Scenario Based Programming for Mobile Applications

Anat Berkman - Chardon
Scenario Based Programming for Mobile Applications

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Anat Berkman - Chardon

Submitted to the Senate of the Technion — Israel Institute of Technology
Av 5775 Haifa August 2015
The research thesis was done under the supervision of Professor Eran Yahav in the Faculty of Computer Science, Technion and Professor David Harel, Dept. of Computer Science and Applied Mathematics, The Weizmann Institute of Science.
# Contents

Abstract

1 Introduction

2 The Language of LSC

3 The Play-in/Play-out approach & PlayGo

4 The Conceptual Flow

5 The Implementation

   5.1 The Architecture
   5.2 The GUI Framework
   5.3 Building a System Model
   5.4 Play-in & Play-out
   5.5 Run As Standalone

6 Tic-Tac-(Pho)-Toe - A Walk-Through Example

7 Related Work

8 Conclusions and Future Work

Hebrew Abstract
List of Figures

2.1 LSC sample - Speed dial ........................................ 6
3.1 PlayGo New Architecture .......................................... 9
3.2 Play-in and Play-out Examples .................................. 10
5.1 Overall Architecture with Android .............................. 13
5.2 Camera Play-In From The System Model ....................... 14
6.1 User Choose A Cell ................................................ 17
6.2 Switching players .................................................. 18
6.3 Win Horizontal First Line ......................................... 19
6.4 Disable click events after a cell was chosen .................. 19
6.5 Hint Buttons Behavior ............................................. 20
6.6 Wait And Changes Background To Grey ....................... 20
6.7 Swipe Over A Cell ................................................ 21
Abstract

We introduce a novel method for creating mobile applications, integrating the Android SDK into PlayGo, a scenario-based behavioral programming framework. The framework we implemented allows creating mobile applications simply by using a visual GUI editor, and then incrementally “playing in” scenarios that construct the application behavior. This allows the developer to focus on the application behavior and interface rather than the syntax and code.
Chapter 1

Introduction

The worldwide smartphone market is in constant growth, and the global inventory of mobile applications, as reflected in the popular application stores, is growing in staggering pace. In the recent decade there is a tremendous growth in the development of applications for mobile devices. This growth is enabled both by growing computational power and memory volume in these devices and by development tools that help developers create nice user interfaces and easily access the mobile components (e.g., camera, GPS, etc.) through well-defined API's. However, to develop a mobile application, one still needs to know “how to program”, i.e., to be able to write code in some conventional programming languages (e.g., Java). In [11] the first-listed author illustrated a vision in which engineers (and potentially end-users) could be freed from the burden of translating their requirements, usually specified as scenarios of the system to be developed, to system structure and code. The terms inter-object and intra-object behavior were used to indicate the two complementary manners in which the system can be viewed. The first captures the system behavior as a set of scenarios in which the systems objects, its users and external environment interact while achieving the system functionality. These scenarios may interact, interleave and maybe even contradict one another. The second captures the system behavior as a set of encapsulated objects’ behaviors, focusing on the objects interfaces and their inner reactions to external inputs and activations. It has been argued that inter-object specification reflects better the user intentions and is more intuitive for the process of requirements elicitation, while the intra-object behavior reflects better the system to be built and is more suitable for the transition from requirements to design. As a result of this paper, a novel approach for requirements elicitation and execution was published under the name “The play-in & play-out approach” together with a supporting tool - the “Play-Engine” [21]. Play-in allows the user to specify the behavioral requirements of systems using an intuitive mechanism. The user “plays” the GUI by clicking buttons or rotating knobs and specifies, using the same tools, the desired reactions of the system. As the scenarios are played in, a formal specification in the language of LSCs is automatically generated. After playing in (part of) the specification, the natural thing to do is to check if the current system behaves as
expected. Play-out is a complementary process to play-in, in which the user uses the same
techniques to manipulate the GUI but this time the scenarios are used to determine the
system behavior and the results are reflected in the GUI, thus allowing the user to actually
see the system in operation.

LSC is a visual language that extends classical message sequence charts (MSCs) \cite{4} with
liveness, denoted by elements temperatures (cold or hot) \cite{9}. LSCs can distinguish between
scenarios that must happen (universal) and scenarios that may happen (existential) in the
system. LSCs also distinguish between messages that must be received (hot) and messages
that may be received (cold). A condition can be cold, meaning that it may be true (if not,
the control moves gracefully out of the current block or chart) or hot, meaning that it must
be true (if not, the system aborts). A bit more detailed description of the language of LSCS
can be found in section \ref{sec:2}.

In a 2008 paper \cite{12}, the same author pointed to three major constraints that have faced
programmers from the early days of computing, and which in many ways still do: (i) the
need to explicitly produce a formal artifact, the program; (ii) the need to produce a second
such artifact, the requirements, written in a totally different formalism, and to then pit one
against the other; and (iii) the need to specify reactive behavior in a way that adheres to the
structure of the system an intra-object approach. That paper suggested that programmers
could be liberated from these, bringing the programming of computers closer to the way we
“program” others; i.e., the way we explain things to our children, employees, students, etc.,
in order for them to attempt to understand what we had in mind and advance accordingly.
The research carried out on this “liberating programming” dream has yielded a large body
of work on scenario-based programming, originating in the LSC language \cite{9,20,18}. It
allows GUI-based or natural-language-based \cite{10} play in of behavior scenarios, and is multi
modal, allowing constraints (for example, forbidden scenarios) to be part of the program.
The approach has been generalized and extended also to other languages including Java,
C++, Erlang, JavaScript and Blockly, and has been termed behavioral programming (BP)
\cite{22}. Research results cover, among others, run-time lookahead (smart play-out) \cite{16,15},
model-checking, compositional verification, synthesis, interactive analysis and detection of
unrealizable specification, an abstraction-refinement mechanism, automatic correction tools,
synchronization relaxation tools, and an SMT solver.

One of the main outcomes of the behavioral programming research effort was the re
placement of the Play-Engine with a new tool called PlayGo \cite{18}. PlayGo is eclipse-based
and in contrast to the Play-Engine, adheres to the UML standard and has an open architecture
that enables extension and integration with other tools. PlayGo also uses a compiler
rather than an interpreter, thus opening the possibility to execute the code on a remote
hardware.

In this paper we show how behavioral programming in the language of LSC, supported
by PlayGo, can make the development of mobile applications much easier and more intuitive,
hopefully making an important step towards a future where end users will be able to do
that by themselves.
The paper is organized as follows: In section 2 we briefly introduce the language of LSC and in section 3 we describe the basic concepts of the play-in / play-out approach. Section 4 describes the basic development flow of a mobile application and is followed by section 5 which describes the main aspects of the implementation. An example is given in section 6 and we conclude with related work and future research directions.

**Main Contributions**

- We provide a novel approach for creating mobile applications using a behavioral programming framework. We present a framework, which allows creating complete android mobile applications using visual tools only. We used PlayGo, a comprehensive tool for behavioral, scenario-based, programming framework.

- We extended PlayGo to communicate with a remote GUI over a TCP connection during development. This makes PlayGo extensible for multiple platforms, both local and remote.

- We created a new technique for automatically extracting a system model from an android application GUI.

- Our system automatically generates objects that act as adapters between an Android application GUI and PlayGo or local generated behavior code.
Chapter 2

The Language of LSC

Message sequence charts (MSC), also known as sequence diagrams, are widely used to describe behavioral aspects of systems, by describing scenarios thereof. However, MSC possess a rather weak partial order semantics that does not make it possible to capture many different types of behavioral requirements. Sequence diagrams can state what might possibly happen, but not what must occur, which conditions should hold but not what happens when they don’t, what can we expect from the system under development but not what is undesired or forbidden.

Live sequence charts were introduced in 1999 by Damm and Harel [9] as a visual scenario based language, enriched with liveness and modalities. LSC distinguish between scenarios that may happen in the system (existential) from those that must happen (universal). They can also specify messages that can be received (cold) and ones that must (hot) and distinguish conditions that may be true (otherwise the system gracefully exits from the current subchart or chart) from those that must be true (otherwise, the system aborts). In the following years the LSC language was extended with the notion of time [19, 17], allowing to specify timing constraints, and with the notions of symbolic messages, lifelines and unification [24] allowing to describe general scenarios referring to variables and classes rather than to specific values and objects.

Fig 2.1 shows an example LSC diagram that describes the process of speed dialing in a cellular phone. After the user dials a digit, the system, in response, should call the number stored under this digit shortcut and make a beep sound every two seconds. We use this example to illustrate the basic concepts of the LSC language.

The diagram consists of vertical lifelines representing objects and classes and horizontal elements denoting interactions between the lifelines. Time is assumed to progress from top to bottom, hence, a higher element on a lifeline precedes a lower one. The first two elements in the diagram are messages indicating actions taken by the user, clicking some digit and then clicking the call button. Both messages are cold (blue), thus, the scenario expects them to happen but will not cause a violation if they do not happen. They are also monitored (dashed), meaning that the execution mechanism shall wait for them to occur rather than
trigger their execution. The first message goes from the user to a lifeline that is indicated as *dynamic*. This means that any actual event with this signature (name, and optional parameters) originating from the user and ending at any of the digit buttons (which are instances of the digit class), will be unified with this message and will advance the LSC. The next element is a cold condition, denoted by a blue dashed hexagon. Conditions are evaluated as soon as possible after the previous event is executed. In this case, the condition requires that only one digit was clicked (it is assumed, and possibly defined in a separate LSC, that each digit clicked is added to the display). If this is the case, the chart will advance beyond it and if not, the chart will exit. If the condition evaluates to true, the chart reaches the next element which is a hot (red) executed (solid line) self message. This message indicates that the *memory* object should retrieve the stored number by calling its *retrieve* method with the proper parameter. Next, the chart reaches an assignment, denoted by a rectangle containing the variable name and the expression to be assigned to this variable. Assignments are internal to the chart (in contrast to the system’s state variables, which can be used in several charts) and, similarly to conditions, are executed as soon as possible after the previous event is executed. Assignments, as well as conditions, may be associated with multiple lifelines which are considered to be synchronized with the assignment. An assignment is *enabled* for execution only after all its synchronized lifelines have reached it and all the variables on the right hand side are bound. After the internal variable *number* is assigned the value of *memory* the chart reaches a *SYNC* element which is actually a shortcut for a cold, constantly true condition. Later, the *display* object displays the phone...
number by calling its show method with the proper parameter, number. Last, the chart reaches a loop, denoted by a subchart with a loop control mark on the top left indicating the number of required loop iterations or ‘*’ for unbounded loops. All participating lifelines are synchronized with the loop start and end. The loop exits after completing the indicated number of iterations or when reaching a false cold condition. At the beginning of the loop we check that the call object is not connected yet. Then, the variable time is assigned with the current time provided by the Clock object which is a predefined object capturing the passage of time in the application. Then the chart reaches a wait hot condition. A wait condition is evaluated continuously until it becomes true (practically, after every occurrence of an event or a clock tick). In this case, the condition is true after two seconds have elapsed from the assignment and then the phone beeps. This loop is executed repeatedly until a connection is established.
Chapter 3

The Play-in/Play-out approach & PlayGo

In [21] a new approach was described termed play-in and play-out for specifying and validating software requirements. While playing-in, the user describes the ‘end-user’ actions and the desired system response. The user uses a graphical interface of the system, and can play-in by clicking on buttons, flipping switches, rotating knobs, etc. The user specifies the desired system reaction in a similar way by setting values for displays or sending messages to (invoking methods of) objects. Internal objects (such that are not part of the GUI) can be changed using the system model diagram [2], which contains all the objects that the system is composed of.

Play-out is a complementary process to play-in, aimed to execute the specification so the system’s behavior can be tested by various stakeholders. During play-out, the user simply plays the GUI application as he/she would have done when executing a system model, or the final system implementation, but limiting himself to “end-user” and external environment actions only. Every execution of an operation is considered a step. Following a user action, the system executes a superstep – a sequence consisting of the steps that follow the user action as dictated by the universal charts. While playing-out, the current objects’ locations in each LSC are represented by a cut.

The play-in/play-out approach is supported by the PlayGo tool [18] - a Java based framework for playing in and playing out LSC specifications. PlayGo is implemented as a set of Eclipse plugins and is packaged as an Eclipse product. PlayGo also includes a scenarios to AspectJ compiler [13] and means for debugging the execution.

A PlayGo application consists of three main parts. The first is a GUI that in most cases serves as a mockup of the final application, but can also be the actual final application. The user can either work with a GUI developed specially for the application or use a GUI which is automatically generated by PlayGo, based on the system model. The second is a system model which consists of all the types, classes and objects in the system. Each class has its
own properties and methods and each object is an instance of a given class. The third part is the application behavior, given as a set of aspects automatically generated by PlayGo from the LSCs, using AspectJ compiler [13]. Having the behavior represented by Java code (rather than interpreting the LSCs in real-time as done in the PlayEngine) opens possibilities to execute the resulting system on remote hardware and not necessarily on the platform that runs PlayGo. PlayGo supports multiple execution strategies, including a strategy that uses model-checking based look-ahead, termed “smart play-out” [16] and synthesis strategy which is based on the technology of controller synthesis. Controller synthesis takes the LSC specification and computes a controller whose behavior is guaranteed to satisfy the requirements set by the specification (if such a controller exists). The computed controller is then used to guide the play-out mechanism [28, 14].

PlayGo was developed to interact with Java/Swing-based GUI applications but has not implemented a specific methodology, framework or interface for working with non-Java GUI applications. To be able to interact seamlessly with such applications, we extended PlayGo and implemented a framework and a set of interfaces that enable using such GUI applications, both for play-in and play-out. Fig. 3.1 outlines the main principles of the new architecture while Fig. 3.2.1 and 3.2.2 show the data flow between the various elements during play-in and play-out respectively.

At the core of the architecture lie two interfaces, IPlayable which abstracts elements that can be “played” (e.g., a GUI application) and IPlayGo which abstracts “behavioral engines” (e.g., the PlayGo tool itself and the behavior automatically generated by it). The behavior then uses IPlayable to communicate with playable elements and the GUI uses IPlayGo to communicate with its associated behavior. The interaction between the GUI and the behavior can be using native Java, or, in case we want to allow the GUI to be written in another language, can be executed using XML messages over a TCP/IP connection. To allow GUI developers to develop their GUI without going into the details of the communication protocol, we provide agents that implement the required interfaces and
can communicate with each other using Java calls or TCP communication and provide the appropriate interface (IPlayable or IPlayGo) to the GUI and the behavior.

During play-in, suppose that the user clicks a button. The button object informs the GUI Agent about the event, the GUI agent forwards the message to the PlayGo Agent over the communication channel (TCP/IP or a direct method invocation), and the last forwards the message to the PlayGo tool. If such a GUI event is triggered during play-out, the messages are forwarded to the generated behavior and not to the PlayGo tool, since we are now interested in the reaction of the system and not in building a new scenario. During play-out, the behavior may inform the GUI that a value of an object’s property was changed and it should be reflected in the GUI. In this case, the generated behavior sends a message through the PlayGo agent to the GUI agent and the last forwards it to the appropriate object.

Figure 3.2: Play-in and Play-out Examples
Chapter 4

The Conceptual Flow

Before going into the details of the implementation, we outline the main steps that compose the process of developing an Android application with PlayGo. Since we want to enjoy the advantages of playing in the specification and not only the benefits of the LSC language as a final artifact, the first step is to create the application GUI. In this step we use the popular Android Development Toolkit (ADT), which has a ‘What You See Is What You Get’ visual GUI editor. We then transform the GUI elements into playable elements so they can be used during play-in and play-out. This can be done, either by using pre-built components or by wrapping existing elements with suitable playable wrapper objects.

The next step is to create a system model, corresponding to the objects in the GUI and the frequently-used mobile components (e.g., camera, GPS, etc.). We do this automatically by scanning the ADT output and extracting all relevant information. The system model is then written to a file in PlayGo format. After creating the system model, the user plays in the application behavior. This is done using the ADT GUI and a TCP/IP protocol connecting PlayGo with the ADT emulator. The specification development interleaves sessions of play-in and play-out where requirements are incrementally added and tested. While playing-out, the user can use PlayGo debugging tools that allow tracking the execution visually in real time and examining the execution trace after it completes. Finally, when the application is ready, an executable APK file is generated in a click and is downloaded and installed on the mobile device.
Chapter 5

The Implementation

5.1 The Architecture

Our solution is based on the architectural principles described in Fig. 3.1 where the GUI application is now an Android application. During play-in, the Android application runs on the Android emulator with a GUI TCP Agent and the behavior runs as an LSC project implementing the PlayGo interface. The two communicate over a TCP/IP channel.

Figure 5.1.1 demonstrates an Android application with a text view (Text1) and a button (Button1) which are instances of the TextView and Button classes respectively. Each GUI element is wrapped with a suitable playable component that handles all its interaction with the playable GUI framework. The playable framework handles the interaction with the behaviour through the various communication agents. In this example, we have also added an invisible playable camera component to the GUI. When the user clicks on Button1 in the emulator, the Playable-Button handles this event and notifies the GUI Agent about this event. The GUI Agent transmits this message over TCP/IP to the Behavior Agent. If PlayGo is in play-in mode, the message is transferred to PlayGo and serves in the process of creating the specification. In play-out mode the message is transferred to the corresponding behavior object, defined in the system model and whose Java code was automatically generated by PlayGo, which can be located either in an LSC project in PlayGo, when the emulator is used, or on the smartphone itself when it runs as a stand-alone application.

5.2 The GUI Framework

While trying to devise an overall solution for the play-in / play-out processes we had to decide between two alternative concepts for implementing the GUI. The first was to develop a Java swing-based GUI template, in which users can place GUI elements. In this case, the play-in and play-out would be very similar to the way other desktop applications are developed with PlayGo, and at the end of the process, this GUI would have been automatically transformed to an Android application according to simple conversion rules. The
The main advantage of this concept is the possibility to easily develop, or to use already existing, Java playable components and to integrate them within our existing Java GUI framework. The second alternative was to use an existing Android emulator and to enhance it so that it can implement the IPlayable interface and thus support the play-in process. Although this option required to develop specific emulator-side code, we eventually chose it, because using a familiar editor keeps the process of creating an application simple and allows an authentic experience of the real GUI during play-in and play-out. We decided to work with the popular Android Development Toolkit (ADT) IDE provided by Google as an add-on to Eclipse, which is PlayGo’s native development environment.

As part of our framework, we provide a single generic Android application that serves as a template so that the developer only needs to create the GUI itself (which is automatically generated by the ADT WYSIWYG editor). An Android GUI is controlled by activities. Activities are objects that respond to events and modify the view accordingly. Our template application contains a single generic activity which implements the IPlayable interface and thus can receive notifications from PlayGo. At system startup, the generic activity uses reflection to inspect the GUI it is running with, wraps each GUI element with a playable wrapper object, and stores the object ID as its key. This allows the activity to redirect method calls from PlayGo to the target wrapper object. A wrapper object serves as an event handler for the wrapped GUI object and manipulates it in response to changes in the behavior object.

### 5.3 Building a System Model

After generating the GUI using ADT, we automatically generate a system model, which corresponds to the classes and objects participating in the Android application. The Android GUI editor is generating an XML file, which describes the GUI layout and the participating objects.
widgets. This file is parsed by our framework and each GUI object is mapped to an object in the system model with compatible properties, methods and a unique ID. E.g., properties of an Android button include displayed text, text size, width, color, visibility, etc. The operation of extracting a system model from the GUI is done in a single right-click by the user. Once generated, the system model can be used in PlayGo for play-in and play-out as if it was created manually inside the tool. We also support “GUI-less” objects, which are objects that are not shown on the GUI but still need to be represented in the system model. A set of predefined GUI-less objects are automatically inserted to the generated system model for every Android application. As an example, consider the smartphone’s camera. There are several operations that the camera provides and are automatically supported by our playable camera object, such as takePicture and getPicture and the boolean property status (indicating the success or failure of the last takePicture action). Using these methods, the user can use the camera without getting into any implementation details. Another “GUI-less” object provided by our framework is a general phone object, which can be used for capturing various general events. For example, this object provides a shake method which is invoked when the phone is physically shaken and can be monitored by an LSC to react respectively. There are, naturally, many other sensors and frequently-used objects and operations that can be supported, such as motion, GPS location, data storage and other media services (playing and recording video, etc.). To demonstrate the concept of seamlessly integrated functionality, our framework also supports a TextToSpeech “GUI-less” object, which allows synthesizing speech from text easily. TextToSpeech has one method speakOut, which receives a string as a parameter and reads it out loud.

![Figure 5.2: Camera Play-In From The System Model](image)

### 5.4 Play-in & Play-out

In “standard” play-in, the developer specifies user actions by actually executing them (e.g., clicking a button, writing text, moving a slider, etc.) and system reactions by right-clicking an object and specifying the desired result. Since there is no mouse in mobile applications, and hence no right-clicks, we use long click, as the natural alternative in the mobile domain, to open a system model menu, change properties and activate methods of other objects. We also added support for touchpad actions, such as swipe. Play-in through the GUI is not always possible since some objects may be invisible sometime during run-time (e.g., a calendar object that is shown upon request to set a date), and other objects are internal and are invisible all the time (e.g, camera). To enable playing in with invisible objects we provide
the play-in functionality through a tree view representation of the system model, which acts as a general-purpose GUI, in which all objects are visible. Using the system model view, the user can also open popup menus and specify physical behaviors that cannot be played-in directly through the GUI such as shaking the phone or flipping it upside down (see Fig 5.2).

5.5 Run As Standalone

Figure 5.1.2 describes the overall architecture of a standalone Android playable application. To run as a standalone application, the generated Java behavior and system model are added and compiled together with the Android GUI application. Here also, the architecture follows PlayGo’s basic architecture guidelines, where the difference is that now no TCP agents are required and the messages are sent directly to the local behavior as method invocations. The messages from the behavior to the GUI are also sent by direct method calls to the generic activity, which implements IPliable.
Chapter 6

Tic-Tac-(Pho)-Toe - A Walk-Through Example

Tic-Tac-Pho-Toe is an adaptation of the well-known tic-tac-toe game to the mobile technology. The winner of the game is the player who marks a row, a column or a diagonal with his or her marks, but instead of placing fixed marks (X & O), each player takes a photo and places it in the desired cell. The game is easier to play when each player uses a ‘selfie’ or harder when the pictures are similar and the players need to memorize their own marked cells. Players can get help in two ways: swiping a cell to uncover its owner’s identity or clicking a Hint T button to color all marked cells according to their owners for T seconds. A short video clip, demonstrating the process of developing this game with PlayGo, as well as playing it, is available at https://youtu.be/qITmuC3S-KE.

The full specification of this game consists of approximately 30 LSCs. In this section we briefly go through some of them to illustrate the nature of a mobile scenario based application.

Fig. 6.1 shows the basic move of a player clicking a cell. The first message is automatically generated by PlayGo, during play-in, when the user clicks a cell in the Android application running on the emulator. The lifeline label indicates the class that is represented by it, in this case ImageButton. Notice that this lifeline is dynamic, meaning that this LSC will be triggered upon a click on any of the cells and the lifeline will bind to the appropriate cell. After the user clicks a cell, we expect the camera to open. Since the camera object is not shown on the GUI, the user plays in this event from the system model as shown in Fig. 5.2. When the process of taking a photo with the camera is completed, the camera sets its status to true. Therefore we wait (i.e., monitor) for this event before we go on. After a photo is taken, we set the cell’s owner to be the current player and the cell’s image to the image taken by the camera.

The second scenario refers to the act of switching the current player at the end of each turn. Fig. 6.2 starts with monitoring the event setPlayer and ends with an alternative subchart that sets the text of the current player name to the appropriate player.
The third scenario refers to a win event, after a player successfully marked all the cells of the first row, as illustrated in Fig. 6.3. The marking events for the cells of the first row are monitored and have the same parameter, `playerName`, making sure that the three markings were indeed done by the same player. Notice that there is no order defined among the three events, as time advances in the vertical axis. After marking the first row, we expect the `controller` to handle the winning event. The `controller` is also an object not shown in the GUI, but as opposed to the `camera` which is part of the mobile device and is provided with the Android application generated from the GUI, this object is internal and specific to the game, and was actually added to the system model by the user. The user may add as many internal objects as he or she likes. Like other events related to hidden objects, the user plays-in this event using the system model tree view representation. Finally, the chart reaches an `EXIT` statement (a syntactic sugar for cold FALSE condition), which forces it to exit. The last message, monitoring the event `newGame`, causes a cold violation when a new game starts. If, for example, Player1 marked two cells, and then a new game starts, we want the LSC to abort, otherwise, marking the third cell is sufficient to cause Player1 to win.

The forth scenario requires that each cell can be chosen at most once. Fig. 6.4 starts with monitoring the event `setPlayer` and ends with a self executing message that sets the `clickable` property to false, which stops the cell from reacting to click events, thus disabling the option to be chosen again.

Figures 6.5 and 6.6 refer to clicking a `Hint T` button in order to color all marked cells according to their owners for T seconds. The scenario starts by monitoring (i.e. waiting
for a click event on a hint button. The application has three hint buttons, each of them having a different property value of `hintLength` (1, 3 and 5 seconds). Notice that this lifeline is dynamic, meaning that this LSC will be triggered upon a click on any of the hint buttons. After the user clicked the hint button, we expect each cell to change its background color accordingly. For this purpose we use a dynamic lifeline of type `universal all` (indicated by three dashed rectangle) which binds to all objects of a given class. To refer only to the cells that are occupied, we use a binding expression, which is dynamically evaluated. In Fig. 6.5.1 we refer only to the cells which their `owner` is `Player 1`, and in Fig. 6.5.2 we refer to the cells which are owned by `Player 2`. The LSC in Fig. 6.6 waits for `setBackground` to finish, using a symbolic parameter `blueOrRed`. Then, the chart reaches its last message, `clearBackgroundAfterMilliseconds`, which returns the background color to default after a specific time.

The last scenario refers to the event of swiping a cell. The first message monitors a `swipe` cell event. This lifeline is also dynamic and therefore this LSC will be triggered upon a swipe over any of the cells. In Fig. 6.7.1 next is an `alternative` subchart which checks the owner of the cell and changes the background color accordingly. Next, the variable `T` is assigned the current system time, the system waits for half a second (using the variable `T`) and the
background color changes back to the default. Since we want to limit the usage of hints, we reward a player with extra points, every time the other player uses help. Fig 6.7.2 refers to the score update. When a player swipes over a cell for hint, the alternative subchart checks who is the current player and updates the score of the other player accordingly.
(6.5.1) Hint Button Changes Background To Blue

(6.5.2) Hint Button Changes Background To Red

Figure 6.5: Hint Buttons Behavior

Figure 6.6: Wait And Changes Background To Grey
(6.7.1) Swipe Over And Change Background

(6.7.2) Swipe Over And Update Points

Figure 6.7: Swipe Over A Cell
Chapter 7

Related Work

Lately Google has announced the IntelliJ IDEA \[3\] based Android Studio \[1\] as the official platform for Android applications development. Porting our current implementation to Android Studio will require some technical changes - finding the right hooks for invoking our code, but as the XML format for describing the layout remained the same, and as most of our logic runs as part of the Android application itself, the majority of our work does not require changes.

There are multiple existing visual programming environments for mobile development. In 2013, IBM Research introduced NitroGen \[5, 6\], a mostly codeless, visual (drag and drop) platform for constructing form-based cross-platform mobile applications for enterprises. NitroGen allows developers to easily create cross platform applications that interact with backend services and databases. Typically, the generated applications contain interactive forms that allows users to add data to a remote database, and allows viewing data from the remote database with a flexible GUI.

Pong Designer \[25\] is an environment for developing 2D physics games through direct manipulation of object behaviors. In order to add behavior the user changes the objects’ initial velocity, and then runs the physics simulation from the current state. Meanwhile, the system displays the outcome in real time on the screen and records internal events, such as object collisions as well as user inputs (mostly multi touch-screen input). After the simulation stops, the user is able to edit the objects from the last simulation.

MIT App Inventor \[29\] originally developed by Google and now maintained by MIT, is a non commercial tool for building mobile apps for Android including a drag-and-drop GUI editor, and a visual block programming editor based on Blockly. Unlike Nitrogen which is limited to form-based data viewing applications of remote databases, App Inventor is much more flexible and allows creating generic applications. The development process with App Inventor is similar to the one we suggest: the developer starts with the design of the GUI using a web-based drag-and-drop GUI editor. Next, the developer specifies the behavior using the visual block editor by defining a set of event handler blocks. The editor allows the user to add a single event handler block for each one of the possible events of the application.
components (GUI and non GUI components). Each event handler is a visual block, and the user can drag action blocks into the event handling block. When the event occurs, the blocks inside the event handling block are invoked. The app that is being developed can be run on a physical device or an emulator, and changes to the GUI and logic are immediately reflected in it.

we believe that our approach which decouples the event handling logic of independent scenarios is more intuitive than the approach taken by App Inventor, where a single event handler contains all the logic for each event, since it allows the developer to focus on a single scenario at a time. Moreover, MIT App Inventor and NitroGen do not include a play-out phase, which allows the user to visually debug the system instead of test it. In our tool, while playing-out, PlayGo changes the LSC cuts accordingly, allowing the user to get deep understanding of the system behavior.

TouchDevelop [30] by Microsoft Research is a cloud-based integrated development environment (CIDE) which allows developing mobile applications on a mobile device. In addition to the IDE, TouchDevelop offers an application-store and a community of users and programmers. TouchDevelop runs entirely in-browser and provides an interactive environment for developing web-based applications. Applications are coded in the TouchDevelop visual scripting language, which can be tested in the browser or be deployed to a mobile device. Expert developers can view and change the code, as well as debug it using GROPG [27], a graphical on-phone debugger.

App Inventor and TouchDevelop allow easy development of mobile applications but they don’t provide an intuitive way to examine the program state as it is being executed. Although TouchDevelop includes a debugger, it is intended for expert users, and understanding the overall state of complicated programs can be hard. In our framework the user can examine the program state during the whole development cycle and can test parts of the program by choosing specific LSCs, and running from a specific state.

Several more approaches exist focusing more on model-driven development than on visual programming. Arctis [23] is based on UML activities, which are used as specification building blocks. The Arctis Editor is essentially a UML editor for activities with state machines as their external contracts. User interfaces are created with the Android SDK and are linked to UML Activities. Arctis, like our tool, provides a set of predefined functionalities such as audio handling, location management and various sensors integration to be used by the user. In [8] a new approach is presented for producing graphical user interfaces (GUIs) for business information system (BIS) prototypes for the Android platform. The new approach creates the GUI based on a model specified by UML diagrams and textual annotations. This approach generates a prototype Android application, which allows a conceptual navigation based on the relationships between the domain entities (as described in a UML class diagram). In our approach, the user can immediately see the resulting GUI right after its creation and can change and debug it continuously during play-in and play-out continuously. Mobichart [26] is a graphical notation which extends the objectchart [7] notation for modeling mobile computing applications, and allows modeling features like
object location, migration, hoarding, cloning, etc.
Chapter 8

Conclusions and Future Work

The contribution of this paper is in the new method for creating playable mobile applications. Given the advanced tools for creating graphical user interfaces for Android applications, our tool takes the development process one step further by allowing the user to focus on the behavior of the application rather than on the syntax and code. The user can simply play-in scenarios with a visual GUI and instantly view an LSC, which formally describes the behavior. The user can then play-out the behavior and visually debug the behavior by tracing the LSCs and the cuts changing over time. We believe that this intuitive manner of application development and debugging is not only faster and easier but may also lead to a deep understanding of the system reactions. The resulting application is packed, deployed and executed on Android devices and can be easily enhanced and modified incrementally by playing in more scenarios, without any need for code or intra-object modelling.

In this paper, we have limited our applications to include a single layout with no navigation, yet, we believe our framework can be easily extended to use play-in for supporting navigations and transition between layouts. Currently we support basic UI components (such as Buttons, TextView, Spinner, etc.), but our framework is extensible and can support any Android components. We believe that as users get familiar with our approach they can easily develop their own playable components and creates a shared open source rich library.
Bibliography


A system model associated with objects appearing in the GUI. The user can long press on an object through the emulator and the menu will open. The Play-out operation is also performed through the Android emulator. During the Play-out, the user can play with the GUI and check the system's response. The user can check the entire system or choose several scenarios. During the game with the GUI, the user can see the changes that occur in the LSC, which is reflected in the GUI (shown by the emulator).

At each stage, the user can return to the Play-in.

The last stage is the stage of creating an application. At this stage, we took the code created by PlayGo, which describes the system behavior, and combined it with the GUI that the user created for an Android application. Finally, we created an .apk file that can be run on Android phones.

The main contributions of this work are: presenting a new approach to creating applications for smartphones in a supporting environment for behavioral programming. Our system allows developing an application for smartphones through visual tools only and without writing a single line of code. Extending PlayGo so that it can communicate with GUIs remotely through an IP protocol. A further contribution is creating the mechanism that allows automatic exporting of the system model and importing it into PlayGo.
למשל, בין אירועים שחייבים להתרחש, לבין אירועים שייתכן ויתרחשו, הודעות שחייבות להtrerוח והודעות שייתכן ויתקבלו, בעיות שmanızיק לערך 'אמת' ובעיות שייתכן ויהיה משוער לערך 'אמת'. הרחבות נוספות הוצגו כגון ציון תרחישים שאסור שיתקימו במערכת, הוספת מוטיב הזמן למערכת ותוספת של אילוצי זמן, אפשרות להתייחס לתרחישים הכללים ללא ציון אובייקטים ספציפיים או ערכים מסוימים (למשל, כאשר אנו מ équipéים את משחק הזיכרון אנו יכולים להתייחס לתרחיש שהמשתמש הפך קלף ללא ציון קלף ספציפי, או למשל במשחק איקס עיגול ניתן להתייחס לתרחיש שהמשתמש בחר בתא כלשהו ללא ציון תאים ספציפיים).

בעבודה זו הראינו איך אפיון המערכת דרך LSC, נתמך ע"י PlayGo, יכולה לשפר את תהליך הפיתוח של אפליקציות לטלפונים חכמים ולהפוך אותו לקל יותר, אינטואיטיבי יותר ובתקווה גם להוביל לעתיד בו כל משתמש, לאו דווקא מתכנת מקצועי, יוכל לבצע זאת בעצמו.

תהליך בניית האפליקציה מורכב מהשלבים הבאים:طلب פיתוח GUI, תיאור המערכת באמצעות Play-in, בדיקת המערכת באמצעות Play-out, ולבסוף יצירת אפליקציה עצמאית وهيжить ב spatvre רחבי. בעבודה זו בחרנו להתמקד בייצור אפליקציות למערכת ההפעלה Android (מערכת הפעלה לטלפונים חכמים שפותחה על ידי Google).

כחלק מהמחקר התלבטנו רבות איך לבצע את שלב ה Play-in וה Play-out ולבסוף החלטנו להשתמש באמולטורים אנדרואיד. PlayGo תומך בעבודה עם ממשק java המבוסס Swing, שпроיקט שלanas אוכלוסיון מהעם עליה בראכטריסות שולש זה_TRANS. בעיות שنزניק לערך 'אמת' ובעיות שייתכן ויהיה משוער לערך 'אמת'

בעבודה זו הראינו איך אפיון המערכת דרך LSC, נתמך ע"י PlayGo, יכולה לשפר את תהליך הפיתוח של אפליקציות לטלפונים חכמים ולהפוך אותו לקל יותר, אינטואיטיבי יותר ובתקווה גם להוביל לעתיד בו כל משתמש, לאו דווקא מתכנת מקצועי, יוכל לבצע זאת בעצמו.

תהליך בניית האפליקציה מורכב מהשלבים הבאים: יצירת GUI, אפיון המערכת באמצעות Play-in, בדיקת המערכת באמצעות Play-out, ולבסוף יצירת אפליקציה עצמאית która אחרים ב PlayGo. בעבודה זו בחרנו להתמקד בייצור אפליקציות למערכת ההפעלה Android (מערכת הפעלה לטלפונים חכמים שפותחה על ידי Google).

כחלק מהמחקר התלבטנו רבות איך לבצע את שלב ה Play-in וה Play-out ולבסוף החלטנו להשתמש באמולטורים אנדרואיד. PlayGo תומך בעבודה עם ממשק java המבוסס Swing, שпроיקט שלanas אוכלוסיון מהעם עליה בראכטריסות שולש זה_TRANS. בעיות שنزניק לערך 'אמת' ובעיות שייתכן ויהיה משוער לערך 'אמת'
The smartphone market worldwide is experiencing constant growth, driven by the proliferation of applications for smartphones, as reflected in the rising number of applications offered in popular app stores. Over the past decade, there has been rapid growth in the development of applications for smartphones. This growth is facilitated, and even accelerated, due to the availability of development tools that simplify the development process. Many tools allow developers to create user-friendly interfaces, and even provide easy access to various components of the smartphone (such as the camera, GPS, etc.). Despite all this, one must develop a leading application, and the developer must know "how to code," not just "how to write code" in a programming language (such as Java).

In 2001, Professor David Harel presented a vision according to which developers, and hopefully also users, could be released from the world of translating specifications (where a detailed description of the system is presented as events) to a system and code. In the same article, two complementary approaches, "inter-object" and "intra-object," were presented, which can be used to describe the system. The "inter-object" approach describes the behavior of the system as a set of events that describe the interaction between different objects in the system (the system, the user, and objects representing the environment).

For example: an event "if the temperature in the classroom exceeds 27 degrees, the air conditioner should start cooling the classroom." The events can be combined with each other and can even conflict with each other. In the second approach, the "intra-object" system is described by a group of objects and a comprehensive description of the behavior required for each object (a part of which is seen in this approach as a "complete story related to the object.") It is claimed that the "inter-object" approach is closer to the description and specification of the system, while the "intra-object" approach is closer to the system's structure.

In the same vision presented, two new methods for describing and creating the system were proposed: \textit{in-}Play and \textit{out-}Play. The \textit{Play}Go engine, initially called \textit{Engine-}, and then renamed to \textit{PlayGo} (originally called \textit{Engine-}), was presented as a tool supporting the use of these methods, named \textit{PlayGo} (initially called \textit{Engine-}). The \textit{in-}Play phase is the phase of describing the system behavior in which the user describes the events through "playing" with the system (or with a \textit{mockup} of the system) in an intuitive way: for example, pressing buttons, rotating the camera, etc. In the same way, the user describes the desired behavior of the system (such as changing text box, operating the air conditioner at a certain temperature, etc.). During the system description process, \textit{PlayGo} generates \textit{LSC} diagrams for the events described by the user. The \textit{out-}Play phase allows intuitive checking of the system and its responsiveness. During the \textit{out-}Play, the user "plays with the system" in the same way as he described the system, and the events described in the \textit{in-}Play phase determine the system's behavior in the interface. In this way, the user can check the system behavior, and in the case of need, change the system behavior and/or add events.

In the year 99, an extended version of the \textit{LSC} language was published by Damm and Harel, which was named \textit{Live-LSC} Sequence Chart.
המחקר נסע בטכניון מחשב, במנהלה, פרופ' ערן יהב בפקולטה למדעי המחשב, הטכניון, פרופ' דוד הראל בפקולטה למתמטיקה ומדעי המחשב, מכון ויצמן.
תכנות מבוסס תרחישים לแอפליקציות מובייל

חסיב על מחקר

לשם מילוי חלקי של הדרישות לקבלת תואר
מגיסטר למדעים במדעי המחשב

ענת ברקמן-חרדון

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
אב התשע"ה  חיפה  אוגוסט 2015

הון למטט הנטוונות – מכון טכנולוגי לישראל
אבר הותש"ה  ייפה  אוגוסט 2015
תכנות.Tabows תרחישים לแอפליקציות מובייל

ענת ברקמן-חרדון