Customizing Model-Based User Interfaces

Raphael Hoffmann
University of Washington
Seattle, WA, USA
raphaelh@cs.washington.edu

Krzysztof Gajos
University of Washington
Seattle, WA, USA
kgajos@cs.washington.edu

Daniel S. Weld
University of Washington
Seattle, WA, USA
weld@cs.washington.edu

ABSTRACT
Previous studies show that user interface customization can significantly increase user productivity [3]. However, as interfaces get more complex and as diverse computing devices are supported, customization becomes progressively harder to provide from the programmers’ point of view and increasingly tedious from users’ perspective. To solve these problems, we implemented a model-based customization framework that supports a variety of customization commands such as moving, copying and deleting parts of the interface, setting default values or skipping dialog windows. A customization which is performed on one device is available on any device in any part of any application which uses SUPPLE to generate its interface. We also demonstrate a sound algorithm for undoing previous customizations in any order, despite the complex interdependencies among subsequent customizations. Finally, we introduce a technique that helps users save time and effort by generalizing previous customizations to similar situations, even across different applications and devices.

INTRODUCTION
Today’s software grows more and more complex with each new release as new features are added to address specialized needs of different user populations. Consequently, the user interfaces that are shipped for these applications frequently miss essential needs of most individual users. Customization is seen as an important means of alleviating this problem. However, customization capabilities are expensive to include from the programmers’ point of view, despite support provided by various platforms (e.g. easy management of toolbars is supported by platforms like KDE). As a result, customization is supported non-uniformly throughout a single application and especially across applications. Furthermore, many beneficial techniques, which are commonly used elsewhere, are not available in the customization process. Most notably, there is no standardized undo capability — frequently the only way to undo a previous customization is to initialize the interface back to the factory default and start customizing anew. Furthermore, there are few tools to conveniently manage repetitive customizations which affect several aspects of a single application, span across different applications, or apply across different hardware devices. Thus we see need for platform-level support for a large range of customization capabilities, that could be available across all applications and devices without requiring much effort on the part of programmers. In this paper, we present three contributions:

1. An implemented, model-based customization framework, built as an extension of our SUPPLE [4] platform. Our system provides uniform customization support for multiple devices and across applications. Furthermore, it enables a rich set of customizations: users can copy, delete or move any interface element anywhere within the application. Users may also set default values anywhere within the interface, specify a widget to be used for a given UI element, or instruct the system to skip certain dialog windows.

2. An algorithm for undoing previous customizations in any order despite complex interdependencies, which often exist between the customization being deleted and the subsequent customizations. We prove that our undo algorithm is sound, runs in polynomial time, and generates the minimal necessary sequence of reversal actions.

3. A technique that helps users save time and effort by generalizing previous customization requests to other similar situations, potentially across different applications and devices.

BACKGROUND
Our customization framework is an extension to SUPPLE, a model-based user interface generation platform. In this section we briefly summarize the relevant
Figure 1: Overview of SUPPLE (without customization) showing a fragment of a graphical representation of a functional specification for an internet application requiring login. The top level Login element is an action: its parameters include information about the user and the target host, while its result type describes the interface element that is to appear after the action is performed, and is not shown in the figure. The intermediate nodes (User and Host) are of a container type and they hold two primitive elements each.

Aspects of our previous work and introduce basic concepts, which will be referred to throughout the paper. SUPPLE [4] treats user interface generation and adaptation as a decision-theoretic optimization problem, where the goal is to minimize the estimated user effort for manipulating a candidate rendering of the interface. SUPPLE takes three inputs: a functional specification of the interface, a device model and a user model. The functional specification defines the types of data that need to be exchanged between the user and the application. The device model describes the widgets available on the device, and the cost functions, which estimate the user effort required for manipulating supported widgets with the interaction methods supported by the device. Finally, a user’s typical activities are modeled with a device- and rendering-independent user trace. SUPPLE’s rendering algorithm is a combination of a constrained branch-and-bound search and constraint propagation.

The functional specification is hierarchical in nature with primitive types (i.e. numbers, strings, booleans) at the leaves. The primitive elements are combined into larger objects by elements of container type. Derivative types are constructed by constraining the domains of either primitive or container elements. Finally actions have a parameter type and a return type – the former specifying what information that needs to be provided for an action to be executed and the later describing the new interface element that should appear as a result of executing the action. Figure 1 shows a graphical representation of a functional specification for a login action for an internet application.

MODEL-BASED CUSTOMIZATION

To incorporate customization features, we extended SUPPLE’s model-based approach by introducing an intermediate user interface representation, the customized specification (Figure 2). SUPPLE now works in two phases. First, it applies a sequence of previously stored customization actions to the functional specification in order to create a customized specification tree, which may have a very different structure. For example, the customized tree may omit parts of the functional tree, may contain multiple copies of certain functional subtrees, and may contain constraints on how to present UI elements. In the second phase, SUPPLE uses the device and user models to convert the customized tree into an “optimal” rendering of the interface.

What kind of customization actions does SUPPLE support? Analysis of a large set of customization examples led us to identify a small set of primitive actions (Figure 3), which may be combined to produce most any customization example we had seen. Users may copy or remove any piece of functionality to another part of the UI (elements can be moved by copying then removal). Default values may be specified for any input element, and a dialog window can be automatically skipped, as long as the user can confidently set default values for required parameters. Finally, the user can specify what concrete widgets are to be used to render any element of the interface.

**Example 1:** Suppose a user wanted to add her own version of one-touch printing (which is predefined as a button in many MS applications). She might first copy the element that invokes the print command to a more convenient location (CopyElement), set the default printer in the corresponding dialog box (SetDefault), and then delete the copied dialog box altogether (RemoveElement). Combining these customizations results in a UI where printing with default settings is ac-
Figure 4: A customization plan which enables one-touch printing

cessable through a conveniently located one-touch trigger. □

Internally, SUPPLE stores customizations explicitly in a constrained sequence called a plan (Figure 4). Plans allow SUPPLE to apply the customizations to any interface, including ones that the user (and SUPPLE) have not yet seen. The plan representation also allows SUPPLE to reason about complex interdependencies among customization actions, when out-of-order undo is performed. Each action in the plan contains detailed information about the type of the customization, the UI elements affected, as well as information to enable reversibility.

In the next section, we demonstrate how customizations can be undone in any order, despite complex interdependencies between actions in the customization plan. Next, we present a generalization algorithm that allows SUPPLE to save user time by automatically proposing to generalize a customization so it applies other parts of the functional specification — for example, generalizing a change in one application’s “Save As” dialog to other, similar applications.

UNDOING CUSTOMIZATIONS

Most applications provide some sort of “undo” capability whereby recent changes can be reversed, restoring the system to its previous state. For example, typing “Z” in Microsoft Word might replace a recently deleted block of text. However, many UI customization systems don’t provide a consistent way to undo changes. For example, “Z” doesn’t remove a recently added button from a Microsoft Office toolbar, nor does it reverse the action of pinning an application to the XP toolbar. In order to succeed in developing a flexible UI substrate capable of “deep deployment” then it must support uniform capabilities. Specifically, with respect to customization, we identify two desiderata:

1. The system must be capable of undoing every customization command that the user can issue (follows from the Golden Rule #6 of UI design [12]).

2. The user should be able to undo any previously issued customization, not just the most recent one.

Most previous systems store user customizations on a stack, and their undo facilities pop the stack, reverting the state one customization at a time. While this approach affords a simple conceptual model to the user, it is especially limiting for a deeply-deployed, integrated UI framework that underlies many intertwined applications. Suppose a user uses customization to create one-touch printing (as illustrated in Figure 4) in her word processor. Next she changes the default style and changes the font control widget to a pulldown menu. If later she changes her mind about one-touch printing customization, should she have to undo all her subsequent customizations as well?

In contrast SUPPLE represents user customizations as a (random access) sequence of customization actions. One might think that this representation would make out-of-order undo very simple to implement — in order to undo , SUPPLE could just remove it from the sequence. Unfortunately, this simple approach fails, for several reasons. First, it doesn’t support redo without additional structures. More importantly, it fails because there may be dependencies between customization commands in sequence.

Example 1 (Continued): Suppose that the user had executed the plan of Figure 4’s, and now wished to undo the default-setting customization, o2. Recall that removing a print confirmation dialog is not possible, unless default print settings have been previously defined. Thus SUPPLE must recognize that removing o2 renders o3 inconsistent, and react accordingly, e.g., warning the user or undoing o3 as well. □

Supple maintains the current state of the user interface with a triple: \( U = \langle S, P, C \rangle \). Recall that \( S \) is a tree of elements, \( e_i \), as shown in Figure 2. \( P \) is a plan, a totally ordered sequence of customization-related actions. Finally, each constraint \( C_{ij}^{ok} \) in the constraint set, \( C \), has two parts. \( o_k \) is the action posting the constraint, and \( x_i \) denotes a conjunctive constraint expression using \( =, <, >, \leq, \geq, \neg \) operators, constants, and a single element property. The plan and constraint parts of \( \langle S, P, C \rangle \) have two uses: they are used for undo, and also as a record of the user’s customizations so that these customizations may be applied to other functional specifications in the future.

A plan may contain any of three types of actions: customizations (described in this section) as well as reversals and generalizations (described below and in the next section, respectively). We use \( a_i \) to denote an action of any type. Every action is formally modeled as a function from one user interface \( U \) to another; thus \( a_i(U) = U \) denotes execution of \( a_i \). The most common actions are customizations, denoted \( o_k \); indeed, all three of the actions in Figure 3 are customizations. Executing a customization action does three things: 1) it changes \( S \) (e.g., by copying a specification subtree, setting an element’s widget type, etc), 2) it postends itself to \( P \);
\[ U \text{ user interface state: } (S, \mathcal{P}, \mathcal{C}) \]
\[ S \text{ current state of the customized spec (e.g., Fig 2) } \]
\[ o_i \text{ an individual customization action } \]
\[ r_{o_i} \text{ the reversal of } o_i, \text{ i.e. an action whose effect } \]
\[ \text{(computed from } o_i \text{'s undo info) is the inverse of } o_i \]
\[ \mathcal{P} \text{ a plan } (a_1, \ldots, a_n) \text{ where action } a_i = o_i \text{ or } r_{o_i} \]
\[ x_i \text{ a conjunctive expression (used in constraints) } \]
\[ c_0^{x_i} \text{ a constraint: } o_i \text{ is posting expression } x_i \]
\[ \mathcal{C} \text{ a set of constraints: } \{ \ldots, c_0^{x_i}, \ldots \}. \]

Figure 5: Summary of notation

and 3) it may add a constraint to \( \mathcal{C} \). If such a new constraint makes \( \mathcal{C} \) unsatisfiable, then it is illegal to execute the action, and SUPPLE will refuse, leaving \( (S, \mathcal{P}, \mathcal{C}) \) unchanged.

The Undoing Algorithm

Reversals, denoted \( r_{o_i} \), are the second type of action. Each type of customization (e.g., as listed in Figure 3) has a schema for its own reversal. For example, the reversal of a CopyElement customization simply removes the copied element, while the reversal of a RemoveElement adds the removed element back to its original parent container. Reversing a SetDefault (or SetWidget) action is more complex, since changing the default value (or required widget) of an element to the most recently named value requires iterating backwards over the plan.

Intuitively, executing \( r_{o_i} \) inverts the effects of a previously executed customization \( o_i \). More precisely, executing \( r_{o_i} \) first checks if its execution will violate constraints; if not, then SUPPLE does following things: 1) updates \( S \) as described in the previous paragraph, 2) the action, \( r_{o_i} \), is added to the end of \( \mathcal{P} \), incrementing its length, 3) if \( o_i \) posted any constraints, these are deleted from the set \( \mathcal{C} \).

Sometimes, however, executing a reversal, \( r_{o_i} \), will violate a constraint, \( c_0^{x_i} \). Indeed, this is what happens in Example 1 — reversing \( o_2 \), the first SetDefault in Figure 4, violates the constraint expression \( printer\_default \neq \emptyset \) posted by RemoveElement, \( o_3 \). In this case the user is given a chance to retract the undo or supply a new default value; however, if she does not then SUPPLE will recursively generate a subplan to eliminate the constraint (e.g., reversing the action which posted it — \( o_3 \) in this simple example). The net effect will be the execution of a sequence of reversal actions, which are also appended to the growing customization plan for future reference. In our example, the resulting plan will be \( (o_1; o_2; o_3; r_{o_1}; r_{o_2}) \). Note that these last two reversal actions remove all constraints originally posted by \( o_2 \) and \( o_3 \). In summary, SUPPLE’s undo algorithm takes as input a user interface \( U \) and a customization \( o_k \) to be undone.

---

**Algorithm 1 ConsistentUndo((S, P, C), \( o_k \))**

1. \( (S, \hat{P}, \hat{C}) := r_{o_k}((S, \mathcal{P}, \mathcal{C})) \)
2. while exists a violated condition in \( \hat{C} \) do
3. let \( c^{o_m}_m \in \hat{C} \) be a violated condition with minimum index \( m \)
4. \( (S, \mathcal{P}, \mathcal{C}) := \) ConsistentUndo((S, P, C), \( o_m \))
5. \( (S, \hat{P}, \hat{C}) := r_{o_k}((S, \mathcal{P}, \mathcal{C})) \)
6. end while
7. return \( (S, \hat{P}, \hat{C}) \)

We now state propositions showing that our algorithm is sound, runs in polynomial time, and generates the minimal sequence of reversal actions necessary to undo an action. Suppose \( U = (S, \mathcal{P}, \mathcal{C}) \) is the current interface. Let \( U_0 = (S_0, \emptyset, \emptyset) \) denote the initial user interface: \( S_0 \) is the original functional specification, and sequentially executing the actions of \( \mathcal{P} \) on \( U_0 \) yields \( U \). A user interface \( (S, \mathcal{P}, \mathcal{C}) \) is consistent, if for all constraints \( c^{o_k}_m \in \mathcal{C} \) the expression \( x_i \) holds. An interface is coherent if for all \( m, 1 \leq m \leq |\mathcal{P}| \) the result of sequentially executing the first \( m \) actions of \( \mathcal{P} \) on \( (S_0, \emptyset, \emptyset) \) returns a consistent interface. In other words, a coherent interface is guaranteed to have been consistent at all steps in its evolution.

For the following propositions, let \( ((S, \mathcal{P}, \mathcal{C}), o_k) \) be the input to ConsistentUndo. We assume that \( (S, \mathcal{P}, \mathcal{C}) \) is coherent and that \( o_k \in \mathcal{P} \). Proofs appear in the paper’s longer version.

**Proposition 1 (Complexity).** ConsistentUndo will terminate; furthermore, its runtime complexity is \( O(n(n - k)^2) \) where \( n \) is the plan length and \( k \) is the index of the customization being undone.

Note that when the customization being removed is the most recently executed action, then \( k = n \) and ConsistentUndo is constant time. If the age of \( o_k \) is bounded, then undo is linear in the length of \( \mathcal{P} \).

**Proposition 2 (Correctness).** ConsistentUndo returns a coherent user interface \( (\hat{S}, \hat{P}, \hat{C}) \) in which \( o_k \)’s effects are absent.

**Proposition 3 (Minimality).** Suppose ConsistentUndo returns \( (\hat{S}, \hat{P}, \hat{C}) \). Suppose there exists a consistent user interface \( (S, \mathcal{P}, \mathcal{C}) \) based on the same functional specification, in which 1) there exists \( \mathcal{P} \) such that \( \hat{P} = Append(\mathcal{P}, \mathcal{P}) \) and 2) \( o_k, \mathcal{P} \in \mathcal{P} \). Then \( |\hat{P}| \leq |P| \).

Guiding Users through Chained Undos

Most undo’s are simple and can be accomplished by adding a single reversal. Yet if the interface has many constraints, SUPPLE’s proposed undo plan may contain many unexpected reversals. While SUPPLE’s programming interface lists reversals as they are performed, target users will be very confused. Thus we are building
an interface that will afford users the opportunity to intercede when undos chain.

GENERALIZING CUSTOMIZATIONS

Supple also has the novel capability to generalize user customizations, proposing additional changes to the user. Generalization is powered by a machine learning technique, called version space algebra [6], which suggests a set of similar UI elements in \( S \), where customization might be desirable.

Example 2: Suppose the user customizes \((o_1)\) her word processor’s print dialog, setting the duplex variable’s default true. Later she sets \((o_2)\) the “double-sided” variable’s default true in a different application. Supple will now propose similar changes to other applications — despite differences in the content and organization of print dialog boxes in those applications. □

Learning a Generalization’s Scope

Every time a user issues a customization, \( \alpha_i \), the action is saved as a training example, and used to learn a concept that defines the parts of the UI where a similar customization might be useful.\(^1\) Each training example may be thought of as a feature vector, describing the UI element, which was modified. Features include the current application, the hardware device in use, and the tree structure of the functional and customized specifications.

Generalizing training examples yields a set of elements, \( \{e_i\} \subset S \) where an analogous customization might be usefully applied. However, Supple doesn’t represent these elements extensionally, because we wish to be able to apply the generalization to other interfaces with their own functional specifications and thus a completely different set of UI elements. Instead, generalizations are represented intensionally with a conjunctive description called a scope. To bias Supple’s learner, only nine atomic conditions may be used in a scope and each conjunct must reference a different primitive. The permissible primitives are shown in Figure 6; as one can see, this bias is quite expressive.

Example 2 (Continued): The user has made duplex printing the default in two instances, and wants to let the scope cover several printer drivers and applications. Figure 7 shows four different functional specifications, corresponding to four different print dialog windows. Setting the scope to all elements named “Duplex” is not sufficient, since it does not cover Figure 7 b). Setting it to all elements named “Duplex” or “Double-sided” falsely selects two elements in Figure 7 d). If the scope also requires ancestor elements “Print” and “Finishing”, it finally contains the intended set of elements. □

---

\(^1\)Later in this section, we explain how to handle situations where the user interleaves examples from multiple concepts.
Guiding Users through Generalizations

After SUPPLE computes a generalized scope, it gives the user the choice to accept or discard the generalization. But how does the scope get represented on the screen? SUPPLE currently displays scopes in a programmer’s syntax, which is difficult for novices to read. We are starting to build a more intuitive interface, that will present scopes using an English-like, intentional description, combined with an extensional description that lists or highlights affected elements. We are also exploring ways of letting the user browse possible generalizations and correct the system by clicking on elements that should be negative or positive examples.

Undoing Generalizations

In order to extend the ConsistentUndo algorithm to handle generalizations, we must confront three questions: 1) How can a generalization action be reversed? 2) May generalizations post constraints, and what should be done if they are violated? 3) What happens if the user (directly or indirectly) reverses a customization which was used as a training example for computing a generalization? Previously, we discussed two of the three kinds of actions that comprised a plan $P = (a_1, a_2, \ldots, a_n)$: customizations $a_i$ and reversal commands $r_{a_i}$. We now focus on the third type: generalizations $g_i$. Each generalization has an associated scope, which represents a set of concrete customization actions: $\{g_j, \ldots, g_k\}$. When a plan $P$, containing a generalization $g_i$, is executed on a customized specification, SUPPLE enumerates all the $a_x$ in $g_i$’s scope and applies each $a_x$ sequentially. For example, $(a_1, a_2, g_3)$ might expand into $(a_1, a_2, g_3, -1, a_3, -2, a_3, -3)$. This new plan no longer contains generalizations, and can therefore be used directly as input to the ConsistentUndo algorithm. Now, if the user wishes to undo a generalization, SUPPLE individually undoes each of the concrete customizations which were instantiated by the generalization. Thus, ConsistentUndo is executed multiple times.

Intuitively, one may think of a generalization as posting the union of the constraints posted by the concrete customizations in its scope. There are three possible ways to recover if one of these constraints, $c_{g_i}$ is violated: 1) undo $a_x$ but otherwise don’t change the generalization, 2) undo the generalization completely, and 3) undo the generalization and then compute a new generalization using $a_k$ as a negative training example. At present, SUPPLE uses strategy one, because it minimizes dis-

---

Example 2 (Continued): When SUPPLE calls VersionSpaceUpdate on $o_1$’s target, the “Duplex” element in the customized specification of Figure 7 a), the atomic ANFR version space has the empty prefix in its upper bound set (selects all elements), and $\{\text{Main}, \text{Print}, \text{PrintDialog}, \text{Finishing}, \text{Options}\}$ in its lower bound set. Next, when VersionSpaceUpdate is called on $o_2$’s target, the “Double-sided” element of Figure 7 b), the lower bound set contains only $\{\text{Main}, \text{Menu}, \text{Print}, \text{PrintDialog}, \text{Finishing}, \text{Options}\}$, whereas the upper bound set remains unchanged. Using SUPPLE’s composite version space structure, this automatically induces the scope, containing the shaded elements of a), b), c) and d).

After its scope has been computed, the third type of action, a generalization $g_3$, is added to the customization plan, much like a regular customization issued by the user. When $P$ is executed on a new functional specification, the scope definition of $g_3$ is used to generate a concrete set of UI elements, where the customization should take place. SUPPLE now attempts to execute each ground customization in turn, but if a ground customization would violate a precondition, it is skipped.

---

2The user also may discard or revise a generalization, when it is applied to a new interface.

3Surprisingly, the order in which the $a_x$ are executed does not matter.
ruption to the UI. However, this approach may confuse users, so we plan a user study to compare the alternatives. When the user explicitly undoes a customization, however, SUPPLE uses approach three and learns a new generalization.

**Learning Multiple Concepts**

Frequently, a user interleaves two (or more) unrelated types of customizations. For example, she might change the printer default in several applications. Meanwhile, she might eliminate certain types of confirmation dialog. This example illustrates a problem with SUPPLE’s generalization algorithm. As presented so far, SUPPLE generalizes by using every past customization as a training example for a single concept it is trying to learn. In fact, the user may best be thought of as interleaving training for two concepts. Unless SUPPLE recognizes this duplicity, it will make terrible predictions.

Fortunately, the problem can be solved simply: a clustering algorithm divides training examples into multiple groups, representing distinct concepts. Now version-space learning is applied on a cluster-by-cluster basis. To apply clustering, however, SUPPLE needs a metric function to estimate the “distance” between two customizations; we compute this distance $D(o_i, o_j)$ as follows. If $o_i$ and $o_j$ are instances of different primitive customizations (e.g., one a SetWidget and the other a CopyElement), then the distance is infinite. Otherwise, SUPPLE uses VersionSpaceUpdate to determine the least general hypothesis, $H$, containing both $o_i$ and $o_j$. $D(o_i, o_j) = \frac{|H|}{|H_0|}$ where $|H|$ denotes the number of elements in $H$ and $|H_0|$ is the number of elements in the unconstrained hypothesis.

**PRELIMINARY EVALUATION**

We conjectured that generalization-based customization is useful in large environments where different interfaces have similar patterns. Because printer drivers often have similar features, but no standardized user interface, we chose them for our study. We issued a survey to 13 subjects in the CS Department at the University of Washington asking how frequently they use non-standard printer settings (e.g., duplex, several pages to one, color, non-default quality, landscape, non-default paper-tray) for different applications, devices, and printers.

We created finite-state machine models for Xerox and HP printer drivers, so that we could compute the minimal number of clicks required to print a specific job in the manner preferred by a given subject. We computed the average number of clicks for each user based on their preferences and reported distribution of jobs. For comparison, we next performed exhaustive search over the space of customizations to find four interfaces that were optimal for a given user. Each of these four interfaces was created by finding the best sequence of two SetDefault customizations, given certain restrictions (we got four “best” interfaces because we considered four different restrictions). The first type of interface, best static, was required to be the same for all printers and applications, while driver-specific allowed different defaults for different printers (useful sometimes if the subject used different printers differently). With the application-specific restriction, defaults could only depend on the application initiating printing. With generalization-based customization, SUPPLE used its version space to compute the scope of the customization.

After measuring the performance of the different SetDefault customizations, we repeated the study considering sequences of two CopyElement customizations. In all cases, generalization-based customization yields the best results, since SetDefault customizations can generalize to non-trivial scopes.

**FUTURE WORK**

While this paper has described the technical underpinnings of a model-based customization framework, many usability issues are unresolved. Therefore, the bulk of our future work will be iterative evaluation and refinement of the new interface by which users will control generalization and undo chaining. Our ultimate goal is a comprehensive user study, demonstrating the benefits of our framework.

In addition, there are open algorithmic issues. At present, SUPPLE keeps growing the plan without bound; this will eventually affect customization’s linear-time
performance. However, causal-link analysis techniques from the automated planning community promise to enable compression mechanisms by which SUPPLE can apply a very short summary plan to new functional specifications.

RELATED WORK

Our work follows decades of research on model-based, user-interface design. The classical work, summarized in [14], focused mostly at helping the designer – consequently those systems provided no end-user customization support, though some systems allowed the designers to mold the design algorithm to conform more closely to their personal design style [2]. An interesting attempt to give the end-user some control within the design-centric approach was to learn user’s task model by observing their interaction with a full-featured application and then using the learned model to automatically compose a customized version of the application from the individual building blocks [8]. The user was given a custom-built application that accurately reflected her needs but rather than just changing the user interface, that paper proposed scaling down the very functionality of the application. Additionally, it required that the learned model be sent back to the software vendor as a template for assembling the custom-build application.

More recent systems like PUC [10] or the “UI on the Fly” [11] create UIs individually for different users but also do not support end-user controllability.

The generalization aspect of our work draws on earlier research on programming by demonstration (for a good overview of the field see [7]) and in particular uses the version space algebra developed initially for a text editing domain [6]. While the earlier systems tended to assist with user’s actual task, ours helps with the meta-task of configuring one’s work environment.

The role of intelligent assistance in user interfaces attracts a lot of controversy [13] but there is evidence that if deployed carefully, it can benefit the user [5] and there is other research proposing the use of automatic assistance for more effective UI customization [1].

CONCLUSIONS

As the complexity of applications, and the number of used devices increases, the need and effort required for users to effectively customize their user interfaces increases. Incorporating customization features into applications, however, is often time-consuming for developers and can lead to inconsistencies where customizations are available at some locations but not at others. This paper addresses these issues by proposing a novel model-based customization framework, which provides uniform and powerful support for multiple applications and across hardware devices.

- We implemented a model-based customization framework, which provides uniform and powerful support for multiple applications and across hardware devices.
- We presented an algorithm that can undo any past customization, not just the one most recently issued, and we prove that our undo algorithm is sound, runs in polynomial time, and generates the minimal necessary sequence of reversal actions.
- We described a generalization mechanism that uses machine learning to compute an expanded scope, suggesting extensions of a user’s past customizations. A preliminary user study shows that generalization offers the potential of significant benefits for user convenience.

Acknowledgments This research is supported by NSF grant IIS-0307906 and ONR grant N00014-02-1-0932. Thanks to Donald Patterson and Kate Everitt for comments.

REFERENCES


