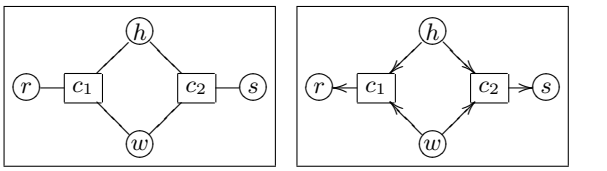
The manner in which values of variables should be propagated in user interfaces cannot in general be captured only with (undirected) relations. The direction to which mappings between variables are applied when interacting with a dialog is significant. For example, whether to compute height from width and aspect ratio or width from height and aspect ratio depends on which user interface elements were altered most recently. In the proposed architecture, this information is not provided by the programmer; rather, the mechanism of *variable priorities*, explained in Section 2.3, determines the direction of value propagation in property models in accordance with the principle of "least surprise", favoring the preservation of values edited more recently over those edited less recently.

A carefully selected representation of property models as multi-way dataflow constraint systems lets us incorporate variable priorities into existing constraint system formulations and leverage known constraint solving algorithms (such as QuickPlan (Zanden 1996)) to determine the direction of propagating values in the system. This section gives a brief introduction to constraint systems and describes how the core parts of the property model can be viewed and implemented as a constraint system.

2.1 Background: multi-way dataflow constraint systems

A constraint system is a tuple $\langle C, V \rangle$, where C is a set of constraints, and V a set of variables that each have a *current value*. 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 ($n = |R|$), and M is a set of *constraint satisfaction methods* (CSM). Executing any CSM in M computes values to some subset of R using another subset of R as inputs, such that the relation r becomes satisfied. If the variables in R satisfy r, we say that the *constraint is satisfied*. The sets of inputs and outputs of a CSM in M are usually required to be disjoint, and their union required to equal R. The code realizing a CSM is considered a "black box": it is the programmer's responsibility to ensure that a constraint is satisfied when any of its CSMs is executed.

The constraint satisfaction problem for a constraint system $\langle C, V \rangle$ is to find a valuation of the variables in V such that each constraint in C is satisfied. Such a valuation is attained if the CSMs, exactly one from each constraint, can be executed in an order where once a variable has been read from or written to by one method, no other method will write to it. Depending on the problem, such an ordering may or may not exist; in this case, variations of the constraint satisfaction problem use different criteria to relax some constraints in order to attain valuations that satisfy a subset of the constraints (Zanden 1996; Freeman-Benson et al. 1990; Sannella 1994).



(a) (b)

Figure 5. An undirected constraint graph (a) and a directed solution graph (b).

The relations in constraint systems can be arbitrary, but often they are equalities. For example, the equation $r = w/h$, name it e_1, could describe what should hold true of the width w, height h, and aspect ratio r of an image. One possible, perhaps the most natural, set of constraint satisfaction methods for this equation is $M_1 = \{w \leftarrow rh, h \leftarrow w/r, r \leftarrow w/h\}$. Another relation, e_2, might connect the width and height to the size of the file necessary to represent the image, say, $s = g(wh)$. We assume that the function g realizes the size calculation, and that its details are not known. For the relation e_2, a possible (singleton) set of constraint satisfaction methods is $M_2 = \{s \leftarrow g(wh)\}$; that is, the constraint can only be satisfied in one direction, by computing s from w and h.

A dataflow constraint system is often represented as a bipartite graph $G = \langle V + C, E \rangle$, with vertex sets V and C representing the variables and constraints of the system, respectively, and E the undirected edges that connect each constraint to its variables. Figure 5(a) depicts the graph arising from the above described constraint system $\langle \{r, w, h, s\}, \{c_1, c_2\} \rangle$, where the constraint $c_1 = \langle \{r, w, h\}, e_1, M_1 \rangle$ and $c_2 = \langle \{s, w, h\}, e_2, M_2 \rangle$. Variables are shown as circles and constraints as rectangles. This graphical representation is useful for visualizing the dependencies that the relations impose between variables, though it does not indicate the inputs and outputs of the CSMs.

A *solution graph* (also, *method graph*) of a constraint graph is a directed acyclic graph formed by selecting exactly one CSM from each constraint, along with its in- and out-edges, so that each variable vertex has at most one in-edge. One possible solution graph for the constraint graph of Figure 5a is shown in Figure 5b. A topological sort of the solution graph's vertices determines a CSM evaluation order that will assign values that satisfy all constraints in the system.

Finding a solution graph for a given constraint graph is referred to as the *planning phase*. Satisfying the constraints by executing the CSMs of the solution graph in a suitable order is known as the *execution phase*.

Constraint systems can be overconstrained, so that there may be no solutions, or underconstrained so that there is more than one solution. The system above has three different solution graphs: any of the three methods of c_1 paired with the sole method of c_2 forms a solution.

2.2 Property model for command parameter synthesis as a constraint graph

Each property in a property model is represented as a variable in a constraint system. Constraint satisfaction methods define how values of variables can be computed from those of other variables. In our example dialog, shown in Figure 2, the various user interface elements each display a value of a variable in a property model. Manipulation of a user interface element translates into a request to

change the value of a variable in the model. The model, according to its relations and CSMs, updates other variables as necessary.

The property model specification in Figure 4 gives rise to a constraint system with the variables:

$$V = \{h_i, w_i, h_a, w_a, h_r, w_r, v_{\text{rat}}, v_{\text{res}}\};$$

w_i and h_i stand for initial width and height, w_a and h_a for absolute width and height, w_r and h_r for width and height relative to initial width and height, and v_{rat} for the Boolean variable that controls preserving the aspect ratio. The variable v_{res} has no direct counterpart as a user interface item; it is the result of the command parameter synthesis in this property model, i.e. the command parameters, a pair comprising the values of the absolute width and height.

The variable v_{rat} corresponds to the **preserve_ratio** variable in the property model specification in Figure 4; correspondence between the other variable names is self-explanatory. The specification categorizes variables into three groups with the **input**, **interface**, and **output** sections. The semantics of the different section labels are described later; the categorization makes no difference for solving the constraint system, but is necessary for other components interacting with the property model, in particular for script recording.

The constraints and their CSMs are derived from the relations that must hold between the variables in the property model. These, in turn, arise from the desired behavior of user interfaces supporting the modeled command parameter synthesis, such as our example dialog box. In our example property model, the relations between the three height and three width variables each give rise to one constraint. In these constraints, we name the CSMs with labels $(a, b, c, d, \ldots)$ so that we may easily refer to them in later discussion:

$$\begin{aligned} c_1 &= \langle \{h_a, h_i, h_r\}, h_a = h_i h_r, \\ &\qquad \{a : h_a \leftarrow h_i h_r/100, b : h_r \leftarrow 100 h_a/h_i\}\rangle \\ c_2 &= \langle \{w_a, w_i, w_r\}, w_a = w_i w_r/100, \\ &\qquad \{c : w_a \leftarrow w_i w_r, d : w_r \leftarrow 100 w_a/w_i\}\rangle \end{aligned}$$

In the Adam specification in Figure 4, the first two **relate** clauses in the **logic** section correspond to constraints c_1 and c_2.

When the preserve aspect ratio flag is set, the ratio of the width and height must equal the ratio of the initial width and height. This relation can be captured as $\neg v_{\text{rat}} \vee h_r = w_r$. It is in principle possible to satisfy this relation with the set of CSMs $\{w_r \leftarrow$ if v_{rat} then h_r else $w_r, h_r \leftarrow$ if v_{rat} then w_r else $h_r\}$. In essence, the variable v_{rat}, however, controls whether the relation $h_r = w_r$ should be enforced or not. To model the condition, we consider two different constraint graphs based on the value of v_{rat}—one with no constraint between variables h_r and w_r, the other with the constraint:

$$c_3 = \langle \{v_{\text{rat}}, h_r, w_r\}, h_r = w_r, \{e : w_r \leftarrow h_r, f : h_r \leftarrow w_r\}\rangle$$

Splitting constraint graphs in this manner to express "conditional constraints" is somewhat undesirable, as it exponentially increases the number of systems one needs to reason about. However, in property models, a valuation satisfying the constraint system is only one of the desired results. The direction in which data flows in the system is also important (see Section 2.3)—conditional constraints affect the direction of the flow of data, whether they are active or not. The counterpart of constraint c_3 in Figure 4 demonstrates the use of the special **when ... relate** construct for expressing conditional constraints.

What remains is to define from which variables the result is collected:

$$c_4 = \langle \{v_{\text{res}}, w_a, h_a\}, v_{\text{res}} = \langle w_a, h_a\rangle, \{g : v_{\text{res}} \leftarrow \langle w_a, h_a\rangle\}\rangle$$

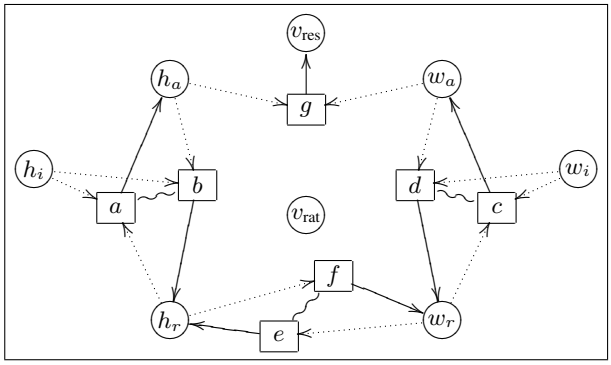


Figure 6. A constraint system for command parameter synthesis for an image-resize command. The rectangles represent method vertices. Dotted arrows are inputs to methods, solid arrows outputs from methods.

This constraint is defined in the **output** section of Figure 4.

Figure 6 shows the constraint graph of the constraint system $S = \langle V, \{c_1, c_2, c_3, c_4\}\rangle$ (the conditional constraint c_3 is included). Instead of the "one vertex per constraint" representation of Figure 5a, here we represent each CSM with a distinct vertex (drawn as a rectangle). Each such *method vertex* has in-edges from vertices that represent the CSM's input variables, and out-edges to vertices that represent its output variables. Note that although the methods in the examples presented in this paper have exactly one output (and the currently deployed version of the Adam language and its property model library impose this restriction), in general, methods can have an arbitrary number of output variables. We mark the method vertices belonging to a single constraint with a wavy arc connecting them. The solution graph, if it exists, is obtained by removing all but one method vertex and its edges from each constraint.

The relations of the constraints are not shown in the graph, and indeed, they do not have an explicit representation in our system. The relations, however, provide the underlying mental model to guide the programmer in developing the CSMs, and should be part of the program documentation. One could envision that for some constraints, instead of the user directly coding the CSMs, they could be automatically generated from the constraint's relation; we have not studied this possibility further.

2.3 Variable priorities

A dialog must show up-to-date values of all properties, no matter which value was last edited. In our example dialog, when the absolute width is edited, the relative width should change accordingly, and vice versa; height cells should work the same way. Which properties to update depends on which value was edited most recently or, more generally, on the order of edits to a set of values.

Constraint systems as described above have many different solution graphs that can satisfy the system. Each of these solution graphs represents a different flow of data amongst the properties. As mentioned, the solution graph selected should be the one that is least surprising to the client, preserving the values of variables that the client has changed more recently over those changed less recently.

To order solution graphs, the property model maintains a measure indicating how recently a variable was edited. We refer to this measure as the *priority* of a variable. We use natural numbers as priority values, where a lower numerical value indicates a higher priority, that is, a more recently edited variable. The task of the solver

is thus to find a solution graph such that data does not flow from a lower priority variable to a higher priority variable, or, if such a solution is not possible, a solution graph that minimizes such flow.

2.4 Property model as a library component

We are now in a position to give a more precise description of a property model as a software component. We identify the following roles and associated categories of operations of a property model component: (1) a property model is a container of variables and constraints between those variables, and thus provides operations for populating the model with variables and constraints; (2) a property model is a constraint system, and thus supports requests to perturb the value or priority of a variable and provides operations for requesting the constraint system to be solved, followed by a recalculation of the values of its variables; (3) a property model serves as the model part of MVC, and thus provides the views with operations that support monitoring the value of a variable; (4) the variable priorities and current computed solution graph of the property model contain information necessary for scripting, user interface element enabling and disabling, automated reasoning, etc., and thus depending on the needs, a property model component provides operations for discovering the variables that contribute to the output variables of a property model, or even direct access to the current solution graph.

A concrete instance of a property model component is the **sheet_t** data structure with its accompanying operations in the Adobe STLab's property model library (Parent 2005).

3. Solving property models

We model variable priorities by adding to the system new constraints that require certain variables' values to remain unchanged. In general, adding such *stay constraints* makes the system over-constrained, so that all the constraints in the system cannot be satisfied simultaneously. We deal with over-constrained systems by giving constraints different strengths, and modifying the constraint satisfaction problem to admit partial solutions that satisfy only some of the constraints, where stronger constraints are favored over weaker ones. Such constraint hierarchies (Borning et al. 1987) commonly have a dedicated strongest class of required constraints that all solutions must satisfy, and a small number of other classes; the constraint strength classes form a total order. Here we associate strength zero with the class of required constraints. Classes of weaker constraints are given strength 1, 2, 3, etc.

In property models, the strength of a variable's stay constraint is determined by the variable's priority; solver algorithms that support constraint hierarchies (Zanden 1996; Freeman-Benson et al. 1990; Sannella 1994) will then favor partial solutions that retain the values of the most recently edited variables.

More precisely, a stay constraint has exactly one method and it involves one variable. The method is a constant function, whose value is the current value of the variable. If a stay constraint is enforced in a particular solution, the constraint is satisfied through its sole method, and the variable's value thus "stays" intact. Figure 7 shows our example constraint graph with stay constraints added (method vertices of stay constraints are shown as doubly framed rectangles). The constraint graph shows one possible assignment of strengths to the stay constraints.

Priorities, and thus stay constraints, are only assigned to variables that can be requested to be changed by a client of the model—in the Adam specifications, these variables are declared in the **interface** section. Thus, neither the output variable v_{res}, nor the input variables h_i and w_i have stay constraints. Section 3.2 explains how stay constraints that remain in a solution graph indicate a special "contributing" role for a variable, which input (or output) variables can never have.

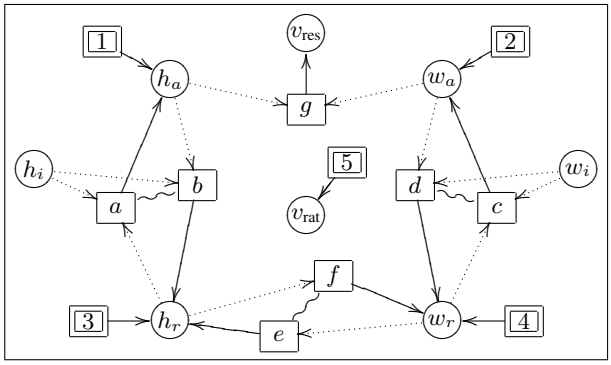


Figure 7. Constraint system for image scaling. Variable priorities are expressed as stay constraints. The rectangles representing the CSM's of stay constraints are drawn with double lines; their numeric labels represent a possible assignment of constraint strengths.

3.1 Solving algorithm

Representing variable priorities as stay constraints allows us to use, for example, the QuickPlan algorithm with hierarchical constraints to solve the system (Zanden 1996). We summarize the algorithm below.

The core of QuickPlan is an algorithm that solves the constraint system, to the extent possible, disregarding constraint strengths. This constraint *elimination* phase proceeds as follows: (1) find a constraint with a CSM that outputs to only *free variables*, i.e., variables that can be output only by that CSM; (2) add the method vertex and edges of that CSM to the solution graph; (3) eliminate the constraint, i.e. all of its method vertices, from the constraint graph; (4) repeat, using the remaining subgraph.

The elimination phase may or may not find a solution. In our example graph, v_{res} is the only free variable, and once it has been removed from the constraint graph, no free variables remain. In this situation, the weakest constraint (any one of them, if there are many of equal strength) is retracted from the current constraint graph, and the elimination phase is entered again. Note that required constraints cannot be removed. These two phases alternate until an empty constraint graph emerges (all required constraints have been satisfied), or until no progress can be made (a cycle of required constraints exists, indicating an unsatisfiable system). This approach is guaranteed to find a solution graph that can satisfy all required constraints, if one exists.

To find the least surprising solution, QuickPlan continues with an *improvement* phase. During this process, the algorithm assembles a new constraint graph consisting of all the constraints that are enforced in the current solution graph, plus the strongest retracted constraint, and retries the elimination phase (including retraction). If the strongest retracted constraint can be enforced, it is added to the solution; if not, it is discarded permanently. The elimination and improvement phases are alternated until the weakest retracted constraint is either added to the solution or discarded permanently.

The above algorithm will find the "best" solution graph, defined in terms of the *locally-predicate-better* comparator (Freeman-Benson and Maloney 1989) between two solution graphs. This comparator defines a solution a to be better than b if there is some constraint strength k, such that a satisfies all the constraints that b satisfies that are stronger than k, and a satisfies one more constraint of strength k than b does. This coincides with favoring flows that preserve the values of the highest priority variables.

QuickPlan has quadratic time complexity in the number of constraints; in practical constraint systems the algorithm is reported to perform in linear time (Zanden 1996).

Finally, the full constraint system formulation, as well as the use of stay constraints and QuickPlan are part of our prototype system. The deployed property model library currently uses a simpler algorithm that may reject certain cyclic constraint graphs that the prototype system can handle.

3.2 Capturing intent from the solution graph

When a constraint system is solved, new values for the property model variables are computed. For the purposes of synthesizing command parameters from a dialog, only the result variable is needed. This is not, however, the only useful data that is produced when the solver is run. The solution graph also gives a way to distinguish the variables intentionally supplied by the user from those that were calculated or those picked up from the surrounding context. This turns out to be essential, for example, when recording or playing back scripts.

Specifically, in our image-resize dialog, several variables are used for command parameter synthesis, even though the result variable will ultimately contain only the values of absolute height and width. But, when recording the outcome of the dialog in a script, however, we are interested in which values the user directly contributed. Consider the case where the preserve ratio flag is not set, and the relative values were changed more recently than the absolute values, say to 50% each. In this case, the resulting absolute width and height, 750 and 1050 are *derived* variables. The initial height and width variables were populated from the document model. We find it useful to refer to initial values like these, i.e., those that are picked up from a particular (document) context rather than the user, as *input* variables. The set of non-derived, non-input variables, that is, the set of variables supplied directly by the user, that contribute in some way to the result will be referred to as *contributing* variables.

What should be recorded in a script as a result of exercising the dialog as described above is the tuple consisting of relative height: 50, relative width: 50, preserve ratio: false. This way, when the script is run later against an image with initial dimensions 600 by 800, it will have the intended effect of scaling the image by 50 percent in each dimension, down to 300 by 400. Had we failed to recognize the user's intent, and stored the initial variables, or the absolute variables, the script would have incorrectly resized the second document to 750 by 1050.

Solution graphs, as we have described them so far, allow us to distinguish derived variables from non-derived variables; non-derived variables are those with no in-edges, except from possibly a node representing a stay constraint. Of non-derived variables, we need to also distinguish between input variables and contributing variables in order to capture intent. As only interface variables are given stay constraints in the constraint graph, an in-edge from a stay constraint for a given variable in a solution graph indicates that a variable is a contributing, rather than input, variable.

3.3 Solver preconditions

We note that finding an acyclic solution to an arbitrary multi-way dataflow constraint system is an NP-complete problem; complexity characteristics of variations of constraint satisfaction problems are described in (Trombettoni and Neveu 1997). However, the problem is polynomial time if the *method restriction* holds, which requires that every CSM of a constraint use all of the constraint's variables as either an input or output of the method. This restriction need not be obeyed in property model specifications, for example, in constraints that are optional.

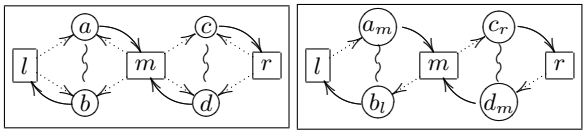


Figure 8. Translation that removes cyclic methods. Current values of the variables l, r, and m are hardwired into the functions of methods a, b, c, and d, which allows removal of the cyclic inputs to these methods.

Consider a case of command parameter synthesis where some variable x can be computed from another variable y, say, according to $x \leftarrow f(y)$, but x can also be specified directly. If y has a higher priority than x, then there is some relation r such that $r(x, y)$ must hold; but if x has a higher priority, nothing is required from x and y. The relation to enforce between variables x and y is thus $\top \vee r(x, y)$, which, of course, is always true. To model such an optional constraint in a way that does not violate the method restriction, we use two CSMs: one that computes $x \leftarrow f(y)$ as the programmer has specified, and another that computes y from x using a function that ignores the value of its input (x), and returns the current value of y instead. Note, this means the functions used in the execution phase may depend on the current values of the constraint system. The general transformation that we apply to optional constraints is to add a CSM, whose outputs are all the constraint's variables that are not outputs of any other CSM in the constraint, and whose inputs are all the constraint's variables.

Another feature disallowed in constraint graphs is the presence of methods whose input and output variables overlap—such methods introduce a cycle immediately, and can never be included in an (acyclic) solution graph. Constraints violating this restriction, however, are encountered frequently. For example, consider the left-hand side of Figure 8: a typical user interface bound to this kind of system would be three sliders, where the "middle" slider is guaranteed to always stay within the limits of the "left" and "right" sliders—moving any of the sliders will cause the others to move to retain this guarantee. For example, the CSM a computes $m \leftarrow \max(l, m)$. To remove the cyclic method (and enforce method restriction), we replace m on the right-hand side with a constant, the current value of m upon entering the execution phase, leaving us the CSM with one input and one output: $m \leftarrow \max(l, m_{\text{current}})$. All other cyclic methods are translated similarly; the result is shown on the right-hand side of Figure 8.

4. Property model system architectures

In Section 1 we stated that when incidental algorithms and data structures in command parameter synthesis code are replaced with explicit property models, reuse is greatly increased. A reusable property model library plays a central role in user interface architectures, relieving the programmer from the burden of the majority of user-interface related tasks. Instead the programmer focuses on the identification and declarative specification of the relevant variables and relationships unique to the commands used.

This section sketches the software architecture surrounding the currently deployed property model component implementation, and gives examples of how the use of explicit property models makes it possible to provide functionality that is reusable across different property model instances—and thus between different interfaces. In particular, we focus on reuse across both dialog handling code and script recording/playback, and the generic mechanism of capturing a user's intent in a script. We provide some measures to quantify the gains in reuse in Section 5.

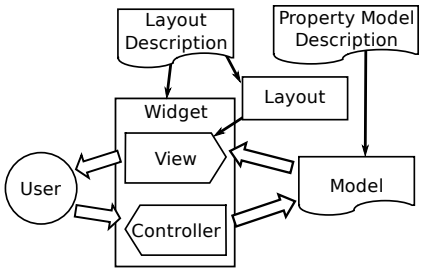


Figure 9. A high level overview of the property, layout and widget library components.

4.1 Architecture

As described in earlier sections, the algorithmic core of a property model library is the prioritized constraint solver. To this we add a declarative language for describing constraints and relationships on a collection of values (typically the parameters to an application command) together with a parser for this language. These components give clients a concise mechanism for explicitly modeling parameters for particular commands.

The property model library provides the model for a clean Model-View-Controller system. In our architecture, the views and controllers are furnished by widget component libraries. Since most widget libraries do not cleanly separate the model, view, and controller, a thin wrapper typically must be provided to adapt the widgets for use with the property model library. (For example, most user interface toolkits tightly couple view and controller functionality, and fail to distinguish, for update notification purposes, the cases where a value is set programmatically versus the cases where it is set by the user.) When bound to a user interface, the property model library provides the logic that controls the user interface behavior.

Systems that use the property model library for user interface command parameter synthesis typically also make use of an independent yet complementary library for widget layout (ASL 2005). The layout library consists of a solver and a declarative language for constructing a user interface. The layout solver takes into account a rich description of user interface elements to automatically achieve a high quality layout rivaling what can be achieved with manual widget placement. With the layout library, a single declarative user interface description is used for multiple operating system platforms and for localizing to different languages. The widget layout library is independent of the widget toolkits used, as explained in (Järvi et al. 2007). A high level overview of the components described so far is given in Figure 9.

4.2 Scripting

Program logic that needs to work with incidental data structures must be repeated several times. For example, the *validation logic*, rejecting invalid input values for individual user interface elements or for combinations of user interface elements, is often entangled with the widget event handling code. Industrial strength applications, however, must also support scripting, and the validation logic for parameters to script commands is typically re-implemented separately from that of the dialog handling code. Two parallel implementations of the same logic, written using different APIs, is a heavy burden and a very common source of defects. To work around this problem, applications may resort to hacks like having scripts launch a dialog, populate its widgets' values, and simulate events that trigger the dialog's validation code.

In system architectures using property models, this situation is notably improved. In Section 3.2 we explained the importance of capturing intent when generating scripts from user-dialog interaction, and how the intent can be obtained from the information captured in the solution graphs. In practice, clients of the property model library need not be aware of the intricacies of a constraint solver. A typical client use case might be as simple as: put up a modal dialog, let the user interact with it, and when the dialog is closed by pressing OK, use the returned data to synthesize parameters for an associated command and capture user's intent if recording a script. We describe a simplified version of the "Modal Dialog Interface Kit API" (ASL 2005), that supports this use case.

The modal dialog interface kit provides a single function that brings up a dialog and manages it with the assistance of the property model, layout, and widget libraries:

```
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 are entirely governed by these specifications.

When the dialog is dismissed, the caller receives the result in a **dialog_result_t** structure, containing (1) a complete parameter set (output variables of the property model), ready to be supplied to the command whose parameters the property model was designed to synthesize; (2) the set of contributing variables that reflect the user's intent in synthesizing the command parameters; and (3) the action that was used to terminate the dialog, for example, an indication as to whether the dialog was dismissed via an "OK" or a "Cancel" button. Enabling and disabling widgets, updating values of widgets as responses to changes in values of other widgets, determining the contributing values, and so forth, are all handled behind the scenes by the libraries: the user only provides the context variables along with the specifications for the property model and widget layout.

Playback of scripts also makes use of the property model library. The sequence of events is as follows. A recorded action in a script specifies which command to execute, together with the context-independent values that contribute to synthesizing the parameters for the command. When a script is executed in a particular context, e.g. against a certain document, the application constructs a property model for the command specified in the script, and populates the input variables of the property model from the context-dependent data, and the contributing interface variables from the values stored in the script. The application then asks the library to solve the property model and uses the resulting output variables to obtain the exact parameters for the command to be executed.

5. Experience

At the time of this writing, Adobe is at the beginning stages of a planned, wide-scale deployment of the property model library, in conjunction with a change of widget library components. It is still too early to judge the overall success of the deployment but we are starting to gather information.

In one application, the event handling and scripting code for a single dialog, which accounted for 781 statements (semicolon count) and contained five known logic defects, was replaced by a property model description with 46 statements. No defects in the resulting dialog have been found. Generally, a reduction of 8–10 to one in statement counts and improved quality are being reported. The long-term expectation is that more features will follow from

this conversion; for example, adding scripting support to those applications that do not currently provide it and improving the scripting support in those that do.

The conversion has so far progressed relatively smoothly, but not without difficulty. There have been a few instances where the existing logic was incomplete or inconsistent, forcing a redesign of the component and user interface. In the version shipped at the time of the initial deployment, some issues were encountered in the design; for example, in disabling user interface elements based on the state of the property model, the granularity for detecting invalid states is too coarse; in certain cases valid operations are disabled unnecessarily.

6. Related work

Systems based on declarative specifications are common in the area of user interfaces. The combination of procedural and declarative program code has found great success in, for example, GUI element layout. The most familiar example is perhaps HTML, CSS, and DOM combined with Javascript. The QTk library (Grolaux and Roy 2001) in Mozart/Oz, Glade (Feldman 2001), XUL (Deakin 2006; Mozilla 2006), XAML (XAML 2008), and XForms (Boyer et al. 2007) serve as examples. Some of the above systems, along with rule-based systems such as Drools (Proctor et al. 2008), Jess (Friedman-Hill 2008), and R++ (Litman et al. 2002), also support the specification of rules for maintaining consistency across values in user interfaces. These systems, like ours, offer the ability to express certain relationships concisely. Property models go beyond this, by not only providing the ability to create rules that assist in producing a valid result, but by providing an explicit model from which the client can programmatically capture notions like intent for script recording, and so on.

The above rule-based systems do not restrict the expressive power of the language specifying the rules. This lack of restrictions gets in the way of realizing the above benefits. For example, XForms supports declarative (one-way) constraints, but also provides open-ended support for user scripts (Javascript) to be bound to events that can, in turn, read and write arbitrary variables in the model. In addition to the dependencies between variables arising from the constraints, the scripts can hide arbitrary dependencies, leading to incidental data structures that cannot be analyzed.

Constraint systems have been studied extensively for use in user interfaces, mostly for automated element layout, but also for maintaining consistency across data in user interface elements, as in command parameter synthesis. A large number of declarative, constraint-based GUI systems have been proposed, for example, Sketchpad (Sutherland 1964), Amulet (Myers et al. 1997), Garnet (Myers et al. 1990), as well as ThingLab I and II, DeltaBlue, and SkyBlue (Sannella 1994), but, except for user interface element layout, they have generally not been adopted in industrial software development. The system described in this paper draws from this line of work; in particular, we use the algorithms and representations for hierarchical multi-way dataflow constraint systems (Zanden 1996).

The basic architecture of property models is based on the Model-View-Controller pattern (Krasner and Pope 1988). This pattern is identified as being important for the separation of concerns in user interfaces in a recent work (Goderis et al. 2007) aimed at untangling application logic from user interfaces.

7. Conclusions and future work

Ask nearly any software engineer what they most dislike in their work and the answer will be "building the user interface." Even working on products where an automatic layout library frees the engineer from mundane tasks such as placing a button at just the right pixel location, the effort to build the human interface is onerous. As code associated with the human interface accounts for nearly a third of the code necessary to implement a new feature for an application, increase in code reuse in this area has a significant impact on productivity.

Property models provide an explicit means for modeling what is traditionally managed with complicated event handling code. It reduces the amount of redundant logic and consolidates common logic for reuse and sharing. Based on our experiences with the property model approach so far, code that implements command parameter synthesis in user interfaces is typically replaced with a declarative description that can be one tenth of the size and have much reduced complexity of the original code.

Increasingly, applications are expected to share major user interface elements as they are targeted to move beyond working in isolated domains, to become parts of suites of application components supporting larger workflows. Integration and code sharing is made difficult because of the disparity between frameworks and object models used by individual applications. The result is that the code comprising the logic behind a user interface is re-implemented in each application's widget and event handling framework client code. The property model library offers a way to consolidate, move, and customize that logic easily between applications, regardless of the underlying framework.

Frequently, a user interface designer is responsible for designing the visuals in an application, perhaps accompanied by a textual description of the intended behavior. This design is then given to a programmer who "codes" both the layout and the behavior. It is only after the design is fully implemented by the programmer that satisfactory user testing can take place. We have prototyped visual tools using the layout and property model libraries to allow the designer to create the interface in a form that can be used directly by the application programmer. This can greatly improve the software construction process as it allows user interface designers to experiment and get feedback about the actual behavior of their designs, not just about the look and feel that they get with traditional builder tools, all without the delay incurred when involving a programmer.

A fundamental assumption in the work described in this paper is that by eliminating incidental data structures, and replacing them with explicit models, we can produce more robust software with very high levels of reuse. We have carried out this process for the incidental data structures and the code that operates on them in the domain of user interface programming, with a focus on command parameter synthesis. The process is not trivial, it requires much the same work as is involved when designing and implementing new reusable library components: a careful analysis of the nature of the relevant data structures and algorithms, an effort to find the correct abstractions, stepwise refinement, a good deal of trial and error, and so forth.

We will continue to refine the abstractions and algorithms for property models, explore alternative mechanisms for representing disjunctive constraints, automate model checking for guaranteeing desired characteristics of property models, work on improving and optimizing the solver algorithms for the particular kind of constraint systems that arise from property models, etc. Future plans include extending property models to support variables that are more complex than single data values, such as lists or trees. Not only will this support richer user interfaces, but it will open the door to property models being applied in the domain of document object modeling.

References

Cocoa Appliation Tutorial. Apple Inc., October 2007. URL `http://developer.apple.com/documentation/Cocoa/Conceptual/ObjCTutorial/`.

ASL. *Adobe Source Libraries*. Adobe Systems, Inc., 2005. URL `stlab.adobe.com`.

Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo. How reuse influences productivity in object-oriented systems. *Commun. ACM*, 39 (10):104–116, 1996.

Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. *SIGPLAN Not.*, 22(12):48–60, 1987.

John M. Boyer, Micah Dubinko, Jr. Leigh L. Klotz, David Landwehr, Roland Merrick, and T. V. Raman. XForms 1.0 (Third Edition), October 2007. URL `http://www.w3.org/TR/2007/REC-xforms-20071029/`.

Steve Burbeck. Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC), 1987. URL `http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html`.

Neil Deakin. XUL tutorial. Webpage, February 2006. URL `http://www.xulplanet.com/tutorials/xultu/`.

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

William B. Frakes and Giancarlo Succi. An industrial study of reuse, quality, and productivity. *Journal of Systems and Software*, 57(2):99–106, June 2001.

Bjorn N. Freeman-Benson and John Maloney. The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver. *Computers and Communications, 1989. Conference Proceedings., Eighth Annual International Phoenix Conference on*, pages 538–542, March 1989.

Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Commun. ACM*, 33(1):54–63, 1990.

Ernest Friedman-Hill. Jess 7, February 2008. URL `http://www.jessrules.com/jess/charlemagne.shtml`.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Sofie Goderis, Dirk Deridder, Ellen Van Paesschen, and Theo D'Hondt. DEUCE: A declarative framework for extricating user interface concerns. *Journal of Object Technology*, 6(9):87–104, October 2007. URL `http://www.jot.fm/issues/isue_2007_10/paper5/`. Special Issue: TOOLS EUROPE 2007.

Donatien Grolaux and Peter Van 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.

Jaakko Järvi, Matthew A. Marcus, and Jacob N. Smith. Library composition and adaptation using C++ concepts. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 73–82, New York, NY, USA, 2007. ACM.

Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

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.

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

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

Brad A. Myers and Mary Beth 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.

Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The Amulet environment: New models for effective user interface software development. *Software Engineering*, 23(6):347–365, 1997. URL `citeseer.ist.psu.edu/article/myers96amulet.html`.

Derek L. Nazareth and Marcus A. Rothenberger. Assessing the cost-effectiveness of software reuse: A model for planned reuse. *Journal of Systems and Software*, 73(2):245–255, October 2004.

Parasol 2008. *Property Models Research Project's Home Page*. Parasol Lab, Computer Science, Texas A&M University, 2008. URL `http://parasol.cs.tamu.edu/groups/pttlgroup/property-models`.

Sean Parent. *Adobe Property Model Library*. Adobe Systems, Inc., 2005. URL `http://stlab.adobe.com`. Part of Adobe Source Libraries.

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

Mark Proctor, Michael Neale, Bob McWhirter, Kris Verlaenen, Edson Tirelli, Fernando Meyer, Alexander Bagerman, Michael Frandsen, Geoffrey De Smet, Toni Rikkola, Steven Williams, Ben Truit, Ritu Jain, Chinmay Nagarkar, and Denis Ahearn. Drools, 2008. URL `http://www.jboss.org/drools/`.

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

Ivan 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.

Gilles Trombettoni and Bertrand Neveu. Computational complexity of multi-way, dataflow constraint problems. In *IJCAI (1)*, pages 358–365, 1997.

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

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