A Model-Engineering Approach to Implementing Personal Universal Controller

Yuan Jin[1]

School of Computer Science, McGill University, Canada

Abstract. The *Personal Universal Controller* (PUC) is a way to allow users conveniently control remote electrical appliances through a peer-to-peer network on their handsets. There are two technical problems the PUC faced: how to communicate between the handset and the appliance, and how to create separate user interfaces for different appliances respectively. As for the first trouble, *Jeffrey Nichols et al.* proposed an interpretation protocol that could be understood by each party. And for the second, an XML specification that includes the states of an appliance and the description of its functions is suggested for UI construction. Based on the received specification, the very application on a handset could interpret its content, select groups that need to be placed together on the same screen, assign actual components to each command or state mentioned in the specification as well as set for a flexible layout for different screens. I am more interested in the user interface construction work and wonder if it's applicable on recently published *Android platform*, and thus, build a naive implementation in a model-engineering way, on *McGill MSDL-based AToM3*. The implementation assumes we've already interpreted the XML specification and read in the appliance's structure: its grouping information, commands, mainly for functionalities and current state information. The first read-in process can be simulated by creating a model by my *AndroidDevice* formalism, with a result of being transformed into a complete appliance's structure in my *AndroidGUI* formalism, without any layout information. Then the AndroidGUI formalism is transformed under certain rules, e.g. *HVGAPGG*, to satisfy the need of some constrained handset screens. The final outcomes are an *AndroidManifest.xml* file which registers all the so-called activities needed for such an appliance, *Java code files* for each user interface layout and *user interface XML files* for each layout. Users are also supposed to draw what they want to place in an Android program in a model-driven way and generate similar files which can be put into use directly.

1 Introduction

Before I move to explaining what I've done and the significance of my work, we need to know what motivates my work and how it shapes the framework I'm creating. And that is Personal Universal Controller (PUC) proposed by Jeffrey Nichols et al. from the Pebbles group of Carnegie Mellon University and MAYA Design Inc.

The background for this proposal is that, as we all can perceive, more and more office and home appliances, such as televisions, VCRs, stereo equipment, ovens, thermostats, light switches, telephones and factory equipment, are created with very advanced and complicated functions. But the fact is that as they become more computerized, their user interfaces are harder for people using them [BJ92]. Another thing is that people are increasingly carrying computerized devices, such as mobile phones, pocket PCs, pagers and PDAs. Many of these devices have or will support wireless networking communications. Short-distance radio networks like Bluetooth and 802.11b enables devices to communicate with other devices that are within close range.

Therefore, the Personal Universal Controller is provided for the handset users to remotely control computerized devices within a short distance like a real remote controller. But there're some realistic problems: how do you communicate with the devices? We, of course, need to know the complete functions of a certain appliance and can read or send control signals to the appliance. This question deals with a low level protocol communication scheme that is shared by both the device and the remote controller. Additionally, since we're talking about a universal controller which means a simple handset device could control different electrical appliances, it naturally raises another question: how do you maintain a generic and comfortable user interface across different appliances?

To answer the first problem, Jeffrey Nichols et al. suggested an intermediate communication protocol that can be understood by both the appliances and the remote controller. Several standards exist for appliance control, including Microsoft's UPnP, Sun's JINI and HAVi, but so far none of these has been widely accepted, many consumer electronics manufacturers have their own proprietary methods for communicating with and between their appliances. Thus, we need to have this device-agnostic agreement/protocol to complete our control. Also, an appliance adaptor specified to each type of appliance was proposed. The adaptor is actually software and hardware that translates from the proprietary communication protocols found on many appliances to the PUC communication protocol. An appliance adaptor is in essence, a translation layer to its built-in protocol. The adaptor would be built for each proprietary appliance protocol that the PUC communicates with.

I'm more concerned over the second problem: how do we build a generic and comfortable user interface across different appliances? This problem, in fact, involves a very complicated process. There're actually several papers written to provide a reasonable solution. The solution extracted from these papers generally has two steps. The first step is to build a user interface which could automatically scale itself to different screen sizes, from a certain specification that details the functionalities of an appliance. For example, if we have a specification that describes the functions and states of a digital camera, we can build its user interface on different handset screens with different sizes: for a HVGA-Portrait screen, only four controls are allowed horizontally, ten vertically; or for a HVGA-Landscape, ten controls are allowed horizontally, four vertically. Some of the complete functions can be put in the same screen, while some could be put

in another screen. The other step, as vividly depicted in *UNIFORM: Automatically Generating Consistent Remote Control User Interfaces*, copes with how to create a user interface that is familiar with the user (which means the user might have seen this UI structure previously). The approach usually adopted is to use a pool or library to store previously constructed layout information and, based on which, to adjust the current layout and then generate a familiar one.

Jeffrey Nichols' approach is to parse the XML specification and read in the appliance's functions and state variables. Most of the manipulable elements could be represented as state variables. Each state variable has a given type that tells the interface generator how it can be manipulated. For example, the radio station state variable has a numeric type, and the interface generator can infer the tuning function because it knows how to manipulate a numeric type. Other state variables include the current track of the CD player and the status of the tape player [JN02]. A function whose result cannot be described easily in the specification is represented as a command. For instance, the seek button on a radio, cannot be represented as state variables. Pressing the seek button causes the radio station state variable to change to some value that is not known in advance.

Each state variable should also possess type information so that the interface generator can understand how it may be manipulated. For example, read-only states are almost always represented by a label component. Boolean states are represented by checkboxes.

A group tree is an n-ary tree that has a state variable or command at every leaf node. 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. A group tree basically categorized similar commands and states in the same group so that the interface generator could allocate them in adjacent locations.

The most innovative component of Jeffrey Nichols' approach is the dependency information. It allows the PUC to know when a particular state variable or command is unavailable. 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.

Based on this fundamental elements, Jeffrey Nichols' approach allows us to either construct a simple user interface that only involves layout information, or a complex-strategy one that could create a consistent user interface. Since I only have a constrained duration for doing this course project, I choose to focus on the implementation of a simple user interface from a model-driven view. By mentioning a model-driven view I mean, instead of conventional direct transformation from XML specification to a concrete user interface, I intervene with models and rules. Models are helpful because we could clearly view, edit, add or delete involved components, and since my work takes two independent formalisms: AndroidDevice and AndroidGUI, I could use the only second formalism to create user interfaces in a model-engineering way and generate necessary Android files.

The work I should've done and have done is as follows: firstly I should create a meta-modeling formalism that could model or say simulate information we retrieve from the XML specification, i.e. grouping information, state variables, command, label information and dependency information. In my words, they're called group, control and command. Label information and dependency information are somehow integrated into the former three components. Using these components, we should be able to recover the complete functionalities of a certain device, or more. Then we need transformation from this AndroidDevice formalism into an AndroidGUI transformation. The reason is that the first formalism does not include any layout information and most importantly, it does not contain any components that are related to actual controls, like a TextView, a CheckBox or a ProgressBar. The AndroidGUI formalism model is then transformed by some rules to a layout-constrained model, which is, structurally, equivalent to the model before transformation. This model should concern about the size of the actual screen (where I naively set the limit for each screen size) and the back-track button or forward button to go across different layouts. And at last, it should output an AndroidManifest.xml file which registers all the existing activities (i.e. layouts), Java source files that each layout would have respectively, and layout XML files. Because on Android platform, the design of user interface and coding is separated, I would need both the layout XML file and Java source file to activate a project.

The work I've done is almost all of the above-mentioned, except intelligent dependency information. By saying this I mean the model I created cannot know which two buttons or two text views should be place together. Instead, they're placed according to their sequence of connecting to the layout, if and only if they're directly connected to the layout. This is quite shameful even though I gave some of the objects a dependency option; but I cannot come up with an algorithm within a short period to separate components intelligently.

The next section, traditionally and formally should examine related work, but since I've done such summary in my reading report, the next section would go directly to explain the first formalism, i.e. AndroidDevice, and its transformation into AndroidGUI. In the third section I'll discuss how I employ certain rules to transform an unconstrained layout into a specified layout (e.g. HVGAP). In the fourth section I'll talk about how to generate necessary files like AndroidManifest and Java source files. I conclude with thoughts on future directions for this project.

2 From AndroidDevice To AndroidGUI

2.1 Formalism Definitions

AndroidDevice Formalism Back to the stage of thinking about what is needed in the first formalism, i.e. what I expect to model and what is the result of this modeling, I had been quite unclear about what I should do. Looking from a high level, I need to sort out the input, the output and its overall role in the project. I supposed that we have already received the XML specification

from a remote electrical appliance, stored in my handset device. I can read out its functions: in the form of command and state variables and which are in the form of group tree. That should be the input of my project, or say I should use the formalism to create such a group tree model. Notice since the function of specification should not contain any layout information, such as the orientation of the volume control group layout, the first model should not contain layout information too. Knowing this, it's still not straightforward to predict how we describe the structure and functions of an appliance completely; I've thought of and created formalism that has device, group, label, state, command, etc. It is what we should include in our formalism according to Jeffrey Nichols' design of the specification. However, for the sake of simplicity, I thought only three components are enough to describe, they're group that stands for the old device and group, control that represents old label and state, and command as one in Nichols' design. Dependency information, however, is included in some controls. Label information is embedded in every control.

Group and command only possess a name attribute; state has an additional type attribute. The type attribute actually realizes Jeffrey Nichols' type information. In essence, the raison d'etre of type information is to make the later decision tree to select an actual component, e.g. TextView, EditText, ProgressBar, CheckBox or Spinner. The Nichol's way of selection is based on whether it's a Boolean, an integer, a string, an enumerated, a fixed point, a floating point or a custom type. But for me and for simplicity, I use control's class name instead. That is to say, If later we find the control has a textview type, then a TextView control would be allocated to the tree and replace the control.

The group may be a device or a functional group, e.g. volume control or tuner. Therefore, in the ClassDiagram design phase, I enable the group may associate with itself. A command is, in fact, a deterministic function, or say a button if incarnated with the AndroidGUI formalism. It can only be associated with a group. But it can be associated with a control, provided control's type is filled with textview. As we can put it further, a control can be associated with a control, which means, for instance, a spinner/control can have a textview label.

The following graph can give you a better understanding of what I'm talking about.

AndroidGUI Formalism The AndroidGUI formalism, in a sense, is quite independent from the previous formalism, namely, AndroidDevice formalism. Because, first, what this formalism cares about is to build a model with some components like TextView and LinearLayout, which are extracted from the real Android System; it does not necessarily need to correlate with a higher level portrayal of what the functions of an appliance should have. Second, this formalism supports layout information, or say panel structure in Nichols words; it only needs to transform, if necessary, according to the requirement of a constrained screen, from an unconstrained model to an appropriate layout model. Because of its independency, this formalism could be also used to create an Android user interface alone.

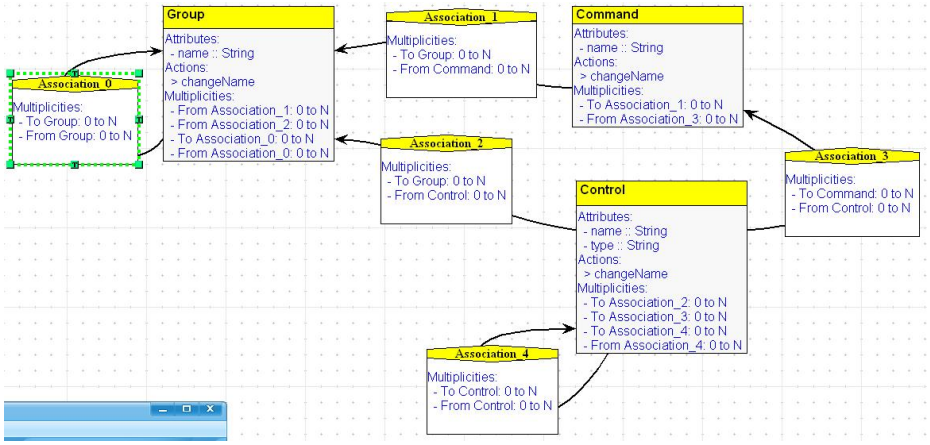

Fig. 1. Class Diagram AndroidDevice

This formalism, if compared with the previous one, is surely more complicated. Its complexity derives from two aspects: first, it simply has more components; the relations between each component would be more complicated, for example, a TextView can only be limited to a LinearLayout once; a LinearLayout can be linked to itself as many as possible, etc. Second, unlike the first formalism which only deals with replacing transformation policy, this formalism has to know what a specific requirement of a screen is, and add, delete, or change components on the layout. It might add another layout if more components are available; it might direct the navigation button to move forward to another layout, or move backward to the previous one, etc.

The following graph could give you a better view of the AndroidGUI formalism structure.

2.2 Device2GUI Transformation

because the sequence and logic of rules really matter. One might not get what he/she expected before the transformation. Rules are also limited by condition. If only the code in the condition happens and successfully returns 1, rules take effect. Action is activated after executing the rule. It might be used to stop/hold/resume the rules. When we're creating rules in AToM3, a Left-Hand Side (LHS) provides a canvas for users to draw a situation before transformation, so that when the system finds an exact circumstance as in the LHS, it rewrites the graph to the one on the Right-Hand Side (RHS). It is a good thing that one can always locate components in both sides simply with some Python code. And then I'll explain to you how do we chat with the system and create situations as we want.

The requirement is simple in this process: transforming models written in AndroidDevice formalism into models in AndroidGUI formalism. And of course,

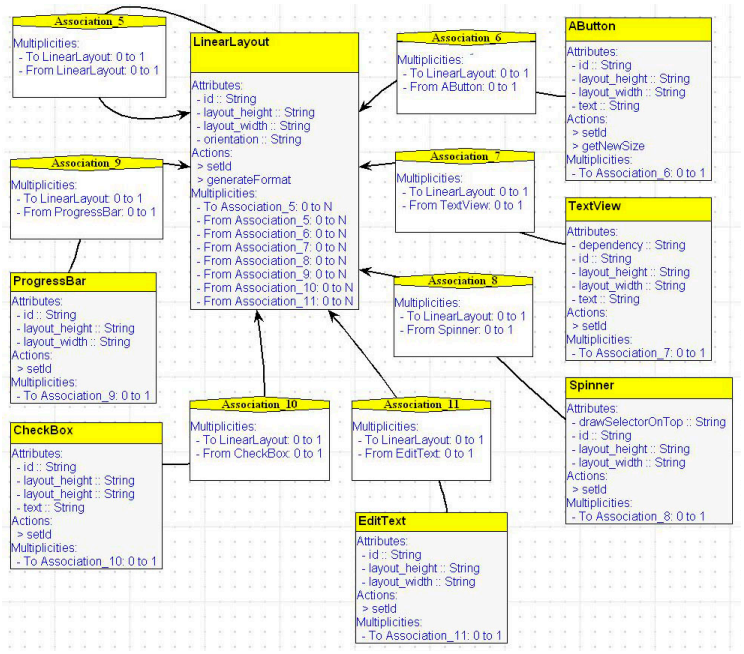

Fig. 2. Class Diagram AndroidGUI

we need to know what equals to what in two formalisms so that we can perform transformation. Generally speaking, a Group in AndroidDevice equals to a LinearLayout; a Command equals to an AButton; a Control with type textview equals to a TextView, a Control with type edittext equals to an EditText; a Control with type spinner equals to a Spinner; a Control with type progressbar equals to a ProgressBar; a Control with type checkbox equals to a CheckBox. Notice here again that the type information I put does not, in reality, means it. In the XML specification, a checkbox might be a Boolean and a textview might be a read-only. The reason I put textview etc. here is only for simplicity.

1. *groupAddLayout*: this rule basically find two group that are linked and insert a LinearLayout in between. And then set the sub-group's isVisited attribute to true, so that it will never take this rule again. This is to insert LinearLayout between all parent and child group. The reason I did this is to in the future replace all the children groups with a LinearLayout.
2. *topGroupAddLayout*: since the first rule won't take any effect on the root group/device, this rule is to add a LinearLayout to the root group/device, with similar purpose.
3. *layoutAddLayout*: this rule is actually a preparation for the fifth rule. It links LinearLayout at a parent group to one at a child group. This rule is applied to take the place of the old groups.

4. *commandAdd2Layout/controlAdd2Layout*: I want to remove groups from the map, but first I need to connect commands and controls that are linked to such groups with its LinearLayout.

5. *deleteGroup*: this is rule is very simple, remove all the groups that has a connection with its parent LinearLayout. In reality, this rule will delete all the groups on the graph.

6. *control2TextViewForCommand/control2TextViewForControl*: now that we have removed all the groups, we want to replace all the commands and controls by appropriate AndroidGUI components. There's also a special control with type textview and linked to a command or a control. This is the label information for that command or control. The sequence of whether replace other components or replace these special label components is important. Because if we take the former replacement first, all the links to their label information would be lost too. So I need to replace this label information first. When the system finds a control with type textview connected to a command or a control, it would add a new child LinearLayout after the command or control's parent LinearLayout, and further move the label and its parent command or control to the new LinearLayout (and set its orientation to horizontal).

7. *command2Button*: the name says it all. When the system finds a command currently linked to a LinearLayout, it simply replaces the command with a Button.

8. *control2TextView/control2EditText/control2Spinner/control2ProgressBar/ control2CheckBox*: applying these rules will replace controls with such types to its corresponding AndroidGUI component.

The following graph displays a model before and after AndroidDevice transformation. On the left-hand side, the leftmost top control has a type progressbar and the rest control all have a type textview.

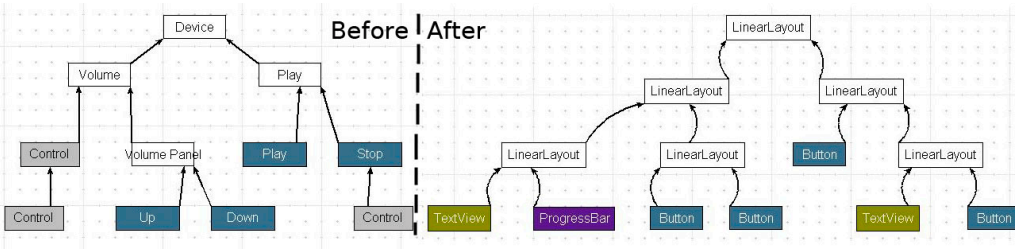

Fig. 3. Comparison between and after the Deive2GUI transformation

3 From Unconstrained To Constrained Layout

3.1 Introduction

After the first transformation, we could receive a model completely written in the AndroidGUI formalism. Basically, this is enough for Android developers to use, what we need to do is output the layout information and generates an XML file. However, sometimes this is not enough. Consider the Personal Universal Controller example, the components we get might be more that we could display on the screen. The following graph shows such a situation.

Fig. 4. Comparison between 7 and 10 buttons

Therefore, before we output files, we need to do some transformation, in order to adjust the layout to a specified screen size, such as HVGA-Portrait or HVGA-Landscape. Because this project is to demonstrate we have the ability to adjust layout according to certain screen size. I only set the rule for HVGA-Portrait. And very naively devised, this rule, HVGAPGG, defines manually by me, the number of components allowed to be showed horizontally and vertically. This means even as the size of a component might change as the result of its text or some attributes, the rule still, stupidly deems a fixed number of components, even if at extreme circumstances, a very few components might fill the screen. The following is also an example to show what I'm talking about.

As illustrated in Figure 5, this graph indicates even with two buttons and full text, like 60 letters, one button would be excluded from the visible canvas. In order to avoid such extreme cases happening, we need to change the structure of the current layout or start a new layout if necessary.

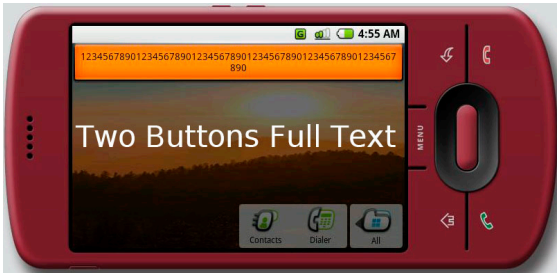

Fig. 5. Two buttons with full text

There're some strategies applying to how to do such transformation. The one that I take requires two steps. .First is to move surplus components on the horizontal line to the next line, so that the whole orientation of the old layout should be changed to vertical. Second, if the system finds more components than its ability to display vertically, start a new layout and move these surplus components or layouts to that new layout. So, logically, my following explanation would focus on the concrete mechanism I employ to achieve the above effect in two steps: horizontal and vertical.

3.2 HVGAPGG Transformation

Horizontal Transformation Here's the rules for horizontal transformation.

1. *setTopNodeAndNewBranch*: this rule first will find a LinearLayout of which the orientation is set to horizontal. And then change its orientation to vertical while add a child LinearLayout to it. The idea is when the system finds a horizontal LinearLayout has more than four components aligned in a line, move them separately to children LinearLayouts whose orientation are horizontal but change the parent LinearLayout's orientation to vertical. There's a problem of this rule: how can we find all the valid horizontally aligned components under this LinearLayout. Because, there might be situations like a child LinearLayout with vertical orientation which should not be calculated, but this child might have its own child LinearLayout whose orientation is horizontal. In this circumstance, this grandchild LinearLayout's children component should all be summed. In order to fulfill such a seeking function, I import a method called iterateSubTree(node). Here's the code.

This job is done before the rule takes effect, so it is executed in the condition. And in order to prevent repeatedly adding new layouts to the child LinearLayout, I add an isVisited attribute to the child LinearLayout in action.

```
import globalVariables

def iterateSubTree(node):
  print ">>>>> new layout: "+str(node)
  if (node.orientation.getValue() == 'horizontal'):
    if node.in_connections_ != []:
      for assoc in node.in_connections_:
        for subNode in assoc.in_connections_:
          if (hasattr(subNode, "orientation")):
            print '@'+str(subNode)+" find a horizontal layout!"
            iterateSubTree(subNode)
          else:
            globalVariables.counter += 1
            print globalVariables.counter
  else:
    if node.in_connections_ != []:
      for assoc in node.in_connections_:
        for subNode in assoc.in_connections_:
          if (hasattr(subNode, "orientation")):
            print '@'+str(subNode)+" find a layout!"
            iterateSubTree(subNode)
```

Fig. 6. iterateSubTree(node)

So, new layout would only be added to the parent LinearLayout not this child.

2. *moveButton2NewBranch/moveTextView2NewBranch/moveEditText2NewBranch/moveSpinner2NewBranch/moveProgressBar2NewBranch/moveCheckBox2NewBranch*: these rules are created with one simple purpose, to move all the components currently linked to the parent vertical LinearLayout to its first horizontal child LinearLayout. A first look might give users an absurd feel. Why should we move ALL of the components to the first child LinearLayout? Why don't we move first four components to the child, and see if we need more child and move? I'm doing so because such reasonable strategy would be more complex to set a rule. The system has to over and over again check if the parent LinearLayout possesses more components even if the focus has been moved to the child LinearLayout. So, my strategy is we move all the components from the parent LinearLayout to the first child and see if has more than allowed components, if it does, apply the third rule (actually a mutated version for rule 1 setTopNodeAndNewBranch). After the transformation and only a link is found on the parent LinearLayout, give the parent node an isReleased attribute. This is, actually to avoid rule 3 setNewBranchWhenFull from happening in advance.

3. itshape setNewBranchWhenFull: this rule is a modified version of rule 1 setTopNoeAndNewBranch. It generally says when the system finds a horizontal LinearLayout with more than four components, it adds a new horizontal LinearLayout to the parent vertical LinearLayout. And this rule is only available when the components of this LinearLayout are more than four.

4. *moveExcessButton2NewBranch/moveExcessTextView2NewBranch/ moveExcessEditText2NewBranch/moveExcessSpinner2NewBranch/ moveExcessProgressBar2NewBranch/moveExcessCheckBox2NewBranch*: these rules are modified version of rule 2 series. It basically says if more than four components are found in one horizontal LinearLayout and another horizontal LinearLayout with less than five components, move the surplus components in the first LinearLayout to the other. These four rules will completely achieve our requirement.

The following graph might give you a straightforward view of the horizontal transformation.

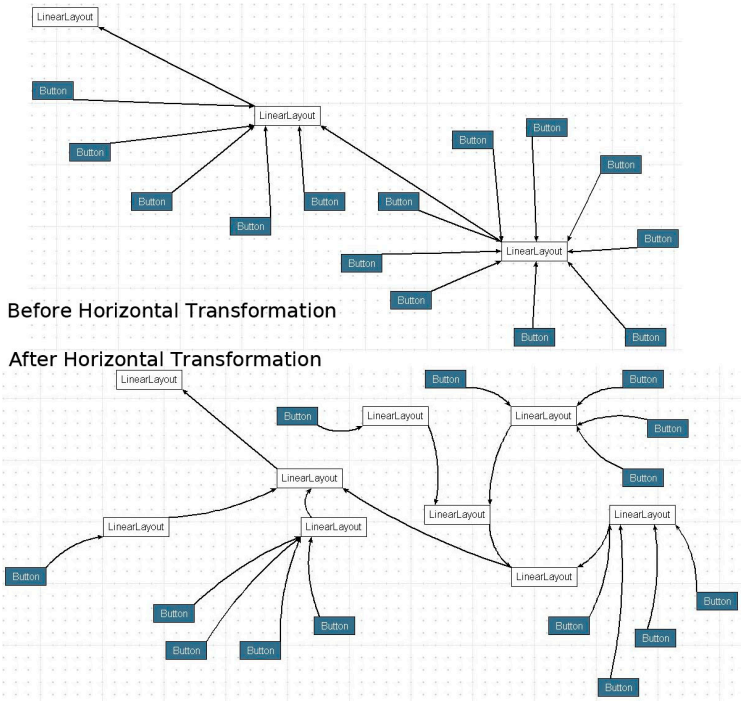

Fig. 7. before and after the horizontal transformation

Vertical Transformation When we perform the vertical transformation, basically the strategy is if there're more than 10 components vertically, start a new layout for the surplus. If there were already an independent layout that has less than ten components, accrue the surplus to the independent one. Of course, we need to add navigation to the parent and independent layouts. Here are rules that I apply.

5. *addButton2AnotherLayout/addTextView2AnotherLayout/ addEditText2AnotherLayout/addSpinner2AnotherLayout/ addProgressBar2AnotherLayout/addCheckBox2AnotherLayout*: this rule is actually the second step in vertical transformation strategy. It says if we find an independent vertical layout already there with less than ten components and simultaneously the current vertical layout has more than ten components; please move the surplus to that one. The following is the condition I take and its python code.

```
def traverseSubTree(node):
  #print "inside the sub-tree of "+str(node)
 if (node.orientation.getValue() == 'vertical'):
   if node.in_connections_ != []:
    for assoc in node.in_connections_:
     for subNode in assoc.in_connections_:
       #print subNode
      if (hasattr(subNode, "orientation")):
       traverseSubTree(subNode)
      else:
       globalVariables.counter += 1
       #print globalVariables.counter
 else:
   globalVariables.counter += 1
   #print globalVariables.counter
   if node.in_connections_ != []:
    for assoc in node.in_connections_:
     for subNode in assoc.in_connections_:
      if (hasattr(subNode, "orientation")):
       traverseSubTree(subNode)
```

condition

```
from iterateSuperTree import *
node1 = self.getMatched(graphID, self.LHS.nodeWithLabel(1))
node2 = self.getMatched(graphID, self.LHS.nodeWithLabel(2))
globalVariables.counter = 0
traverseSubTree(node1)
print str(globalVariables.counter)
if globalVariables.counter > 10:
  globalVariables.anotherCounter = 0
  traverseSubTreeAnotherCounter(node2)
  if globalVariables.anotherCounter < 10:
   return 1
```

Fig. 8. rule 5 conditions

6. *segregateButtonFromLayout/segregateTextViewFromLayout/ segregateEditTextFromLayout/segregateSpinnerFromLayout/ segregateProgressBarFromLayout/segregateCheckBoxFromLayout*: these rules are comparatively simple because it only needs the system to find whether a vertical LinearLayout is connected to more than ten components; if it is, then start a new layout and move the surplus to the new layout. The difference between rule 6 and 5 is rule 5 already has an independent layout. The reason for this difference is initially we would only have one layout but possesses a lot of components; this situation satisfies the condition of rule 6, so an independent layout would be created. Then rule 5 finds an independent layout and is executed.

7. *removeSubTreeFromParent*: Having solved the generation of new layout when the parent layout is connected by controls and commands (in AndroidDevice's perspective). The system would still produce problems if surplus groups cannot be separated from the parent vertical LinearLayout.

8. *addLinkButton*: This rule is the last one; it traverse the graph and finds root LinearLayouts and then assign a navigation button to them respectively. How the button should work is left to the generator of Java source file.

As usual, a graph is presented to demonstrate vertical transformation.

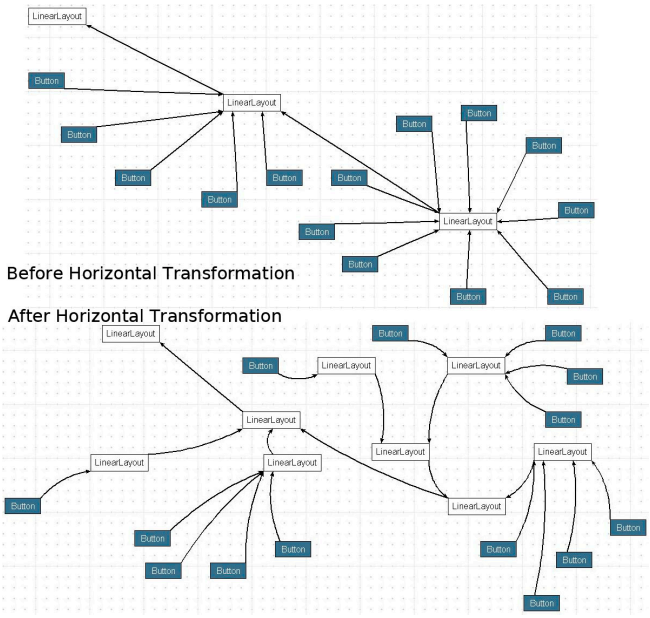

Fig. 9. before and after the horizontal transformation

4 From Model To Files

There're four things I need to do during this course. Since I have already get a Android user interface model, a user interface XML file for each layout, Java source files that direct what components should act, as well as an AndroidManifest.xml file which registers all the activities (i.e. layout in some sense in this project), are the outcomes we expect to see. But, except for those, we also need to assign the android:id attribute for every component. Since if I assign the id for every component when it's created (action: create post), there would be some problems in creating new components during the horizontal or vertical transformation.

In this section, I would like to clarify how I created the user interface XML file for every layout, Java source files and the AndroidManifest file.

4.1 User Interface XML Files

The most important thing to create the XML file is to find its tree structure. Every XML file should have a grammatically complete structure, Android observes. It starts with a declaration of what coding scheme it is using, and then goes on to describe which components this model has. So, it would be quite simple if we can start from the very first root LinearLayout of every layout, fills in its branch and leaves with LinearLayout, Buttons, TextViews, etc. it is connected

to. To summarize, this is a tree-traversal problem. The following code segment is extracted from the "Save2XML" button's action, which shows how I do it.

```
#generate ui xml files
for key in self.ASGroot.listNodes.keys():
  if self.ASGroot.listNodes[key] != []:
    for node in self.ASGroot.listNodes[key]:
      if node.out_connections_ == []:
        output = open('page'+str(i)+'.xml', "w")
        output.write("<?xml version="+'"1.0"'+" encoding="+'"utf-8"'+'?>'+'\n')
        generate(node, output, True)
        output.close()
        print "save to file "+'page'+str(i)+'.xml'+' successfully!'
        i += 1
```

```
import globalVariables

def generate(node, outfile, isFirstNode):
 if (node.getClass() == 'AButton'):
   outfile.write('<'+'Button'+'\n')
 else:
   outfile.write('<'+node.getClass()+'\n')
 if (isFirstNode):
   outfile.write('xmlns:android='+'"http://schemas.android.com/apk/res/android"'+'\n')
 for attr in node.realOrder:
   outfile.write('android:'+attr+'="'+str(node.getAttrValue(attr).getValue())+'"\n')
 outfile.write('>'+'\n')
 if (node.in_connections_):
   for association in node.in_connections_:
     for childnode in association.in_connections_:
       generate(childnode, outfile, False)
 if (node.getClass() == 'AButton'):
   outfile.write('</'+'Button'+'>\n')
 else:
   outfile.write('</'+node.getClass()+'>\n')
```

Fig. 10. generate XML files

Because in the META.py file, a self only indicates AToM3 itself; I need to find an abstract root node of the current graph which lists all the instantiated classes on the graph. This is what self.ASGroot.listNodes returns. But it returns a list of nodes and their classes which one cannot easily play with. So I select the node by their attributes (keys()). For example, if node's out connections attribute is [], it means it's surely a root LinearLayout. Beginning from every root LinearLayout found on the graph, I traverse all the children it has and insert them into the XML file. Notice that the children discovery strategy is recursive: if it finds a component other than a LinearLayout, then add it to the file, or else go deep into the LinearLayout and find what it has. Another thing to notice is since I always used AButton to replace a pre-defined Button class in AToM3, we need to replace every found AButton by Button manually.

4.2 AndroidManifest.xml

The responsibility of an AndroidManifest.xml file is, as I have repeated many times in this paper, registers current available activities (or says layout information in this project). The essence of creating such a file is to find the entire available root LinearLayouts and assign them consistent names. As we've already known if only the out connection of a component is zero, it should be a

```
<?xml version="1.0" encoding="utf-8" ?>
- <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android" android:id="@+id/LinearLayout_1389" android:layout_height="wrap_content"
    android:layout_width="wrap_content" android:orientation="vertical">
  - <LinearLayout android:id="@+id/LinearLayout_1391" android:layout_height="wrap_content" android:layout_width="wrap_content" android:orientation="vertical">
    - <LinearLayout android:id="@+id/LinearLayout_1945" android:layout_height="wrap_content" android:layout_width="wrap_content" android:orientation="horizontal">
        <Button android:id="@+id/Button_1418" android:layout_height="wrap_content" android:layout_width="wrap_content" android:text="text" />
        <Button android:id="@+id/Button_1419" android:layout_height="wrap_content" android:layout_width="wrap_content" android:text="text" />
        <Button android:id="@+id/Button_1420" android:layout_height="wrap_content" android:layout_width="wrap_content" android:text="text" />
        <Button android:id="@+id/Button_1412" android:layout_height="wrap_content" android:layout_width="wrap_content" android:text="text" />
      </LinearLayout>
    - <LinearLayout android:id="@+id/LinearLayout_1956" android:layout_height="wrap_content" android:layout_width="wrap_content" android:orientation="horizontal">
        <Button android:id="@+id/Button_1413" android:layout_height="wrap_content" android:layout_width="wrap_content" android:text="text" />
        <Button android:id="@+id/Button_1414" android:layout_height="wrap_content" android:layout_width="wrap_content" android:text="text" />
        <Button android:id="@+id/Button_1415" android:layout_height="wrap_content" android:layout_width="wrap_content" android:text="text" />
        <Button android:id="@+id/Button_1416" android:layout_height="wrap_content" android:layout_width="wrap_content" android:text="text" />
      </LinearLayout>
    - <LinearLayout android:id="@+id/LinearLayout_1963" android:layout_height="wrap_content" android:layout_width="wrap_content" android:orientation="horizontal">
        <Button android:id="@+id/Button_1417" android:layout_height="wrap_content" android:layout_width="wrap_content" android:text="text" />
      </LinearLayout>
    </LinearLayout>
    <Button android:id="@+id/Button_1966" android:layout_height="wrap_content" android:layout_width="wrap_content" android:text="linkButton" />
  </LinearLayout>
```

Fig. 11. user interface XML files

root LinearLayout, and then our job is simply to follow a certain format and output it.

```
#generate the androidmanifest file
i = 0
firstPage = True
for key in self.ASGroot.listNodes.keys():
  if self.ASGroot.listNodes[key] != []:
    output = open('AndroidManifest.xml', "w")
    output.write('<?xml version="'+'"1.0"'+' encoding="'+'"utf-8"'+'?>'+'\n')
    output.write('<manifest xmlns:android=')
    output.write('"http://schemas.android.com/apk/res/android"'+'\n')
    output.write('package='+'"yuan.android.translucent"'+'>\n')
    output.write('<application android:icon="'+'"@drawable/icon"'+'>\n')
    for node in self.ASGroot.listNodes[key]:
      if node.out_connections_ == []:
        if firstPage:
          generateAndroidManifest(output, firstPage, i)
          firstPage = False
        else:
          print 'print branch layouts now'
          generateAndroidManifest(output, firstPage, i)
        i += 1
    output.write('</application>'+'\n'+'</manifest>')
    output.close()
    print "successfully write to AndroidManifest.xml"

def generateAndroidManifest(outfile, isFirstPage, number):
  if isFirstPage:
    print str(isFirstPage)
    outfile.write('<activity'+' android:name="'+'.AndroidPage'+str(number)+'"'+' android:label="'+'@string/app_name"'+'>\n')
    outfile.write('<intent-filter>'+'\n')
    outfile.write("<action android:name="+'"android.intent.action.MAIN"/>'+'\n')
    outfile.write("<category android:name="+'"android.intent.category.LAUNCHER"/>'+'\n')
    outfile.write('</intent-filter>'+'\n')
    outfile.write('</activity>'+'\n')
  else:
    print str(isFirstPage)
    outfile.write('<activity'+' android:name="'+'.AndroidPage'+str(number)+'"'+'>\n')
    outfile.write('</activity>'+'\n')
```

Fig. 12. generate AndroidManifest.xml

And its outcome is presented as follows.

4.3 Java/Android Source Files

When generating Java source files for the AndroidGUI models, one big problem is to analyze whether this layout/activity has a certain instances of a class. If it does, we need to declare its class in advance. For example, if this is a null layout, it is unreasonable to statically generate LinearLayout or Button or other classes' declaration code. Another thing is if this graph holds more than one layout, we

```
<?xml version="1.0" encoding="utf-8" ?>
- <manifest xmlns:android="http://schemas.android.com/apk/res/android" package="yuan.android.translucent">
  - <application android:icon="@drawable/icon">
    - <activity android:name=".AndroidPage0" android:label="@string/app_name">
      - <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
      </activity>
      <activity android:name=".AndroidPage1" />
    </application>
  </manifest>
```

Fig. 13. AndroidManifest.xml

need to set an OnClick listener and an event handler for the navigation button to perform actual navigations. Besides, how to navigate is also a problem.

The first two scenarios are easy to settle because we could, as we want, find any instance on the graph or determine whether this presumed instance exists or not. Another technical problem is to avoid repeatedly adding declarations to the source file. In terms of the navigation algorithm, mine is quite intuitive. If this layout finds it's the last layout on the graph, then navigate back to the previous layout; if not then navigate to the next layout.

Following is the source file generator's code. It's apparently not clear enough (for limited space), but if you're interested in how I do it, please write to me.

Fig. 14. compact source file generator

And its result as the Java source file.

5 Future Work

There're already three proposals for the future work. The first one is to develop a market-driven modeling tool for the user interface design of Android platform. My current prototype is simple designed for demonstration only. It lacks support

```
package yuan.android.translucent;

import android.app.Activity;
import android.os.Bundle;
import android.widget.Button;

import android.content.Intent;
import android.view.View;
import android.view.View.OnClickListener;

public class AndroidPage0 extends Activity {
   private Button linkButton;

   public void onCreate(Bundle icicle) {
      super.onCreate(icicle);
      this.setTheme(R.style.Theme_Translucent);
      setContentView(R.layout.page0);
      linkButton = (Button) findViewById(R.id.linkButton);
      linkButton.setOnClickListener(listener);
   }

   private OnClickListener listener = new OnClickListener() {
      @Override
      public void onClick(View view) {
         Intent intent = new Intent(AndroidPage0.this, AndroidPage1.class);
         intent.setLaunchFlags(Intent.NEW_TASK_LAUNCH);
         startActivity(intent);
      }
   };
}
```

```
package yuan.android.translucent;

import android.app.Activity;
import android.os.Bundle;
import android.widget.Button;

import android.content.Intent;
import android.view.View;
import android.view.View.OnClickListener;

public class AndroidPage1 extends Activity {
   private Button linkButton;

   public void onCreate(Bundle icicle) {
      super.onCreate(icicle);
      this.setTheme(R.style.Theme_Translucent);
      setContentView(R.layout.page1);
      linkButton = (Button) findViewById(R.id.linkButton);
      linkButton.setOnClickListener(listener);
   }

   private OnClickListener listener = new OnClickListener() {
      @Override
      public void onClick(View view) {
         Intent intent = new Intent(AndroidPage1.this, AndroidPage0.class);
         intent.setLaunchFlags(Intent.NEW_TASK_LAUNCH);
         startActivity(intent);
      }
   };
}
```

Fig. 15. Java source file for two layouts: AndroidPage0 and AndroidPage1

for complete components, complete attribute editing function and a better and intuitive graph design environment. I don't think this would take a long time to finish, though it apparently needs a complete rewriting and re-consideration of its meta-modeling formalism. If it could be, as Prof. Hans Vangheluwe told me, applied on the web version AToM3 with better graph support, I guess this would be a popular tool for many Android developers.

Another direction is to enable intelligent component distribution. Currently, Jeffrey Nichols' work has already given solution how to do this. The dependency information is the important key to do it. But, since I have a quite limited time on this course project, it is thus impossible for me alone to crack down all the problems. Of course, mine present available version is somewhat reasonable in distribution, because it selects the components according to their sequence of connecting to the parent LinearLayout. But it would be nice if I or somebody else could continue working this intelligent mechanism out.

The third proposal is to provide a consistent user interface [3] to users. This, too, has been documented by Jeffrey Nichols, but not in a model-engineering way. A consistent user interface offers uses similar layout and component, labels as ones they've seen before. This is done through a pool that stores previously constructed layouts. But, how to do in a model-engineering way is presently unknown to me. Perhaps, it would even need to upgrade some parts of AToM3. If anybody who's interested in doing this, please don't hesitate to contact me.

6 Acknowledgement

I want to take this opportunity to thank Prof. Hans Vangheluwe for his wonderful lectures and assignments. Because of these previous works, I can, in a very short time, be confident and finish this great project. His warm-hearted help within and beyond the office hour is really a sample to many others who are devoted to

teaching. Without his patient explanation and analysis on what I should work, I don't think I myself could work out this course project.

I also want to appreciate my gratitude to our TA (although she's on strike now) Ms. Sadaf Mustafiz. Her attentiveness and patience makes me never dare to be careless. She repeatedly reminded me what I should've submitted and when I should meet her to explain my assignments. I think this is really a good habit to help students learning.

7 References

[1] Jeffrey Nichols et al. "Generating Remote Control Interfaces for Complex Appliances", UIST 2002

[2] Brouwe-Janese et al. "Interfaces for consumer products: "how to camouflage the computer?"" CHI 1992

[3] Jeffrey Nichols et al. "UNIFORM: Automatically Generating Consistent Remote Control User Interfaces" CHI 2006