Android Executable Modeling: Beyond Android Programming

Olivier Le Goaer, Franck Barbier, Eric Cariou, and Samson Pierre
Université de Pau / LIUPPA, PauWare Research Group, BP 1155,
F-64013 PAU CEDEX, France
Email: {firstname.name}@univ-pau.fr

***Abstract*—This paper demonstrates through an example how a modeling effort can substitute to a programming effort so that a main part of the code of apps for smart devices can be replaced by a model. We focus on the behavioral model of an application and then instrument its direct execution on an Android device thanks to the PauWare API. The proposed installation of PauWare on Android OS sets up the foundation for a whole range of mobApps, provided they are modeled with the statechart formalism.**

I. Introduction

As early highlighted by Parnas [13], abstraction has always been a key factor to successful software engineering. Among the manifold forms taken by abstraction, modeling has proved its efficiency for handling complexity of software development, contrasting with the classical programming all focused on source code. Indeed, models abstract away details to concentrate on particular, high-level, viewpoints on the system to be built. As such, they offer unparalleled reasoning supports to designers. Models are so powerful that they have earned their place in the software engineering through a dedicated sub field called Model-Driven Engineering (MDE). After having been intensively used as contemplative assets till the mid 2000 (a recurring analogy was blueprints in architecture), models have been turned into productive assets, relying on automated transformation chains ending predominately to source code. It is worthwhile mentioning that this evolution owes much to the OMG's MDA initiative [7]. A more recent trend is to see a model as an end in itself by directly executing it [10], [5]. The analogy could be now those of an "animated" or "actioned" blueprint which may serve as way of simulation of course but also as a full-fledged executable system so that the implementation stage is totally skipped. This vision shift from static (albeit productive) model to dynamic model tends to abolish the boundaries between modeling and programming, and its slogan might be "*What you model is what you get*" (WYMIWYG).

The entirely model-centered and hence fast development allowed by executable – or more precisely, interpretable – model approach is particularly interesting when focusing on tiny devices or on embedded software like for Smart-* (Phones, Watches, Glasses, TVs, ...). Indeed, these applications are characterized by a high time-to-market pressure, a rapid fluctuation of user's requirements, while they run on top of fast-paced platforms. This situation is going to explode with the future Internet of Things, and the arrival of a multitude of connected objects. Because a growing number of these systems are running with Android, we choose the successful operating system from Google to propose our work.

The rest of this paper is organized as follows. Section II sketches a homemade mobApp built utterly with a state machine model. Consequently, the section III explains how to reuse the PauWare API, a toolkit dedicated to the execution of statechart models, on the Android platform. Section IV gives technical details about how all this stuff works on an Android device, as a proof of concept. Related work is exposed in Section V before we conclude in Section VI.

II. A Working Example

A prominent example of modeling is the finite state machine viewpoint to represent the behavior of a running system. When looking at the lines of code you will never see any "state", "transitions", "guards" or something like that, but rather a bunch of statements, controls, function calls, data passing, etc. Hence, abstracting away all of that with the statechart formalism is a pure mental effort, which is the essence of modeling. Moreover, even if the system behavior has been modelized through a state machine, this model is usually used in a static way by the developer for helping him in writing the code, through code generation or not. In this paper, we show that this model can contrariwise directly be used as a running part of the system. The model is turned into a dynamic item, suppressing the necessity of its implementation within the application code.

A. Energy-aware mobApp

We can simply imagine an Android mobile application embodying an assistant or companion for energy-saving/aware that reacts to power-related events and give some (not too serious) advices to the end-user as feedbacks. This kind of application is arguably one of the most amenable to be entirely modeled with a state machine. The result is presented in Fig. 1.

The statechart is composed of four states, including a composite state, and seven transitions. All the focus is put on the model, so that no Android programming skills are required here. The states are enough expressive and even the labels set on transitions, namely the couple event/action (no guards here), just call for a superficial knowledge of Android OS. Indeed, it is quite intuitive and it works very similarly in further mobile OS. Description of this Android-related vocabulary is provided below.

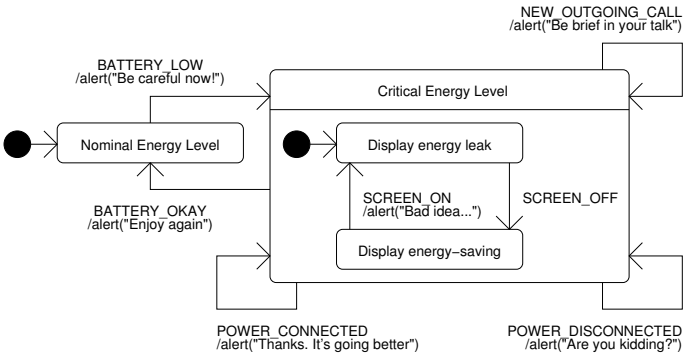

Fig. 1. Specification of the behavior of the mobApp by means of an UML 2 State Machine Diagram

B. Android-related Events and Action

Among the list of native events supported by Android (120 available for API 17 for example[1]), we kept seven distinct ones. The selected events are first relevant from a power viewpoint, but also self-described: their name is enough and do not require handling additional data conveyed in the event and called "extras" in the Android jargon[2].

The `BATTERY_LOW` and `BATTERY_OKAY` events refer to a variation of the threshold value of the status of charge of the battery. `SCREEN_ON` and `SCREEN_OFF` occur when the display surface (OLED basically) is activated or deactivated by the user or by a timer, as well. `POWER_CONNECTED` and `POWER_DISCONNECTED` means that the owner intentionally decided to plug or unplug his device to an external energy source. Incoming calls are ignored by the assistant insofar they do not result from a desire of the user, but every outgoing call is caught by a new `NEW_OUTGOING_CALL` event.

The unique kind of action triggered when crossing transitions is to alert the end-user with a comment that aims to provide friendly guidance for the energy management of her/his smart device.

III. Building a mobApp with the PauWare API

Building a mobApp through a real executable model approach requires a set of three elements: an engine responsible for model execution, the model of your application of course, and a connector to glue the engine's inputs/outputs with the underlying platform features, here Android.

A. PauWare Engine

PauWare engine is a lightweight execution engine for state machines which implements the full UML 2 semantics that deals with sophisticated features: concurrency, deep history, nested state, etc. It is coded in Java and hence runs itself on top of a JVM (and ultimately on a Dalvik VM). It is released for several platforms, depending on whether reflection capabilities are available like for J2SE APIs and Android APIs (`java.lang.reflect`) or missing like for J2ME. It can be imported in any Java-based project as a library (Jar file). No additional dependencies are required.

From a performance point of view, the apk file solely increases of 101 Kilo Bytes required for the library, and the memory footprint is small even in case of complex statemachine. PauWare engine worked fine for older devices running with J2ME and hence the additional layer introduced by the engine is now negligible when considering modern devices running with Android OS.

B. PauWare Model

The regular way to describe an UML state machine in PauWare consists simply to write raw Java code for instantiating states (`Statechart` PauWare Java class) and to build the structure of the state machine by combining the states and adding between them transitions. Another way consists in inflating the code from serialized formats: either defined with your favorite UML modeler (through a XMI file) or in SCXML (State Chart XML[3]) which is a W3C standard for defining state machines. As a consequence, either by writing or generating its code, a UML state machine is present within the application code (through the instances of PauWare dedicated classes representing states and their relationships). This state machine is semantically equivalent to a UML state machine modelized through any modeler but has here simply a Java representation.

A preview of raw Java code corresponding to the model depicted in Fig. 1 is given bellow:

```
//States definition
AbstractStatechart nominal, critical;
nominal = new Statechart("Nominal Energy Level");
critical = new Statechart("Critical Energy Level");
//Sets the initial state
nominal.inputState();
//Combination (mutually exclusive) of the
//states for building the state machine
Statechart_monitor machine = new
    Statechart_monitor(nominal.xor(critical),
    "Energy Assistant Behavior");
//Transitions definition
//Setup of a reflective call to alert() method
machine.fires(Intent.ACTION_BATTERY_LOW, nominal,
    critical, true, this, "alert", new Object[]
    {"Be careful now"});
...
```

Operations (i.e. method calls) can be attached to transitions or states. These operations are typically implementing the business logic or data management of the application. The implementation of these operations is at the developer responsibility but there is no more need for her/him to implement the behavioral part of the system as it is already specified through the state machine model. This way to build an application offers important gains in terms of development. First, there is no more need to implement the code dedicated to the behavior of the system as it is directly defined with the state machine model. Second, there is a good separation of concerns between the behavior and the business logic implementation making them easier to define and to maintain.

[1] The list can easily be found at `android-sdks\platforms\android-17\data\broadcast_actions.txt`

[2] It is a little bit confusing but what Android calls "actions" refer to our event's names in the statechart, where actions are hence calls to operations.

[3] `http://www.w3.org/TR/scxml/`

Once loaded in memory, the engine will perform "run-to-completion" cycles onto the model and in the strict respect of the UML 2 semantics. Such a cycle processes an event occurrence by triggering the right transitions and calling the required operations.

C. Android Connector

As the PauWare engine is a core library that works in an agnostic manner, it is not an out-of-the-box product. It is required to define a "connector" whose purpose is to ensure that the model's elements are tied to the context in which they are immersed. Notably, it is important to bind abstract events and actions described at the model-level with their platform-specific counterparts. Thus, the Android connector will sniff out the system events that occur and will bring them up to the engine for their processing. In the opposite way, the Android connector will ensure that concrete actions are enacted onto the device from the statechart under execution.

IV. Anatomy of the Android Connector

The realization of the Android connector is a projection onto the application component framework provided by Android and must adhere to the conventions thereof. Below are the key technical choices that have been made.

A. Message Notification

We simply focus here on the `alert("message")` action that will be mapped with the notification mechanism, which is an important part of Android UI. For recall, notifications are messages displayed to the user in the notification area that appears in the top bar on the device's screen. Other notification modes are supported like sound, LED pulse or vibration, but it goes without saying that these will be counterproductive in the case of a mobApp that aims to save energy. Each new notification replaces the former one in the notification area; they are not stacked. The code will be located in a private method called by reflection from the engine service (see Section IV-C).

```
private static final int uniqueId = 514054615;

private void alert(String msg) {
  NotificationCompat.Builder mBuilder =
           new NotificationCompat.Builder(this)
           .setSmallIcon(R.drawable.notification_icon)
           .setContentTitle(msg);
  NotificationManager mNotif =
       (NotificationManager)
       getSystemService(Context.NOTIFICATION_SERVICE);
  mNotif.notify(uniqueId, mBuilder.build());
}
```

B. Event Sniffer

The straightforward way to implement this part of the connector is to create a Broadcast Receiver component registered to listen to the seven native events mentioned in Sect. II-B and whose purpose is just to delegate them to the engine. Every seven event registration is done programmatically (see the next section) because some of them do not work when manually declared in the manifest. And further, this fosters the possibility to dynamically configure the receiver from an arbitrary statechart. This solution will only require a quick analysis phase when the model is loaded, to elicit the set of events actually used, and to register (or unregister) the receiver accordingly.

```
class EventSniffer extends BroadcastReceiver {

  @Override
  public void onReceive(Context arg0, Intent arg1) {
    Intent engine;
    engine = new Intent(arg0, WrappedEngine.class);
    engine.setAction(arg1.getAction());
    arg0.startService(engine);
  }
}
```

C. Engine Execution Task

At glance, the mobApp looks like a background task and henceforth does not require a plain UI (i.e. screens). That means that the PauWare engine has to be wrapped into a Service, which is an application component that performs longer-running operations. The corresponding code is sketched below.

```
public class WrappedEngine extends Service {

  private AbstractStatechart_monitor machine;

  private void installBroadcast() {
    EventSniffer s = new EventSniffer();
    registerReceiver(s, new
        IntentFilter(Intent.ACTION_SCREEN_ON));
    registerReceiver(s, new
        IntentFilter(Intent.ACTION_POWER_CONNECTED));
    //The same for the 5 other events
  }

  //Here goes the raw code of the model
  //Refer to Section III.B
  private void loadModelFromRawCode() { ... }

  //Here is the method invoked by reflection
  //Refer to Section IV.A
  private void alert(String msg) { ... }

  @Override
  public void onCreate() {
    installBroadcast();
    loadModelFromRawCode();
  }

  @Override
  public int onStartCommand(Intent intent, int
      flags, int startId) {
    machine.run_to_completion(intent.getAction());
    return Service.START_NOT_STICKY;
  }
}
```

The service is used here in unbounded mode (versus bounded mode) where the `startCommand()` method is responsible for launching a run-to-completion step. Indeed, it will be invoked every time a new event is caught by the receiver in the purpose to evolve the current active state of the state machine. Notice that the absence of a final state in Fig. 1 means that the model's execution never ends (until the end-user closes the app of course), which is the expected behavior for this kind of mobApp.

V. Related Work

There is a large body of work in the literature dealing with the productive usage of models (MDA-compliant or not) in order to develop software for mobile platforms [6], [1], [2], [12], [11], [8], [14]. Most of them are based on UML 2 models, notably Class Diagrams and Activity Diagrams. This is an unsurprising situation because the static model approach is not related to any technological frame and works as well for building desktop applications, mobile applications, Web applications and so on. Instead, the newest approach of dynamic model implies to be closer to the target platforms because the model ought to be directly executed on the latter. To our knowledge, there were no feedbacks for smart devices so far.

The most closely related works to ours can be found on the side of cross-platform mobile development techniques. Indeed, to deal with the technology independence, some of them like Cordova (formerly PhoneGap) or Rhodes, require an execution engine running on the target platform (e.g. Android) [9], whether embodied by the Web browser for the former or by a specific Ruby VM for the latter. But anyway, the engineer is expected to stay at the program-level, and does not rise at the model-level as in the approach presented in this paper.

VI. Conclusion

In this paper, we showed that development of mobile applications is not only mandatory programming but can also be about modeling, and thereby this is a radical paradigm shift. The proposed development approach is currently hybrid: some parts of the application will be programmed in a classical manner while others parts will rely on executed models. For instance, we have defined the behavior of an application with a state machine model whereas the associated business operations have to be programmed. Nevertheless, the Holy Grail consisting in suppressing all the programming parts is not so unrealistic. Indeed, the OMG proposes fUML[4] that enables to define, at the UML diagrams level, the equivalent of "code" for fully specifying for instance the methods of classes in class diagrams. Then, through our state machine example, it is easy to foresee that if the operations associated with states and transitions are defined at the UML level, we will get a complete application directly through models.

Behind the scene, executable statecharts pursue a more ambitious goal: ease software maintenance and adaptation, as well [3], [4]. Indeed, as reified as a model, the behavior of the application is not hard-coded into the installed apk. It is willing to see it just as an input (meta)data of the engine, communicable and exchangeable over the network in the SCXML format for example so that it is possible to download a new model on-demand. Doing so, one may have a completely new application without impelling any disruption to the end-user (typically uninstall/reinstall apk on her/his device).

Appendix
About the material used in this paper

The PauWare API and documentation thereof are downloadable at `http://www.pauware.com`. The complete sources of the Energy Assistant mobApp presented in the paper are downloadable through the "Quick Start" menu. Further more complex case studies are also available.

[4]Semantics of a Foundational Subset for Executable UML Models (fUML): `http://www.omg.org/spec/FUML/1.1/`

References

[1] F. Balagtas-Fernandez, M. Tafelmayer, and H. Hussmann. Mobia modeler: easing the creation process of mobile applications for non-technical users. In *Proceedings of the 15th international conference on Intelligent user interfaces*, IUI '10, pages 269–272. ACM, 2010.

[2] P. Braun and R. Eckhaus. Experiences on Model-Driven Software Development for Mobile Applications. In *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2008)*, pages 490–493. IEEE, 2008.

[3] E. Cariou, O. Le Goaer, and F. Barbier. Model Execution Adaptation? In *7th International Workshop on Models@run.time (MRT 2012) at MoDELS 2012*. ACM Digital Library, 2012.

[4] E. Cariou, O. Le Goaer, F. Barbier, and S. Pierre. Characterization of Adaptable Interpreted-DSML. In *European Conference on Modelling Foundations and Applications (ECMFA 2013)*, volume 7949 of *LNCS*, pages 37–53. Springer, 2013.

[5] B. Combemale, X. Crégut, and M. Pantel. A Design Pattern to Build Executable DSMLs and associated V&V tools. In *The 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*. IEEE, 2012.

[6] J. Dunkel and R. Bruns. Model-driven architecture for mobile applications. In W. Abramowicz, editor, *Business Information Systems*, volume 4439 of *Lecture Notes in Computer Science*, pages 464–477. Springer Berlin Heidelberg, 2007.

[7] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[8] F. A. Kraemer. Engineering android applications based on uml activities. In *the 14th International Conference on Model Driven Engineering Languages and Systems (MODELS 2011)*, pages 183–197. Springer, 2011.

[9] O. Le Goaer and S. Waltham. Yet Another DSL for Cross-platforms Mobile Development. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, GlobalDSL '13, pages 28–33. ACM, 2013.

[10] G. Lehmann, M. Blumendorf, F. Trollmann, and S. Albayrak. Meta-Modeling Runtime Models. In *Models@run.time Workshop at MoDELS 2010*, volume 6627 of *LNCS*. Springer, 2010.

[11] B.-K. Min, M. Ko, Y. Seo, S. Kuk, and H.-S. Kim. A uml metamodel for smart device application modeling based on windows phone 7 platform. In *TENCON 2011 - 2011 IEEE Region 10 Conference*, pages 201–205, Nov 2011.

[12] A. Parada and L. de Brisolara. A model driven approach for android applications development. In *the 2012 Brazilian Symposium on Computing Systems Engineering (SBESC 2012)*, pages 192–197, Nov 2012.

[13] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[14] Z. Wang. The study of smart phone development based on uml. In *the 2011 International Conference on Computer Science and Service System (CSSS 2011)*, pages 2791–2794, June 2011.