Synthesis for Programming on Mobile Platforms

Jeff Foster  
University of Maryland, College Park

Armando Solar-Lezama  
Massachusetts Institute of Technology
Android Popularity

• Android — most popular smartphone platform
  – 80% of the market in 2Q13
  – 850K Android apps on Google Play (4/13)
  – 40 billion apps downloaded (4/13)

• Apps written in Java
  – Extensive framework that apps hook into
  – Everything Android-specific happens in Java
    • Native code calls back into Java to access phone features

• Lots of exciting app development underway
• Interesting apps don’t need to be large
Many Opportunities for Synthesis

• Synthesis for tool developers
  – Making existing analysis technologies work for Android presents a programming challenge that synthesis can help address

• Synthesis for expert programmers
  – Similar challenges to other “Big Code” environments

• Synthesis for the end user
  – Can every phone user be a developer?
Synthesis for Android Tool Developers

New Collaborative Project enabled by ExCAPE

- Leverages combined expertise in
  - Analysis for mobile code
  - Constraint based synthesis
- Project has recently been joined by Xiaokang Qiu
  - Former student of PI Madhusudan
  - New collaboration brings in additional expertise in verification
Synthesis for Android Tool Developers

• Motivation:
  – Program analysis and synthesis tools for Android need to understand the Android framework
  – Example: SymDroid, symbolic execution for Dalvik bytecode
    • Can’t execute any app without modeling the framework

• Challenges
  – App code is tightly tied to the Android framework
  – Framework is big and complex
    • Symbolic execution of the framework is infeasible
    • Hand creation of models is brittle and expensive

• Can we synthesize a model of Android?
  – Model can be used in later synthesis work as well
Event driven programming model
• No `main()`
• Many entry points invoked in response to events from software or hardware

Framework provides an abstraction around hardware
• Convey hardware generated events
• Provide API access to hardware state

Goal: Automatically discover a simple model of the framework
Qualities of an Android Model

• Tailored for analyzability
  – Simpler than original framework
    • E.g., may need to model GUI callbacks, but need not worry about exact placement of widgets on screen
  – Code need *not* be extensible and maintainable

• Need not be precise
  – Over- or under-approximations can be acceptable
    • Depends on the goal
The Synthesis Problem

• Reactive synthesis with a few twists
  – Model is not *just* a state machine
    • Although many of its components are
  – Requirements come in the form of multiple artifacts
    • *Record of interactions* between framework and application
    • High-level structural constraints in terms of *design patterns*
    • High-level properties and *invariants*

• Note: Same ideas can apply to many software frameworks
Current Approach

- Logs gathered from range of executions
  - Use bytecode instrumentation framework to record all calls/returns between app and framework
- Templates represent knowledge of system
  - Type signatures from APIs
  - Design patterns from SDK documentation, naming conventions
  - Other insights from reading documentation
  - Key challenge: Minimizing necessary knowledge
Creating Templates: Example

- What the user knows

Users register with Talk

Talk notifies users about change
- Which users?
- In what order?

Talk observes changes to the date

User

Date

Talk
Template Using Observer Pattern

Pattern elements

- Class and method signatures
- Methods identified by function based on design pattern
- Main must implement an event listener loop

```java
class User {
    String _name;
    @Notified void update(Talk t, Date arg);
}
class Main {
    public static void main() {
        ... // setup
        @React;
    }
}
class Date { int _date; }
class Talk {
    int _date;
    String _title;
    @Event(Date) void dateChanged(Date date);
    @Attach void addObserver(User user);
    @Detach void deleteObserver(User user);
}
```
Annotation Translation

• Expand structural information from annotations
  – Basically, implement the design patterns

• Add unknowns as appropriate
  – Holes ?? indicating unknown values
  – Regular expression generators \{ e_1 \mid \ldots \mid e_n \} 

• Translate from Java to Sketch
  – Map class hierarchy and dynamic dispatch into appropriate Sketch primitives
```java
class Talk {
    @Event(Date) void dateChanged(Date date);
}
```

```java
void dateChanged_Talk(Talk self, Date date) {
    // General Event handling code:

    // log function entry and parameters

    // iterate over some of the registered users
    // call the @Notified methods of users who pass a test

    // log the function exit
}
```
void dateChanged_Talk(Talk self, Date date) {

    // log function entry and parameters
    void f(User u) {
        if (P(u, self, date)) {
            update_User(o, self, date);
        }
    }

    { | (forwardIter| backwardIter)(f, self.obs) | }

    // log the function exit
}
Log Translation

... Date change event # event (external)
> Talk.dateChanged(Talk@4885, Date@2013) # call (internal)
...
< Talk.dateChanged() # ret (internal)
...

harness void sample() {
    Event e0 = new Event(kind=0, date=...);
    ... // event setting: insert events into the queue
    main(); // main program
    ...
    log = read@log(idx++);
    assert log == /* Talk.dateChanged(Talk@4885, Date@2013) */
    ...
    log = read@log(idx++);
    assert log == /* Talk.dateChanged() */
    ...
}
(Very) Preliminary Results

• Synthesis for observer pattern
• Template:
  – 3 classes, 26 lines of code, 4 observer-related annotations
• Samples:
  – 3 total, with 40, 60, and 18 calls, including 5, 5, and 2 events
• Intermediate outputs:
  – 619 lines of sketch code
    • 308 lines for encoding classes and scenarios
    • 246 lines for encoding logs/samples
• Running time
  – Translation – less than 1 second
  – Sketch synthesis time – 17 seconds
• Final output
  – 147 lines of Java code
Scaling Up to Android

• Support for more design patterns
  – Accessors, state machines, single-/multiton, factories, builders, plus a few others

• Reduce knowledge encoded in templates
  – Increase burden on synthesizer
  – Modularity may be important here

• Test results against actual executions

• Try on several different Android platform versions
  – (1) Synthesize model supporting additional features
  – (2) Detect changes in platform (though not likely)
Other Synthesis Challenges

• Synthesis to help programmers
  – CodeHint: Interactive synthesis for IDEs (PI Bodik)
    • Sort of a super “autocomplete” feature
    • Focus is Java in general, but works with Android

• Synthesis to help end users
  – Programming by Demonstration
    • Autolayout by dragging/resizing elements of graphs (PI Bodik)
    • LifeJoin query by demonstration (PI Solar-Lezama)
      – Both may be very useful for designing mobile apps
Broader Impacts

• A huge number of people use Android today
• Synthesizing model could help researchers building Android analysis and synthesis tools
  • Which in turn will help Android users
  • E.g., tools that analyze Android apps for security concerns
• Making Android programmers more productive could help apps be easier to write, maintain, and be more reliable and secure
• Enabling end-users to build apps they want could lead to innovative new mobile device ideas
End
Synthesis for Expert Programmers

• Motivation:
  – Give Android programmers tools to make their job easier

• Build on experience with synthesis tools for frameworks
  – Prospector, MatchMaker

• Key problems
  – How to describe what you want
  – How to cope with complexity

• Technical directions
  – Combining data-driven techniques with logic-based engines
Synthesis for End Users

• Motivation:
  – Lots of people have ideas for apps they want to build
    • Can’t do it if they don’t know Java at a fairly high level
  
• Can we make every phone user an app developer?
  – Focus: “mashups” that combine abilities of existing apps
    • Ex: Map contacts whose addresses are nearby current location
    • Ex: Auto-translate SMS messages from certain phone numbers
    • Ex: Convert units from recipe app into metric

• Technical directions
  – Finding valid combinations of high-level “pathways” through apps
Android Model as a Reactive Program

• Inputs
  – External events
    • Timers, user gestures,
    • Include method being called + parameters
  – Application calls
  – Application return

• Outputs
  – Calls from the framework to the application

• Model is not just an FSM
  – Basic building blocks include FSMs, Stacks, internal data structures
  – User-provided sketch helps guide the choice of building blocks
    • Leverage known design patterns
High-level User Knowledge

• Sort of Knowledge
  – Type signatures, *design patterns*, partial programs, etc.

• Source of Knowledge
  – SDK documentations, naming conventions, insight(!)

• Sketch of Knowledge
  – Java: best fit for type signatures
    • Android platform and apps are written in Java
  – Annotations
    • Knowledge can be represented in a *declarative* manner
    • Designed to add meta-data into Java code
Synthesis Challenges

• Conveying Structure
  – Sketch is too low level
  – Solution: use design patterns to give complex structures concisely

• Scalability
  – Early experiments are encouraging
  – Modularity will be crucial

• Client-Synthesis feedback loop
  – Model should only be as complex as required by the analysis
Translation into Sketch

1. parsing high-level templates written in Java
   - modify Java grammar to allow annotations at every level, including class, method, field, and expression
   - keep AST to generate final results easier

2. rule-based annotation reduction
   - introduce appropriate data structures, fields, or expressions to reflect semantics of annotations

3. Java-to-C translation
   - virtual structure for class hierarchy
   - class per file, due to the scope of static fields
Interpretation of Synthesis Result

- find what operators/methods/values are selected
- dump out AST after substituting holes with such chosen operators/methods/values

```c
void dateChanged_Talk (Talk self, Date date, ref global int[5][50] ev_s532, ref global int log_cnt_s533)/*Talk.sk:6*/ { 
  ...
  bit __sa4 = ((self._obs.elts[0]) != (null)) && 1;
  while(__sa4) {
    User o;
    o = self._obs.elts[idx];
    if((self._date) == (date._date))/*Talk.sk:13*/ { 
      update_User(o, self, date, ev_s532, log_cnt_s533);
    }
    idx = idx + 1;
    __sa4 = ((self._obs.elts[idx]) != (null)) && (idx < 6);
  }
  ...
} /*Talk.sk:33*/
```
Current Results

• templates
  – 3 participating classes
    • 26 lines of Java code with 4 observer-related annotations
  – main harness methods for 3 scenarios
    • 42 lines of Java code with 4 reaction annotations

• samples
  – 40, 60, and 18 lines of call sequences
  – including 5, 5, and 2 events to react
Current Results (cont’d)

• intermediate outputs
  – 614 lines of sketch code
    • 303 lines for encoding classes and scenarios
    • 245 lines for encoding samples
  – translation time: within a second
  – sketch running time: 9.5 seconds

• final results
  – 144 lines of Java code
Towards Android Modeling

- **Accessors**
  - `@Get, @Set, @Is, @Has, @Put` based on naming convention

- **Uncertain methods**
  - `@Tag("tag-name")` or `@All()`

```java
class ClipboardManager {
    @Get @Set @Has
    Clipdata primaryClip;
}

class Activity {
    @Tag("ActivityLifecycle")
    void onCreate(...);
    @Tag("ActivityLifecycle")
    void onResume(); ... }

... class Platform {
    void step_up_act (Activity act) {
        act.@Tag("ActivityLifecycle");
        act.@Tag("ActivityLifecycle");
        act.@Tag("ActivityLifecycle");
    } ... }
```
More Patterns

• State machine
  – a field to maintain a @State
  – an expression to retrieve the designated @State
  – an expression to @Update the state

```java
class Activity {
    @State(Show("ActivityLifecycle"))
    int _state;
    boolean isFinishing() {
        return @State(this) == ??;
    }
    @Tag("ActivityLifecycle") void onCreate(...);
    ...
}
```
More Patterns (cont’d)

• @Singleton and @Multiton
  – one and only Activity stack, a.k.a. back stack
  – one and only instance for resource managers, associated with distinct name tags

```java
class Platform {
    @Singleton
    Stack<Activity> back_stack;
    ...
}
```
```java
class Context {
    final static String LOCATION_SERVICE = "location";
    @Multiton({ LocationManager, ... })
    Map<String, Object> systemService;
    ...
}
```
More Patterns (cont’d)

• **@Factory method**
  – returns a new instance of the target class whose constructor has a private modifier

    ```java
    class PowerManager {
        @Factory WakeLock newWakeLock(...);
        class WakeLock { ... }
    }
    ```

• **Builder pattern**
  – @Append and then @Assemble

    ```java
    class Uri {
        @Get @Set(Builder)
        private String scheme;
        @Get @Set(Builder) @Append(Builder)
        private String path;
        ... // cont’d
        @Factory
        Builder buildUpon();
        class Builder {
            @Assemble
            Uri build();
        }
    }
    ```