Scenario-Based Programming for Mobile Applications

Anat Berkman-Chardon
Technion
Haifa, Israel

David Harel
Weizmann Institute of Science
Rehovot, Israel

Yaarit Goel
Weizmann Institute of Science
Rehovot, Israel

Rami Marelly
Weizmann Institute of Science
Rehovot, Israel

Smadar Szekely
Weizmann Institute of Science
Rehovot, Israel

Guy Weiss
Weizmann Institute of Science
Rehovot, Israel

ABSTRACT

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

Categories and Subject Descriptors

D.1.7 [**Software**]: Programming Techniques—*Visual Programming*

General Terms

Design, Languages

Keywords

Behavioral Programming, PlayGo, Modeling, Mobile Development, Android

1. INTRODUCTION

The worldwide smartphone market is in constant growth, as is the global inventory of mobile applications. In the recent decade, there has been a tremendous growth in the development of applications for mobile devices. This is enabled both by the increased computational power and memory of these devices and by new development tools that help developers create nice user interfaces that 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 language (e.g., Java).

In [9] the author described 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, into system structure and code. The terms *inter-object* and *intra-object* behavior were used to indicate the two complementary manners in which system behavior can be viewed. The first captures behavior as a set of multi-modal scenarios describing how the system's objects, its users and external environment interact to achieve some desired functionality. These scenarios may interact and interleave, and they can specify mandatory, possible and forbidden behavior. The second approach — the more classical one — captures behavior as a set of behaviors of the system's encapsulated objects, focusing on their interfaces and their inner reactions to external inputs and activations. It has been argued that inter-object specification better reflects user intentions and is more intuitive for the process of requirements elicitation, while intra-object behavior reflects the system to be built and is more suitable for the transition from requirements to design. As a result of that paper, a novel approach for requirements elicitation and execution was developed, titled "the play-in&play-out approach" together with a supporting tool — the "Play-Engine" [19]. Recently, the Play-Engine has been replaced with a new tool called PlayGo [20].

Play-in allows the user to specify behavioral requirements by working with a GUI mock-up of the system. 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 *live sequence charts* (LSC; see [30]) 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, and the results are reflected therein, thus allowing the user to see the system in operation.

LSC is a visual language that extends classical message sequence charts (MSCs) [5] with modalities and liveness [30]. LSCs can distinguish between scenarios that must happen (universal) and scenarios that may happen (existential) in the system and 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 more detailed description of the language can be found in section 3.

In this paper we show how specifying behavior in the language of LSC using play-in and play-out, can render the development of mobile applications much easier and more intuitive. We believe that this constitutes an important step towards a future where end users are able to develop even complex applications for mobile devices on their own.

The paper is organized as follows: Section 2 describes the basic development flow of a mobile application using our method. In Section 3 we briefly introduce the language of LSC, while Section 4 describes the basic concepts of the play-in&play-out approach. Section 5 describes the main aspects of our implementation, and is followed by a detailed example in Section 6. We conclude with related work and future research directions.

2. THE CONCEPTUAL FLOW

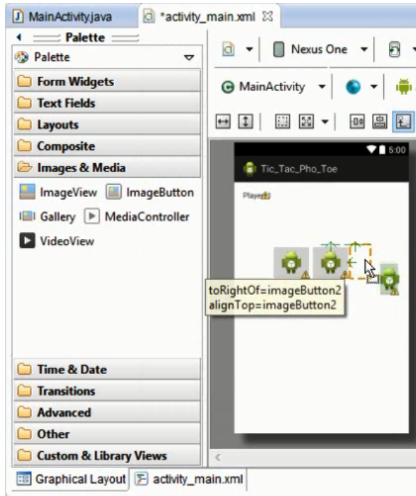

Figure 1: GUI creation

Figure 2: Final GUI

We now illustrate the conceptual flow of developing an Android application using our framework, by walking through a Tic-Tac-Toe example. We have modified the game so that it makes use of smartphone features, and call it "Tic-Tac-Photo". This is a Tic-Tac-Toe game, but instead of placing fixed marks on the board (X & O), each player takes a photo and places it in (i.e., marks) the desired cell. The game is easier to play when players use "selfies" and harder when the pictures are similar and the players have to memorize their own marked cells. The winner is the first player to mark a row, a column or a diagonal with his or her photos, as in classical Tic-Tac-Toe.

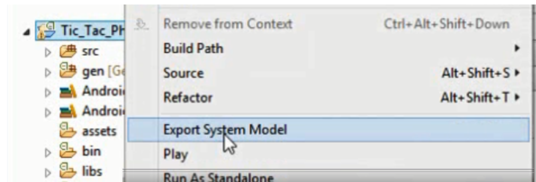

Figure 3: Extract system model

The first step in building the application is to create a GUI. We use the popular Android Development Toolkit (ADT), which has a 'What You See Is What You Get' visual GUI editor. Fig 1 describes the creation of our GUI. The user drags and drops a text view for displaying the current player, nine imageButtons representing nine cells, and a text view for displaying the winner's name. The resulting GUI is shown in Fig 2.

The next step is to generate the PlayGo view of the system's structure. This is done by right-clicking on the project and selecting "Export System Model" (See Fig 3). As a result, a *system model* containing the GUI classes and objects and frequently-used mobile components (e.g., camera, GPS, etc.) is automatically generated by PlayGo.

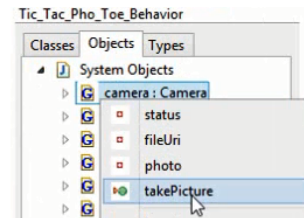

Figure 4: Camera play-in from system model

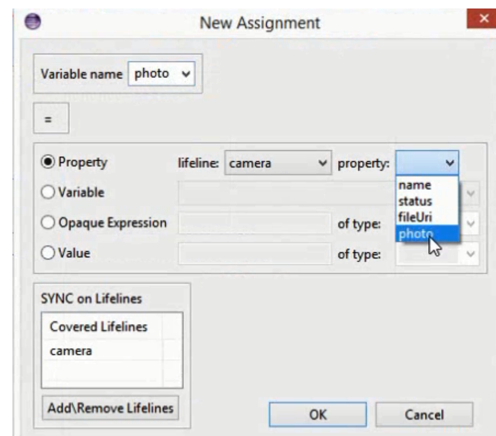

Figure 5: The assignment dialog

The user can now start playing in desired behavior via the GUI in the Android emulator. The first scenario we want is the ability to click a cell and take a photo. We want the photo taken to be displayed and the cell to "remember" which player picked it. So, the user clicks on a cell to indicate his/her selection and then right-clicks on the camera (from the system-model in PlayGo, See Fig. 4) and picks

"takePicture". The camera is then enabled and the user takes a photo. In order to store the photo the user right-clicks the camera and stores its photo property under the variable name *photo*, see Fig 5. Next, the user long-clicks the selected cell, and sets its source image to the value of the *photo*. After specifying this behavior, the emulator displays the photo in the selected cell. The user then similarly sets the property *player* of the selected cell to the current player, as displayed in the text view.

During play-in, while the user is specifying the behavior from within the Android emulator, PlayGo constructs the corresponding LSCs, thus allowing the user to view the visual code of the specified behavior and, if necessary, to change it. Fig 6 shows the generated LSC. At any point during the playing in of scenarios, the user can play out the behavior specified so far, thus testing the system under various conditions and examining its reactions.

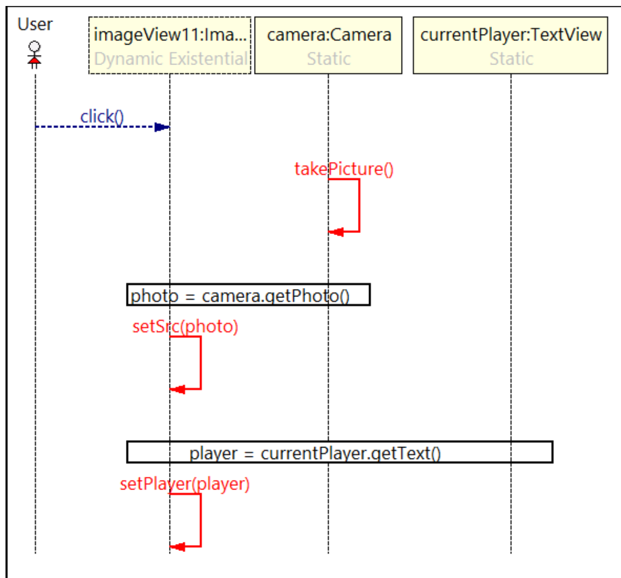

Figure 6: Camera use

During play-out the user now notices that the system does not handle switching players after each turn. Adding this scenario is simple. The system should wait for the event of a player becoming associated with a cell and then switch players. This is done as follows: the user long-clicks on a cell, selects its *setPlayer* method and marks it as monitored (i.e., an event that the system looks out for, but the event does not drive its execution). The user then long-clicks on the current player text view, and adds a simple if-then-else case that switches between the two players (see Fig 7).

The final step is to specify the winning scenarios. The user can describe each winning scenario separately, and thus add or remove winning cases incrementally in a flexible manner. For example, specifying winning by marking a full row is done by long-clicking each of the three cells and monitoring their *setPlayer* events, constraining them to use the same value (which is to say the same player name). If this scenario occurs, the winning text view is made visible and set to contain the winning player's name; see Fig 8.

As can be gleaned from this short section, development interleaves sessions of play-in and play-out where require-

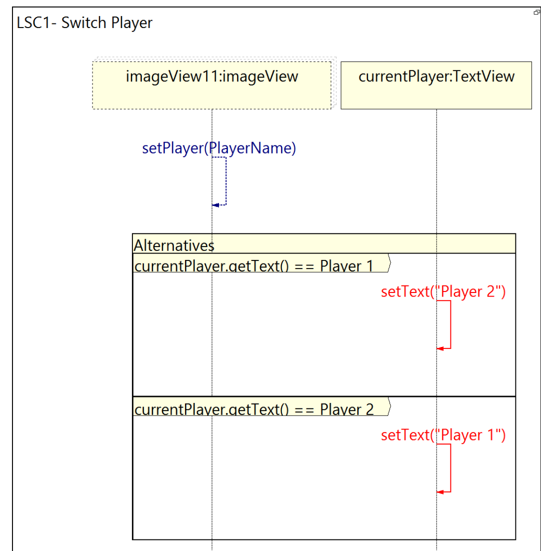

Figure 7: Switching players

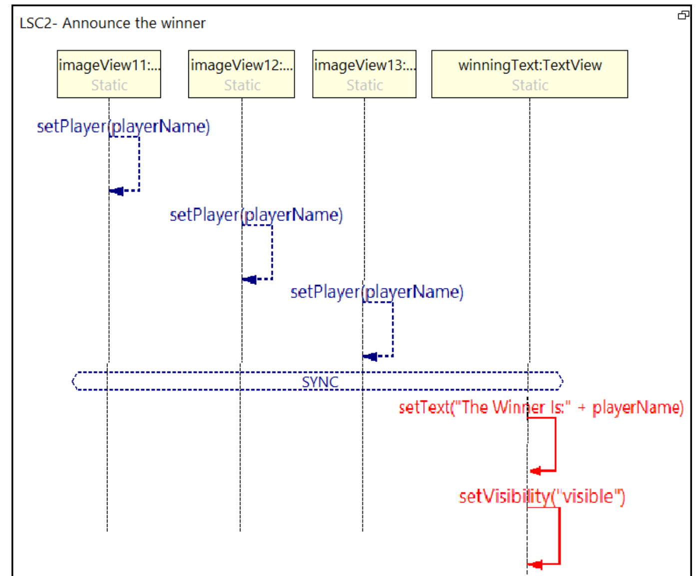

Figure 8: Announce winnner

ments and desired behaviors are incrementally added and tested. While playing-out, the user can use PlayGo's debugging tools, which allow tracking the execution visually in real time, and examine the execution trace after it completes. In addition, of course, since one can apply powerful analysis and verification tools to the PlayGo specification, for deeper insights into system behavior and performance. This, however, is beyond the scope of the paper; see for example, [16]. Finally, once all the scenarios have been specified and analyzed, and the application is ready for use, an executable APK file is generated, and can be downloaded and installed on the mobile device with a simple single menu click.

A short video clip, demonstrating the process of developing the Tic-Tac-Photo game with PlayGo, as well as playing it, is available at `https://youtu.be/qITmuC3S-KE`.

3. THE LSC LANGUAGE

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; they can express which conditions should hold, but not what happens when they don't; and one can specify what we can expect from the system under development but not what is undesired or forbidden.

To remedy these deficiencies, *live sequence charts* were introduced in 1999 by Damm and Harel [30] as a visual scenario based language, enriched with liveness and modalities. LSCs can distinguish 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 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 years that followed, the LSC language was extended in many ways, among which is a notion of time [14, 17], allowing one to specify timing constraints, and the ability to use symbolic messages and lifelines along with a notion of unification [29], which allows one to describe general scenarios referring to classes and variables rather than to specific objects and values.

Fig 9 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 in the corresponding shortcut and sound a beep 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, so that 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), meaning that the scenario expects them to happen but will not cause a violation if they do not . They are also monitored (dashed), meaning that the execution mechanism will wait for them to occur but will not 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 preceding 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 indeed 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*.

Finally, the chart reaches a loop, denoted by a subchart with a loop control mark on the top left corner, 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 the current time provided by *Clock* — 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 (more specifically, after each occurrence of an event or a clock tick). In this case, the condition becomes true after two seconds have elapsed from the assignment, and then the phone beeps. This loop is executed repeatedly until a connection is established.

4. PLAY-IN, PLAY-OUT AND THE PLAYGO TOOL

In [19], two complementary methods were described for specifying and validating software requirements termed, respectively, *play-in* and *play-out*. In fact, they can be thought of as methods for programing and executing LSCs. While playing-in, the user describes the 'end-user' actions and the desired system response, using a graphical interface of the system. This is done 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, often by first right-clicking the relevant GUI object. Internal objects that are not part of the GUI can be changed using the system model diagram [3], which contains all the system's objects.

Play-out is a complementary process to play-in, aimed to execute the specification or program (e.g., for testing its behavior 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 him/herself 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 situation is repre-

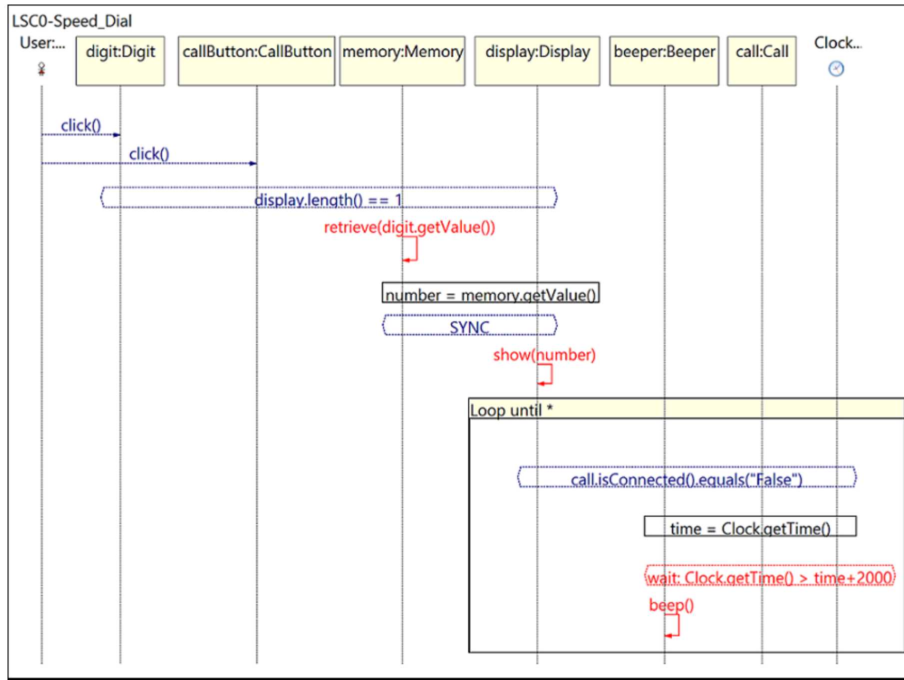

Figure 9: LSC sample: speed dial

sented graphically by a cut in each LSC, which is a roughly horizontal line that indicates where the execution is right now for each lifeline.

The play-in/play-out approach is supported by the PlayGo tool [20], which is a Java based framework for programming and analyzing LSC specifications. PlayGo is implemented as a set of Eclipse plugins and is packaged as an Eclipse product. It also adheres to the UML standard and has an open architecture that enables extension and integration with other tools. PlayGo includes a scenario-to-AspectJ compiler [11] and means for debugging the execution. Thus, in contrast to the original Play-Engine [19], which interpreted the LSC in runtime, PlayGo uses a compiler, thus opening the possibility to execute the code on remote hardware.

A PlayGo application consists of three main parts. The first is a GUI, which in most cases serves as a mock-up of the final application, but which can also be the actual final application. The user can work with a GUI developed specially for the application or use a GUI that is automatically generated by PlayGo, based on the system model. The second component of PlayGo is the aforementioned 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's behavior, given as a set of aspects automatically generated by PlayGo's compiler from the LSCs [11].

PlayGo supports multiple execution strategies that can be applied during play-out. Besides the 'naïve' method, which chooses the next event arbitrarily from all the possible enabled ones, these include a strategy that uses model-checking based look-ahead, termed "smart play-out" [16] and a strategy that is based on controller synthesis. The controller synthesis technique computes a controller from the LSC specification, 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 [13, 25].

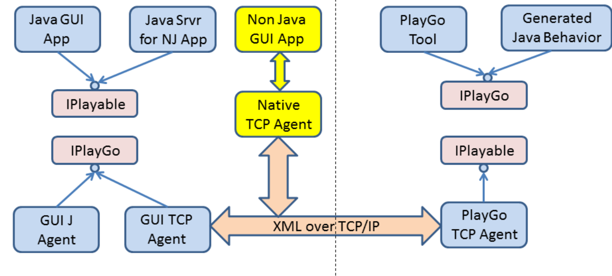

Figure 10: PlayGo new architecture

PlayGo was originally developed to interact with Java/Swing-based GUI applications, but does not employ a specific methodology, framework or interface for working with non-Java GUIs. To be able to interact seamlessly with such applications, we have now extended PlayGo and implemented a framework and a set of interfaces to support this, both for play-in and play-out. Fig. 10 outlines the main principles of the new PlayGo architecture, while Fig. 11 and 12 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 carried out using native Java, or, if we want to allow the GUI to be written in another language, it can be executed using XML messages over a TCP/IP connection. To allow GUI developers to develop their GUI without getting 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 they provide the appropriate interface (IPlayable or IPlayGo) to the GUI and the behavior.

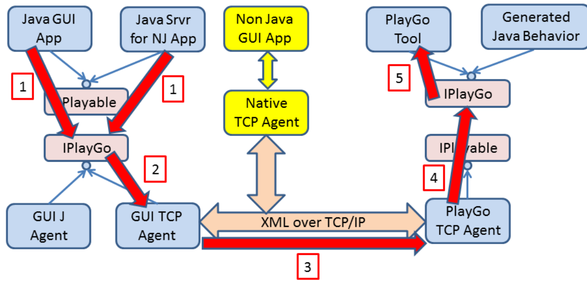

Figure 11: Play-in example

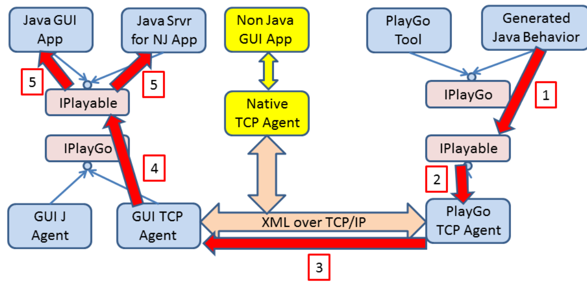

Figure 12: Play-out example

During play-in, suppose that the user clicks a button on the GUI. 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 agent forwards the message to the PlayGo tool. In contrast, if such a GUI event is triggered during play-out, the message is 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, which should be reflected in the GUI. In this case, the generated behavior sends a message through the PlayGo agent to the GUI agent, who forwards it to the appropriate object.

5. THE IMPLEMENTATION

5.1 The architecture

Our system is based on the architectural principles described in Fig. 10, 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.

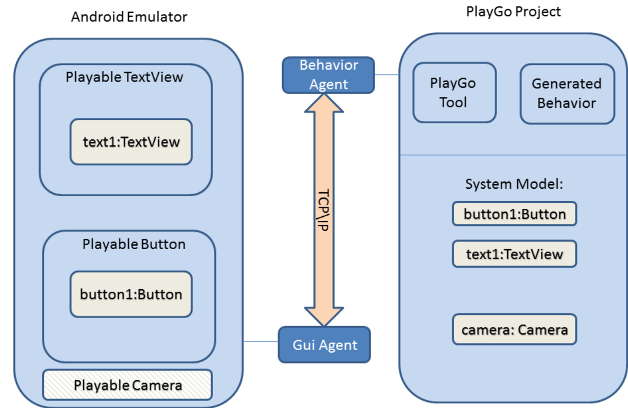

Figure 13: Android application with PlayGo architecture example

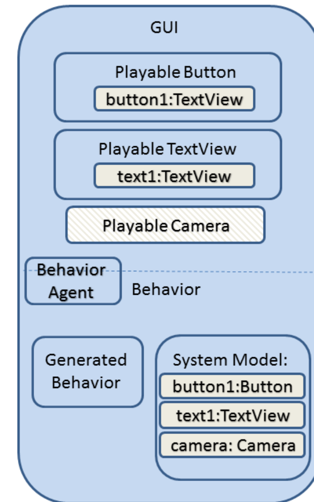

Figure 14: Standalone play-in-able Android application

Figure 13 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 endowed with a suitable playable component that handles all its interaction with the playable GUI framework. The playable framework handles the interaction with the behavior through the various communication agents. In this example, we have also added an invisible

playable camera component to the system model. When the user clicks on *Button1* in the emulator, the *PlayableButton* handles this event and notifies the GUI Agent about it. 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, which is defined in the system model, and whose Java code was automatically generated by PlayGo. It can be located either in the 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

When 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, 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 into an Android application according to simple conversion rules. The main advantage of this method 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 other alternative was to use an existing Android emulator and to enhance it so that it is able to implement the *IPlayable* interface and thus support the play-in process. Although this option required developing specific emulator-side code, we eventually chose it, because a familiar editor keeps the process of creating an application simple and allows a more authentic experience of the real GUI during play-in and play-out. We decided to work with the popular Android Development Toolkit (ADT) IDE [1] 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 create the GUI itself (which is auto-generated by the ADT WYSIWYG editor). An Android GUI is controlled by activities. Our template application contains a single generic Android activity that implements the IPlayable interface and can thus receive notifications from PlayGo. At system startup, the generic activity uses reflection to inspect the GUI with which it is running, 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, a single right-click by the user causes automatic generation of a system model, which captures the classes and objects participating in the Android application. The system model is written to a file in PlayGo format. The Android GUI editor generates an XML file, which describes the GUI layout and the participating 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. For example, properties of an Android button would include displayed text, text size, width, color, visibility, etc.

Once generated, the system model can be used in PlayGo for play-in and play-out as if it were created manually inside the tool. We also support so-called GUI-less objects, which are not shown on the actual GUI but still need to be represented in the system model. A set of predefined GUI-less objects is automatically inserted into 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 which are automatically supported by our playable camera object, such as *takePicture* and *getPicture*, as well as 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 the general *phone* object, which can be used to capture various general events. For example, it provides a *shake* method that is invoked when shaking the phone physically and can be monitored by an LSC to react respectively. Obviously, there are 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.

5.4 Play-in and play-out

Before starting the actual programming, we automatically 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 automatically wrapping existing elements with suitable playable wrapper objects.

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 the system's reactions by right-clicking an object and selecting the desired result. Since there is no mouse in mobile applications, and hence no right-clicks, we use long-click as the natural alternative, to open a system model menu, change properties and activate methods of other objects. We have also added support for touchpad actions, such as swipe.

Play-in through the GUI is not always possible since in a run some objects may be invisible at times (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., the *camera*). To enable playing in with such invisible objects we support the play-in functionality also 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.

5.5 Run as standalone

Figure 14 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; the difference is that here no TCP agents are required and messages are sent directly to the local behavior as method invocations (using the local Behavior Agent). The messages from the behavior to the GUI are also sent by direct method calls to the generic activity, which implements IPlayable.

6. TIC-TAC-PHOTO: A WALK-THROUGH EXAMPLE

In this section we extend the Tic-Tac-Photo game described in section 2 with several features. In the extended game, players can get help about cell owners 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.

The full specification of this game consists of approximately thirty LSCs. In this section we discuss some of them to illustrate the nature of a mobile scenario-based application.

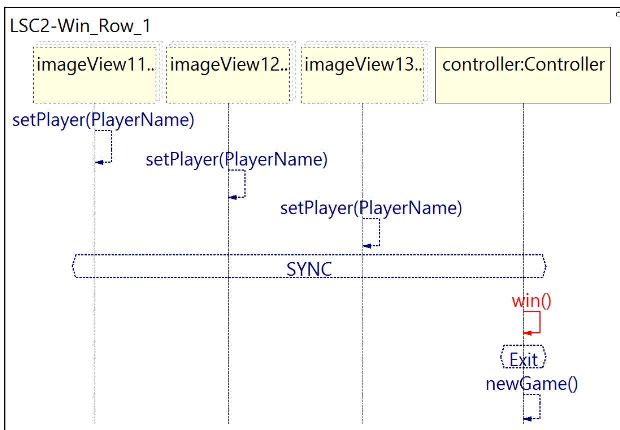

Figure 15: Win horizontal first line

Let us first revisit the winning scenario, after a player successfully marks all the cells of the first row, as illustrated in Fig. 15. The marking events for the cells of the first row are monitored, and have the same parameter, *playerName*, thus making sure that the three markings were indeed made by the same player. Notice that there is no order defined among the three events, as time advances along the vertical axis of each lifeline separately. After marking the first row, we expect the *controller* to handle the winning event. In contrast to the *camera*, which is part of the mobile device and is provided with the Android application generated from the GUI, the *controller* object is GUI-less and is internal and specific to the game; it was actually added to the system model by the user/developer, who 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 (syntactic sugar for a 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.

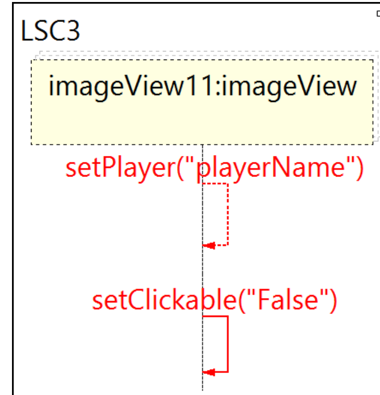

Figure 16: Disable click events after a cell was chosen

The second scenario prescribes that each cell be chosen at most once. Fig. 16 starts with monitoring the event *setPlayer* and ends with a self-executing message that sets the *clickable* property to *false*. This stops the cell from reacting to click events, thus disabling the option of being chosen again.

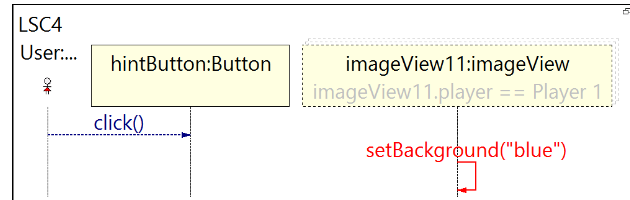

Figure 17: Hint button changes background to blue

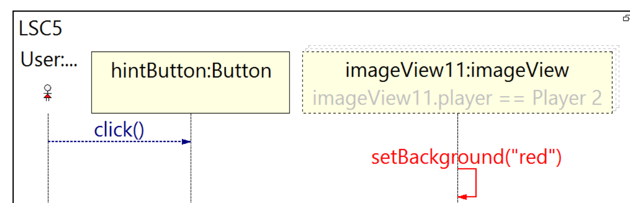

Figure 18: Hint button changes background to red

Figures 17, 18 and 19 refer to clicking a *Hint T* button in order to show, for a duration of *T* seconds, all cells marked according to their owners. 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, respectively). Notice that this lifeline is dynamic, meaning that the LSC will be triggered upon a click of any of the hint buttons. After the user clicks 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 a dashed rectangle) which binds to all objects

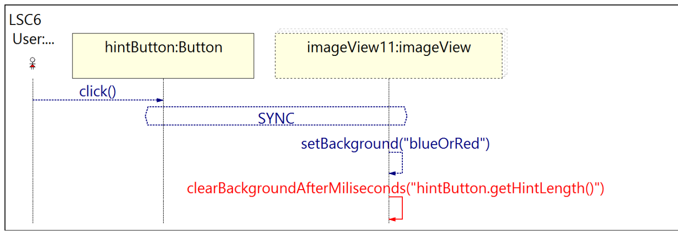

Figure 19: Hint button waits and changes background to grey

of a given class. To refer only to the cells that are occupied, we use a binding expression, which is dynamically evaluated. In Fig. 17, for example, we refer only to the cells whose *owner* is *Player 1*, and in Fig. 18 we refer to the cells owned by *Player 2*. The LSC in Fig. 19 waits for *setBackground* to finish, using a symbolic parameter *blueOrRed*. The chart then reaches its last message, *clearBackgroundAfterMiliSeconds*, which returns the background color to default after a specific time.

The final scenario we would like to describe refers to the swiping of a cell. The first message in Fig. 20 monitors a *swipe* cell event. This lifeline is also dynamic and hence the LSC will be triggered upon a swipe over any of the cells. Following this, there 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 requests a hint. Fig 21 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.

7. RELATED WORK

Google has recently announced the IntelliJ IDEA based Android Studio as the official platform for Android applications development; see [2, 4]. Porting our current implementation to Android Studio will require some technical changes, such as finding the right "hooks" for invoking our code. However, since the XML format for describing the layout remains the same, and since 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, a mostly codeless, visual (drag and drop) platform for constructing form-based cross-platform mobile applications for enterprises; see [6, 7]. NitroGen allows developers to easily create cross platform applications that interact with back-end services and databases. Typically, the generated applications contain interactive forms that allows users to add data to a remote database and to view data from the remote database with a flexible GUI.

Pong Designer [23] 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 out-

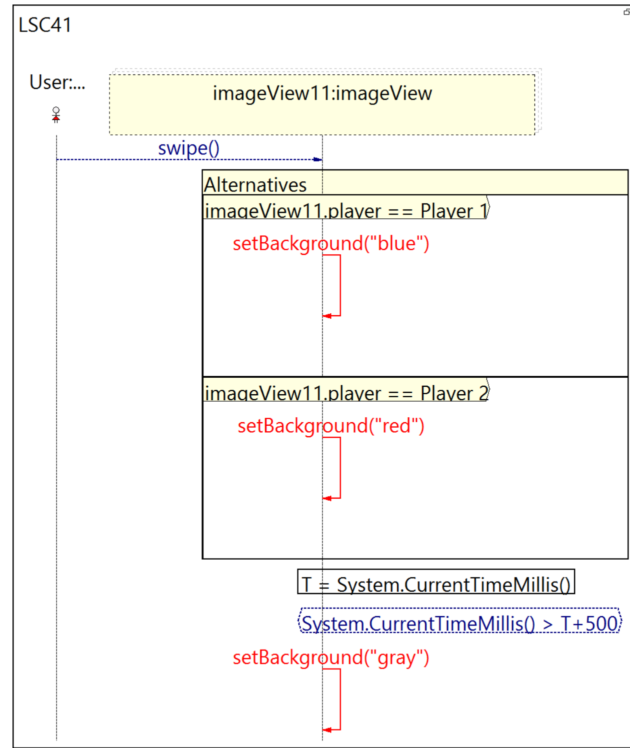

Figure 20: Swipe over and change background

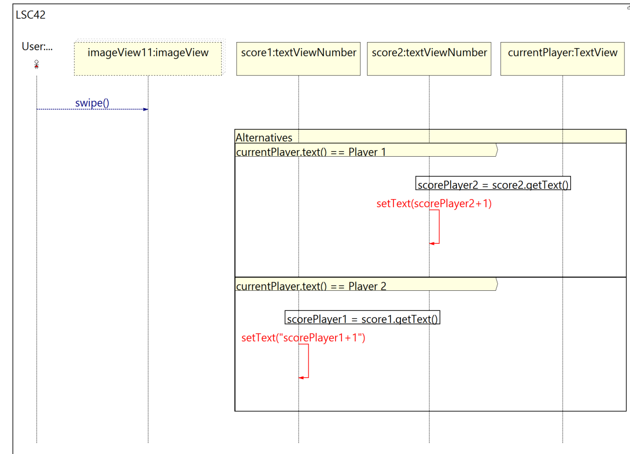

Figure 21: Swipe over and update points

come 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 [28], 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 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 that of 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's 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 obtain deep understanding of the system's behavior. We leave an appropriate evaluation effort as future work.

TouchDevelop [26] by Microsoft Research is a cloud-based integrated development environment (CIDE) that allows developing mobile applications on a mobile device. In addition to the IDE itself, 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 indeed 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 entire 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 [22] 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.

In [21] a new approach is presented for producing graphical user interfaces (GUIs) for business information system (BIS) prototypes for the Android platform. This approach creates the GUI based on a model specified by UML diagrams and textual annotations. It generates a prototype Android application, which allows 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.

Mobichart [24] is a graphical notation that extends the objectchart [8] notation for modeling mobile computing applications, and allows modeling features like object location, migration, hoarding, cloning, etc.

In [10], it is argued that programmers should be liberated from some major constraints such as the need to produce the program or pit the program against the requirements. 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 [18, 20, 30]. 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) [12]. Research results cover, among others, run-time lookahead (smart play-out) [15,16], model-checking, compositional verification and synthesis. It would be desirable to expend our method to be supported also by these non-visual languages that implement scenario-based programming.

8. CONCLUSIONS AND FUTURE WORK

We have described a 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 the generated LSC that formally describes the behavior. The user can then play-out the behavior and visually debug it 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 deeper understanding of the system's reactions. The resulting application is packed, deployed and executed on Android devices, and can be easily enhanced and modified incrementally by playing in additional scenarios, including negative ones that constraint behavior, without any need for explicit coding or intra-object modeling.

In this paper, we have limited our applications to include a single layout with no navigation. However, we believe that 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 the framework is extensible and can support any Android components. We believe that as users become familiar with our approach they will be able to easily develop their own playable components and create a rich shared open source library.

9. ACKNOWLEDGMENTS

Part of this research was carried out with the aid of grants to DH from the Israel Science Foundation (ISF) and from the Erica Drake & Benoziyo Fund. We are indebted to Eran Yahav for his support, and Assaf Marron for his helpful thoughts throughout the research.

10. REFERENCES

[1] Android development toolkit (adt). http://developer.android.com/tools/help/adt.html. Accessed 6/2015.

[2] Android studio. http://developer.android.com/tools/studio/index.html. Accessed 6/2015.

[3] Documentation of the unified modeling language (uml), available from the object management group(omg). http://www.omg.org.

[4] Intellij idea. https://www.jetbrains.com/idea. Accessed 6/2015.

[5] ITU-TS, ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)ITU-TS, geneva. 1996.

[6] A. Abadi, and Y. Dubinsky, and A. Kirshin, and Y. Mesika, and I. Ben-Harrush, and U. Hadad. Nitrogen: rapid development of mobile applications. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity*, pages 15–16. ACM, 2013.

[7] A. Abadi, and Y. Dubinsky, and A. Kirshin, and Y. Mesika, and I. Ben-Harrush, and U. Hadad. Developing enterprise mobile applications the easy way. In *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices*, pages 78–83. ACM, 2014.

[8] D. Coleman, and F. Hayes and S. Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *Software Engineering, IEEE Transactions on*, 18(1):8–18, 1992.

[9] D. Harel. From play-in scenarios to code: An achievable dream. *Computer*, 34(1):53–60, 2001.

[10] D. Harel. Can programming be liberated, period? *Computer*, 41(1):28–37, 2008.

[11] D. Harel, and A. Kleinbort, and S. Maoz. S2A: A compiler for multi-modal uml sequence diagrams. In *Fundamental Approaches to Software Engineering*, pages 121–124. Springer, 2007.

[12] D. Harel, and A. Marron, and G. Weiss. Behavioral programming. *Communications of the ACM*, 55(7):90–100, 2012.

[13] D. Harel, and H. Kugler. Synthesizing state-based object systems from lsc specifications. *International Journal of Foundations of Computer Science*, 13(01):5–51, 2002.

[14] D. Harel, and H. Kugler, and A. Pnueli. Smart play-out extended: Time and forbidden elements. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, pages 2–10. IEEE, 2004.

[15] D. Harel, and H. Kugler, and R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *FMCAD*, volume 2, pages 378–398. Springer, 2002.

[16] D. Harel, and H. Kugler, and R. Marelly, and A. Pnueli. Smart play-out. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 68–69. ACM, 2003.

[17] D. Harel, and R. Marelly. Playing with time: On the specification and execution of time-enriched lscs. In *Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002. MASCOTS 2002. Proceedings. 10th IEEE International Symposium on*, pages 193–202. IEEE, 2002.

[18] D. Harel, and R. Marelly. *Come, let's play: scenario-based programming using LSCs and the play-engine*, volume 1. Springer Science & Business Media, 2003.

[19] D. Harel, and R. Marelly. Specifying and executing behavioral requirements: The play-in/play-out approach. *Software and Systems Modeling*, 2(2):82–107, 2003.

[20] D. Harel, and S. Maoz, and S. Szekely, and D. Barkan. PlayGo: towards a comprehensive tool for scenario based programming. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 359–360. ACM, 2010.

[21] Luís Pires da Silva and Fernando Brito e Abreu. Model-driven gui generation and navigation for android bis apps. In *Model-Driven Engineering and Software Development (MODELSWARD), 2014 2nd International Conference on*, pages 400–407. IEEE, 2014.

[22] Frank Alexander Kraemer. Engineering android applications based on uml activities. In *Model Driven Engineering Languages and Systems*, pages 183–197. Springer, 2011.

[23] M. Mayer, and V. Kuncak. Game programming by demonstration. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 75–90. ACM, 2013.

[24] Hrushikesha Mohanty, Satyajit Acharya, RK Shyamasundar, and RK Ghosh. Mobichart for modeling mobile computing tasks. In *TENCON 2003. Conference on Convergent Technologies for the Asia-Pacific Region*, volume 1, pages 193–197. IEEE, 2003.

[25] N. Piterman, and A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. In *Verification, Model Checking, and Abstract Interpretation*, pages 364–380. Springer, 2006.

[26] N. Tillmann, and M. Moskal, and J. de Halleux, and M. Fahndrich. Touchdevelop: programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pages 49–60. ACM, 2011.

[27] Tuan A. Nguyen, Christoph Csallner, and Nikolai Tillmann. Gropg: A graphical on-phone debugger. In *Proc. 35th ACM/IEEE International Conference on Software Engineering (ICSE), New Ideas and Emerging Results (NIER) track*, May 2013. To appear.

[28] Shaileen Crawford Pokress and José Juan Dominguez Veiga. Mit app inventor: Enabling personal mobile computing. *arXiv preprint arXiv:1310.2830*, 2013.

[29] R. Marelly, and D. Harel, and H. Kugler. Multiple instances and symbolic variables in executable sequence charts. In *ACM SIGPLAN Notices*, volume 37, pages 83–100. ACM, 2002.

[30] W. Damm, and D. Harel. Lscs: Breathing life into message sequence charts. *Formal methods in system design*, 19(1):45–80, 2001. Preliminary version in: Ciancarini, P., Fantechi, A., Gorrieri,R. (eds.) Proc.

3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS 99), Kluwer Academic Publishers, 1999, pp. 293–312.