# Report for the Paper Reading of Model-based Interface Generation on Google Android

Yuan Jin *
School of Computer Science
3480 University Street
Montreal, Quebec
yuan.jin@mail.mcgill.com

## ABSTRACT
In this paper, we present the main ideas of two papers by Jeffery Nichols et al. and a problem formation of my course project. The first paper about *Personal Universal Controller* explains how to engage in a two-way communication with other electrical appliances, and then goes to depict an approach for improving the interfaces to complex appliances by extracting information from a specification that includes functions of appliances. The other paper, *UNIFORM* steps further into exploring how to automatically generate consistent remote control user interfaces on hand-held devices. Its main purpose is to screen inconsiderable dissimilarity in function descriptive files like a specification but highlight interface consistency in similar devices. The essence of this method is to use information from a knowledge base to replace similar but apparently different functions as well as labels, or maintain the same location of a similar control as the user saw before. This paper goes on to talk about what needs to implement if we want to have a model-based, instead of specification-based engineering to generate user interfaces on a constrained device, say a Google Android platform.

## Keywords
Personal Universal Controller (PUC), Uniform, consistency, hand-held computers, mobile phones, Pebbles, remote control, appliances, Android

## 1. INTRODUCTION
The motivation for us to explore a model-based interface generator for the Google Android derives from two understanding based on my previous study. The first is that Android, as a newly released advance mobile platform offers a large amount of application-level APIs and interfaces to developers, and lots of freshmen claimed they could easily develop a fancy hello-world program within the first hour of acquaintance. However, the problem of efficiency is people by large needs more time to get familiar with Android framework (this process takes as long as a whole month to get acquainted.)In comparison, Domain-Specific Modeling (DSM) allows faster development, since it's based on models of the product rather than on models of the code. Industrial experiences of DSM have shown important improvements in productivity, lower development costs as well as evidently better quality [1]. For instance, Nokia [2] affirms that with DSM it now develops mobile phones up to 10 times faster and Lucent [3] says that they improved their productivity by 3-10 times depending on the product. Looking at other company's' success stories, I'm thinking about if we could copy their concept into Android.

The other understanding originates from the idea of a Personal Universal Controller introduced and implemented by Carnegie Mellon University Pebbles Project and Pittsburgh-based MAYA company. Their intent [4] is to use a handheld device to remote control all the enabled electrical appliances in a room through peer-to-peer network, such as Bluetooth or infrared. Since their approach is dependent on a specification, basically an XML file, to generate user interfaces and achieve consistency among similar devices by the following algorithm so-called UNIFORM [5], we're thinking about if that method is able to be applied to our model-based design, so that after we drag, click, program in a model-driven way, the design could be transformed correspondingly into a device-specific and constrained user interface. In the rest part of this section, I'm going to go through introductions into the above-mentioned two papers.

The first paper *Generating Remote Control Interfaces for Complex Appliances* deals with offering a variety of devices a uniform remote control. The context for this design is that people are increasingly carrying computerized devices, such as mobile phones, pagers, and PDAs. Many of these devices contain processors powerful enough to support speech applications and hardware for wireless networking. Short-distance radio networks, such as 802.11b and Bluetooth are expected to enable many devices to communicate with other devices that are within close range [4]. So people at CMU and MAYA were beginning to consider how hand-held devices can communicate with home and office appliances through the underlying network and, how hand-held devices and

*Yuan Jin is a Master of Science student at McGill SoCS. His interests include embedded systems, Android mobile platform, virtual machines, MP3 compression, modeling as well as wireless networks.

speech can improve the interfaces of these appliances.

A personal universal controller (PUC) provides an intermediary interfaces with which the user interacts as a remote control for any appliance. PUCs can run on a myriad of platforms including hand-held devices with graphical interfaces or hidden PCs with speech recognition software. The PUC differentiates from many of today's universal remote controls, such as the Philips Pronto or the inVoca speech remote, simply because it is self-programming. This means that the PUC can exchange information with the appliance in two directions, retrieving specification that describes functions of the appliance and then automatically generate a high quality interface. Another difference from ordinary remote controller is PUC's ability to provide an interface with complete functionality of an appliance.

The description of the appliance's functions must have enough information to allow a PUC to generate a high quality user interface, but not contain specifics about how the interface should look or feel. Instead, an interface generator is responsible for rendering their appearances. The specification language, a common language used to depict the appliance's functions, contains a hierarchical group tree of all appliances functions. This tree is determined by the understanding of a designer of the specification of an appliance.

Another feature of the specification language is its dependency information, which describes the availability of each function relative to the appliance's state. Organization improves because dependency information is objective, rather than subjective like the group tree. Another significant component in the PUC framework is the ability to control actual appliances from the interfaces created by the interface generator. This is done through the appliance adaptors: software and hardware that translate from the proprietary communication protocols found on many appliances to the PUC protocol.

The other paper that I read is *UNIFORM: Automatically Generating Consistent Remote Control User Interfaces.* The background for conducting research in generating consistent interfaces is two-fold: the high quality interface produced by a PUC system definitely relies on the quality of a specification that describes the functions and how it groups related components together. Conducted by Jeffery Nichols in Carnegie Mellon University, a consistency survey showed that for the same author, similar devices which might distinguishes from each other in architecture or data storage, poses different specification. This results in a quite remarkable dissimilarity in user interfaces of these similar devices. Another group of survey showed that for the same appliance, different authors' understanding of its architecture and dependency information disagrees. This means if an end user has downloaded different specification for an appliance, he or she might not be able to quickly know how to control, which has decreased the benefit [5] for using a universal remote controller.

When we talk about consistency in user interface, we are really mean 7 rules discovered by Jeffery Nichols et al to assure what we logically expect, i.e. interfaces should manipulate similar functions in the same way; interfaces should

**Figure 1: User interfaces generated with consistency for a complex and simple copier.**

locate similar functions in the same place; interfaces should use familiar labels for similar functions; interfaces should maintain a similar visual appearance as well as, usability of unique function is more important than consistency for similar functions; interface generators must provide a method to find similar functions across multiple appliances and users must be able to choose to which appliance consistency is ensure.

The Uniform, meaning *Using Novel Interfaces For Operating Remotes that Match*, 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 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 attempt to generate interfaces with a similar visual appearance.

Uniform is implemented on top of PUC, 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 illustrating the working mechanism for communication between appliances and handhelds as well as components that are demanded. Then we go on to introduce specification: what is need to describe a device; how to express a hierarchical organization of functions and so forth. Afterwards, a combination of PUC and Uniform is concretely carried out to explain how a consistent user interface is generated. At last, a brief look-ahead at my course project, i.e. what is needed; what should I do, etc. is discussed.

## 2. THE ARCHITECTURE

The architecture is composed by four parts: a specification language, an adapter, a communication protocol and an in-

terface generator. The PUC architecture is designed to allow for automatic interface generation by a wide range of devices in a wide range of modalities. This is enabled by a two-way communication protocol and a specification language that allows each appliance to describe its functions to an interface generator. The specification language separates the appliance from the type of interface that is being used to control it, such as a graphical interface on a hand-held or a speech interface on a mobile phone.

A key part of the architecture is the network that PUCs and appliances use to communicate. Both papers assume that each appliance has its own facility for accepting connections from a PUC. The peer-to-peer aspect of this choice allows the PUC architecture to be more scalable. A PUC could discover appliances by intermittently sending out broadcast requests, as in the service discovery portion of the Bluetooth protocol.

Although we assume network layer communication between PUC and electrical appliances, we have no idea of how to communicate with actual appliances. Several standards exist for appliance control, including Microsoft's UPnP, Sun's JINI, and HAVi. Many consumer electronics manufacturers have their own proprietary methods for communicating with and between their appliances. Often these methods are not available for use by the general public. Most devices in the low-end market have no means of being controlled externally, except for one-way IR-based remote control. These appliances must be altered at the hardware level to achieve two-way communication with a PUC.

To connect the PUC to any real appliance, we must need an appliance adaptor, that is, a translation layer to the proprietary protocols. An adaptor is built for each proprietary appliance protocol that the PUC communicates with. This architecture allows a PUC to control virtually any appliance; provided the appliance has a built-in control protocol or someone has the hardware expertise to add one.

## 3. SPECIFICATION LANGUAGE

The *raison d'etre* of a specification language is to describe the functions of an appliance so as to automatically generate user interface in PUC system. This description must include enough information to generate a good user interface, but it should not contain any information about look or feel. The PUC paper actually determined what components should be embraced in a specification by hand-made user interfaces and then studied that in order to provide enough data to interface generator, a typical specification language should include: state variables and commands; type information; label information; group tree and dependency information. In the rest of this section, I will elaborate the definition, function and relationship of these components respectively.

*State variables* are representation of most of the manipulable elements. Discovered from the hand-made implementation of a user interface, researchers find that interface designers must know what can be manipulated on an appliance before they can build an interface for it.Each state variable has a *type* that tells the generator how it can be manipulated. *Commands* are, in a sense, complimentary to the state variables to describe unmanipulable functions. In general, state variable and commands tell the functions of an appliance.

Each state variable must be specified with a type so the interface generator knows how it can be manipulated. There're 7 generic types defined: Boolean, integer, fixed point, floating point, enumerated, string and custom.

The interface generator must also have information about how to label the interface components that represent state variables and commands. Providing this information is difficult because different form factors and interface modalities require different kinds of label information. An interface for a mobile web-enabled phone will probably require smaller labels that an interface for a PocketPC with a larger screen. PUC chooses to provide this information with a generic structure that we call the label dictionary. Each dictionary contains a set of labels, most of which are plain text. The assumption underlying the *label dictionary* is that every label contained within will have approximately the same meaning. Thus the interface generator can use any label within a label dictionary interchangeably.

Interfaces are always more intuitive when similar elements are grouped close together and different elements are kept far apart. PUC explicitly specifies grouping information using a group tree. State variables and commands may be present at any level in the tree. Each branching node is a group, and each group may contain any number of state variables, commands, and other groups.

The two-way communication feature of the PUC allows it to know when a particular state variable or command is unavailable. This can make interfaces easier to use, because the components representing those elements can be disabled. The specification contains formulas that specify when a state or command will be disabled depending on the values of other state variables. These formulas can be processed by the PUC to determine whether a component should be enabled when the appliance state changes.

## 4. INTERFACE GENERATION

Personal Universal Controller proposed two phases as shown above: the phase from specification to abstract user interface and the phase from abstract user interface to concrete user interface. Since PUC does not assure that consistent user interface could be generated for similar devices, Uniform adds another four phase: Mapping phase, which finds out mapping graphs by comparing current specification with specification stored in a knowledge base; Functional phase, in which interface generator transforms similar functions into the same button and which also promises that interface should manipulate similar functions in the same way. The third additional phase is called an Abstract Structural phase, where two sub-phases moving and re-ordering promise similar functions would appear on the same location on an interface. The last Concrete phase ensures that the visual appearance of the final interface is as similar as possible. In the rest of this section, I'll illustrate the merits of two papers by combining the interface generation into one flow, since PUC is a subset of Uniform.

When the interface generator receives specification from a device, it first goes through the *Mapping Phase*. The main

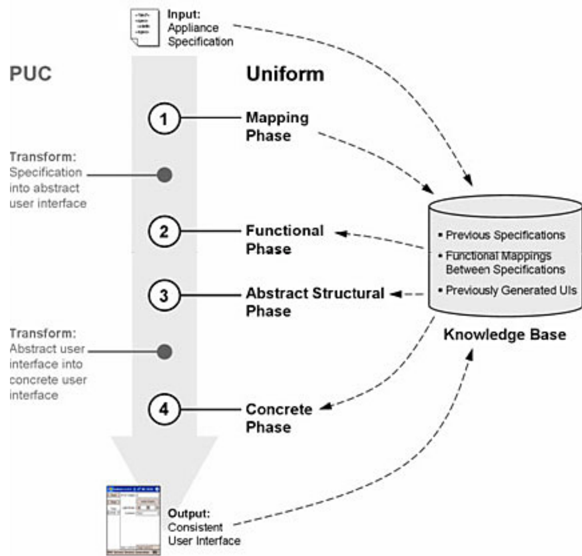


**Figure 2: he interface generation process of Uniform.**

function of the mapping phase is to automatically extract mappings between the new specification and previous specifications that Uniform already knows about. Mappings between similar functions in multiple specifications are grouped in a mapping graph. The central purpose of a mapping graph is to help determine which appliance should be used as the basis for consistency for a function. Each set of similar functions has separate mapping graphs containing mappings specific to those functions.

In the following process, 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) for each function. The abstract user interface does not contain any information about layout. Also during this phase, panel structure, which regulates the relocation of specific components, is determined by dependency information. The graphical interface generator uses dependencies to decide how to divide the screen into panels, and assign branches of the group tree to each panel. It is more often to compare the dependencies of different state variables and commands to determine if they are never available at the same time. This is to find so-called mutual exclusive information of each state. If such mutual exclusive information is found, typically three rules are used to generate panel structure.

Each functional mapping is examined in the corresponding *Functional Phase* and the abstract user interface may be modified to ensure functional consistency. This consistency is ensured 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. 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.

A step before transform abstract interface into concrete layout is the *Abstract Structural Phase*. This phase helps to ensure structural consistency by modifying organization of the abstract user interface. There are two sub-phases are involved, i.e. the *Moving Phase* and the *Re-Ordering Phase*. In the moving phase, the interface generator traverses through the group tree to find out mappings between two specifications. If found, nodes of mapping which share the same function in two specifications are placed with their parent nodes in a containment stack. This helps to compare in detail the differences between two mappings. Nuances in the new specification is then targeted and transformed in accordance with the previous one for consistency. By doing this, generator ensures similar function has the same location as previous ones. The second possible transformation derives from re-ordering. It also compares two specifications and place nodes of mapping with their parent nodes in some block lists. The re-ordering phase's responsibility is to change the order of new specification according to the previous ones, so that components in the same group/panel has exactly the same location as interfaces the user has seen before.

Afterwards, the interface generator goes to make the *abstract interface concrete*. The first step is to choose what kind of component to assign to each state variable and command. The generator uses a decision tree to make these choices. Once the components have been selected, the interface generator recursively traverses the group tree and inserts each component into an interface tree, which represents the panel structure of the generated interface and is used for translating abstract layout relationships to a concrete interface. After the group tree has been traversed, the interface is made concrete by recursively traversing the interface tree twice. The first traversal allocates space for each component and determines the size and location of every panel. The second traversal places and sizes the components within their rows, and then assigns labels by picking the largest label that will fit in the space allocated.

The last phase modifies the concrete interfaces to ensure a similar visual interface. Two concrete consistency rules are implemented. 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 sometimes 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. When this phase is complete, an interface is presented to the user.

## 5. DISCUSSION

There are some advantages and disadvantages for the above-mentioned approach. Benefits include automatic interface generation; can be applied to any device with a specification and always a consistent user interface. However, all of this setup is dependent heavily on XML and does not

provide any information for user's ability to change the interface, nor the transformation from specification to code. My course project is able to provide a model-driven domain-specific design of Google Android interface, which allows for transformation from an abstract user interface to a platform-dependent interface and also transformation into Android XML and code files.

I start from the assumption that we have read in all the functions of an appliance, which is simulated by a modeling tool instantiated by a self-designed meta-modeling language. In this state, the interface would have no idea of any specific layout; it simply and logically assemblies all the functions in the right way. The next step, which is activated by a transformation, considers the physical characteristics of an interface on a device/platform, i.e. the screen size, which functions would never appear on the same panel - the so-called mutual exclusives derived from mapping comparisons. Having determined which buttons should appear at this or that panel (the panel structure), the environment should give additional organizational navigation "keys" - it might be a button or a tab or anything else, to correctly lead users to the right panel. When this is done, code transformation is activated. Developers could install the generated Android interface XML file on the emulator QEMU to see the result.

Having a global perspective of my course project, I decided, with the help of Prof. Hans Vangheluwe, to use Montreal version ATOM3 for implementation. Future work includes design of a meta-modeling language; design an abstract interface for the device; transformation into constrained interfaces and a final generation of code files.

## 5.1 References


[1] White paper of Domain-Specific Modeling with MetaEdit+: 10 Times Faster than UML. MetaCase. 2007

[2] Nokia Mobile Phones Case Studies. MetaCase. 2003

[3] Software Product-line Engineering. Weiss, D., Lai, C. T. R. Addison Wesley Longman. 1999

[4] Generating Remote Control Interfaces for Complex Appliances. Jeffery Nichols et al. UIST '02

[5]UNIFORM: Automatically Generating Consistent remote Control User Interfaces. Jeffery Nichols et al. CHI 2006