Program Synthesis for Program Analysis: Models and Drivers
(Extended Abstract)

Jinseong Jeon
University of Maryland, College Park
jsjeon@cs.umd.edu

1. Introduction

In recent years, many program analysis tools have been proposed, with several promising results, e.g., using abstract interpretation [4] or symbolic execution [9] to find bugs in software systems [2, 3]. Despite a diversity of techniques on which those tools rely, almost all tools share a common problem: they focus on analyzing the application code. But applications are no longer written in isolation—they are built on top of frameworks consisting of infrastructure, libraries, and the operating system, at the very least. Moreover, framework code might not be available; it might be available but too large and complex to analyze in practice; or it might be written in a different language from the ones handled by tools. As a result, program analysis cannot be performed properly when it reaches the boundary between an app and the framework.

Suppose we want to analyze a Java Swing app. Figure 1 shows a subset of call-return sequences and event creations recorded under one particular run of the target app, which registers itself as an event listener for two buttons (lines 5–6). After GUI elements are built, it is ready to react to user interactions; user’s button clicks will trigger ActionEvents (lines 8 and 12), which are in turn followed by the corresponding event handler, actionPerformed.

From the client point of view, control-flows from ActionEvent creation to client’s actionPerformed method (lines 8–9 and 12–13) are implicit. That is, part of client code is called back from the framework, thus program analysis tools cannot properly analyze such program without a model of the framework.

To resolve this issue, most analysis tool developers resort to manually writing models of frameworks for analysis purpose. For example, Java Pathfinder [14] includes models of Java GUI libraries [12], and the KLEE project [3] uses a specially modified, embedded systems version of libc. Such models of framework, however, are tedious and error-prone to write, and must often be modified if the analysis or framework changes.

With or without such models of a framework, it is challenging for program analysis tools to achieve (re)usability. Program analysis techniques, such as abstract interpretations [4] and symbolic execution [9], define how to analyze programs, but what properties to analyze are up to what problems those tools aim for. In most cases, program analysis tools are either specific to particular problems, e.g., buffer overflow detection [19], or so general that driving them to analyze desired properties is not trivial.

In this article, we propose to apply program synthesis [10, 11] to automatically generate two artifacts—models of framework and client-oriented drivers—to enable program analysis tools to analyze framework apps more easily and efficiently. The first one will enable program analysis to proceed even when it reaches the boundary between an app and the framework. The second one will supply end-users with easy-to-understand code that drives program analysis tools to analyze properties of interest.

2. Program Synthesis

Program synthesis [10, 11] is an attractive programming paradigm in which an automated algorithm derives a program from a given specification. In addition to SMT solvers [13], other key ideas that make program synthesis feasible are samples and templates, which serve as behavioral and structural constraints, respectively. In this section, we explain what samples and templates are, and propose novel sorts of samples and templates for our synthesis problems.

Figure 2 illustrates how our proposed system can be organized into three major subsystems: logger, rule encoder, and Sketch synthesizer [17]. We will explain how the logger can create logs and translate them into behavior constraints; how the rule encoder translates high-level templates into structural constraints; and how those constraints are combined and resolved together.

2.1 Sample

The inference algorithm inside program synthesis depends on the given specification. Among various ways to represent a specification, input-output examples are much preferred by programmers because they are intuitive and easy to provide. Unlike previous research [6, 16], in which input-output domains were just bits or integers for function arguments, our problems—synthesizing models of frameworks and client-oriented drivers—deal with much higher-level samples: behavior of the example programs.

We will represent a behavior as call-return sequences and environment changes, e.g., button click, mouse movements, etc., as shown in Figure 1. We will obtain such call sequences either by instrumenting example programs via a binary rewriter [8] or by attaching a logging feature to the environment where programs run [1].

```
1 ButtonDemo.createAndShowGUI()
2 ButtonDemo.ButtonDemo()
3 JButton.setActionCommand(JButton@8, "disable")
4 JButton.setEnabled(JButton@4, false)
5 JButton.addActionListener(JButton@8, ButtonDemo@9)
6 JButton.addActionListener(JButton@4, ButtonDemo@9)
7 ...
8 ActionEvent.ActionEvent(JButton@8, 0, "disable")
9 ButtonDemo.actionPerformed(ButtonDemo@9, ActionEvent@7)
10 ActionEvent.getActionCommand(ActionEvent@7)
11 JButton.setEnabled(JButton@8, false)
12 ActionEvent.ActionEvent(JButton@4, 0, "enable")
13 ButtonDemo.actionPerformed(ButtonDemo@9, ActionEvent@5)
14 ActionEvent.getActionCommand(ActionEvent@5)
15 JButton.setEnabled(JButton@8, true)
```

Figure 1. Sample log output (abbreviated).
2.2 Template

A template [18], a.k.a. a partial program, is a program with holes that need to be filled by a synthesizer. It aids synthesis by dramatically reducing the search space explored by the synthesizer. Ironically, such templates can also hinder scalability of program synthesis because writing templates can be quite burdensome. Another key idea behind our approach is to leverage user’s high-level knowledge, such as which design patterns might be used or what kinds of temporal properties are of interest, to make templates easy to develop.

2.2.1 Artifact A: Models of Framework

We have observed that framework code often makes intensive use of design patterns [5]. Some patterns, e.g., structural patterns, are easy to write actual code, whereas some patterns, e.g., behavioral patterns, require developers to go from the idea of the design pattern to the details. Either way, design patterns themselves can serve as templates, and any non-deterministic choices inside design patterns will be determined by the synthesizer.

For example, the observer pattern is an extremely common pattern in event-driven frameworks, involving three different roles: observers register (lines 5–6 in Figure 1) to be notified (lines 9 and 13) when specific events happen on observable objects (lines 8 and 12). By leveraging the structure of the observer pattern, our technique can synthesize models of observer code in widely used frameworks. For performance purpose, users are required to supply our tool with a couple of additional hints, namely, annotations to specify which set of classes play a role in the observer pattern:

```java
@ObserverPattern(ActionEvent)
public class AbstractButton ...

@ObserverPattern(ActionEvent)
public interface ActionListener ...
```

Here the annotations indicate that class AbstractButton and sub-classes, and interface ActionListener and implementers, may be involved in the observer pattern with event ActionEvent. Notice, however, that we do not need to specify which role each class/interface plays, nor the roles of particular methods.

2.2.2 Artifact B: Client-Oriented Drivers

A client-oriented specification [7] is a code snippet that drives program analysis tools to certain states of interest, and then asserts that the desired property holds. We regard it as a client-oriented driver (co-driver in short). For program analysis tools that input user-given specs, a co-driver is a better option than parameter-based configurations, since it intuitively shows how the tool is going to simulate the target program with what inputs, user interactions, etc. On the flip side, it is up to users to write such spec, which is neither desirable nor scalable.

To make writing such co-drivers easier, our technique inputs users’ rough hints (e.g., a time line diagram like storyboard [15]) about forms of temporal properties they are looking for, rather than concrete formulæ. Our technique then translates those hints into a template composed of conditional branches and loops, according to the semantics of temporal logics. Together with behavioral constraints encoded from the logs, the synthesizer will generate driver code that represents the desired property.

References