MORE for Less: Model Recovery from Visual Interfaces for Multi-Device Application Design

Yves Gaeremynck, Lawrence D. Bergman, Tessa Lau
IBM TJ Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532 USA
+1 914 784 7946
bergmanl@us.ibm.com

ABSTRACT

An emerging approach to multi-device application development requires developers to build an abstract semantic model that is translated into specific implementations for web browsers, PDAs, voice systems and other user interfaces. Specifying abstract semantics can be difficult for designers accustomed to working with concrete screen-oriented layout. We present an approach to *model recovery*: inferring semantic models from existing applications, enabling developers to use familiar tools but still reap the benefits of multi-device deployment. We describe MORE, a system that converts the visual layout of HTML forms into a semantic model with explicit captions and logical grouping. We evaluate MORE's performance on forms from existing Web applications, and demonstrate that in most cases the difference between the recovered model and a hand-authored model is under 5%.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *user interfaces.* D.2.2 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *restructuring, reverse engineering, and reengineering.*

General Terms: Algorithms, Design, Human Factors, Experimentation.

Keywords: Model recovery, reverse engineering, semantic modeling, rule systems, multi-device application development.

1. INTRODUCTION

Developing applications for multiple devices is a difficult task. Devices have widely varying characteristics, input mechanisms, and output capabilities. Designing a single application to be delivered on devices as diverse as a desktop browser, a handheld PDA, and a voice interface raises unique challenges for the application designer. In an attempt to facilitate the development and maintenance of multi-device applications, many development frameworks ([5],[8],[12],[13]) force developers to design applications using an abstract specification language that can be automatically transformed into executable applications for each target device. In the PIMA system [3], for example, rather than designing an application using a device-specific widget set, designers author abstract models that specify abstract user interface elements at a semantic level, such as multi-way choices and input questions with explicit caption and hint text. PIMA automatically converts abstract models into applications by mapping each interactor (abstract widget) in the model to the appropriate device-specific widget. For example, an n-way choice that prompts for the user's country could be rendered in a web form as a drop-down list whereas a voice interface could have the user speak the country name. A complete description of the process of converting an abstract model into device-specific applications can be found in [3].

A major barrier to multi-device application development is that designers must be trained on a new suite of content-creation tools. These abstract design tools can be unnatural for designers accustomed to working with concrete UI elements and layouts. In addition, this approach only works for newly created applications; legacy applications must be reimplemented from scratch.

To address these problems, we propose an automatic solution for inferring abstract models from existing applications—a process we call *model recovery*. Simply identifying individual UI elements in an application is not sufficient; model recovery must induce semantic *relationships* between elements in the application (Figure 1). In the figure, the text "Opportunity Listings" is a caption of the select-many list of position titles, and the month/year selectors for degree date are grouped as a logical

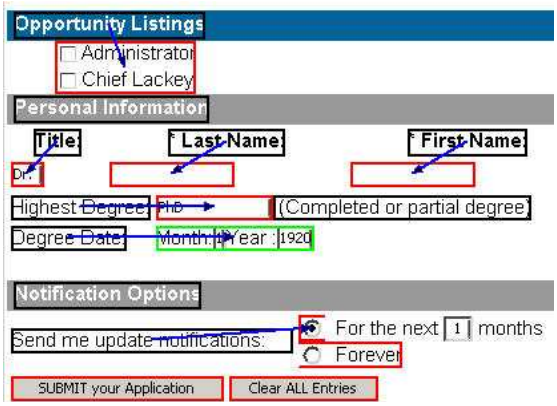

Figure 1. Semantic model of a web form shown in MORE's visual model editor. Boxes indicate strings or interactors. Arrows represent relationships, such as captions, between elements.

IUI'03, January 12–15, 2003, Miami, Florida, USA.

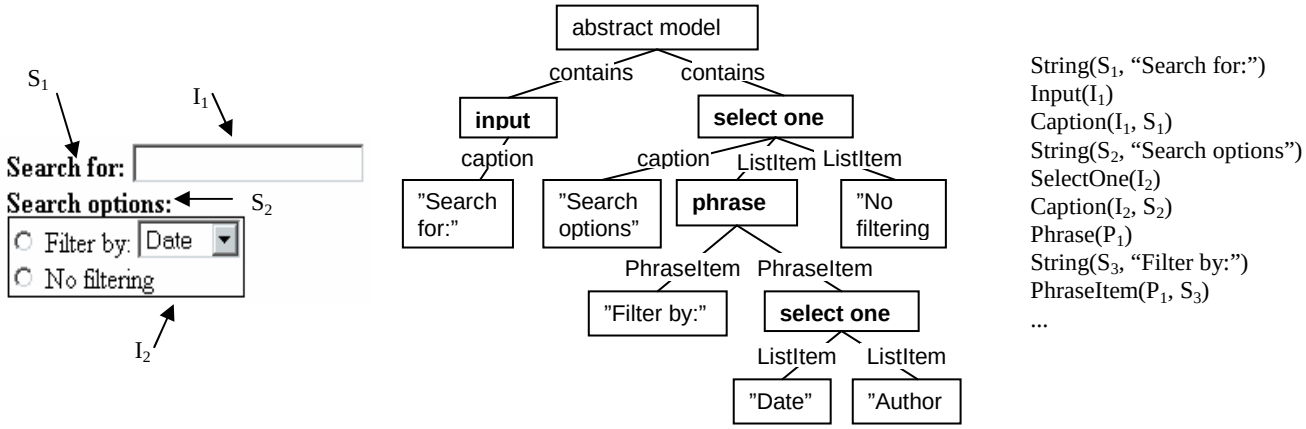

Figure 2. A sample HTML form, the corresponding abstract semantic model represented as a tree, and the same model represented as a collection of facts.

phrase. These semantic relationships are based on common patterns we have observed in UI design.

On the surface, model recovery resembles a parsing problem: building parse trees from a sequence of symbols. However, our problem is complicated by the fact that UI elements are laid out in a two-dimensional user interface. Thus, traditional one-dimensional parsing algorithms are unsuitable for model recovery.

We believe an effective model recovery system must be:

- Interactive: designers must be able to update their applications and see the results quickly; and
- Extensible: as new UI design patterns emerge, the system must be reconfigurable to recognize them.

This paper presents MORE, an interactive, extensible rule-based approach to model recovery from visual user interfaces. We have constructed a set of rules that capture common design patterns in web application forms; while specific to the HTML domain, we believe that these rules are easily extended to cover new features or non-HTML interfaces. We first give a formal definition of the model recovery problem. Next we describe our algorithm and its implementation. We then evaluate the performance of our system on HTML forms drawn from existing web applications, and conclude with ideas for future work.

2. THE MODEL RECOVERY PROBLEM

Let a concrete application $A = \{F_A\}$ where $\{F_A\}$ is a collection of facts in predicate form describing entities (strings and widgets) in the original application, relationships between those entities (such as containment), stylistic properties (font size, weight, style) of those entities, and their geometry (width, height, position). In the case of HTML we extract these facts from the Document Object Model (DOM) of the web page; similar object models can be extracted, with more difficulty, from Java or Windows applications, either by runtime traversal of the widget hierarchy or through visual generalization [10].

Let an abstract model $M = \{F_M\}$ where $\{F_M\}$ is a set of facts describing entities in the model and relationships between those entities. An entity in an abstract model is either a string or an abstract interactor such as an input field, a select-one list, or a container object. Entity relationships include caption assignments, membership in a multiple-choice interactor, and group containment. Every abstract model has an equivalent tree representation in which nodes correspond to entities and edges correspond to relationships between the entities. Note that the structure of an abstract model, which captures semantic relationships, may be very different from the structure of the original application, which is based on presentation layout.

A model recovery system defines a mapping $f(A) \rightarrow M$ that maps a concrete application A to an abstract model M.

This paper describes MORE, a system that converts HTML forms to abstract models for use in the PIMA system. For example, Figure 2 shows a simple web form and the abstract model recovered from this application in both tree and fact representation. Although the current implementation converts each HTML form into a separate abstract model, we expect that extending it to handle multi-page applications will be straightforward.

3. AUTOMATED MODEL RECOVERY

We view model recovery as a search through the space of all possible models. A model is a set of facts. Each state corresponds to a partial model. The initial state contains the starting set A. Each search operator augments a partial model by adding facts to it. The allowable operators at each state add facts that can be derived from and are consistent with the current model. The goal state is the highest-scoring model out of all models that could be generated from the application.

We assume that each entity plays a unique role in the abstract model. For example, if a string is a caption for one interactor, it cannot also be a hint for the same interactor (see Figure 3). (This is why the abstract model can be represented as a tree instead of a DAG or graph.) Two facts that assign different roles to the same entity are *mutually exclusive*. Due to mutual exclusion, the order in which facts are added to a partial model is important; a fact selected at a given step may preclude adding a higher scoring fact later on.

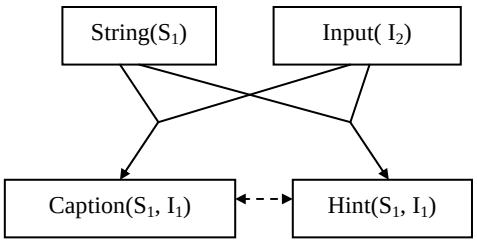

Figure 3. Dependency graph illustating mutual exclusion

MORE is based on a forward chaining rule system. Standard rule-based systems use a fixed heuristic to resolve mutual exclusion conflicts. In contrast, optimal conflict resolution in model recovery often depends on properties of the interactors involved in the conflict, such as relative position or text style. MORE handles mutual exclusion using a look-ahead strategy that reasons about potential conflicts, and resolves conflicts using a scoring system.

Table 1 presents an overview of the MORE algorithm. Input consists of a set of initial facts and a set of rules. Facts are kept in a *working memory*. At any given moment, every fact in the working memory must be either:

- Selected: part of the abstract model,
- Eliminated: not part of the abstract model, or
- Unprocessed.

The working memory may contain conflicting facts, but no selected fact may conflict with another selected fact. When the algorithm terminates, the set of selected facts forms the completed abstract model.

Domain knowledge is expressed as three types of rules:

- *Deduction rules*, which produce new facts from selected facts,
- *Exclusion rules*, which determine whether two given facts are mutually exclusive, and
- *Scoring rules*, which assign scores to facts based on properties of the interactors involved.

The algorithm proceeds as follows. At initialization, the working memory is loaded with an initial set of facts representing the concrete application *A*, which are all selected facts. Deduction rules are iteratively applied to the selected facts in the working memory. These deduction rules capture common patterns of UI design in the target domain, such as combining a string and an input into a captioned text input, or grouping selectors together into a phrase. The facts generated by each iteration of the algorithm are added to the working memory as unprocessed facts.

To select facts that are to be added to the model, our hill-climbing algorithm performs look-ahead by constructing *conflict closures*. A conflict closure is a transitive closure of facts with respect to mutual exclusion. In other words, it is a minimal collection of facts such that each fact conflicts with at least one other fact in the closure but none outside of it. Facts that conflict with no other fact form singleton conflict closures. Thus, conflict closures form a partition of the working memory.

Input:
A collection *initialFacts* of initial facts
A collection R_d of deduction rules
A collection R_s of scoring rules
A collection R_e of exclusion rules

Output:
An abstract dialog

```
1   /* Initialization: */
2   WorkingMemory memory = {};
3   for f ∈ initialFacts do
4       addFact( memory, f );
5
6   /* Execution cycle. */
7   while containsUnprocessedFact( memory ) do
8
9       /* Create conflict closures and select one. */
10      ClosureList list = getConflictClosures( memory, Re );
11      Closure c = selectConflictClosure( list, memory );
12
13      /* Select the best non conflicting subset. */
14      for fact ∈ selectBestSubset( c, memory, Rs ) do
15
16          /* Generate the deductions. */
17          for f ∈ getDeductions( Rd, memory, fact ) do
18              addFact( memory, f );
19
20  /* Termination. */
21   return getSelectedFacts( memory );
```

Table 1. Pseudocode for the model recovery algorithm

The algorithm selects the conflict closure deemed least likely to create conflicts in the future and performs *conflict resolution* on it. Using the scoring rules, conflict resolution selects the best subset of the closure that does not contain incompatible facts. The facts in the highest-scoring consistent subset are marked as selected. The remainder of the conflict closure (facts which are now inconsistent with the selected facts) is eliminated. Deduction rules are applied to the newly selected facts and the loop continues.

The algorithm terminates when all facts have been either selected or eliminated. It returns the consistent set of selected facts, which form an abstract model containing interactors, groups and their properties. The following subsections describe the steps of the algorithm in more detail.

3.1 Conflict closure selection

Conflict closure selection (Table 1, line **10**) is based on the notion of completeness. A conflict closure is said to be *complete* if the selection of an unprocessed fact in the working memory cannot produce, directly or indirectly, a deduction that conflicts with the contents of the closure. If the algorithm selects a complete closure, it ensures that no future deductions will conflict with this local decision. Conceptually, conflict closures partition the page

Method:
selectConflictClosure
Input:
A collection *closures* of conflict closures
A working memory *memory*
Output:
The conflict closure to test during the cycle

```
1   /* Find a complete conflict closure. */
2   for closure ∈ closures do
3      markAsComplete( closure );
4      TypeList types = getTypes( closure );
5      TypeList prerequisites = getPrerequisites( types );
6
7      for p ∈ prerequisites do
8         if containsUnprocessedFactOfType( memory, p )
          then
9            markAsIncomplete( closure );
10           break for;
11
12     if isMarkedAsComplete( closure ) then
13        return closure;
14
15  /* Return a default conflict closure. */
16  return getOldestClosure( closures );
```

Table 2. Pseudocode for the closure selection algorithm

into locally ambiguous sections, enabling us to apply a divide-and-conquer approach, resolving each section in turn.

We illustrate conflict closure selection using the example in Figure 2. Assume the working memory contains three selected facts:

String(S_1), the string built from S_1
String(S_2), the string built from S_2
Input(I_1), the input interactor built from I_1

and three unprocessed facts:

A = Caption(I_1, S_1), a caption built from S_1 and I_1
B = Caption(I_1, S_2), a caption built from S_2 and I_1
C = Input(I_2)

A, **B**, and **C** form two conflict closures: {**A**, **B**} and {**C**}. {**A**, **B**} involves two mutually exclusive caption assignments, which depend on strings and interactors. As **C** represents an interactor that has not been processed yet, our algorithm considers the {**A**, **B**} closure incomplete. {**C**} involves the construction of a new interactor. No unprocessed fact may produce a deduction that conflicts with **C**, therefore this conflict closure is considered complete. As a result, our algorithm examines {**C**}. After fact **C** is processed, two new potential captions are generated:

D = Caption(I_2, S_2), the caption built from S_2 and I_2
E = Caption(I_2, S_1), the caption built from S_1 and I_2

Having made these deductions, the working memory contains four conflicting facts: **A** conflicts with **B, B** conflicts with **D**, **D** with **E**, and **E** with **A**. Thus, **A**, **B**, **D** and **E** belong to the same conflict closure {**A**, **B**, **D**, **E**}. During the next execution cycle, the conflict closure {**A**, **B**, **D**, **E**} will be selected as complete.

During a single execution cycle, the algorithm typically produces more than one conflict closure. During the first cycle, for instance, each initial fact is a member of its own conflict closure, since the facts do not conflict. The algorithm then selects one of these closures for further examination.

To determine whether a conflict closure is complete, one must execute the deduction rules for all unprocessed facts in the working memory, which would be computationally infeasible. Instead, we use a conservative heuristic for approximating a complete conflict closure, presented in Table 2. The heuristic relies on knowing the fact types consumed and produced by each deduction rule. Using these type dependencies, our algorithm computes a dependency graph over deduction rules and fact types (Figure 3). The dependency graph is used to calculate the prerequisites of a conflict closure (Table 2, lines **4** and **5**). If the working memory contains an unprocessed fact whose type is a prerequisite for a conflict closure, one of its deductions could potentially conflict with the content of the closure. Therefore, the closure is not considered complete (Table 2, lines **9** and **10**). This heuristic is sound but not complete. The closure it selects is guaranteed to be complete; however, if a complete conflict closure exists, the algorithm is not guaranteed to select it.

In certain cases, the closure selection algorithm may not find a complete conflict closure. This situation may occur if there is a cycle in the dependency graph. In such cases, we heuristically select a conflict closure by choosing the closure with the oldest fact.

3.2 Conflict resolution

After a conflict closure is selected, the conflict must be resolved. Conflict resolution selects the best subset of the closure that does not contain incompatible facts. The algorithm assigns scores to the facts using scoring rules associated with each fact, then selects the non-conflicting subset with the highest score. Scores are functions of properties and relationships between the interactors, such as distance, direction, size, font style, and textual content.

A complete solution to conflict resolution would consider all possible subsets of the conflict closure. The complexity of this algorithm scales exponentially with the size of the conflict closure, which is a problem since the closures manipulated by MORE may be large. For example, closures involving caption and hint assignments may include many of the elements on a page.

Instead, we perform a greedy search to resolve the conflict, as shown in Table 3. At each step, the resolution algorithm selects the fact from the closure with the highest score, removes it and its alternatives from the closure, then repeats until the conflict closure is exhausted. Although this approach is not guaranteed to compute the optimal conflict resolution, it worked well in practice on the web pages we tested.

In the example given in the previous section, the conflict closure {**A**, **B**, **D**, **E**} contains two consistent subsets: {**B**, **E**} and {**A**, **D**}. The latter has a higher score, based on proximity and orientation of the captions with respect to their associated interactors.

Method:
getBestSubset

Input:
A conflict closure *closure*
A working memory *memory*
A collection R_s of scoring rules

Output:
The locally optimal subset that does not contain incompatible facts

```
1   FactList list = {}
2
3   /* Iterate until the conflict closure is empty. */
4   while ! isEmpty( closure ) do
5
6     /* Select the best fact. */
7     Fact bestFact;
8     float bestScore = 0;
9     for fact ∈ closure do
10      float score  = getScore( fact, list, memory, Rs );
11      if score > bestScore then
12        bestFact = fact;
13        bestScore = score;
14
15    /* Add it to the set and update the closure. */
16    list.add( bestFact );
17    markAsSelected( bestFact );
18    remove( closure, bestFact );
19    for fact ∈ getAlternatives( bestFact ) do
20      remove( closure, fact );
21      markAsEliminated( fact );
22
23  return list;
```

Table 3. Conflict resolution algorithm

```
if      are_close( string, interactor )
and     not interactor_between( string, interactor )
then    new Neighbors( string, interactor );

if      Neighbors( string, interactor )
and     is_valid_caption( string )
then    new Caption( string, interactor );

if      Neighbors( string, interactor )
and     is_valid_hint( string )
then    new Hint( string, interactor );
```

Table 4. Deduction rules for caption and hint assignments

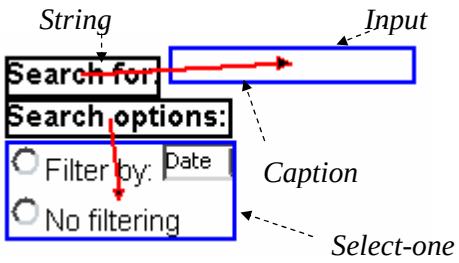

Figure 4. User interface for editing recovered models

4. IMPLEMENTATION

We have implemented the MORE model recovery system as part of the PIMA framework for writing and deploying multi-device applications [3].

We began by collecting a set of 98 HTML web forms. We attempted to ensure broad coverage by selecting pages that differ both in their goals (registration, search, survey forms, etc.) and their design. From this collection, we chose 50 at random and used them to design MORE's rule set; the remainder were held out as a test set for the evaluation described in the next section. Each page provides three sources of information: the DOM structure of the web page, the style of the UI elements, and their geometry. The observations collected during analysis of the training set were translated into a rule set containing 33 deduction rules, 3 exclusion rules, and 4 scoring rules. The rules are implemented in Java.

Table 4 shows pseudocode for three of the deduction rules in MORE that generate caption and hint assignments. These rules are based on the observation that captions and hints always occur in the vicinity of their corresponding interactor. In addition, a string is very unlikely to be associated with an interactor if there is another interactor between them. Therefore, when building pairs of neighboring strings and interactors, the first rule checks for the presence of possible obstacles. This test is rather complex because it requires knowledge of the complete content of the page. The second rule builds captions from neighbors and the third one deduces hints from strings. Both rules perform additional tests not shown in the table: for instance, we observed that captions are never enclosed in parentheses.

Exclusion rules are straightforward and not shown here. Scoring rules are more complex, but we omit them due to space limitations. These rules typically take into account the style of the interactors and their relative distances, orientations, and positions. For example, a italicized hint has a higher score than a boldfaced hint, and a caption relationship between a string and a nearby interactor has a higher score than the same string with a distant interactor.

4.1 User Interface

We have designed a user interface that visualizes the recovered abstract model by overlaying markup directly on a rendered web page (Figures 1 and 4). Rectangular boxes of different colors represent interactors, strings, and groups. Arrows link these boxes to show the various property assignments.

The UI allows designers to modify the automatically constructed model in cases where MORE makes incorrect inferences. Boxes and arrows can be created, moved, or deleted; these changes are then propagated to the underlying model.

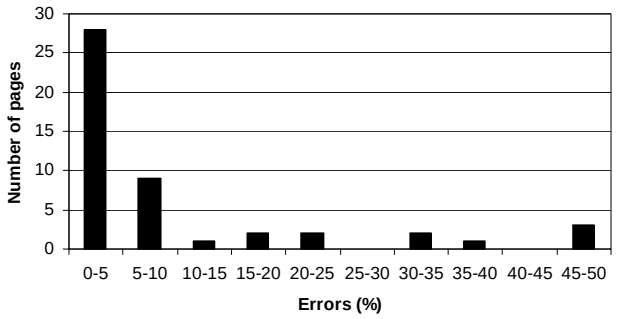

Figure 5. Histogram of evaluation results, showing the number of pages versus the percent errors on a page.

5. EVALUATION

In this section, we present an empirical evaluation of the accuracy of our system. We measure the performance of our algorithm on a given page by comparing the model computed by MORE with an ideal model of the same page created manually.

In order to compare two models, we first convert the model into its canonical tree representation (see Figure 2 for an example). Nodes in the tree represent entities (strings and interactors), while edges represent relationships between entities. We use a standard *edit distance* metric to compare two trees, which counts the number of editing operations required to convert one tree into the other. The edit distance depends on the set of allowable editing operations. For a conservative metric, we limited the set of operations to node additions and deletions. For instance, if a piece of text is recognized as a caption when it should be a hint, two mistakes are counted: the removal of the node representing the caption, and the addition of the same node as a hint. The number of mistakes is then normalized by the number of nodes in the correct model. We do not count all nodes, however. For instance, we ignore list items inferred from options contained in HTML `<select>` elements. This is because it is common for a page to contain `<select>` elements with a large number of options, and recognizing them correctly is trivial. Underestimating the size of the web page in this way makes our metric even more conservative.

We evaluated the performance of our approach on a test set of 48 pages. The ideal models contained on average 80 nodes; the largest model contained 173 nodes. Figure 5 shows the results of the evaluation. Overall, MORE recovered the correct model from 40% of the web pages with no error. 53% of the web pages were recovered with less than 5% error, while 77% of the web pages contained less than 10% error. As the figure shows, MORE can handle a large percentage of web pages with few mistakes. These results confirm that it is possible to identify common patterns in user-interface design, and use these patterns to infer abstract models from existing web applications. Although the figure shows that some pages were recovered with a large fraction of errors, the edit distance metric is misleading because the number of errors increases rapidly whenever MORE incorrectly groups UI elements. Each element must first be removed from the group it belongs to and then added to the correct group. In particular, the rule that groups checkboxes into a selectable list did not correctly identify large checkbox groups on several pages, resulting in several pages with very low accuracy.

Although MORE does not recover models with 100% accuracy, mistakes can be easily corrected using MORE's visual model editor. In addition, we are currently investigating techniques for automatically applying the same changes to future models generated from the same application.

6. RELATED WORK

Many research projects have addressed the problem of re-engineering user interfaces, particularly in the domain of web pages. HTML is interesting for many reasons: its popularity, its simplicity, the availability of an object model, and the power of its content creation tools.

We distinguish between two types of re-engineering tools: reverse engineering and re-authoring systems.

Reverse engineering tools extract content from user interfaces. Several such tools extract a model-based representation of a user interface. Vaquita [4] addresses the problem of making a web site accessible to a wide range of computing platforms. It allows developers to translate web pages into a model-based representation according to multiple reverse engineering options. Vaquita focuses on the XIML [16] presentation model of the web site: it extracts the hierarchy of the UI elements contained in HTML pages and their layout relationships. Semantic relationships between elements are not inferred automatically and must be specified by the user. MORE, on the other hand, explicitly captures semantics such as captions and hints.

Ware [2] aims to simplify the maintenance of a web application spanning multiple web pages by converting it to a UML representation. Ware extracts the architecture of the application, the dynamic interactions, and typical scenarios of use based on an automatic static analysis, and user-driven dynamic and behavioral analyses. While Ware focuses on dynamic interaction, MORE is concerned with identifying static relationships between elements on a single page. Similarly, Mathaino [11] uses execution traces to migrate a legacy application to a new platform, focusing on user interaction rather than static layout.

Paganelli and Paterno [6] present a system that extracts task models from multi-page web applications. Their system traverses the DOM of each page, extracting relevant tags and links and transforming them into a hierarchical model. The rules in this system unambiguously transform DOM nodes into model nodes, capturing only relationships explicitly specified in the HTML. In contrast, MORE infers semantic relationships between page elements and utilizes additional information such as geometry and style.

Re-authoring techniques apply a set of transformations to a user interface. Therefore, their output is necessarily a user interface of the same type as the input. For example, Digestor [1] provides device-independent access to a web site. It is implemented as a proxy that automatically applies transformations such as elisions to make the page fit into a browser. Re-authoring techniques are applied recursively. At every step, the search process determines which transformations may be applied and an evaluation function selects the document with the smallest display area requirements.

Like Digestor, ReWeb [5] applies rewrite rules to web pages but with the perspective of improving their maintainability, usability,

and portability. Rewrite rules may target a single page, such as the correction of mistakes in the DOM, or multiple pages, such as the reorganization of a web site into frames.

Several companies and consortia are actively designing and standardizing languages to represent device-independent applications: XForms [15], and UIML [14] are two such languages. Our approach to model recovery from concrete applications enables developers to continue using familiar content-creation tools, while the PIMA framework can be extended to translate the abstract model to any of these target platforms.

Although we considered formulating model recovery as a constraint-satisfaction problem (e.g., [8]), we believe that expressing these transformations as a set of rules is more natural. Our approach to building a model using a set of rules is similar to a standard forward-chaining rule engine, but with several differences.

Most of the deduction rules are aggregation rules. For example, transforming a set of checkboxes into one or more selectable lists requires partitioning the set of checkboxes according to proximity and style. Although it is possible to write declarative rules to perform these aggregations, the logic of each of them must be broken into several rules. As a result, the size and the complexity of the rule set increases rapidly.

Many deduction rules are complex and are difficult to express declaratively. For instance, a string is very unlikely to be the caption of a given interactor if there is another interactor in between. Therefore, the rule that finds captions needs to be aware of the complete content of the user-interface to return a relevant result.

More importantly, the generated model may not contain two mutually exclusive deductions. For instance, if one rule deduces that a given string is a caption while another one assigns it as a hint, only one deduction—preferably the best one—should be kept. The standard way to express mutual exclusion in a rule system is by using negative conditions. However, even with negative conditions, it is difficult to control which rule instance will be chosen, since this choice typically depends on a fixed heuristic encoded into the rule engine such as selecting the rule with the fewest conjuncts. A better solution consists of executing all conflicting rule instances and adding another set of rules to pick the best non-conflicting combination of deductions. This solution provides better control over the deduction process. However, it is very complex to implement it in a declarative manner. Depending on the features offered by the forward-chaining system, implementation might not even be possible. In both solutions, rules must implement the logic that performs the aggregations as well as the logic that handles mutual exclusion. As a result, the rule set is difficult, if not impossible, to develop and maintain. In contrast, our approach uses conflict closures to handle mutual exclusion, and incorporates scoring rules to dynamically select the best rule based on characteristics of the interactors involved.

7. CONCLUSIONS AND FUTURE WORK

We have presented MORE, an interactive, extensible rule-based approach to the model recovery problem: translating a visual application into an abstract application model for use in multi-device application development. Model recovery is feasible because it is possible to identify design rules (i.e., patterns) that are common to visual applications. We have constructed a set of rules that capture common design patterns in web application forms; we believe that these rules could be easily extended to cover new features or non-HTML interfaces.

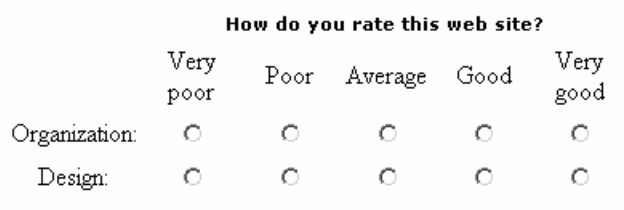

Figure 6. A form whose interactors share captions

Our algorithm is based on a modified forward-chaining rule engine that incrementally builds the model by generating new facts at each step. Domain knowledge is encoded in the system by three sets of rules. Deduction rules assign semantics to UI elements and aggregate them. Mutual exclusion rules specify which deductions conflict which each other. Scoring rules determine which facts should be preferred over others based on characteristics of the interactors involved.

Future work on MORE will include enriching and refining the rule set so as to improve the quality of the generated models. In addition, we plan to investigate techniques for modeling non-visual elements of application interfaces, such as inferring abstract event-handlers from JavaScript code. Our assumption that each user interface element has a unique role does not hold in all cases. Figure 6 presents such a case: it shows a portion of survey form that contains two lists of radio buttons. Both lists share the same captions—the strings across the top. In the future we will investigate how to model this kind of application in our system.

Another interesting possibility is to add learning capabilities to the model recovery process. We could use feedback from human correction of the system's mistakes to automatically update the weights in the scoring rules. This mechanism would allow us to automatically determine the weights that produce optimal results, and additionally it would allow MORE to adapt to the design conventions of a particular UI designer or web site.

8. REFERENCES

[1] Bickmore, T.W., Shilit, B.N., Digestor: Device-independent Access to the World-Wide-Web, Proceedings of the 6th WWW Conference, 1997.

[2] Di Lucca, G.A., Di Penta, M., Antoniol, G., Casazza, G., An Approach for Reverse Engineering of Web-Based Applications, Proceedings of WCRE '01, pp. 231-240.

[3] Bergman, L.D., Banavar, G., Soroker, D., Sussman, J., Combining Handcrafting and Automatic Generation of User-Interfaces for Pervasive Devices, Proceedings of CADUI III (2002), pp. 155-166.

[4] Bouillon, L., Vonderdonckt, J., Souchon, N., Recovering Alternative Presentation Models of a Web Page with VAQUITA, Proceedings of CADUI '02, pp. 311-322.

[5] Puerta, A. and Eisenstein, J., Towards a General Computational Framework for Model-Based Interface Development Systems, Proceedings of IUI 99, pp.171-178.

[6] Paganelli L., Paterno, F., Automatic Reconstruction of the Underlying Interaction Design of Web Applications, Proceedings of SEKE ’02, pp. 439-445.

[7] Ricca, F., Tonella, P., Baxter I.D., Restructuring Web Applications via Transformation Rules, Proceedings of SCAM ‘01, 150-160

[8] Sannella, M, SkyBlue: A Multi-Way Local Propagation Constrain Solver for User Interface Construction, Proceedings of UIST ’94, pp. 137-146.

[9] Singh, G., Kok, C. and Ngan, T., Druid: A System for Demonstrational Rapid User Interface Development, Proceedings of UIST 1990, pp. 167-177.

[10] St. Amant R., Lieberman H., Potter R., Zettlemoyer L., Visual Generalization in Programming by Example, Communications of the ACM, v.43 n.3, 107-114, March 2000.

[11] Stroulia, E., Kapoor, R.V., Reverse Engineering Interaction Plans for Legacy Interface Migration, Proceedings of CADUI ’02, pp. 295-310.

[12] Sukaviriya, P., Foley, J., and Griffith, T., A Second Generation User Interface Design Environment: The Model and the Runtime Architecture, Proceedings of ACM INTERCHI'93, pp.375-382

[13] Szekely, P., Luo, P., and Neches, R., Beyond Interface Builders: Model-Based Interface Tools, Proceedings of ACM INTERCHI'93, p.383-390.

[14] UIML, http://www.uiml.org

[15] XForms, http://www.w3c.org/Markup/Forms/

[16] XIML, http://www.ximl.org/