

2006

UNIFORM: Automatically Generating Consistent Remote Control User Interfaces

Jeffery Nichols
Carnegie Mellon University

Brad A. Myers
Carnegie Mellon University

Brandon Rothrock
Carnegie Mellon University

Follow this and additional works at: <http://repository.cmu.edu/hcii>

This Conference Proceeding is brought to you for free and open access by the School of Computer Science at Research Showcase @ CMU. It has been accepted for inclusion in Human-Computer Interaction Institute by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

UNIFORM: Automatically Generating Consistent Remote Control User Interfaces

Jeffrey Nichols, Brad A. Myers, Brandon Rothrock

Human Computer Interaction Institute

Carnegie Mellon University

5000 Forbes Avenue

Pittsburgh, PA 15213

{jeffreyn, bam, rothrock}@cs.cmu.edu

<http://www.cs.cmu.edu/~jeffreyn/uniform/>

ABSTRACT

A problem with many of today's appliance interfaces is that they are inconsistent. For example, the procedure for setting the time on alarm clocks and VCRs differs, even among different models made by the same manufacturer. Finding particular functions can also be a challenge, because appliances often organize their features differently. This paper presents a system, called *Uniform*, which approaches this problem by automatically generating remote control interfaces that take into account previous interfaces that the user has seen during the generation process. Uniform is able to automatically identify similarities between different devices and users may specify additional similarities. The similarity information allows the interface generator to use the same type of controls for similar functions, place similar functions so that they can be found with the same navigation steps, and create interfaces that have a similar visual appearance.

Author Keywords

Automatic interface generation, consistency, familiarity, handheld computers, personal digital assistants, mobile phones, personal universal controller (PUC), Pebbles

ACM Classification Keywords

D.2.2 Design Tools and Techniques: User interfaces – automatic generation. H5.2. User Interfaces: Graphical user interfaces (GUIs).

INTRODUCTION

The number and diversity of computerized appliances in our homes and offices is greatly increasing. Even basic interactive elements, such as light switches, are being aug-

mented with processing capability, which allows a formerly simple appliance to have complex behavior and even be controlled wirelessly. A challenge for consumers is to learn how to use all of these new computerized devices, especially when appliances of the same type often have substantially different user interfaces [3]. A common problem when traveling, for example, is being stumped by the user interface for setting the alarm on the clocks in hotel rooms, even though people may not have trouble setting their own alarm clocks at home.

Appliance interfaces could be more usable if they were consistent [3], meaning that users could take their knowledge of an appliance they have used in the past and apply it to a new appliance with similar capabilities. Unfortunately, this is not simply a problem of representing identical functions in the same way. Many appliances have similar functions with a few extra features. For example, the user may be familiar with a copier with only one kind of stapling, and then be confronted with a new copier that can staple on any corner. How might additional functionality be presented to the user while still providing an interface that is consistent with the old one?

This paper presents *Uniform*, Using Novel Interfaces For Operating Remotes that Match, a system that automatically generates appliance interfaces that are *personally consistent*, meaning that the interfaces generated for each user are consistent with that particular user's past interface experiences (see Figure 1). Uniform attempts to use similar representations for the same function, including substituting previous labels for similar functions. If two appliances share many of the same functions, Uniform will also try to create a consistent *organization* so that users can navigate in much the same way on both appliances. For appliances that share nearly all of the same functions, Uniform will also attempt to generate interfaces with a similar visual appearance.

The personalized aspect of Uniform means that different users may see different interfaces for the same appliance depending on their interface history. Our architecture allows for users to change their consistency profile if they prefer the interaction style of a different appliance. It is

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CHI 2006, April 22-28, 2006, Montréal, Québec, Canada.

Copyright 2006 ACM 1-59593-178-3/06/0004...\$5.00.



Figure 1. User interfaces generated by Uniform for a complex and simple copier without and with consistency.

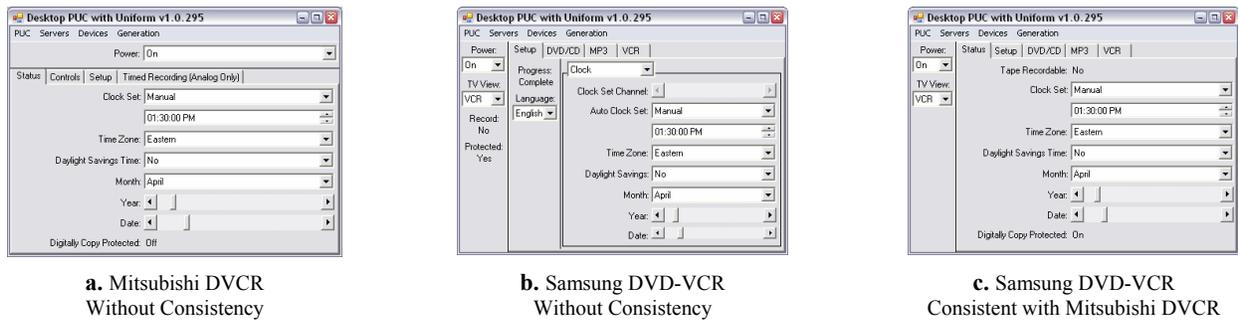


Figure 2. User interfaces generated by Uniform for a Mitsubishi DVCR and a Samsung DVD-VCR without consistency and the Samsung DVD-VCR generated to be consistent with the Mitsubishi DVCR. Note that the clock functions are located under the Status tab for the Mitsubishi DVCR, under Setup for the Samsung DVD-VCR, and in a new Status tab in the consistent interface. Also note that Controls and Timed Recordings from the DVCR are located under the VCR tab on the Samsung DVD-VCR.

even possible to choose a different representation for a particular function without affecting where that function is placed in the user interface.

Of course, Uniform cannot modify the physical interfaces on existing appliances. Instead users interact with appliances through remote control interfaces presented on a handheld computer, such as a PDA or mobile phone. Using a handheld has several benefits:

- The handheld provides a consistent way to interact with *all* electronics, and allows Uniform to keep track of every remote control interface that the user has seen.
- The handheld provides a consistent platform for interaction, including a standard UI toolkit and a set of user interface guidelines.
- Handhelds have rich interface hardware that would not be economical to put on every appliance, such as color LCDs, touch-sensitive screens, and text-entry technologies. This hardware can make the creation of high-quality appliance interfaces easier.

Uniform is implemented on top of our Personal Universal Controller (PUC) system [8], which generates remote control interfaces for handhelds and connects to appliances. The PUC specification language defines the functional and organizational representations of appliances.

In this paper, we start by putting Uniform in context with other systems that have addressed user interface consistency. A key requirement is the ability to find similarities between PUC appliance specifications, and our design is informed by two studies on the variability found in appliance specifications. In the first study, we compare specifications of three different VCRs written by the same author to understand how specifications can vary between appliances. The second study compares several specifications for the same appliance written by paid subjects to understand how specification authors can affect the design of a specification. We use our studies and previous work in interface consistency to establish a set of requirements for creating consistent interfaces.

Based on these requirements, we designed and implemented Uniform. Uniform is made up of a rule-based system to gather similarity information automatically by comparing new specifications to the set of specifications already seen, a knowledge base that stores both automatically-inferred and user-specified mappings, and a set of interface generation rules that ensure consistency by inspecting the mapping information and making changes to both the abstract and concrete interfaces during generation.

Uniform has been tested with specifications for copiers (see Figure 1) and VCRs (see Figure 2) on the PocketPC, mobile phone, and desktop platforms. Although copiers and VCRs are a subset of the appliances that exist today, our experience from the PUC project suggests that these appliances, especially VCRs, may be representative of the most complex appliances. The issues we have experienced with VCRs have carried over to many other appliances, such as car navigation systems, DVD players, and others.

RELATED WORK

Consistency has been a subject of research for the user interface community for many years, and there has been much debate about what consistency is. According to Grudin, “a two-day workshop of 15 experts was unable to produce a definition of consistency” [4]. Reisner said that consistency is loosely defined as “doing similar things in similar ways” and that inconsistency occurs when “the designer and the competent user employ different assignment rules” [12].

A number of methods have been developed to help ensure interface consistency. Platform interface guidelines and toolkits, like those developed by Apple for the Macintosh, help designers know how to make their applications consistent with others on the same platform. Usually these guidelines work best for common functions, such as defining a standard menu structure that makes it easy to open and save files and access clipboard functions. More general consistency guidelines have been proposed, such as those described by Nielsen [10]. These include maxims such as “the same information should be presented in the same location on all screens.” Another process for maintaining consistency is called “design languages” [13], which are used by designers to ensure that common features and branding are shared across a family of products.

Systems have addressed user interface consistency in two different areas: evaluation of consistency and generation of consistent user interfaces. Sherlock [6] and GLEAN [5] are two examples of systems that address evaluation of consistency. Sherlock uses a heuristic approach to evaluate task-independent qualities of user interfaces for consistency. GLEAN makes predictions about human performance on a user interface based on a GOMS model, which can accurately predict transfer times between tasks and find consistency problems between similar tasks. Uniform borrows some of Sherlock’s heuristic techniques, such as maintain-

ing the use of similar labels, but does not use a model with enough power to employ GLEAN’s evaluation techniques.

ITS, SUPPLE, and DiamondHelp are all systems that address the consistency of user interfaces with automatic design. The ITS system [17] was successful in producing consistent interfaces across a family of applications (such as all the displays for a World’s Fair) and for multiple versions of the same application. Interfaces were generated using a rule-based approach, and consistency resulted because the rules applied the same interaction technique in every place where the same condition was found. Note that although Uniform does achieve some consistency by using the same generation rules for each interface, it cannot rely on the underlying appliance models being the same. Appliances of the same type may have substantially different underlying models, and one of Uniform’s contributions is finding similarities between underlying models and creating consistency based upon these similarities.

SUPPLE [1] automatically generates layouts for user interfaces using an optimization approach to choose controls and their arrangement. Some initial research has been conducted on adapting SUPPLE to support the creation of consistent user interfaces for the same application across different platforms [2]. Like ITS, SUPPLE cannot create consistent interfaces if the underlying appliance model differs.

DiamondHelp [14] attempts to address the consistency problem for consumer electronics devices by combining a task-model based approach with a direct-manipulation interface. While one of DiamondHelp’s goals is to provide consistency, the current system relies on designers to create the direct-manipulation portions of the interface and for each appliance to supply its own task model. DiamondHelp does not provide a way to search for possible inconsistencies across devices or to automatically adjust the interfaces to help the user transfer knowledge between interfaces.

Software engineering researchers have also been addressing the issue of behavioral equivalence for some time, for applications such as finding equivalent code fragments and simulation models [18]. There are two main differences between this work and Uniform. First, the primitives in the software engineering models are known to be equivalent in advance, which reduces the problem to finding whether the models have the same structure. Uniform must determine whether its primitives (functions) are equivalent and also deal with the problem of functions that are similar but not equivalent. Second, the software engineering models are designed to analyze the order in which operations occur, which is not required by PUC specifications or Uniform. This work may be applicable, however, to systems that use task models as their basis for interface generation.

SPECIFICATION STUDIES

We started work on Uniform by studying how inconsistency can arise in the user interfaces created by our target platform: the PUC. The PUC interface generator uses a

rule-based process that is guaranteed to produce the same interface given the same appliance specification, so any inconsistencies arising in the interface will be due to differences in the specifications of two similar appliances. In order to understand how specifications can differ, we conducted two studies to investigate the following questions:

- How can specifications vary for different appliances that share similar functions?
- How can specifications vary for different authors?

These studies focus on specifications for VCRs, which in our experience are among the most complicated appliances to specify.

Background: PUC Specification Language

We start with some background on the PUC specification language. The functions of an appliance are represented by *state variables* and *commands*. State variables have primitive types that define the data they contain, such as integer, string, or enumeration. The interface generators infer from the type the operations that are possible on the state variable, so it is not required that a command be supplied for the common manipulations of the state variable. Commands can be used to specify manipulations that cannot be inferred directly from a variable's type. One example of a command is the seek function for a radio station. The station itself might be represented as a variable, but seek cannot be inferred from the variable because the controller cannot know in advance what the next radio station with good reception will be. After the seek command is invoked, the appliance can change the radio station variable's value as appropriate. Uniform uses state variables and commands as the basic elements of the mappings that describe the similarities between appliance functions.

Organization is specified in the PUC language via a hierarchy called the *group tree*. Variables and commands can be placed anywhere in the hierarchy, not just at the leaves. The hierarchy is used for structuring the interface and making layout decisions.

Human-readable labels are also an important part of the specification language. The PUC specification language supports different form factors with a generic structure called a *label dictionary*. Each dictionary may contain multiple labels of different types and lengths, many of which will be plain text, but can also be pronunciations, icons, etc. A label dictionary is specified everywhere that a label is needed. State variables and commands must have labels, for example, and groups should also be given labels. Label dictionaries are particularly important for Uniform, because they are the best source of information that can help identify similarities.

Study 1: Differences Among Appliances

Our first study addressed the question of how specifications can vary for different appliances with similar functions by examining specifications written for three different VCRs.

Two of the VCRs were complicated with many features, a Mitsubishi HD-HS2000U Digital VCR and a Samsung DVD-V1000 DVD-VCR combo-player, and the final VCR was the cheapest model that we could find at our local Best Buy store: a Panasonic PV-V4525S. We recruited an expert specification author, the first author of this paper, to write these specifications so that we could be confident that the specifications were of high quality. Only one author was used for this study because we wanted to control for differences that might arise between authors. The specifications took a total of about one week to complete.

In order to analyze the VCR specifications, we identified the state variables and commands, hereafter referred to as objects, which seemed to be shared across the appliances. Some objects were identical, such as the counter and status variables that tracked whether a tape was in the VCR. Many objects were similar but not completely identical. For example, the only differences between some objects were the labels, such as for the "TV/VCR" or "VCR/TV" Boolean states that are present on each VCR. Other objects contained some of the same values, but also supported other features that were not present across all of the VCRs. For example, all of the VCRs have a state variable that specifies whether the coax input is coming from the antenna or cable. The Panasonic VCR supports only these two options, and the Samsung adds an extra option called "Auto" in which it will automatically select the appropriate value. The Mitsubishi VCR does not have the auto value, but it supports two additional input types not found on the other VCRs ("cable box" and "digital cable").

Other functions shared across all of the VCRs were specified quite differently. For example, each of our VCRs supports a timed recording feature to specify TV programs that the user wishes to record in the future. The way to specify the time that the program would be recorded differed across devices. The Mitsubishi and Panasonic VCRs both have variables for the start time and stop time of the recording, while the Samsung has a matching variable for start time but a different variable that specifies the duration of the recording. In this case, the underlying data is quite different even though the function is identical.

We also analyzed the organization of the VCR specifications and found a few differences. In general, it seems that most of the same high-level groups were shared between the specifications, though the exact placement of those groups varied somewhat. For example, all of the VCRs have the Power state at the top-level with groups for Status and Setup. The Mitsubishi and Panasonic VCRs also have groups for Controls and Timed Recordings at the top-level. The Samsung DVD-VCR has these same groups, but they are located in a top-level VCR group because this appliance also must support its DVD and MP3 players.

Study 2: Differences Among Specification Authors

Our second study examined the variations in specifications written for the same VCR by several different authors. For

this study, we were particularly interested in seeing how the organization will vary between specifications. We used the Panasonic VCR from the first study and recruited 3 students in our university's electrical and computer engineering department to be subjects. We chose these subjects because we expect that specification authors in industry would have this background.

Unlike in the first study, these subjects had no knowledge of the PUC specification language when they started. Before writing the VCR specification, subjects were trained on the language through a written document with several exercises and examples from a to-do list application. We chose to use written training so that we could ensure that the learning experience was the same for each subject. The to-do list application was used for examples because it incorporated every feature of the language, but was different enough from the VCR that we did not believe it would affect the subject's specifications. Training and authoring took a substantial amount of time, about 1.5 hours and 5 hours respectively, so we allowed subjects to take the materials and VCR home with them and complete the study over the course of two weeks. Subjects were paid for their participation: \$15 for completing the training and \$50 upon returning the VCR and turning in a valid specification.

All three of the authors' specifications contained two top-level groups for setup functions and basic controls. All also had a group for timed recordings, but not all placed the group in the same location. Two of the three made timed recordings a top-level group, while the other chose to place it in the basic controls group. Two of the three had an advanced controls group, with one placing this group at the top-level and the other putting it inside the basic controls group. Within the common groups, the subjects used different strategies to further organize the functions. For example, one subject organized functions based on whether they belonged to the TV and VCR, using this method to organize within both the basic controls and setup groups.

The subjects also placed objects at different locations in their hierarchy. For example, the repeat-play command was put in the advanced playback controls group of one specification and in the setup group in another one. The functions were also defined differently in some cases, as one subject used commands for the play, stop, and pause buttons while the other two used state variables.

Discussion

In these studies we found that specifications will have differences, even if written by the same author or for the same appliance. These differences may be found in the specification of similar functions and the organization of these functions. The number and variety of differences was particularly surprising and demonstrates the challenges that Uniform faces when creating consistent interfaces. In the next section, we will combine these results with prior work on interface consistency to synthesize a set of requirements for the Uniform system.

REQUIREMENTS FOR CONSISTENCY

For Uniform, we define a consistent user interface as one that is easier to learn because it incorporates elements and organization that are familiar to the user from previous interfaces. In the context of appliance interfaces, this might mean that a new copier interface is easier to learn because it uses the same labels as a previously learned copier interface (see Figure 1) or that the clock is easier to set on a new VCR interface because the clock controls are located at the same place in the interface hierarchy (see Figure 2). In order to facilitate this knowledge transfer between interfaces, previous work suggests that tasks must have similar steps and there must be sufficient external cues in both applications [11]. To facilitate this, Uniform has the following requirements for its consistent user interfaces:

R.1 Interfaces should manipulate similar functions in the same way

R.2 Interfaces should locate similar functions in the same place

These two requirements help to ensure that user tasks will have similar steps, which can facilitate knowledge transfer. They also illustrate a fundamental separation in Uniform between *functional consistency* and *structural consistency*. Two interfaces are functionally consistent if the same set of controls is used for similar functions. Two interfaces are structurally consistent if similar functions can be found in similar organizational groups. These two types of consistency are independent and are addressed separately by Uniform.

In order for knowledge transfer to occur, we also need sufficient external cues to indicate that the applications are the same. In many cases, Uniform gets an important external cue for free, because users are often aware of the type of appliance they are using and will have some memory of using similar appliances in the past. To reinforce that cue, we have the following requirements to help increase users' perceptions of consistency between user interfaces:

R.3 Interfaces should use familiar labels for similar functions

R.4 Interfaces should maintain a similar visual appearance

We know from our studies that situations may arise where these requirements cannot be followed. For example, similar functions may have different representations that cannot use the same control. Unique functions may also affect the order in which controls appear on the screen or affect the layout if they require larger controls or have wider labels. In these situations there is a fundamental trade-off between maintaining consistency to a previous interface and appropriately rendering all of the new appliances' functions. To address this problem, we could favor consistency. In this case, we could move the unique functions to a separate panel so that they cannot affect the layout. This solution has many negative consequences for usability however: impor-

tant functions could be moved to a non-intuitive location, and the extra features of a similar function might appear to not exist. It seems better to favor usability in these situations, and therefore we have the following requirement:

R.5 Usability of unique functions is more important than consistency for similar functions

We have found that a common result of this requirement is that our consistent user interfaces do not always have a similar visual appearance. However there may be some benefit to having a different visual appearance: work by Satzinger [15] found that users were able to learn the user interface for a similar application more easily when the interface used the same labels but had a different visual appearance.

The first five requirements apply to the user interfaces that Uniform generates, and illustrate the actions that Uniform will take to ensure consistent interfaces. A pre-requisite for all of these requirements however, is:

R.6 Interface generators must provide a method to find similar functions across multiple appliances

Although this is a general requirement for any system that wants to create consistent interfaces, the implementation of this method is likely to be specific to the particular type of input that the system receives. In the case of Uniform, the input is written in the PUC specification language, which provides a functional model of each appliance. Uniform's method for finding similarities may be applicable to other systems that use functional models, but may not apply to systems that use other types of input, such as task models.

The final requirement applies to Uniform's design. Uniform makes consistency decisions based on the previous interfaces that users have experienced, so it is possible for Uniform to generate consistently poor interfaces if users start with poor interfaces. In order to address this, Uniform must handle the following requirement:

R.7 Users must be able to choose to which appliance consistency is ensured

This requirement affects Uniform at a fundamental level, because its data structures must include information about each of the possible consistency choices and it must have some means to keep track of the current choice. To support this, we developed a mapping graph structure, which is used throughout Uniform and discussed in the next section.

ARCHITECTURE

Most interface generation systems, like the PUC, use a two-step process to create user interfaces:

1. An interface specification is transformed into an *abstract user interface*. The PUC's abstract interface is a platform-independent representation with a tree structure that contains Abstract Interaction Objects (AIOs) [16] for each function. The abstract user interface does not contain any information about layout.

User Interface Generation Process & Architecture

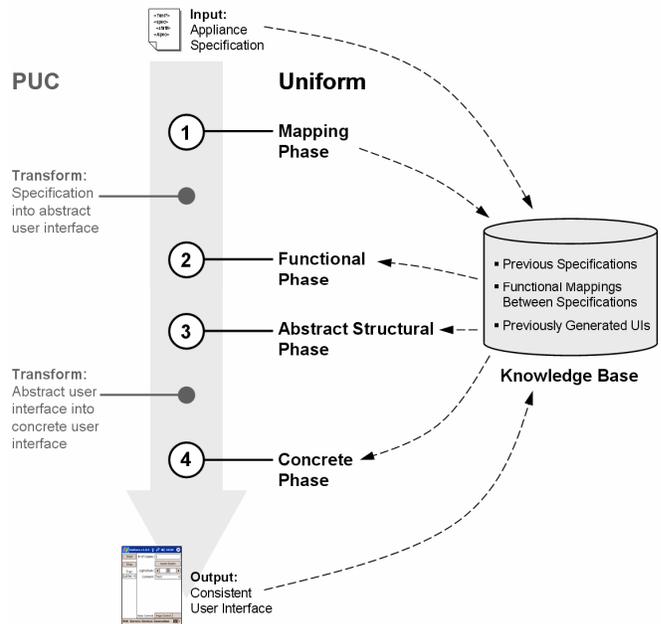


Figure 3. The user interface generation process of Uniform integrated with the PUC's existing generation process. Each phase consists of a set of rules that use Uniform's knowledge base, which stores information about previous interfaces and functional mappings between specifications.

2. The abstract user interface is transformed into a platform-specific *concrete interface*, which can be displayed to the user.

Uniform attempts to separate itself from the interface generator by operating on the inputs, outputs, and intermediate results of the generation process. Uniform necessarily must know about the data structures the interface generator uses, but it should be possible to translate its rules to use similar structures from other interface generators. Consistent interfaces are ensured in four phases (see Figure 3):

1. **Mapping Phase:** the new specification is compared to known specifications and the similarities between the specifications are extracted and saved in a knowledge base. Following this phase, the PUC transforms the specification into an abstract user interface.
2. **Functional Phase:** each functional mapping is examined and the abstract user interface may be modified to ensure *functional consistency*.
3. **Abstract Structural Phase:** the organization of the abstract user interface is modified for consistency. This phase helps to ensure *structural consistency*. Following this phase, the PUC transforms the abstract interface into a concrete user interface.
4. **Concrete Phase:** the concrete user interface is modified to ensure a similar visual appearance.

Table 1. Mapping types in the Uniform system

| Name | Description |
|----------|---|
| general | Allows a series of operations on one appliance to be matched with series of operations on another appliance, with support for repetition. The possible operations are invoking a command or changing the value of a state variable. |
| state | Maps two state variables. Particular values of the state may be mapped together, and a shortcut is available to define that the two states have entirely equivalent values. |
| node | Specifies that a node in the group tree from one specification is similar to a node in another specification. A node could represent a group, command, or state variable. |
| list | Specifies that two lists contain the same data. |
| group | Groups multiple mappings together. Groups cannot be nested. |
| template | Maps two Smart Templates [9], a special feature of the PUC specification language. Smart Templates allow the PUC to render domain-specific design conventions, such as the standard number pad layout on a telephone or the media control icons on a VCR. |

Each of these phases consists of a set of rules. The rules of the mapping phase are the only rules that add new information to the knowledge base. The remaining rules, which we call consistency rules, modify the abstract or concrete user interface based on information from the knowledge base. Most of the work to ensure consistency takes place in the second and third phases, which operate on the abstract user interface. The concrete phase is then used to clean up visual consistency problems created by the interface generator's transformation process, such as changing the orientation of panels or adding additional organization.

The knowledge base is an important piece of the Uniform architecture. It stores previously generated specifications, mappings between specifications, and information about the interface designs built from those specifications. The most important elements of the knowledge base are the mappings between functions on different appliances. A mapping defines a relationship between similar functions in two specifications. These mappings may be automatically generated by rules in mapping phase, manually specified by the user, or downloaded from the Internet.

Uniform supports six different types of mappings, each of which was identified from the specifications written for our authoring studies. We believe that most of these types are generally applicable for creating mappings with any type of functional specification language, though one type is based on a special feature of the PUC system called Smart Templates [9]. Each mapping type is described in Table 1.

Mappings between similar functions in multiple specifications are grouped together in a *mapping graph*. The central purpose of a mapping graph is to help determine which

Media Controls Mapping Graph Example

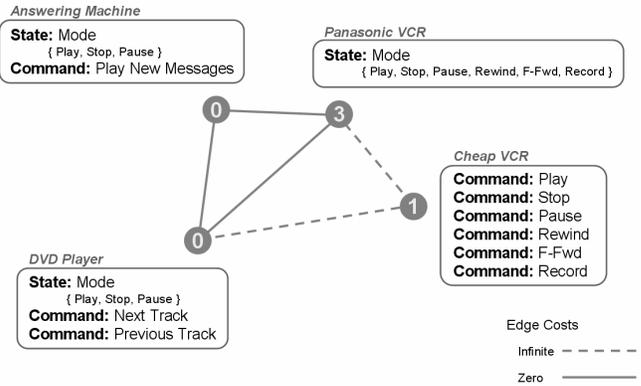


Figure 4. An example mapping graph for the media control functions, e.g. play, stop and pause, on four appliances. The node counts indicate that the Panasonic VCR has been the basis for consistency three times (for itself, the answering machine, and the DVD player) and the Cheap VCR has been the basis for consistency once (just for itself). The answering machine and DVD player were generated to be consistent with the Panasonic VCR, and thus both have counts of zero.

appliance should be used as the basis for consistency for a function. Every mapping belongs to a mapping graph, and there is a mapping graph for each set of similar functions in the knowledge base. For example, the power, media controls, and VCR/TV functions all have separate mapping graphs containing mappings specific to those functions. An example mapping graph for the media controls function is shown in Figure 4.

To find the specification to ensure consistency to, Uniform starts at the node that represents the appliance being generated and traverses the mapping graph to find the node that the user has seen most often. Each node maintains a count of the times it has been used as the basis for consistency. As discussed earlier, it may be impossible to ensure consistency between similar functions if their specifications are too different. To represent this case, the edges of a mapping graph have cost values, currently limited to zero and infinite. For example, in Figure 4 the Panasonic VCR represents play, stop and pause as a state variable while the Cheap VCR uses only commands. It is not possible to convert between these representations, so the mapping between them will have infinite cost. Costs allow mapping graph traversals to ensure that consistency can be maintained between the endpoints of the traversal result. You may wonder why we bother to include infinite cost edges in the mapping graph. When no zero cost edges are available, infinite cost edges may be traversed to ensure consistency by using the labels of a similar function.

MAPPING PHASE

The main function of the Mapping Phase is to automatically extract mappings between the new specification and previous specifications that Uniform already knows about. The challenge of automatic mapping generation is the lack of

Table 2. Uniform's functional consistency rules.

| Name | Description |
|-------------------------------|--|
| state | Ensures that similar variables use the same label and, if possible, the same control |
| invoke-to- invoke | Ensures that similar commands use the same label |
| change-to- repeated-invoke | Converts between the situation where changing a variable on appliance is the same as repeatedly invoking a command on another |
| change-invoke- to-change | Converts between the situation where one appliance has a state variable and the other has a state variable with a command that must be invoked before a variable change will take affect |
| time-end-to- duration | Converts between the situations where one appliance uses a start time and an end time and another appliance uses a start time and a duration |
| template | Ensures that smart templates use the same label and control style |
| node | Ensures that nodes mapped with a node mapping use the same label |
| generic-group | Takes a group of mappings and processes the other mapping rules on each of them. |

substantial semantic information about each function within the specifications.

We have built two separate matching systems. The first is based on our intuition of the PUC data structures, and makes use of names, label dictionaries, and variable types. The second is based on the similarity flooding technique [7], which also incorporates organization. The first system performs the best, finding about 60% of the mappings in our VCR test cases with about 20% of the total mappings found being false positives. We are currently investigating other means to improve our matching algorithms, such as incorporating a thesaurus of appliance terminology. A promising technique might leverage existing mappings found among other specifications, but we have not yet explored this. Mappings not found by our current algorithms must be specified by hand or downloaded from the Internet.

FUNCTIONAL PHASE

This phase ensures functional consistency by inspecting each function in the new specification, determining whether there is a previous function with which the new function should be consistent, and then making changes to the abstract interface to ensure consistency. Although this phase precedes the abstract structural phase, these two phases could be executed in the opposite order. We chose this order for implementation reasons, because it is easier to find functions in the abstract user interface before the structural phase moves them around.

The previous function to be consistent with is found by traversing the mapping graph. If a previous function is found, Uniform uses functional consistency rules to transform the new specification into a form that is consistent with the previous specification. In order to determine the particular rule to apply, Uniform iterates through the rules until it finds a rule that matches the mapping. Uniform will use the first rule that is found, so the rules must be carefully ordered to ensure that those with more specific matching conditions will be tested before those with more generic matching conditions. Uniform currently implements eight functional consistency rules, as shown in Table 2. Each rule modifies a portion of the new specification to match the previous specification. For example, the `change-invoke-to-change` rule will convert between a state variable and a state variable with a command that must be invoked before a variable change will take effect. As part of this conversion, a command in the new specification must be hidden or a new command must be added. If the command is hidden, the converted state variable will automatically invoke the command when its value is set by the user. If a command is added, then the appliance will not be informed of a variable change until the user invokes the new command in the interface.

ABSTRACT STRUCTURAL PHASE

The goal of the abstract structural phase is to ensure that functions are located in the same place in new interfaces. This phase is divided into two sub-phases: moving and then re-ordering. Moving rules only need to ensure that functions are placed in similar groups, and then the re-ordering rules can ensure that the functions have a consistent ordering within their groups. Both of these sub-phases rely on mappings, such as node, template, and list mappings, that identify similar groups across specifications. Uniform uses this information to rearrange groups so that they have the same structure as a previous specification. Uniform does not currently add new organization to specifications.

Moving

The moving sub-phase traverses the abstract user interface's group tree and searches for mappings.

An important feature of the moving sub-phase is its data structure, called the "containment stack." The purpose of the containment stack is to keep track of similar parent groups as the sub-phase traverses through the tree. Two stacks are created, one for the new appliance and another for the previous appliance. Only mappings between these two specifications are included in the containment stack, so any entry in the stack is known to refer to an existing location in both specifications. For example, the containment stack for the clock group in the Mitsubishi DVCR and Samsung DVD-VCR is shown in Figure 5a. The clock is located in a different group in these specifications, which is reflected in the containment stack.

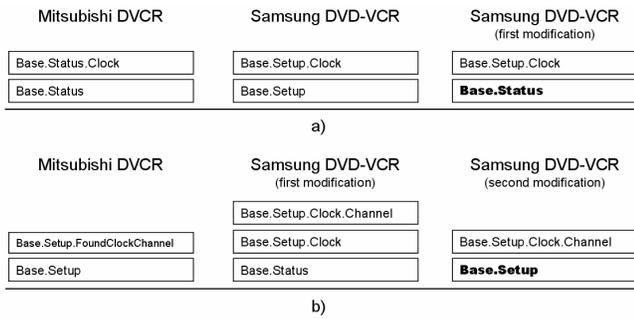


Figure 5. Containment stacks for the previous specification (Mitsubishi DVCR), the new specification (Samsung DVD-VCR), and the results of two consecutive movements. a) Shows the movement of the clock group, and b) shows how the rule chains with the movement of the clock channel state.

Our current moving rule checks the containment stacks to see if the top-most group mappings are different. If the mappings are different, then the mapping’s objects are moved to the group that corresponds to the previous specification’s top-most group mapping. For example, suppose that we are generating the Samsung DVD-VCR interface to be consistent with the Mitsubishi DVCR. The top-most group mappings are Base.Setup for the DVD-VCR and Base.Status for the DVCR, which are different. Because of this, the moving rule will take the clock group and move it to the DVD-VCR’s Base.Status group, which is similar to the group with the same name on the DVCR (see Figure 5a). Note that this algorithm also chains appropriately. For example, the clock group on the DVD-VCR contains a variable that specifies the channel from which clock information can be extracted. The DVCR has a similar state variable, but is located in the setup group instead of the clock group. Before Uniform moved the clock group, the clock channel variable had the same containment in both specifications, but afterward this is no longer true. When the algorithm is applied to this channel variable, the difference in containment is found because the containment stacks are calculated from the variable’s current location. The algorithm will then move the channel variable back to its consistent location in the setup group (see Figure 5b). Note that this movement is visible in the generated interfaces: Figure 2b shows the clock group under the Setup tab with the clock set variable, and Figure 2c shows the clock group in the Status tab without the clock set variable.

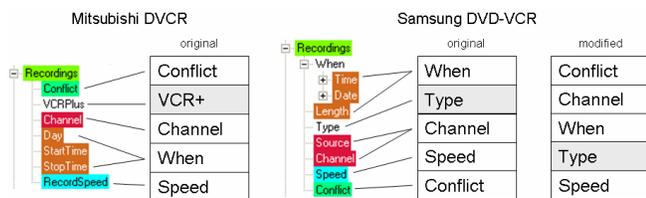


Figure 6. Block lists created for the timed recordings groups of the Mitsubishi DVCR and Samsung DVD-VCR. “VCR+” and “Type” are unmapped blocks in the block lists.

Reordering

The reordering sub-phase moves functions within groups to ensure a consistent ordering. For example, Figure 6 shows that the parameters for a timed recording have a different ordering between the Mitsubishi DVCR and the Samsung DVD-VCR. This sub-phase, like the moving sub-phase, traverses the abstract user interface until it encounters a mapping. Unlike the moving sub-phase, reordering rules are not applied to leaf nodes in the group tree.

Before the reordering rules are executed for a group, the sub-phase determines the previous specification with which the group will be made consistent. The sub-phase then creates a “block list” data structure for the group in the new specification and its equivalent group in the previous specification. The block list is important because it allows rules to analyze and manipulate functions as if the functions are in a list, when the underlying representation of structure in the abstract user interface is a tree. The tree structure can become problematic when a function’s objects span multiple levels of tree hierarchy, as in the case of the “When” mapping on the Samsung DVD-VCR (shown in Figure 6). Each set of adjacent objects with the same mapping becomes a block, which is stored in the list in the order they would appear in a generated interface (see Figure 6). Consecutive unmapped objects are also stored as blocks in the list. The block lists are processed by the reordering rules, resulting in a new block list that specifies the final ordering for the group.

Our current reordering rule starts by searching the block lists from the new and previous specifications to find blocks with the same mapping. These blocks are re-ordered to match the previous specification. Unmapped blocks are moved with the block that precedes them. For example, notice in Figure 6 that the Type block followed the When block to its new location. This heuristic seems to perform reasonably for unmapped blocks, but we are also experimenting with additional rules that extract semantics from the unmapped blocks and establish stronger links with mapped blocks.

CONCRETE PHASE

The goal of the concrete phase is to ensure that the visual appearance of the final interface is as similar as possible. Following the advice of Satzinger [15], our concrete consistency rules emphasize the placement of functions in similar structure over recreating a previous layout. Our current rules modify the concrete user interface to add organizational elements or modify the orientation and sizing of some visual elements.

We have implemented two concrete consistency rules. The first rule adds overlapping panel organization to a user interface if it was used in the previous interface and more than one control can be placed on each of the overlapping panels. The exact type of overlapping panel widget is chosen based on the previous interface. The second rule modifies the side panels that the PUC interface generator some-

times creates around overlapping panels. This rule checks the orientation of the side panel, which may be either horizontal or vertical, and ensures that the orientation is the same as in the previous interface. Both of these rules are used to generate the consistent interfaces in Figure 1.

DISCUSSION AND FUTURE WORK

Uniform represents the first attempt to build a system that automatically generates consistent interfaces from potentially inconsistent interface specifications. Its strengths and weaknesses suggest future areas of work for us and the CHI community.

One of the biggest challenges throughout Uniform is appropriately dealing with the unique functions and organization found in similar appliances. In this paper we have suggested several heuristics to approach this problem, but our solutions are often limited because of a lack of useful semantic information about the unique functions. In part, this is due to Uniform's use of "relative semantics" to understand similarity; Uniform only knows that two functions are similar, not *why* they are similar, *what* they do, or *how* they relate to other functions in the appliance. With better information, Uniform could make more informed decisions about where to place functions and when to create new organization. Of course, better information would come at the cost of additional modeling for each appliance because it seems unlikely that detailed information could be automatically extracted from available appliance information.

Uniform's consistent interface designs are based on seven basic requirements, as discussed earlier. These requirements are based on current work in consistency, and would likely benefit from some elaboration. More work to understand how consistency can be best operationalized in an automatic system would help greatly.

There are several directions to pursue with Uniform. Most importantly, the system needs to be evaluated with users to understand how much of an improvement it provides. We would also like to replicate Satzinger's study [15] to understand the effect of visual appearance within our interfaces. More specific studies of our rules would also be helpful to determine which rules are contributing most to consistency and whether any are detrimental. Uniform also needs to be tested with many more appliances to validate its generality.

Another area of future work is to create additional rules throughout all of Uniform's phases. Rules are particularly needed to in the structural consistency phase to deal better with unmapped objects and in the mapping phase for automatically inferring new mappings.

CONCLUSION

Uniform's ability to generate consistent interfaces has implications for the future of user interface research and design. For research, the absence of work on which to base consistency rules suggests there is still research required in the area of interface consistency, which has seen little activ-

ity in the past decade. For design, it demonstrates that automatic interface generation systems can extend the capabilities of human designers by adding features that would be difficult or impossible to implement by hand.

ACKNOWLEDGMENTS

We would like to thank the reviewers and Andrew Ko for providing insightful feedback that helped us improve this paper substantially, and Duen Horng Chau for assisting with the analysis of the specification studies. This work was conducted as a part of the Pebbles project, and was funded in part by grants from NSF, Microsoft, and General Motors, and equipment grants from Mitsubishi Electric Research Laboratories. The National Science Foundation funded this work under Grant No. IIS-0534349. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

REFERENCES

1. Gajos, K., Weld, D. SUPPLE: Automatically Generating User Interfaces, in *Proc. of IUI*. (2004) 93-100
2. Gajos, K., Wu, A., and Weld, D.S. Cross-Device Consistency in Automatically Generated User Interfaces, in *Proceedings of the 2nd MUI3I*. (2005) 7-8
3. Gomes, L., Appliances Have Become Like PCs: Too Complex for Their Own Good. *The Wall Street Journal Online*. (2003)
4. Grudin, J., The Case Against User Interface Consistency. *CACM*, (1989) **32**(10): 1164-1173
5. Kieras, D.E., Wood, S.D., Aboitel, K., and Hornof, A. GLEAN: A Computer-Based Tool for Rapid GOMS Model Usability Evaluation of User Interface Designs, in *Proc. of UIST*. (1995) 91-100
6. Mahajan, R. and Shneiderman, B., Visual and Textual Consistency Checking Tools for Graphical User Interfaces. *IEEE Transactions on Software Engineering*, (1997) **23**(11): 722-735
7. Melnik, S., Garcia-Molina, H., and Rahm, E. Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching, in *Proc. of 18th ICDE*. (2002) 117-128
8. Nichols, J., Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R., and Pignol, M. Generating Remote Control Interfaces for Complex Appliances, in *Proc. of UIST* (2002) 161-170
9. Nichols, J., Myers, B.A., and Litwack, K. Improving Automatic Interface Generation with Smart Templates, in *Proc. of IUI* (2004) 286-288
10. Nielsen, J., *Usability Engineering*. 1993, Boston: Academic Press.
11. Polson, P.G., The consequences of consistent and inconsistent user interfaces, in *Cognitive science and its applications for human-computer interaction* (1988) Lawrence-Erlbaum. Hillsdale, NJ
12. Reisner, P. What is inconsistency? in *INTERACT*. 1990. 175-181.
13. Rheinfrank, J. and Evenson, S., Design Languages, in *Bringing Design to Software*, T. Winograd, Editor (1996) Addison-Wesley 63-80
14. Rich, C., Sidner, C., Lesh, N., Garland, A., Booth, S., and Chimani, M. DiamondHelp: A Graphical User Interface Framework for Human-Computer Collaboration, in *Proc. of IEEE International Conference on Distributed Computing Systems Workshops*. (2005) 514-519
15. Satzinger, J.W. and Olfman, L., User Interface Consistency Across End-User Applications: The Effects on Mental Models. *Journal of Information Management Systems* (1998) **14**(4): 167-193
16. Vanderdonckt, J., Advice-Giving Systems for Selecting Interaction Objects. *User Interfaces to Data Intensive Systems* (1999) 152-157
17. Wiecha, C., Bennett, W., Boies, S., Gould, J., and Greene, S., ITS: A Tool for Rapidly Developing Interactive Applications. *ACM Transactions on Information Systems* (1990) **8**(3): 204-236
18. Yucesan, E. and Schruben, L., Structural and Behavioral Equivalence of Simulation Models. *ACM Transactions on Modeling and Computer Simulation* (1992) **2**(1): 82-103