# Synthesizing framework models for symbolic execution


http://dx.doi.org/10.1145/2884781.2884856

### Citation


### As Published

http://dx.doi.org/10.1145/2884781.2884856

### Publisher

Association for Computing Machinery (ACM)

### Version

Author's final manuscript

### Accessed

Wed Nov 21 14:37:02 EST 2018

### Citable Link

http://hdl.handle.net/1721.1/103076

### Terms of Use

Creative Commons Attribution-Noncommercial-Share Alike

### Detailed Terms

http://creativecommons.org/licenses/by-nc-sa/4.0/
Synthesizing Framework Models for Symbolic Execution

Jinseong Jeon*  Xiaokang Qiu†  Jonathan Fetter-Degges*
Jeffrey S. Foster*  Armando Solar-Lezama†

*University of Maryland, College Park, USA  †Massachusetts Institute of Technology, USA
{jsjeon,jonfd,jfoster}@cs.umd.edu  {xkqiu,asolar}@csail.mit.edu

ABSTRACT
Symbolic execution is a powerful program analysis technique, but it is difficult to apply to programs built using frameworks such as Swing and Android, because the framework code itself is hard to symbolically execute. The standard solution is to manually create a framework model that can be symbolically executed, but developing and maintaining a model is difficult and error-prone. In this paper, we present Pasket, a new system that takes a first step toward automatically generating Java framework models to support symbolic execution. Pasket’s focus is on creating models by instantiating design patterns. Pasket takes as input class, method, and type information from the framework API, together with tutorial programs that exercise the framework. From these artifacts and Pasket’s internal knowledge of design patterns, Pasket synthesizes a framework model whose behavior on the tutorial programs matches that of the original framework. We evaluated Pasket by synthesizing models for subsets of Swing and Android. Our results show that the models derived by Pasket are sufficient to allow us to use off-the-shelf symbolic execution tools to analyze Java programs that rely on frameworks.

Categories and Subject Descriptors
I.2.2 [Automatic Programming]: Program Synthesis

Keywords
Program Synthesis, Framework Model, Symbolic Execution, SKETCH.

1. INTRODUCTION
Many modern applications are built on frameworks such as Java Swing (a GUI framework) or the Android platform, among many others. Applying symbolic execution [5] to such applications is challenging because important control and data flows occur via the framework [11]. For example, consider a Swing application that creates a button, registers a callback for it, and later receives the callback when the button is clicked. A symbolic executor that simulates only application code would miss the last step, since the control transfer to the callback happens in the framework.

One possible solution is to symbolically execute the framework code along with the application, but in our experience this is unlikely to succeed. Frameworks are large, complicated, and designed for extensibility and maintainability. As a result, behavior that appears simple externally is often implemented in complex ways. Frameworks also contain details that may be unimportant for a given analysis. For instance, for Swing, the details of how a button is displayed may not be relevant to an analysis that is only concerned with control flow. Finally, frameworks may contain native code that is not understood by the symbolic executor.

The standard solution to this issue is to manually create a framework model that mimics the framework but is much simpler, more abstract, and can be symbolically executed. For example, Java PathFinder (JPF) includes a model of Java Swing [24] that is written in Java and can be symbolically executed along with an application. However, while such models work, they suffer from several potential problems. Since the models are created by hand, they likely contain bugs, which can be hard to diagnose. Moreover, models need to be updated as frameworks change over time. Finally, applying symbolic execution to programs written with new frameworks carries a significant upfront cost, putting applications that use new or unpopular frameworks out of reach.

In this paper, we take a first step toward automatically synthesizing framework models by introducing Pasket (“Pattern sketcher”), a tool that synthesizes Java framework models by instantiating design patterns. The key idea behind Pasket is that many frameworks use design patterns heavily, and that use accounts for significant control and data flow through the framework. For example, the Swing button click callback mentioned above is an instance of the Observer pattern [12]. Thus, by creating a model that includes an equivalent instantiation of the observer pattern, Pasket helps symbolic execution tools discover control flow that would otherwise be missed.

Overview. Figure 1 gives an overview of Pasket. Its two main inputs are a set of tutorial programs that exercise relevant parts of the framework, and a summary of the framework API to be modeled. For scalability of the synthesis problem, Pasket is designed to be used with tutorial programs that each exercises a small part of the framework, and Pasket then combines the information from each tu-
We introduce PASKET, a new tool that takes a first step toward automatically synthesizing framework models sufficient for symbolic execution.

We formulate the synthesis problem as design pattern instantiation and show how to use the framework API and log of framework/client calls to constrain the design pattern instantiation process. (Sections 3 and 4)

We show how to encode the synthesis problem as a Sketch synthesis problem. (Sections 5 and 6)

We present experimental results showing PASKET can synthesize a model of a subset of Swing and a subset of Android, and that model is sufficient to symbolically execute a range of programs. (Section 7)

2. RUNNING EXAMPLE

As a running example, we show how PASKET synthesizes a Java Swing framework model from the tutorial program in Figure 2, which is a simplified extract from one of the tutorials for Java Swing.

Here the main method (not shown) calls createAndShowGUI (line 15), which instantiates a new window and adds a new instance of ButtonDemo to it. The ButtonDemo constructor


3. LOGGING AND LOG CONFORMITY

As explained earlier, PASKET executes the tutorial program to produce a log of the calls between an application and the framework. Figure 4 shows a partial log from ButtonDemo. Each log entry records a call or return. In the figure, this is the first parameter to each call, and we use indentation to indicate nested calls. Constructor calls and object parameters are annotated with a Java object id. For example, JButton@8 is a JButton with object id 8. Using object ids provides us with a simple way to match the same object across different calls. Thus, the log contains detailed information about both the values that flow across the API and the sequencing of calls and returns.

That detailed information is exactly what is needed to sufficiently constrain the synthesis problem. For example, line 67 has a call to addActionListener with arguments JButton@8 and ButtonDemo@99. Subsequently, on line 71 an ActionEvent associated with this button is created and immediately posted into the EventQueue; after this, the run method in the EventDispatchThread is called. The details of what happens inside the framework after the call to run are ignored by the logger because it does not involve methods in the given API.

The next log entry in line 74 corresponds to the framework’s call to the actionPerformed method in the application. It will
4. DESIGN PATTERN INSTANTIATION

Pasket synthesizes the code in Figure 3 by instantiating design patterns. To understand the synthesis process, consider Figures 5 and 6, which show two of the four design patterns supported by Pasket. The UML diagrams in these figures have boxes for classes and interfaces, with fields at the top and methods at the bottom, arrows for subclass or implements relationships, and diamond edges for containment. Unless marked private, fields and methods are public.

The key novelty in these diagrams are design pattern variables, indicated in colored italics. These are unknowns that Pasket solves to determine which classes and methods play which roles in the patterns. For example, the observer pattern in Figure 5 includes several different design pattern variables, including the names of the Subject and Observer classes, the name of the IObserver interface, and the names of the attach and detach methods. The main technical challenge for Pasket is to match these pattern variables with class, interface, and method names from the API description. In our running example, Pasket determines there must be an observer pattern instance with AbstractButton as the Subject and addActionListener as the attach method. This raises the framework model, Pasket instantiates the field olist from the pattern as a new field of AbstractButton, and it instantiates the body of the attach method into addActionListener. The other roles are instantiated to other classes in the API.

In addition to design pattern variables, the design pattern descriptions also leave certain implementation details to be discovered by the synthesizer. For example, inside the handle method, the synthesizer can decide what event types should invoke which individual handlers, and in the handle-handle_i, the synthesizer is left to choose in what direction to iterate over the observer list. Note that if the synthesizer chooses to iterate forward through the list, Pasket replaces the while loop with a for loop as seen in Figure 3.

Pasket uses the same basic idea of design pattern instantiation to create the entire framework model. We next discuss the patterns currently supported by Pasket, and then we discuss the problem of synthesizing multiple patterns simultaneously. We selected this set of patterns to support the experiments in Section 7, but we have designed Pasket to support extensibility with more patterns; if necessary, it is even possible to create specialized patterns when we need very platform-specific behavior.

Observers and Events. We have already discussed several aspects of the observer pattern in Figure 5. The Subject maintains a list of IObserver’s, initialized in the constructor. Observers can be attached or detached to the list, and both methods are optional, i.e., they may or may not be present. Notice update_i has no code in the pattern, since the Observer is part of the client rather than the framework. For example, in Figure 2, the update_i method is actionPerformed.

We mark the methods handle and handle_i as auxiliary to indicate they are not part of the original framework. The real framework has some (possibly complicated) logic to determine how to call the update_i methods when the run method of the EventDispatchThread is called, and the methods handle and handle_i are our way of modeling this logic. Because we do not need to match them with methods in the API, their names are not pattern variables. This is why they were added with these same names to AbstractButton in Figure 3, where the synthesizer instantiated handle to just call handle_1 and handle_1 to iterate forward through olist while calling the update method actionPerformed.

Accessors. Figure 6 shows the accessor pattern, used for classes with getters and setters. The class has k fields f1 through fk. As in Java, each field has a default value before

Figure 4: Sample output log from ButtonDemo.

be up to Pasket to infer that this sequence of log entries is part of the observer design pattern. In addition to design pattern instantiation to create the entire framework model. We next discuss the patterns currently supported by Pasket, and then discuss the problem of synthesizing multiple patterns simultaneously. We selected this set of patterns to support the experiments in Section 7, but we have designed Pasket to support extensibility with more patterns; if necessary, it is even possible to create specialized patterns when we need very platform-specific behavior.

Observers and Events. We have already discussed several aspects of the observer pattern in Figure 5. The Subject maintains a list of IObserver’s, initialized in the constructor. Observers can be attached or detached to the list, and both methods are optional, i.e., they may or may not be present. Notice update_i has no code in the pattern, since the Observer is part of the client rather than the framework. For example, in Figure 2, the update_i method is actionPerformed.

We mark the methods handle and handle_i as auxiliary to indicate they are not part of the original framework. The real framework has some (possibly complicated) logic to determine how to call the update_i methods when the run method of the EventDispatchThread is called, and the methods handle and handle_i are our way of modeling this logic. Because we do not need to match them with methods in the API, their names are not pattern variables. This is why they were added with these same names to AbstractButton in Figure 3, where the synthesizer instantiated handle to just call handle_1 and handle_1 to iterate forward through olist while calling the update method actionPerformed.

Accessors. Figure 6 shows the accessor pattern, used for classes with getters and setters. The class has k fields f1 through fk. As in Java, each field has a default value before
any initialization or update (0 for int, false for boolean, and null for all object fields). There are also r getter methods
get_f1 through get_f_r and s getter methods set_f1 through
set_f_s. Each getter method get_f_i retrieves the value of
a field chosen from f1 through f_k; similarly, each setter method
updates a field chosen from f1 through f_k with the input v.
The Accessor class also has a single constructor that ac-
cepts j arguments, for some j \leq k. The i-th argument is
used to initialize the i-th field f_i, respectively. This incurs no
loss of generality since Pasket can choose to enumerate the
fields in any order. For those fields beyond t_j, i.e., fields t_j+1
through t_k, Pasket may opt to initialize some of them im-
plcitly with either a new instance of some class cls or some
constant value (indicated by a hole ??), depending on field's
Type. For the former case, we assume that the new instance
is constructed by a public, no-argument constructor cls().
Before these fields are initialized, the constructor may or
may not call the superclass constructor with a subset of the
j arguments, written \[ [01 \ldots j] \]. For example, in Figure 3
we see that ActionEvent's constructor passes only two para-
ters to its superclass AWTEvent, which in turn passes only
one parameter to its superclass EventObject. Finally, the
constructor initializes the fields appropriately.

Other Patterns. Pasket also supports the singleton pat-
tern and the adapter pattern, which are not shown due to
lack of space. The singleton pattern supports classes that
have a single instance, such as system-level services. The
adapter pattern is used to delegate calls to another object,
e.g., in Swing, InvocationEvent is an adapter that dispatches
run calls to a Runnable object stored in a field. More details
about other patterns, along with UML diagrams, can be found in [16].

Multi-pattern Synthesis. In practice, frameworks may have
zero, one, or multiple instances of each pattern, and they
may use multiple patterns. Currently, the number of in-
stances of each pattern is a parameter to Pasket. In our
experiments, for each framework we fix these numbers across
all tutorial programs, and then discard any unused pattern
instances, as discussed further in Section 6.
Since the same class might be involved in multiple pat-
tterns, the design patterns in Figures 5 and 6 should be taken as
minimal specifications of classes—Pasket always allows classes
to contain additional fields and methods that are not
listed in a diagram. Those additional class members either
get their code from a different pattern (or different instance
of the same pattern), or are left with empty method bodies
(or return the default value of the return type). In our
running example, the AbstractButton class is involved in both
the observer pattern and the accessor pattern: its methods
addActionListener, removeActionListener and fireActionPer-
formed instantiate an observer pattern, and its methods
getActionCom-
mand and setActionCommand instantiate an accessor pattern.
Currently Pasket requires that each method body be
instantiated from at most one pattern.

5. FRAMEWORK SKETCHING
Pasket uses Sketch to discover how to instantiate the
design patterns from Section 4 into the method bodies in
Figure 3 to satisfy log conformity.

Background. The input to Sketch is a space of programs
in a C-like language. The space is represented as a pro-
gram with choices and assertions. The choices can include
unknown constants, written ??, as well as explicit choices
between alternative expressions, written \{c1 \ldots cm\}. The
goal of Sketch is to find a program in the space that sat-
isfies the assertions [35]. For example, given a program
void double(int x) \{ int t = ?????; assert t = x + x; \}
 Sketch will choose 2 for the constant ?? and x for the choice.
Full details about Sketch can be found elsewhere [34, 35].

Figure 5: Observer pattern in Pasket.

Figure 6: Accessor pattern in Pasket.
assert Subject ≠ Observer;
assert subclass[Subject]|belongsTo[attach]|;
assert subclass[Subject]|belongsTo[detach]|;
assert argNum[attach] == 1;
assert argNum[detach] == 1;
assert argType[attach][0] == IObserver;
assert argType[detach][0] == IObserver;
assert returnType[attach][0] == VOID;
assert returnType[detach][0] == VOID;
assert subclass[Observer][Observer];
assert attach ≠ detach;

Figure 7: Constraints on design pattern variables (partial).

The Encoder component in Pasket consumes the framework API, the tutorial and the log, and produces a framework sketch, which is a Sketch input file. The framework sketch is comprised of four main pieces: (1) the tutorial code, (2) driver code to invoke the framework/tutorial with the sequence of events captured in the log, (3) the framework API filled in with all possible design pattern implementations guarded by unknowns that allow the synthesizer to choose which roles of which patterns to use in each method, and (4) additional code to assert log conformity and other constraints, e.g., from subtyping relationships. When Sketch finds a solution, it will thereby discover the implementations of framework methods such that when the framework is run in combination with the app, log conformity will be satisfied.

From Java to Sketch. The first issue we face in building the framework sketch is that it must include Java code, e.g., for the client app and framework method implementations. However, Sketch's language is not object-oriented. To solve this problem, Pasket follows the approach of JSKetch [20], a tool that adds a Java front-end to Sketch. We currently do not use JSKetch directly, for two reasons. First, for log conformity, we need to retrieve runtime instances, which requires modifying an object allocation function. Second, to check log conformity only for calls that cross the boundary between the framework and the client app, we need to slightly modify method signatures and call sites to include a framework/client flag.

Like JSKetch, we introduce a new type V_Object, defined as a struct containing all possible fields plus an integer identifier for the class. More precisely, if \( C_1, \ldots, C_m \) are all classes in the program, then we define:

```c
struct V_Object {
    int class_id; fields-from-C_1 ... fields-from-C_m
}
```

where each \( C_i \) gets its own unique id.

Pasket also assigns every method a unique id, and it creates various constant arrays that record type information. For a method id \( m \), we set \( \text{belongsTo}[m] \) to be its class id; \( \text{argNum}[m] \) to be its number of arguments; and \( \text{argType}[m][i] \) to be the type of its \( i \)-th argument. We model the inheritance hierarchy using a two-dimensional array \( \text{subcls} \) such that \( \text{subcls}[i][j] \) is true if class \( i \) is a subclass of class \( j \).

Using this encoding, we can translate the client app directly into the framework sketch.

```c
void addActionListener(V_Object self, V_Object l) {
    /\* addActionListener has id 19 */
    int[] params = { 19, self.obj_id, l.obj_id};
    check_log(params);
    /\* Check that "params" is the next log entry */
    /\* and advance the log counter by one */
    if (attach == 19) { /\* code for attach */
        else if (detach == 19) { /\* code for detach */
            else if ...
        int[] ret = {-19}
        check_log(ret);
    }
```

Figure 8: Framework sketch (partial).

Driving Execution. The next piece of the framework sketch is a driver that launches the client app and injects events according to the log. More specifically, looking at Figure 4, we see three items that come from "outside" both the client app and the framework: the initial call to main (line 62) and the user inputs on lines 71 and 78. The driver is responsible for triggering these events, which it does by calling the appropriate (hard-coded) method names in Figure 5 for the event queue (or the appropriate names for Android if applying Pasket to that domain).

Design Pattern Implementations. The next component of the framework sketch is the framework API itself, with code for the design patterns, checks of log conformity, and constraints on design pattern instantiation.

For each possible pattern instantiation, and each possible design pattern variable, we introduce a corresponding variable in the framework sketch, initialized with a generator. For example, to encode the observer pattern, every role name (in italics in Figure 5) will be a variable in the framework sketch:

```c
int Subject = [[ 1 | 2 | ... ]]; int Observer = [[ 1 | 2 | ... ]];
int attach = [[ 18 | 19 | ... ]]; int detach = [[ 18 | 19 | ... ]]; ...
```

Here each design pattern variable’s generator lists the possible class or method ids that could instantiate those roles. (If there were multiple occurrences of the observer pattern, there would be multiple variables attach1, attach2, etc.)

Next, Pasket generates a series of assertions that constrain the design pattern variables according to the structure of the pattern. Figure 7 shows some of the constraints for the observer pattern. The first line requires that two different classes are chosen as Subject and Observer. The next lines check that the attach and detach methods are members of or inherited by the Subject, and that those methods have the same signature—taking a single argument of an appropriate type (a superclass of Observer) and returning void. Finally, it checks that distinct roles (e.g., attach and detach) in the design pattern are instantiated with different methods.

Finally, for each API method, we add a corresponding function to the framework sketch that checks log conformity at entrance and exit of the method, and in between conditionally dispatches to every possible method of every possible design pattern.

For example, Figure 8 depicts the framework sketch code corresponding to addActionListener (Figure 3). The first statement (line 105) creates a call descriptor that includes the method’s id and the object ids of the parameters. This call descriptor is passed to check_log (on line 106), which asserts
it matches the next entry in the global log array (created in the driver) and advances the global log counter. Next the code dispatches to various design pattern method implementations based on the role chosen for this method. Finally, the code checks that the return (indicated by negating the method id) matches the log: here the method returns void. (Note that void returns are included in the actual log though we omitted them from Figure 4.)

Putting this all together, the check_log assertions will only allow this method to be called at appropriate points in the trace, specifically lines 67 and 68 of Figure 4. Sketch will determine that attach is 19, hence the attach method code will be called in the function body.

Model Generation. After Sketch has found a solution, the last step is to generate the framework model. Pasket uses Sketch’s solution for each variable (attach, detach, etc.) to emit the appropriate implementation of each method in the model. For example, since we discover that addActionListener is the attach method of the observer pattern, we will emit its body as shown in Figure 3, along with the other methods and fields involved in the same pattern.

In some cases, methods in the framework API will be left unconstrained by the tutorial program. In these cases, Pasket either leaves the method body empty if it returns void, or adds a return statement with default values, such as 0, false, or null, according to the method’s return type.

6. IMPLEMENTATION

We implemented Pasket\(^1\) as a series of Python scripts that invoke Sketch as a subroutine. Pasket comprises roughly 14K lines of code, excluding the Java parser.

We specify name and type information for the framework via a set of Java files containing declarations of the public classes and methods of the framework, with no method bodies. Pasket parses these files using the Python frontend of ANTLR v3.1.3 [28] and its standard Java grammar. After solving the synthesis problem, Pasket then unparses these same Java files, but with method bodies and private fields instantiated according to the synthesis results. We use partial parsing [10] to make this output process simpler.

There are several additional implementation details.

Logging. For Swing tutorials, Pasket gathers logs via a logger agent, which is implemented with the Java Instrumentation API [2] using Javaassist [8]. This allows Pasket to add logging statements to the entry and exit of every method at class loading time. Pasket also inserts logging statements before and after framework method invocations. In this way, it captures call-return sequences from the framework to clients, and vice versa. Altogether, the logger agent is approximately 368 lines of Java code.

For Android tutorials, Pasket uses Redexer [18], a general purpose binary rewriting tool for Android, to instrument the tutorial bytecode. Similarly to our approach for Swing, we use Redexer to add logging at the entry and exit of every method in the app, and also insert logging statements before and after framework method invocations. The logging statements emit specially tagged messages, and we read the log over the Android Debugging Bridge (adb).

Java Libraries. Recall that several of our design patterns use classes and interfaces from the Java standard library, typically for collections such as List. Client applications also use the standard library. Thus, as part of our translation from Java to Sketch, we provide Sketch implementations of standard library methods used in our experiments.

Android Layouts. Android apps typically include XML layout files that specify what controls (called views in Android) are on the screen. In addition to the class of each control and its ID, the layout may specify the initial state of a control, such as whether a checkbox is checked, or in some cases an event handler for the control. Since layout information is needed to analyze an app’s behavior, we manually translate the layout files for each tutorial and subject app into equivalent Java code. The translated layout files instantiate each view in the layout file, set properties as specified in the XML, and add it to the Activity’s view hierarchy.

Multi-pattern Synthesis. Recall from Section 4 that we need to synthesize models with multiple design patterns at once; thus Pasket needs to know how many possible instances of each pattern are needed. For Swing, we choose 5 observer patterns, 9 accessor patterns, 1 adapter pattern, and 1 singleton pattern per tutorial program, and for Android, we choose 1 observer pattern, 10 accessor patterns, and 5 singleton patterns per tutorial program. These counts are sufficient for the tutorial programs in our experiments.

Most of the time, not all pattern instances will actually be needed. If this is the case, the input we pass to Sketch will underconstrain the synthesis problem, allowing Sketch to choose arbitrary values for holes in unused pattern instances. In turn this would produce a framework model that is correct for that particular tutorial program, but may not work for other programs. Thus, Pasket includes an extra pass to identify and discard unused pattern instances.

Merging Multiple Models. As described so far, Pasket processes a single tutorial program to produce a model of the framework. In practice, however, we expect to have many different tutorials that illustrate different parts of the framework. Thus, to make our approach scalable, we need to merge the models produced from different tutorials.

Our merging procedure iterates through the solutions for each tutorial program, accumulating a model as it goes along by merging the current accumulated model with the next tutorial’s results. At each step, for each design pattern, we need to consider only three cases: either the pattern covers classes and methods only in the accumulated model; only in the new results for the tutorial program; or in both. In the first case, there is nothing to do. In the second case, we add the new pattern information to the accumulated model, since it covers a new part of the framework. In the last case, we check that both models assign the same classes or methods to design pattern variables, i.e., that the results for those classes and methods are consistent across tutorial programs. (Note for this check to work, we must ensure class and method ids are consistent across runs of Pasket.)

7. EXPERIMENTS

We evaluated Pasket by using it to separately synthesize a Swing framework model and an Android framework model

\(^1\)https://github.com/plum-umd/pasket
from tutorial programs. Table 1 summarizes the results, which we discuss in detail next.

**Synthesis Inputs.** To synthesize the Swing model, we used ten tutorial programs distributed by Oracle. The names of the tutorials are listed on the left of Swing group in Table 1, along with their sizes. In total, the tutorials comprise just over 1,900 lines of code. The tutorial names are self explanatory, e.g., `CheckBoxDemo` illustrates JCheckBox’s behavior. The last row of the Swing section reports statistics for the merged model.

We ran each tutorial manually to generate the logs. For instance, for the `ButtonDemo` code from Figure 2, we clicked the left-most button and then the right-most button; only one is enabled at a time. It was very easy to exercise all features of these small, simple programs. The third column in the table lists the sizes of the resulting logs. We also created Java files containing the subset of the API syntactically used by these programs. It contains 95 classes, 263 methods, and 92 (final constant) fields.

To synthesize an Android model, we used three tutorial apps, listed in the Android group of Table 1. Two of them, `UIButton` and `UICheckBox`, were examples in a 2014 Course era class on Android programming. The third tutorial app, `Telephony`, is from an online tutorial site.\(^2\) Table 1 gives the size of each tutorial after translating the layout files into Java, as described above. We treated the tutorial apps similarly to the Swing programs: we ran the Android apps manually to generate logs, and we created a subset API containing the 50 classes, 153 methods, and 36 (final constant) fields referred to by these programs.

**Synthesis Time.** Given the logs and API information, we then ran **Pasket** to synthesize a model from each tutorial program individually. The middle set of columns in the table summarizes the results. Performance reports are based on seven runs of the synthesis process on a server equipped with forty 2.4 GHz Intel Xeon processors and 99 GB RAM, running Ubuntu 14.04.3 LTS.

The column **SKETCH** lists the sizes of the framework sketch files. We should emphasize that this is a very challenging synthesis problem, and these sketches are much larger than **SKETCH** has typically been used for, both in terms of lines of code and search space. For example, based on the combinatorics of the classes and methods available to fill the roles, the search space for the Swing framework is at least size $95^{21} \times 263^{87}$. In fact, one of the sketches is so hard to solve that **SKETCH** runs out of memory.

To address this problem, we adopted Adaptive Concretization (AC) [19], an extension to **Sketch** that adaptively combines brute force and symbolic search to yield a parallelizable, and much more scalable, synthesis algorithm. The remaining columns under **SKETCH** in the table report the results of running both with and without AC. The Std column lists the median running time under **SKETCH** without AC. The || column lists the median number of parallel processes forked and executed before a solution is found under AC. The next column reports the median running time of a single trial that found a solution. The last column lists the median total running time under AC. We can see that overall, synthesis just takes a few minutes, and AC tends to reduce the running time, sometimes quite significantly for larger programs.

The bottom row of each section of the table lists the time to merge the individual models together, which is trivial compared to the synthesis time.

**Synthesis Results.** The next group of columns summarizes how many instantiations of each design pattern (O for observer, Ac for accessor, Ad for adapter, and S for singleton) were found during synthesis. The last four columns report the median running time under AC. The next column reports the median number of parallel processes forked and executed before a solution is found under AC. The remaining columns under **SKETCH** in the table report the results of running both with and without AC. The Std column lists the median running time under **SKETCH** without AC. The || column lists the median number of parallel processes forked and executed before a solution is found under AC. The next column reports the median running time of a single trial that found a solution. The last column lists the median total running time under AC. We can see that overall, synthesis just takes a few minutes, and AC tends to reduce the running time, sometimes quite significantly for larger programs.

In Swing, most tutorials handle only one kind of event and one event type, and hence have a single instance of the observer pattern. Looking at the bottom row of the table, we can see there is a lot of overlap between the different tutorial programs—in the end, the merged model has five observer pattern instances.

In terms of the accessor pattern, again there is a lot of overlap between different tutorials, resulting in nine total

---

pattern instances in the merged model. Finally, all tutorials have exactly one instance of the adapter pattern for InvocationEvent and one instance of the singleton pattern for Toolkit, which are part of the Swing event-handling framework.

We manually inspected the set of empty methods in the merged model, and found that most of these methods influence how things are displayed on screen. E.g., Window.pack() resizes a window to fit its contents, and Component.setVisible() shows or hides a window. Thus, while these methods are important in an actual running Swing program, they can be left abstract in terms of control flow.

We also found some (5 of the 30 empty methods) cases of setter-like methods that were called in a tutorial, but the set value was never retrieved, hence it did not affect log conformity. Thus, for this set of tutorial programs these are safe to abstract, while another set of tutorial programs might cause these to be matched against the accessor pattern.

In general, synthesis results in Android are similar to those in Swing. Most tutorials in Android also handle only one kind of event and one event type, resulting in a single instance of the observer pattern. Similarly, for the observer pattern and the accessor pattern, there is a lot of overlap between different tutorials.

One noticeable difference between Swing and Android is the number of instances of the singleton pattern. In Android, many system-level services are running in background and providing useful features to applications. For easier maintainance, those system-level services are usually implemented as singletons.

Correctness. To check the correctness of the merged Swing model, we developed a sanity checker that verifies that a tutorial program produces the same logs when run against the merged model as when run against Swing. Recall that the logs include the events, i.e., the user interactions, that produced the original logs used for synthesis. Thus, we developed a script to translate the logged events into a main() method containing a sequence of Java method calls simulating reception of those events. Then we replay the tutorial under the model by running this main() method with the tutorial and model code, recording the calls and returns in the execution. We then compare against the original log. Using this approach, we successfully verified log conformity for all ten tutorial programs.

To check the correctness of the merged Android model, we ran the tutorial apps under the SymDroid [17] symbolic executor. Since the Android model is much smaller than that of Swing, we manually examined SymDroid’s outputs to verify the correctness of the model: we ran SymDroid and recorded its detailed execution steps; checked branching points of interest, while walking through those symbolic execution traces; and double-checked that expected branches were taken and that expected assertions passed accordingly.

Java PathFinder’s Model. Next, we compared our synthesized Swing model to an existing, manually created model: the Swing model [24] that ships as part of Java PathFinder [30] (JPF). We ran JPF, under both models, on eight of the ten tutorials. We omitted two tutorials, ColorChooserDemo and FileChooserDemo, since those cannot easily be run under JPF due to limitations in JPF’s Swing event generator. Note that there are no symbolic variables in this use of JPF, i.e., we explore only the path taken to create the original log.

Surprisingly, we found that, run with JPF’s own model, JPF failed on all tutorial programs, for a fairly trivial reason: Some method with uninteresting behavior (i.e., that our synthesis process left empty) was missing. In contrast, all eight tutorials run successfully under JPF using PASKET’s merged model. This shows one benefit of PASKET’s approach: By using automation, PASKET avoids simple but nonetheless frustrating problems like forgetting to implement a method.

Applicability to Other Programs. Finally, we ran symbolic execution on several other programs under each model, to demonstrate that a model derived from one set of programs can apply to other programs.

We chose eight Java Swing code examples from O’Reilly’s Java Swing, 2nd Edition [23] that use the same part of the framework as the Oracle tutorials we used. Table 2 lists the eight examples, along with their sizes. All ran successfully using JPF under our merged model. The rightmost column lists which Oracle tutorials are needed to cover the framework functionality used by the O’Reilly example programs. Interestingly, we found that in addition to the “obvious” Oracle tutorial (based on just the name), often the O’Reilly example programs also needed another tutorial. For example, ToolbarFrame3 needed functionality from both ToolBarDemo (the obvious correspondence) and CustomIconDemo.

We also ran two apps under the synthesized model of Android; they are listed in Table 3. Visibility is an activity extracted from the API Demos app in the Android SDK examples.3 “Bump” is an app (created for an earlier project [25]) that looks up a phone number and/or device ID from the TelephonyManager, depending on the state of two check boxes. We manually translated the layout files to Java for these two apps, as we did for the tutorial apps. As with the O’Reilly examples, these apps needed framework functionality from multiple tutorials.

In our earlier project [25], we introduced interaction-based declassification policies along with a policy checker based on symbolic executions. Using the model generated by PASKET, we conducted similar experiments. We ran the policy checker against the original, secure version of the Bump app, and found the checker yielded the correct results with the

---


<table>
<thead>
<tr>
<th>Name</th>
<th>LoC</th>
<th>Tutorials</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToolBarDemo</td>
<td>76</td>
<td>ToolBarDemo</td>
</tr>
<tr>
<td>ToolBarFrame2</td>
<td>76</td>
<td>ToolBarDemo</td>
</tr>
<tr>
<td>ToolBarFrame3</td>
<td>156</td>
<td>ToolBarDemo + CustomIconDemo</td>
</tr>
<tr>
<td>JBugleEvents</td>
<td>40</td>
<td>ButtonDemo + CheckBoxDemo</td>
</tr>
<tr>
<td>JToggleButtonEvents</td>
<td>43</td>
<td>ButtonDemo + CheckBoxDemo</td>
</tr>
<tr>
<td>SimpleSplitPane</td>
<td>45</td>
<td>SplitPaneDividerDemo + FileChooserDemo</td>
</tr>
<tr>
<td>ColorPicker</td>
<td>35</td>
<td>ColorChooserDemo + ButtonDemo</td>
</tr>
<tr>
<td>ColorPicker3</td>
<td>72</td>
<td>ColorChooserDemo + ButtonDemo</td>
</tr>
<tr>
<td>SimpleFileChooser</td>
<td>94</td>
<td>FileChooserDemo</td>
</tr>
</tbody>
</table>

Table 2: Examples from O’Reilly’s Java Swing, 2nd Edition.

<table>
<thead>
<tr>
<th>Name</th>
<th>LoC</th>
<th>Tutorials</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visibility</td>
<td>114</td>
<td>UIButton + UICheckBox</td>
</tr>
<tr>
<td>Bump</td>
<td>50</td>
<td>UIButton + UICheckBox + TelegraphyManager</td>
</tr>
</tbody>
</table>

Table 3: Example apps for Android.
synthesized framework model. For the Visibility app, we conducted the same correctness check as the other tutorial apps: we ran the app under SymDroid, and double-checked that the simulated events of user clicks were properly propagated to the app’s event handlers via our synthesized framework model.

8. RELATED WORK

Modeling. As mentioned earlier, symbolic execution tools for framework-based applications usually rely on manually crafted framework models. For example, as discussed earlier JPF-AWT [24] models the Java AWT/Swing framework. The model is tightly tied to the JPF-AWT tool and cannot easily be used by other analysis tools. Moreover, as we saw in Section 7, the model is missing several methods.

There are some studies that attempted to automatically create models of Swing [7] and Android [40] for JPF. The techniques from these papers are quite different as they rely primarily on slicing. One advantage of Pasket is that it could generate more concise models for complex frameworks because it is unconstrained by the original implementation’s structure. Nonetheless, the techniques used in those papers could help identify which parts of the framework to model.

Several researchers have developed tools that generate Android models. EDGEMiner [6] ran backward data-flow analysis over the Android source code to find implicit flows. Modelgen [9] infers a model in terms of information flows, to support taint analysis. To learn behaviors of the target framework, it inputs concrete executions generated by Droidrecord, similarly to our logging using Redexer [18]. Both of these systems target information flow, which is insufficient to support symbolic execution.

Given an app, Droidel [4] generates a per-app driver that simulates the Android lifecycle. This enables some program analysis of the app without requiring analysis of the Android framework, which uses reflection to implement the lifecycle. A key limitation of Droidel is that it is customized to the lifecycle and to a particular Android version.

Mimic [15] aims to synthesize models that perform the same computations as opaque or obfuscated Javascript code. Mimic uses random search inspired by machine learning techniques. Mimic focuses on relatively small but potentially complex code snippets, whereas Pasket synthesizes large amounts of code based on design patterns.

Samini et al. [31] propose automatically generating mock objects for unit tests, using manually written pre- and post-conditions. This is also quite different from Pasket, which synthesizes a model using knowledge of design patterns.

Synthesis. There is a rich literature on algorithmic program synthesis since the pioneering work by Plumel and Rosner [29], which synthesizes reactive finite-state programs. Most of these synthesizers aim to produce low-level programs, e.g., synthesis techniques that are also sketch-based [36, 37, 38]. The idea of encoding a richer type system as a single struct type with a type id was also used in the Autograder work [33]. Component-based synthesis techniques [14, 22] aim at higher-level synthesis and generate desired programs from composing library components. Our approach is novel in both its target (abstract models for programming frameworks) and its specification (logs of the interaction between the client and the framework, and an annotated API).

The idea of synthesizing programs based on I/O samples has been studied for different applications. Godefroid and Taly [13] propose a synthesis algorithm that can efficiently produce bit-vector circuits for processor instructions, based on smart sampling. Storyboard [32] is a programming platform that can synthesize low-level data-structure-manipulating programs from user-provided abstract I/O examples. Transit [39], a tool to specify distributed protocols, inputs user-given scenarios as concolic snippets, which correspond to call-return sequences Pasket logs. In our approach, the synthesis goal is also specified in terms of input (event sequences) and output (log traces), and our case studies show that the I/O samples can also help synthesize complex frameworks that use design patterns.

Design Patterns. In their original form, design patterns [12] are general “solutions” to common problems in software design, rather than complete code. That is, there is flexibility in how developers go from the design pattern to the details. Several studies formalize design patterns, detect uses of design patterns, and generate code using design patterns.

Mikkonen [26] formalizes the temporal behavior of design patterns. The formalism models how participants in each pattern (e.g., observer and subject) are associated (e.g., attach), how they communicate to preserve data consistency (e.g., update), etc. Mikkonen’s formalism omits structural concerns such as what classes or methods appear in.

Albin-amiot et al. [1] propose a declarative meta-model of design patterns and use it to detect design patterns in user code. They also use their meta-model to mechanically produce code. Jeon et al. [21] propose design pattern inference rules to identify proper spots to conduct refactoring. These approaches capture structural properties, but omit temporal behaviors, such as which observers should be invoked for a given event. In contrast, Pasket accounts for both structural properties and temporal behaviors. We leverage design patterns as structural constraints and logs from tutorial programs as behavioral constraints for synthesis.

Antkiewicz et al. [3] aim to check whether client code conforms to high-level framework concepts. They extract framework-specific models, which indicate which expected code patterns are actually implemented in client code. This is quite different from the symbolically executable framework model synthesized by Pasket.

9. CONCLUSION

We presented Pasket, the first tool to automatically derive symbolically executable Java framework models. Pasket consumes the framework API and logs from tutorial program executions. Using these, it instantiates the observer, accessor, singleton, and adapter patterns to construct a framework model that satisfies log conformity. Internally, Pasket uses Sketch to perform synthesis, and it merges together models from multiple tutorial programs to produce a unified model. We used Pasket to synthesize a model of a subset of Swing used by ten tutorial programs, and a subset of Android using five tutorial programs. We found that synthesis completed in a reasonable amount of time; the resulting models passed log conformity checks for all tutorials; and the models were sufficient to execute the tutorial programs and other code examples that use the same portion of the frameworks. We believe Pasket makes an important
step forward in automatically constructing symbolically executable Java framework models.

Acknowledgments
Supported in part by NSF CCF-1139021, CCF-1139056, CCF-1161775, and the partnership between UMIACS and the Laboratory for Telecommunication Sciences.

References
[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994.


