Mobile Device Application Synthesis from Domain-specific Models

Raphael Mannadiar

McGill University rmanna@.cim.mcgill.ca

Abstract. A meta-model for modelling mobile device applications (MDA) is presented along with transformations to translate domain-specific models into Google Android (GA) code that runs on GA enabled devices.

1 Introduction

From bits to assembly code, from assembly code to Object-Oriented programming languages, from OO languages to the UML, software development has been steadily moving away from the machine language of 0s and 1s to the human language of English for several decades. However, despite the UML being far more intuitive than bits and opcodes to a human developer, it is still very close to the underlying OO philosophy which, it can be argued, should not concern higher level domain experts who wish to develop domain-specific applications. Hence, further movement away from machine language and code and toward the domain model is still required. The following presents a meta-model at a level of abstraction appropriate to the development of mobile device applications. This meta-model alone would not solve the problem introduced above so transformations are described to translate domain-specific models into running code on GA enabled devices.

Section 2 will describe related work and provide some useful background. Section 3 will describe the pertinent meta-models while section 4 will describe the transformations that lead from one the other. Finally, section 5 will explore example models that were built using the introduced meta-model.

2 Related Work and Background

The most pertinent related work is that of Nokia who designed a meta-model for MDA using MetaCase's MetaEdit+ [1]. Their meta-model, currently used to develop applications targeted to the public smartphone platform Series 60, is very similar to the one introduced in this work but the lack of published documentation makes it difficult to judge the span of the meta-model's features.

Another noteworthy effort is DroidDraw [2]. This applet produces GA XML code for a graphically specified GA GUI.

Before proceeding, a brief description of a general GA [3] application seems appropriate. A GA application has 3 key components:

– A manifest file, *AndroidManifest.xml*, which holds some information regarding the application's features and classes.
– XML files, *res/layout/*.xml*, which each describe a visible screen on the mobile device.
– Java files, *src/package/name/*.java*, which describe the control flow of the application.

This is of course a very crude description but provides the key notions to understand the following. Much more information can be found at [3].

3 The Meta-Models

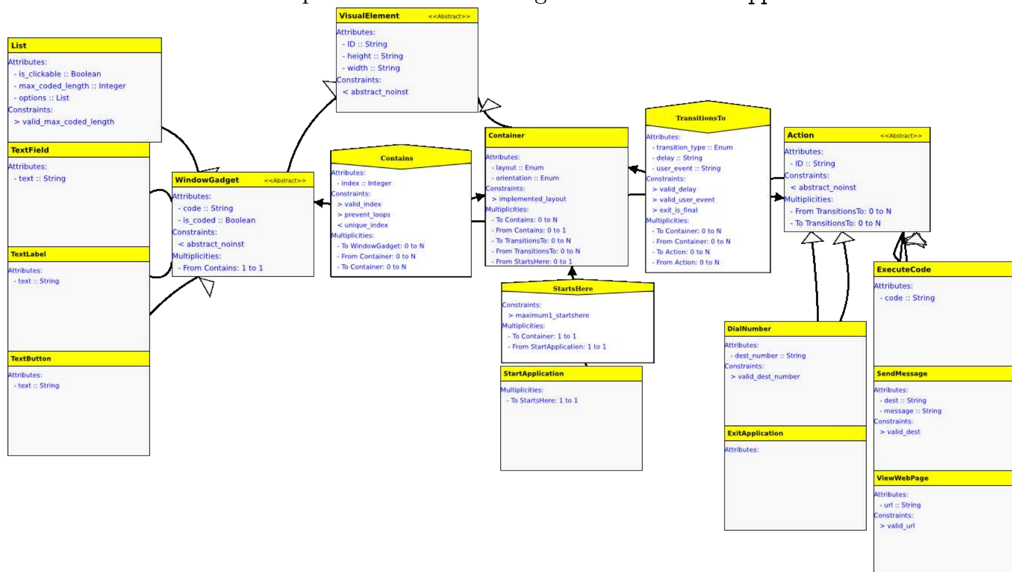
The are in fact 3 meta-models in play: `PhoneApps`, `TextualPhoneApps` and `GA`. A sufficient description of the latter was already given; this section will describe the 2 others which were both developed using the `AToM3` [4] modelling environment.

3.1 PhoneApps

This is the domain-level meta-model that is used by the mobile device application developer. It provides a sufficient set of components to express a wide variety of applications. Note however that it is not sufficient to express all of the features of `GA`. Restrictions were imposed both by the limited device emulator provided by `Google` and by the time constraints for the development of this work. Future work on both limiting factors should lead to a complete meta-model with the same expressiveness as `GA`.

3.1.1 Model Components and Associations

Below is a detailed description of the class diagram of the `PhoneApps` meta-model:



- `VisualElement`: an abstract class that represents anything related to the GUI i.e. what appears on the screen of the mobile device; all subclasses inherit the following attributes: *height*, *width* and *ID* where *ID* is a unique identifier that can be used to reference the component elsewhere in the model.

  - `Container`: contains – containment is represented by `Contains` edges – some number of `VisualElement`s whose positions are determined by the *index* attribute of the appropriate containment edge (further positioning information is encoded in a `Container`'s *layout* and *orientation* attributes); `Container`s that aren't contained in any other represent full screens on the mobile device and are referred to as top-level `Container`s.

  - `WindowGadget`: an abstract class that represents simple widgets that are used to populate `Container`s; all subclasses inherit the following attributes: *code* and *is_coded* where *code* is non-mandatory modeller-specified Java code that can generate the contents of the widget when this content is the result of complex computations (e.g. retrieving data on the Web, computing mathematical equations, etc.) that can not be modelled with the provided meta-model.

    - `TextLabel`: represents a label; the *text* attribute of this component can either be generated by Java code or be explicitly specified.

    - `TextField`: represents a editable textfield; the *text* attribute of this component can either be generated by Java code or be explicitly specified.

    - `TextButton`: represents a clickable button; the *text* attribute of this component can either be generated by Java code or be explicitly specified.

    - `List`: represents a list of options; the *options* attribute of this component can either be generated by Java code or be explicitly specified; the *is_clickable* attribute determines if the options in the list should be rendered as buttons or labels.

- `Action`: an abstract class that represents an action, be it a user-specified action (e.g. `ExecuteCode`) or a built-in action provided by the mobile device (e.g. `SendMessage`); all subclasses inherit the following attribute: *ID*, a unique identifier.

  - `SendMessage`: represents the sending of a textual message to some recipient (phone number, email address); the *dest* and *message* attributes are explicitly specified.

  - `DialNumber`: represents the dialing of number; the *dest_number* attribute is explicitly specified.

  - `ViewWebPage`: represents the launching of a web browser at some explicitly specified URL, *url*.

  - `ExecuteCode`: this component enables the modeller to specify Java code to be run; it can be especially useful to overcome some of the limitations of the

introduced meta-model such as reading/writing `GA` data structures.

- `ExitApplication`: represents closing the application and returning to whatever the mobile device was doing before its launch.

– `StartApplication`: represents the starting point of the application; it is connected to a `Container` that will become the initial screen of the application.

One type of class association has yet to be discussed: `TransitionsTo`s. In short, under certain conditions, `Container`s and `Action`s can transit to `Container`s and `Action`s. This represents transiting from screen to screen, from screen to action to screen, from action to action... Such transitions are parametrized via the modeller-specified mutually exclusive *delay* and *user_event* attributes of `TransitionsTo` edges.

– The *delay* attribute enables the modeller to specify that a screen should transition elsewhere after some number of milliseconds or that the application should do something/go somewhere after an action has completed.
– The *user_event* attribute enables the modeller to specify that a screen should transition elsewhere after the user clicks on some option of a clickable list or on some button.

3.1.2 References

The previous subsection mentioned that it was possible to reference model components and to specify "user events" based on list option and button clicks. This subsection explores the syntax to accomplish this.

There are 3 scenarios were one can reference other model components:

1. When specifying a component's attribute:
   e.g. a modeller may want some action to trigger the sending of a text message where the message text contains text that was specified by the user in a textfield from a previous screen as well as a choice the user made in a list from a previous screen; this could be accomplished by specifying the *message* attribute of the `SendMessage` instance to "the user entered $<<< textFieldID.text >>>$ and chose $<<< listID.??? >>>$ sometime ago".

2. When specifying a button-click user event:
   e.g. a modeller may want the click of button to trigger the exiting of the current `Container`; this could be accomplished by specifying the *user_event* attribute of the appropriate outgoing `TransitionsTo` edge to "$onClick(<<< buttonId >>>)$".

3. When specifying a clickable list option-click user event:
   e.g. a modeller may want the click of one or any of a clickable list's options to trigger the exiting of the current `Container`; these 2 cases could be accomplished by specifying the *user_event* attribute of the appropriate outgoing `TransitionsTo` edge to "$onClick(<<< listId = optionText >>>)$" and "$onClick(<<< listId = ??? >>>)$" respectively.

3.1.3 Code

Code can be specified either to generate content for a coded `WindowGadget` or as the *code* attribute of an instance of `ExecuteCode`. Modeller-specified Java code can reference model components as explained above (hence yielding non-compilable Java code). Furthermore, it must follow a strict syntax: all the code must be encompassed in a class named *ExecuteCode* that must contain a public method named *executeCode()* with return type *String*, $java.util.ArrayList < String >$ or *void*.

3.1.4 Constraints

Many constraints are applied on the meta-model to ensure that only valid models are saved and to ensure that certain assumptions made in the transformations below hold. The following is a sample of these constraints:

- `abstract_noinst`: ensures that it is impossible to instantiate abstract classes (e.g. `Action`, `WindowGadget`)

- `prevent_loops`: ensures that a `Container` can not contain a `Container` that directly or transitively contains it.

- `exit_is_final`: ensures that no `TransitionsTo` edges exit instances of `ExitApplication`.

- `valid_*` where * is some attribute name: ensures that the given attribute is valid (e.g. the *ID* attribute matches this pattern "$[a-zA-Z_0-9]+$")

- `valid_references`: ensures that every "$<<< elementId.[attributeName|???] >>>$" pair references a valid $elementId$.

- `container_large_enough`: ensures that a `Container` is large enough to display all of its direct and transitive contents.

- `flag_unreachable_containers`: ensures that there are no unreachable `Container`s in the model.

- `validate_ALL`: ensures that constraints that were never checked (e.g. because they are triggered by the editing of an element that was never edited) are checked.

Note that many constraints such as multiplicity constraints do not have an explicit model constraint like those above but are implicit from the meta-model definition (i.e. the class diagram).

3.1.5 Notes

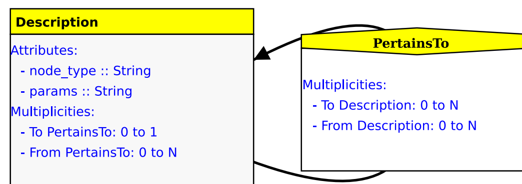
Some of the design choices above may seem bizarre or absurd. Many of such choices were made to accommodate the modelling environment. Examples of this are:

- Due to a problem with the matching of *Text* type variables in the `AToM3` transformation mechanism, the *code* attribute needs to be specified by a *String* rather than a *Text*. This poses the obvious inconvenience of coding in a 20 character long textfield. This is however no more than a minor inconvenience since it is assumed that most modellers would develop their code snippets using a Java editor rather that the bundled `AToM3` Python editor.

– Due to a problem with multiple levels of inheritance and associations among classes being ported to their sub-sub-classes, no ModelElement with attribute *ID* was created; the *ID* attribute was instead created once for Actions and once for VisualElements.
– Due to limitations in specifying when a constraint should be checked, it was decided to create the validate_ALL constraint, checked before saving, rather than to create twin constraints for many already existing constraints that would be checked after editing and before saving respectively.

3.2 TextualPhoneApps

This meta-model is not seen by the modeller. It is merely a simplified representation of a PhoneApps model where superfluous information has been removed and important information has been aggregated to make the translation to GA code easier. All TextualPhoneApps models are generated by the transformations that will be explored in the next section.



As shown in the previous class diagram, a TextualPhoneApps model is simply a collection of possibly connected instances of Descriptions where the *node_type* and *params* attributes take on values that summarize the values of the attributes of corresponding PhoneApps model elements. This process will become clearer in the following section.

4 The Transformations

The meta-models have been introduced. It is now time to explore the steps that lead from a PhoneApps model to a GA application.

4.1 PhoneApps2TextualPhoneApps

This transformation translates PhoneApps models to TextualPhoneApps models. It is made up of the following rules:

1. Initial Action
   Every element in the PhoneApps model is marked as visited.

2. StartApplication2Desc (Order = 1)
   The unique instance of StartApplication in the PhoneApps model is replaced by a new instance of Description where *node_type* is *StartApplication* and *params* is $< start : StartingContainerId/start >$ where *StartingContainerId* is the *ID* of the Container to which the instance of StartApplication was connected.

3. `Container2Desc` (Order = 1)
   A new instance $d1$ of `Description` is connected to every `Container` in the `PhoneApps` model. $d1$ has *node_type* set to $Container$ and *params* set to $< ID : ContainerId/ID >< height : ContainerHeight/height >< width : .../width >< layout : .../layout >< orientation : .../orientation >< containsI : VisualElementId/containsI >$ where there is one $containsI$ for every `VisualElement` contained in the `Container` and I represents the index of the corresponding `Contains` edge.
   A second instance of `Description` is also created and connected to $d1$; this instance has *node_type* set to $Transitions$ and *params* set to $< src_ID : ContainerId/src_ID >< transitionsto0 : DestId0/transitionsto0 >< onevent0 : Event0/onevent0 > ... < transitionstoN : DestIdN/transitionstoN >< oneventN : EventN/oneventN >$ where there is one $transitionstoI$ for every outgoing `TransitionsTo` edge; $EventI$ is the event that triggers the transition to the `Action` or `Container`, described by $DestIdI$, that lies on the other end of the `TransitionsTo` edge. To ensure this rule is only executed once per `Container`, the *visited* flag set in the initial action is checked before execution and set after execution.

4. `TextField2Desc` (Order = 2)
   Every `TextField` in the `PhoneApps` model is replaced by a new instance of `Description` where *node_type* is $TextField$ and *params* is $< ID : TextFieldId/ID >< height : TextFieldHeight/height >< width : .../width > VALUE$ where $VALUE$ is either $< text : ValueOfTextAttribute/text >$ or $< code : ValueOfCodeAttribute/code >$ depending on whether the particular `TextField` is coded or not.

5. `TextLabel2Desc` (Order = 2)
   Same as `TextField2Desc` but for `TextLabel`s instead of `TextField`s.

6. `TextButton2Desc` (Order = 2)
   Same as `TextField2Desc` but for `TextButton`s instead of `TextField`s.

7. `List2Desc` (Order = 2)
   Every `List` in the `PhoneApps` model is replaced by a new instance of `Description` where *node_type* is $List$ and *params* is $< ID : ListId/ID >< height : ListHeight/height >< width : .../width >< is_clickable : True|False/is_clickable > VALUE$ where $VALUE$ is either $< options : [ValueOfOption0]...[ValueOfOptionN]/options >$ or $< code : ValueOfCodeAttribute/code >< max_length : ValueOfMaxCodedLengthAttribute/max_length$ depending on whether the particular `List` is coded or not.

8. `SendMessage2Desc` (Order = 2)
   A new instance $d1$ of `Description` is connected to every `SendMessage` in the `PhoneApps` model. $d1$ has *node_type* set to *Action* and *params* set to $< ID : SendmessageId/ID >< dest : .../dest >< message : .../message >< type : SendMessage/type >$.
   A second instance of `Description` is also created and connected to $d1$; this instance has *node_type* set to $Transitions$ and *params* set to $< src_ID : SendMessageId/src_ID >< transitionsto0 : DestId/transitionsto0 >< onevent0 : onComplete/onevent0 >$ where $DestId$ is the ID of the `Action` or the `Container` on the other end of the `TransitionsTo` edge. Note that unlike `Container`s, the only possible trigger out of an `Action` is the completion of the associated action which can only lead to one "place" to preserve determinism. To ensure this rule is only executed once per

SendMessage, the *visited* flag set in the initial action is checked before execution and set after execution.

9. DialNumber2Desc (Order = 2)
   Same as SendMessage2Desc but for DialNumbers instead of SendMessages. The only difference is that $d1$'s *params* attribute is now set to $< ID : DialnumberId/ID >< dest_number : .../dest_number >< type : DialNumber/type >$.

10. ViewWebPage2Desc (Order = 2)
    Same as SendMessage2Desc but for ViewWebPages instead of SendMessages. The only difference is that $d1$'s *params* attribute is now set to $< ID : ViewWebPageId/ID >< url : .../url >< type : ViewWebPage/type >$.

11. ExecuteCode (Order = 2)
    Same as SendMessage2Desc but for ExecuteCodes instead of SendMessages. The only difference is that $d1$'s *params* attribute is now set to $< ID : ExecuteCodeId/ID >< code : .../code >< type : ExecuteCode/type >$.

12. RedirectContainedContainers (Order = 4)
    Every Contains edge from Container $c1$ to Container $c2$ is replaced by an edge from the Description that represents $c2$ to the one that represents $c1$.

13. RedirectContentsToContainerDesc (Order = 5)
    Every edge from Container $c1$ to Description $d1$ – such edges were created in rules 4 to 7 – is replaced by an edge from $d1$ to the Description that represents $c1$.

14. ExitApplications2Desc (Order = 6)
    Every ExitApplication in the PhoneApps model is replaced by a new instance of Description where *node_type* is *ExitApplication* and *params* is $< ID : ExitApplicationId/ID >$.

15. CleanContainers, CleanSendMessages, CleanDialNumbers, CleanViewWebPages, CleanExecuteCodes (Order = 6)
    Every remaining PhoneApps model element is removed.

The initial PhoneApps model has now been reduced to a much simpler version of itself on which the TextualPhoneApps2Android transformation will now be applied.

You may have noticed that some of the above rules seem redundant; this is due to the fact that AToM3 does not seem able to apply rules declared on parents to their sub-classes.

4.2 TextualPhoneApps2Android

This transformation translates generated TextualPhoneApps models to GA applications. It is made up of the following rules:

1. Initial Action
   - Initializes and populates several global dictionaries and lists that are read and written by the rules below.

- Removes old output directory if any and creates new one.
- Replaces any occurrence of $<<< ElementId.AttributeName >>>$ by $pastEntries.get("<<< ElementId.AttributeName >>>")$ in all the values of the *params* attributes of all the model elements. *pastEntries* is a global *HashMap* in the final generated `GA` Java code.

2. `TextField_Label_Button2Android` (Order = 1)
Deletes every instance of `Description` with *node_type* *TextField*, *TextLabel* or *TextButton* and...
   - Produces appropriate `GA` XML code and stores it in global dictionary *XML_code* under *ElementId*.
   - Produces Java code to set the text of the button/label/textfield and stores it in global dictionary *settext_code* under *ElementContainerId* where *ElementContainerId* is the *ID* of the top-level `Container` in the `PhoneApps` model that contained this element. If the element is not coded, this code explicitly sets the text to that specified by the modeller. If it is, this code is a call to *setText_ElementId*(), a generated function that calls modeller-specified Java code and sets the text of the element to the result of the *public String executeCode()* function. The code for the *setText_ElementId*() function, also generated within this rule, is stored in global dictionary *android_code* under *ElementId*.
   - If *node_type* is *TextField* and its *ID* – encoded in the `Description`'s *params* attribute – is referenced elsewhere – references are resolved as part of the initial action –, produces Java event listener code that will update $pastEntries("<<< TextFieldId.text >>>")$ as the user types in the textfield and stores this code in global dictionary *event_listeners* under *ElementContainerId*.

3. `List2Android` (Order = 1)
Deletes every instance of `Description` with *node_type* *List* and...
   - Produces appropriate `GA` XML code and stores it in global dictionary *XML_code* under *ListId*. This code is essentially a *LinearLayout* of buttons or labels depending on if the list is clickable or not.
   - Produces Java code to set the text of the buttons/labels that make up the list and stores it in global dictionary *settext_code* under *ElementContainerId*. If the list is not coded, this code explicitly sets the text to that specified by the modeller. If it is, this code is a call to *setList_ListId*(), a generated function that calls modeller-specified Java code and sets the text of the buttons/labels to the results of the *public java.util.ArrayList*$<$ *String* $>$ *executeCode()* function. The code for the *setList_ListId*() function, also generated within this rule, is stored in global dictionary *android_code* under *ListId*.
   - If this list is clickable, produces Java event listener code with cases for each option in the list and stores this code in global dictionary *android_code* under under *ListId*. This code is encapsulated in a function called *onListItemClick_ListId*(*String selection*). Some portions of the *onListItemClick_ListId*(*String selection*) function are not available when its code is generated. To overcome this, special comments are generated in the method body (e.g. "//ACTION_FOR_ListOption" and "//DEFAULT_ACTION") and are replaced by the appropriate code in the `Transitions2Android` rule.
   - If this list is clickable, produces Java event listener code to ensure each "list-button" calls *onListItemClick_ListId*(*ButtonText*) upon being clicked and stores it in global dictionary *event_listeners* under *ElementContainerId*.

– If the list is clickable and is referenced elsewhere, produces Java code that will update $pastEntries$(" <<< $listId$.??? >>> ") when the user clicks on a "list-button". This code becomes the first line of the $onListItemClick_ListId(String\ selection)$ function.

4. `SendMsg_DialNum_ViewUrl2Android` (Order = 1)
Deletes every instance of `Description` with *node_type Action* and *params* containing $<$ *type* : $SendMessage$/*type* $>$, $<$ *type* : $DialNumber$/*type* $>$ or $<$ *type* : $ViewWebPage$/*type* $>$ and...
   – Produces Java code that carries out the action (e.g. makes the required `GA` function calls to send a text message) and stores it in global dictionary *android_code* under $ActionId$. This code is encapsulated in a function called $action_ActionId()$.
   – If *type* is $DialNumber$, a special line of `GA` XML indicating that the application requires the permission to dial the phone is produced and stored in global list *permissions*.

5. `ExecuteCode2Android` (Order = 1)
Deletes every instance of `Description` with *node_type Action* and *params* containing $<$ *type* : *ExecuteCode*/*type* $>$ and...
   – Produces Java code that carries out the action (i.e. instantiates the *ExecuteCode* class provided by the modeller and calls its member function *public void executeCode()*) and stores it in global dictionary *android_code* under $ExecuteCodeId$. This code is encapsulated in a function called $action_ExecuteCodeId()$.

6. `Container2Android` (Order = 2)
Deletes every `Description` with *node_type Container* that has no incoming connections from `Description`s with *node_type Container* (i.e. either there were never any such connections or the ones that existed were removed by previous applications of this rule; this condition actually implies that all of the original `Container`'s contents have already been translated by this rule or previous ones) and...
   – Produces appropriate `GA` XML code and stores it in global dictionary *XML_code* under $ContainerId$.
   – If the concerned `Description` describes a `Container` that was contained in no other, outputs the produced XML to a file name $screen_ContainerId.xml$ and, produces Java method $transitionTo_screenContainerId()$ and stores it in global dictionary *android_code* under $ContainerId$. This method is structured as follows:
      (a) Sets the mobile device's current screen to the one defined in $screen_ContainerId.xml$.
      (b) Invokes every statement in global dictionary value *settext_code[*$ContainerId$*]*.
      (c) Invokes every statement in global dictionary value *event_listeners[*$ContainerId$*]*.
      (d) If there is a delay before the exiting of this screen, invokes statements that will implement the timeout.

   Some portions of the $transitionTo_screenContainerId()$ function are not available when its code is generated. To overcome this, special comments are generated in the method body (e.g. "//DELAY_ON_EXIT" and "//EVENT_LISTENERS") and are replaced by the appropriate code in the final action.

7. `Transitions2Android` (Order = 3)
   Deletes every `Description` with *node_type* $Transitions$ and generates code corresponding to the events stored in the *params* attribute (recall its structure is $< src_ID : SrcId/src_ID >< transitionsto0 : DestId0/transitionsto0 >< onevent0 : Event0/onevent0 > ... < transitionstoN : DestIdN/transitionstoN >< oneventN : EventN/oneventN >$) according to the logic below:
   For each $< transitionstoI : DestIdI/transitionstoI >< oneventI : EventI/oneventI >$ where I takes on values 0 to some N
   - If $EventI$ is *onComplete*, produce Java code that will invoke $transition_code$ (defined below) upon completion of the action described by $SrcId$ and store this code in global list *oncomplete_events*.
   - If $EventI$ is $after(millis_delay)$, produce Java code that will invoke $transition_code$ after *millis_delay* have gone by since entering the screen described by $SrcId$ and replace the single occurrence of "//DELAY_ON_EXIT" in global dictionary value *android_code[SrcId]* by the produced code. Reminder, the value of *android_code[SrcId]* is the definition of the $transitionTo_screenSrcId()$ function.
   - If $EventI$ is *onClick*($<<< ButtonId >>>$), produce Java event listener code that will invoke $transition_code$ whenever the user clicks on the button described by $ButtonId$ and store this code in global dictionary *event_listeners* under $ElementContainerId$ where $ElementContainerId$ is the *ID* of the top-level `Container` in the `PhoneApps` model that contained the concerned button.
   - If $EventI$ is *onClick*($<<< ListId =??? >>>$), produce Java code that invokes $transition_code$ and replace the single occurrence of "//DEFAULT_ACTION" in global dictionary value *android_code[ListId]* by the produced code. Reminder, the value of *android_code[ListId]* is the definition of the $onListItemClick_ListId(String\ selection)$ function.
   - If $EventI$ is *onClick*($<<< ListId = ListOption >>>$), produce Java code that invokes $transition_code$ and replace the single occurrence of "//ACTION_FOR_ListOption" in global dictionary value *android_code[ListId]* by the produced code.

   The $transition_code$ mentioned above is set according to the logic below:
   - If $DestIdI$ was the *ID* of an `Action` in the original `PhoneApps` model, $transition_code = action_DestIdI(); return;$
   - Else If $DestIdI$ was the *ID* of a `Container` in the original `PhoneApps` model, $transition_code = transitionTo_screenDestIdI(); return;$
   - Else If $DestIdI$ was the *ID* of an `ExitApplication` in the original `PhoneApps` model, $transition_code = finish(); return;$ where *finish()* is a `GA` function that terminates the current *Activity* and returns the mobile device to its state prior to launching the current *Activity*.

8. `StartApplication2Android` (Order = 4)
   Deletes the unique `Description` with *node_type* *StartApplication* and sets global variable *starting_container_ID* to the container *ID* specified in the `Description`'s *params* attribute.

9. `CleanExitApplication` (Order = 6)
   Deletes every `Description` with *node_type* *ExitApplication*.

10. `Final Action`
    This is where all the pieces of the puzzle come together: the global variables that were populated throughout the `TextualPhoneApps2Android` transformation are cleverly pieced together to produce a `GA` application equivalent to the original `PhoneApps` model. Below are the steps that make up this final rule:
    - For each *ID ContainerId* that describes a top-level `Container` in the original `PhoneApps` model, replace the single occurrence of "//EVENT_LISTENERS" in global dictionary value *android_code[ContainerId]* by global dictionary value *event_listeners[ContainerId]*
    - Produce and output `AndroidManifest.xml`. This is very straightforward: all of the contents the manifest file are static at the exception of the permissions section which is populated with the contents of the global list *permissions*.
    - The contents of the global list *on_complete_events* are placed in the definition of `GA` method *public void onWindowFocusChanged(boolean hasFocus)*, which is called any time an action (i.e. SendMessage, ExecuteCode, ...) terminates, and the resulting code is stored in local variable *on_complete_events_code*
    - The final step consists in producing and outputting `PhoneApp.java`, the Java file that contains all of the application logic.
      (a) Static package and import statements are produced followed by
      (b) Global variable declarations (e.g. *pastEntries*) followed by
      (c) The `GA` equivalent for a main, *public void onCreate(Bundle icicle)*, which contains a unique function call: *transitionTo_screenStartingContainerId*(); where *StartingContainerId* is retrieved from global variable *starting_container_ID* followed by
      (d) The contents of local variable *on_complete_events_code* followed by
      (e) All the contents of the global dictionary *android_code*.

The 3 key components of a `Google Android` application have now been produced:

- A collection of screen_*.xml files
- The `AndroidManifest.xml` file
- A Java file that contains application logic

All of the output files are generated in *CURRDIR/_dest_dir/* where *CURRDIR* is the directory from where `AToM3` was launched. The only remaining steps are to move these files to their appropriate locations as detailed in Section 2 and to run the generated application on the `GA` mobile device emulator.

5 Example Models

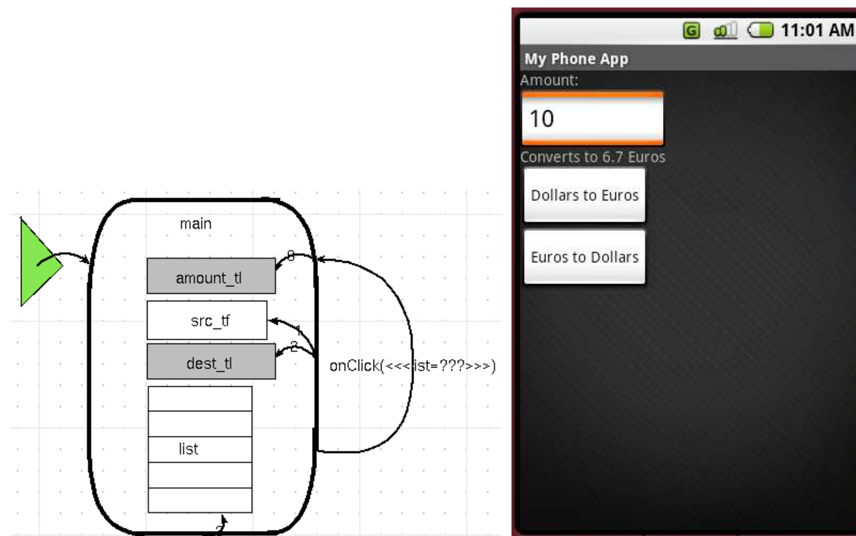
Three example models were created to demonstrate the various features of the introduced DSMM.

5.1 `Menu` Application

This application demonstrates the meta-model's ability to express the very common `Main Menu` application that ships with modern phones. In short, a list of options is displayed each representing available applications such as dialing a number, sending a text message or browsing the web. The `PhoneApps` model for this example is located in *User Formalisms/PhoneApps/test_MENU_MDL.py*.

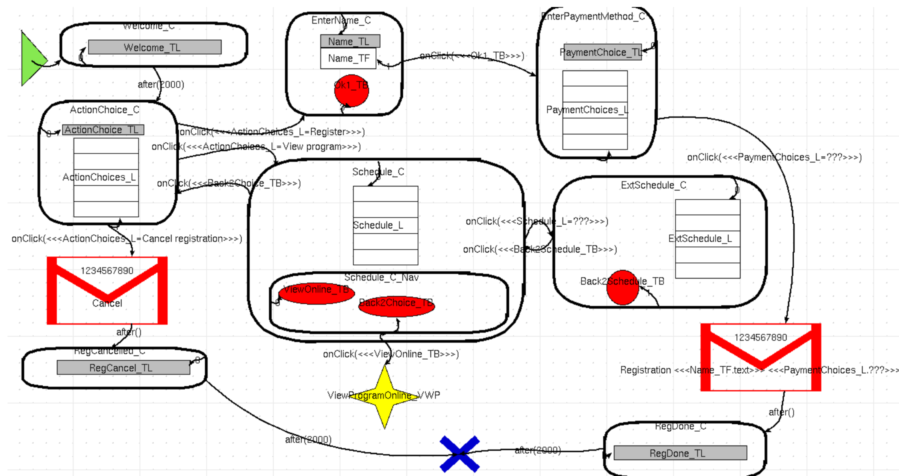
5.2 `Currency Converter` Application

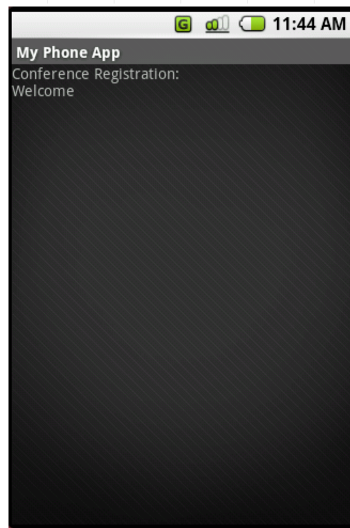
This application demonstrates the meta-model's ability to express another very common application on hand-held devices: a currency converter. In short, a screen is displayed where the user can enter an amount and choose to convert it from Dollars to Euros and vice versa. This application demonstrates coded `TextFields` (the conversion is computed in modeller-specified code that could potentially get live exchange rates of the Web and returned by *public String executeCode()*), clickable `Lists` and `TransitionsTo` self-loops. Below are screenshots of the `PhoneApps` model (located in *User Formalisms/PhoneApps/test_CURRENCY_CONVERTER_MDL.py*) and the application running on the `GA` mobile device emulator.

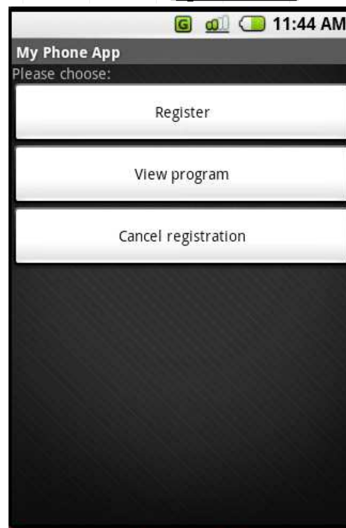


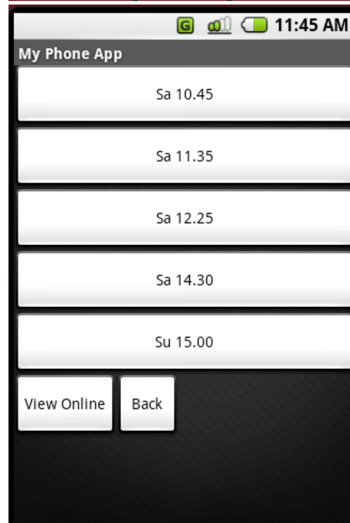
5.3 Conference Registration

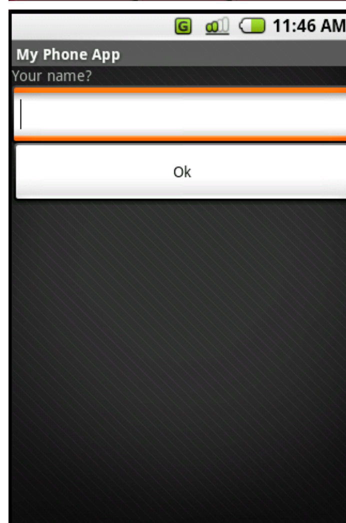
This application is much more complex than the other two. It demonstrates the meta-model's ability to express applications that span several screens, that make use of mobile device features such as sending text messages and viewing web pages, that use timeouts to transit from one screen to the next, and more. Below are screenshots of the `PhoneApps` model (located in *User Formalisms/PhoneApps/test_CONFERENCE_REGISTRATION_MDL.py*) and the application running on the `GA` mobile device emulator.


Welcome_TL
after(2000)
EnterName_C
Name_TL
Name_TF
Ok1_TB
onClick(<<<Ok1_TB>>>)
EnterPaymentMethod_C
PaymentChoice_TL
PaymentChoices_L
ActionChoice_C
ActionChoice_TL
ActionChoices_L
onClick(<<<ActionChoices_L=Register>>>)
onClick(<<<ActionChoices_L=View program>>>)
onClick(<<<Back2Choice_TB>>>)
Schedule_C
Schedule_L
onClick(<<<PaymentChoices_L=???>>>)
ExtSchedule_C
ExtSchedule_L
onClick(<<<Schedule_L=???>>>)
onClick(<<<Back2Schedule_TB>>>)
Back2Schedule_TB
onClick(<<<ActionChoices_L=Cancel registration>>>)
1234567890
Cancel
Schedule_C_Nav
ViewOnline_TB
Back2Choice_TB
after()
RegCancel_TL
onClick(<<<ViewOnline_TB>>>)
ViewProgramOnline_VWP
1234567890
Registration <<<Name_TF.text>>> <<<PaymentChoices_L.???>>>
after()
RegDone_C
RegDone_TL
after(2000)
after(2000)


11:44 AM
My Phone App
Conference Registration:
Welcome


11:44 AM
My Phone App
Please choose:
Register
View program
Cancel registration


11:45 AM
My Phone App
Sa 10.45
Sa 11.35
Sa 12.25
Sa 14.30
Su 15.00
View Online
Back


11:46 AM
My Phone App
Your name?
Ok

6 Conclusion and Future Work

A domain-specific meta-model for developing mobile device applications has been introduced as well as the means to synthesize `Google Android` applications from domain-specific models. No research efforts have attempted to compare the 'exact" speed up of developing applications using the introduced DSMM rather than implementing them directly on the `GA` platform. Nevertheless, the obvious similarities between the example models shown in the previous section and the synthesized applications is a testament to the fact that the `PhoneApps` meta-model is indeed the proper level of abstraction for the development of MDA.

As for future work, full support for `GA` widgets along with more complete integration with all `GA` features and data structures would definitely make MDA modelling with `PhoneApps` a strong alternative to coding in `GA`. Finally, interesting and useful extensions to `PhoneApps` would be to add an elegant way to do user input checking before transiting out of `Container`s and providing an efficient mechanism to self-loop on `Container`s without having to reload widget contents.

References

1. MetaCase: Domain-specific modelling with MetaEdit+: 10 times faster than UML 2007
2. DroidDraw: http://www.droiddraw.org/
3. Google Android: http://code.google.com/android/
4. AToM3: http://atom3.cs.mcgill.ca/
5. Android Demo : http://www.youtube.com/watch?v=1FJHYqE0RDg
6. Skypop on Android : http://www.youtube.com/watch?v=PyxWnIalDcY
7. Androidology - Part 1 of 3 - Architecture Overview : http://www.youtube.com/watch?v=Mm6Ju0xhUW8
8. Androidology - Part 2 of 3 - Application Lifecycle : http://www.youtube.com/watch?v=ITfRuRkf2TM
9. Androidology - Part 3 of 3 - APIs : http://www.youtube.com/watch?v=iiD4fGjjXcc
10. A first hand look at building an Android application : http://www.youtube.com/watch?v=ENRcZcnoptM