ABSTRACT

Title of dissertation: FRAMEWORK SYNTHESIS FOR SYMBOLIC EXECUTION OF EVENT-DRIVEN FRAMEWORKS

Jinseong Jeon, Doctor of Philosophy, 2016

Dissertation directed by: Professor Jeffrey S. Foster
Department of Computer Science

Symbolic execution is a powerful program analysis technique, but it is very challenging to apply to programs built using event-driven frameworks, such as Android. The main reason is that the framework code itself is too complex to symbolically execute. The standard solution is to manually create a framework *model* that is simpler and more amenable to symbolic execution. However, developing and maintaining such a model by hand is difficult and error-prone.

We claim that we can leverage *program synthesis* to introduce a high-degree of automation to the process of framework modeling. To support this thesis, we present three pieces of work. First, we introduced SymDroid, a symbolic executor for Android. While Android apps are written in Java, they are compiled to Dalvik bytecode format. Instead of analyzing an app's Java source, which may not be available, or decompiling from Dalvik back to Java, which requires significant engineering effort and introduces yet another source of potential bugs in an analysis, SymDroid works directly on Dalvik bytecode.

Second, we introduced PASKET, a new system that takes a first step toward automatically generating Java framework models to support symbolic execution. PASKET takes as input the framework API and tutorial programs that exercise the framework. From these artifacts and PASKET's internal knowledge of *design patterns*, PASKET synthesizes an *executable* framework model by *instantiating* design patterns, such that the behavior of a synthesized model on the tutorial programs matches that of the original framework.

Lastly, in order to scale program synthesis to framework models, we devised *adaptive concretization*, a novel program synthesis algorithm that combines the best of the two major synthesis strategies: symbolic search, *i.e.*, using SAT or SMT solvers, and explicit search, *e.g.*, stochastic enumeration of possible solutions. Adaptive concretization parallelizes multiple sub-synthesis problems by partially concretizing highly influential unknowns in the original synthesis problem.

Thanks to adaptive concretization, PASKET can generate a large-scale model, *e.g.*, thousands lines of code. In addition, we have used an Android model synthesized by PASKET and found that the model is sufficient to allow SymDroid to execute a range of apps.

FRAMEWORK SYNTHESIS FOR SYMBOLIC EXECUTION OF EVENT-DRIVEN FRAMEWORKS

by

Jinseong Jeon

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2016

Dedication

To Yunkyung, my better half.

forever and always.

To Lukas and/or Seungwoo, our new generation,

a naughty boy who kinda stole our hearts.

And to my supportive parents.

Acknowledgments

I could not have performed the research that this dissertation encompasses without the help of others, who I would like to thank now. First and foremost, I would like to thank my advisor Jeff Foster for his advice, support, and patience. During the course of my Ph.D., there were several once-in-a-life events, such as marriage (yes, she is perfect to me), childbirth (yes, once is enough!), and a family bereavement (hope there will be no more). Of course, I was not able to do research a couple months, and he generously allowed me to support my family. Needless to say, this is just one evidence that he is a great advisor. From him, I also learned how to conduct (ambitious) research, discuss effectively, develop critical thinking, to name a few.

As a close second, I thank my family, especially my wife. She has been patient and really enjoying the life as a graduate student's wife, even though we have the terrible twos, who often tested our limits. My son Lukas and/or Seungwoo, born in the middle of my Ph.D., became another catalyst for graduation in some sense. And thanks to my parents for their unfailinig support.

I was fortunate to collaborate with Armando Solar-Lezama, Xiaokang Qiu, Kris Micinski, Jeff Vaughan, Todd Milstein, Jon Fetter-Degges, Tanzirul Azim, Jaeyeon Jung, Ravi Bhoraskar, and Josh Reese. These collaborators have contributed significantly to this dissertation and enriched my time as a graduate student. I must also thank my committee members for their useful feedback.

There is a story that I would like to mention. As an alternative way to do

a mandatory military service, I was a researcher at a government agency, which is somewhat similar to DARPA. As expected, the security matters, and the way we, as agents, could access to internet was quite annoying: we use two-in-one computers each of which uses seperate networks. That is, we need to turn it on and off to switch to the other part.

After finishing the service and arriving here as a student, who was kinda allowed to enjoy free network access, what do you think I wanted to do very first? Well, reading emails via a cell phone on my palm! Thus, the very first thing I did was visiting an Apple store so as to buy an iPhone. All the crew members were busy while talking to other visitors, but I dared to interupt a random member. Surprisingly (now I understand I did wrong), "don't you see me serve others?", she snapped. The very first culture shock I experienced. That made me go to another store and choose an Android phone.

At the first meeting with my advisor, he listed potential projects that I could involve, including Android security and program synthesis. Since I was using an Android phone, I chose Android security first. (I also let him know that I'm interested in program synthesis as well, but it just looked a bit difficult at that moment.) This is an unofficial, informal story of how one's dissertation begun. While finalizing my dissertation, I'd like to thank that crew member who drove me to use Android phones, not iPhones.

This work was supported in part by a research award from Google, NSF CNS-1064997, CCF-1139021, CCF-1139056, CCF-1161775, and the partnership between UMIACS and the Laboratory for Telecommunication Sciences.

Table of Contents

List of Figures

List of Abbreviations

AAR	Android Archive
AC	Adaptive Concretization
ADB	Android Debugging Bridge
API	Application Programming Interface
CEGIS	Counter-example-guided Inductive Synthesis
CO-spec	Client-Oriented Specification
CTS	Compatibility Test Suite
DVM	Dalvik Virtual Machine
GUI	Graphical User Interface
JAR	Java Archive
JPF	Java Path Finder
JVM	Java Virtual Machine
MCMC	Markov Chain Montecarlo
RC	Random Concretization
SAT	Satisfiability
SDK	Software Development Kit
SIQR	Semi-interquartile Range
SMT	Satisfiability Modulo Theory
SyGuS	Syntax-Guided Synthesis
UML	Unified Modeling Language
URI	Uniform Resource Identifier
XML	Extensible Markup Language

Chapter 1: Introduction

In recent years, the research community has proposed many useful program analysis tools. Among many others, symbolic execution [53, 14, 41, 66] is an appealing technique because it is able to either verify properties of interest or else provide counterexamples, which could be used as test inputs. Depending on the input type, a symbolic executor is essentially a binary or bytecode interpreter, but with the additional ability to operate on *symbolic expressions*, which represent potentially unknown quantities. The executor may branch its execution if a symbolic expression is used as the guard of a conditional, in the case that both true and false branches are feasible. The main feature of symbolic execution is to maintain path conditions comprised of symbolic expressions at such branching points, which are in turn used to check assertions and discard infeasible paths in later executions. At the same time, those path conditions are used to represent paths that have been taken by symbolic execution. Such path conditions could explain under what conditions apps may reach program points of interest, *e.g.*, assertions, system calls, permission-associated APIs, to name a few.

Although symbolic execution is a powerful program analysis technique, applying it to event-driven frameworks, such as Android, is challenging. The main reason is that apps that run on top of event-driven frameworks are tightly tied to the framework: Apps can define their own event handlers which the framework will callback when events happen. The control-flows from event creations through the framework to apps' own event handlers are implicit, and thus symbolic execution cannot be performed properly if it reaches the boundary between apps and the framework. For example, consider an Android app 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 Android, the details of how UI elements are 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) [71], a static analysis framework for Java, includes a model of Java Swing [59] 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 dissertation, we propose to leverage *program synthesis* [58, 57] to introduce a high-degree of automation to the process of generating a model of framework that can make symbolic execution more effective and efficient. The model, which abstracts away unimportant details yet encompasses essential behavior of the framework, will enable symbolic execution to proceed even when it reaches the boundary between an app and the framework.

Our thesis is following:

> *Program synthesis can help apply symbolic execution to event-driven frameworks and make that technique more effective and efficient by generating a framework model that enables symbolic execution to perform even across the boundary between apps and the framework.*

To support this thesis, we present three pieces of work. First, we introduced SymDroid, a symbolic executor for Android. In our preliminary study using a manual model, SymDroid was able to reason under what conditions privacy-sensitive APIs are used in a tutorial app. Although that study was promising, such hand-written model hindered us from rapidly applying the tool to newer apps. The main bottleneck was incrementally adding more APIs into the model whenever SymDroid stopped due to missed APIs, which motivated this dissertation. Second, we introduced Pasket, a new system that synthesizes a Java framework model. We replaced the manual model in SymDroid with the model automatically synthesized by Pasket and showed that the synthesized model is sufficient to enable SymDroid to execute a range of apps. Lastly, in order to scale program synthesis to framework model domain, we devised *adaptive concretization*, a new program synthesis algorithm that combines the best of two other extreme approaches and naturally parallelizes. Each piece will be more described in the next three sections.

1.1 Symbolic Execution for Android

Google's Android is currently the most popular mobile device platform, running on a majority of all smartphones. Android provides a certain level of security by protecting sensitive framework APIs using *permissions*. When installing an app, the user will be presented with the list of permissions requested by the app, and have only two choices: either proceeding with installation, which grants the app all the permissions it asks for, or aborting installation. In this context, the user can see *what* permissions apps request, but the reasons *why* apps require those permissions are unclear. We opt to use symbolic execution in order to reason under what circumstance apps use permission-protected, privacy-sensitive APIs.

While Android apps are written in Java, they are compiled to Google's Dalvik Virtual Machine bytecode format. Thus, while existing Java-based program analysis tools could potentially be used to reason about properties of apps, including correctness, security, and privacy, in practice doing so requires either access to an app's

Java source, or decompilation from Dalvik back to Java. The former is problematic for many uses (*e.g.*, any case where we want to analyze an app without source), and the latter requires significant engineering effort and introduces yet another source of potential bugs in an analysis.

To address this limitation, we opt to develop a program analysis tool that works directly on Dalvik bytecode. In Chapter 2, we introduced SymDroid [43], a symbolic executor [53, 14, 41, 66] for Dalvik. Due to the constrained mobile environment, Dalvik is carefully designed to reduce the overall size of the application, resulting in redundant instructions. To make as clean and simple symbolic execution rules as possible, we designed μ-Dalvik, a simpler language that contains just 16 instructions, compared to more than 200 Dalvik bytecode instructions, and to which it is easy to translate Dalvik. In addition to modeling the bytecode itself, for purposes of the evaluation, we manually created a model of Android and Java libraries in part.

We evaluated the correctness and feasibility of the prototype of SymDroid via Android Compatibility Test Suite (CTS) [7] and a case study that discovers under what conditions certain privileged operations can be used in a tutorial app. CTS acts as a collection of unit tests that thoroughly exercise Dalvik bytecode semantics and platform functionality. As long as input apps do not introduce symbolic variables, we can regard SymDroid as a sort of Dalvik virtual machine, since it will simply execute given apps step by step. Tests in CTS do not involve in symbolic variables, and thus we can use CTS as regression tests. We found that SymDroid passed all test cases that did not require more modeling of Android and Java libraries. We also ran SymDroid on a tutorial app, named PickContact, and found it could discover the correct conditions under which the READ_CONTACTS permission was used.

1.2 Framework Synthesis

Although the preliminary study on SymDroid was promising, using a manual model obstructed us from swiftly expanding the target apps. We manually modeled parts of the framework that were sufficient only to a selection of apps; figured out which parts should be for other apps; and repeated the aforementioned process in an incremental manner. Not unexpectedly, that process was tedious and error-prone, which motivated the next system.

In Chapter 3, we introduced Pasket [48], a new system that applies program synthesis to generating framework models. In particular, Pasket's focus is to create an *executable* model of event-driven frameworks, which is able to produce implicit control-flows due to the event handling mechanism. Besides, to synthesize a *symbolically* executable model, Pasket abstracts away unimportant parts from the analysis perspective, *e.g.*, GUI layouts.

To automatically synthesize framework models, Pasket relies on two key insights: First, from the perspective of a client app, only the surface behavior of the framework matters, and not the framework internals. That is, a framework model is sufficient as long as it produces the same call-return sequences between the framework and the app. Second, looking at event-driven frameworks in more detail, we find that most of the major control flow decisions are captured in design patterns [29] used by the framework. For example, event-driven frameworks make heavy uses of the observer pattern, in which observers register for callbacks with a subject.

Pasket builds on these insights with a multi-modal (*i.e.*, consuming several different kinds of input) synthesis process. The first input to Pasket is the framework API, in terms of its classes, methods, and types; this gives the basic structure of the expected model. The second input is *logs* gathered by running a tutorial program against the actual framework and recording the call-return sequence between the program and the framework. Pasket uses such tutorials to enforce *log conformity*: a correct framework model, run against the same tutorial program, should produce the same logs.

In addition to these inputs, Pasket has internal knowledge of design patterns, which place *structural constraints* on the space of possible framework models that Pasket considers. For example, in the observer pattern, Pasket knows there must be subject and observer classes, and the subject must have a method to attach an observer, among other things. Given all this information, Pasket performs *design pattern matching*: instantiating the design patterns to the framework API, such that log conformity is satisfied.

We used Pasket to produce a model of the Java Swing GUI framework and the Android framework. For the given ten Swing tutorials, Pasket took just a few minutes and generated a model consisting of 95 classes and 2,676 lines of code, making it one of the largest pieces of code ever synthesized using constraint-based synthesis. For the given three Android tutorials, Pasket took just a few seconds and produced a model consisting of 50 classes and 1,419 lines of code.

We empirically validated those models: We ran the Swing tutorials under Java PathFinder [71] (JPF), and found that JPF with our synthesized model could successfully execute input tutorials, whereas JPF's own model failed due to some missing methods. We selected eight code examples from a different source that use the same part of the framework and verified that they ran under JPF using our synthesized model for Swing. We also selected two Android code examples and verified they ran under SymDroid using our synthesized model for Android.

Currently, to specify the expected behaviors and structures of the framework model, Pasket relies on logs—calls that cross the boundary—and design patterns. However, it is likely that there exist some more important features of the target framework whose behaviors and structures may not be properly captured via logs and/or design patterns, *e.g.*, APIs with internal side effects, framework's own programming idioms, *etc.* As a consequence, a model synthesized by Pasket is neither sound nor complete: it may include wrong behaviors and/or may miss some important features. Nonetheless, the key point is that we can easily regenerate an enhanced model by using Pasket's semi-automated process. We discuss the abstraction, soundness and completeness of synthesized models in detail.

1.3 Adaptive Concretization

The way we reduced the automatic construction of a framework model to program synthesis is groundbreaking, and so is the size of encoded problems. In the course of Pasket development and evaluation, we often encountered performance bottlenecks, where Sketch simply failed due to timeout or out of memory. Those performance issues motivated us to study and devise a scalable program synthesis algorithm.

Program synthesis [58, 57] is an attractive programming paradigm in which an automated algorithm derives a program from a given specification. One popular style of program synthesis is *syntax-guided synthesis*, which starts with a structural hypothesis describing the shape of possible programs, and then searches through the space of candidates until it finds a solution. The structural hypothesis is represented as a partial program or *template*, which encompasses *unknowns* that are resolved by the synthesizer. There are two common approaches to syntax-guided synthesis: *explicit search*—either stochastically or systematically enumerating the candidate program space—and *symbolic search*—encoding the search space as constraints that are solved using a SAT solver. The SyGuS competition has recently revealed that neither approach is strictly better than the other [4].

In Chapter 4, we proposed *adaptive concretization* [46, 45], a new algorithm that greatly improves scalability of program synthesis. Adaptive concretization combines many of the benefits of explicit and symbolic search while also parallelizing very naturally, allowing us to leverage large-scale, multi-core machines.

The key observation behind our algorithm is that in synthesis via symbolic search, the unknowns that parameterize the search space are not all equally important in terms of solving time. However, it is impossible to accurately calculate how influential an unknown is with respect to solution time in adaptive concretization, and thus we estimate the influence of an unknown by means of its dependent terms in the translated formulae. Since the influence is an estimate, there is a chance that concretizing even highly influential unknowns may not affect solving time. Thus, we opt to randomize what to concretize and how much to concretize by introducing a notion of *degree of concretization*. Then the influence estimation as well as degree of concretization will determine the probability of concretizing certain unknowns.

The degree of concretization poses its own challenge: an optimal degree varies from benchmark to benchmark. Due to the lack of fixed optimal degrees, the crux of adaptive concretization is to search for the optimal degree online. We assume that the tradeoff between degree and solution time forms a "vee": at degree 0, which corresponds to pure explicit search, the synthesis may take long due to the low chance of finding a solution; at degree ∞, which corresponds to pure symbolic search, the synthesis may take long, too, due to the complexity of the synthesis problem itself; and between them, there would exist a sweet spot, where partially concretizied problems may not take that long, while the probability of finding a solution is much higher, achieving overall improvement in solution time. We empirically validate our insight about "vee" and utilize its shape and the existence of optimum: Our algorithm begins with hill climbing to dramatically reduce the range of target degrees and performs binary search between the refined range.

The adaptive concretization algorithm involves several "magic numbers" in the influence estimations and concretization probability. We thoroughly explored potential design choices and empirically showed that our algorithm is robust to those design decisions. We implemented our algorithm for Sketch and evaluated it against 26 benchmarks from a number of synthesis applications. We found our algorithm outperforms Sketch on 22 of 26 benchmarks, sometimes achieving significant speedups of 3× up to 18×. In one case, adaptive concretization succeeds where Sketch runs out of memory. We also ran adaptive concretization on 1, 4, and 32 cores, and found it generally has reasonable parallel scalability.

Chapter 2: Symbolic Execution for Android

In this chapter, we take a first step toward developing a suite of program analysis tools that work directly on Dalvik bytecode. We introduce SymDroid [43], a symbolic executor [53, 14, 41, 66, 33, 75, 17, 34, 70] for Dalvik. SymDroid is essentially a Dalvik bytecode interpreter, but with the additional ability to operate on *symbolic expressions*, which represent potentially unknown quantities. SymDroid uses an SMT solver to test whether assertions involving those expressions are always true; if not, SymDroid can produce a counter-example showing a cause of the assertion failure. SymDroid may also branch its execution if a symbolic expression is used as the guard of a conditional, in the case that both the true and false branch are feasible.

One way to view a symbolic executor is as a runnable operational semantics, and thus we envision that SymDroid's semantics for Dalvik might be of independent interest. Thus, we aimed to develop as clean and simple a semantics as possible. The result is μ-Dalvik, a language that contains just 16 instructions, compared to more than 200 Dalvik bytecode instructions, and to which it is easy to translate Dalvik. μ-Dalvik achieves its compactness through three basic transformations. First, it coalesces multiple Dalvik instructions that are distinguished only by bit widths, *e.g.*, `goto +AA`, `goto/16 +AAAA`, and `goto/32 +AAAAAAAA` become a single μ-Dalvik goto statement. Second, it encodes operand types in the operand, rather than in the operator, *e.g.*, `aput-byte`, `aput-char`, and `aput-short` all map to the same μ-Dalvik instruction, and we store the operand type in the operand. Finally, μ-Dalvik expands some complex Dalvik instructions into sequences of simpler instructions, *e.g.*, `packed-switch` becomes a sequence of conditions. Note that μ-Dalvik aims to minimize the number of instructions (and thus keep the semantics cleaner), whereas Dalvik's goal is to maximize performance and minimize code size. (Section 2.1 presents μ-Dalvik in detail.)

The core of SymDroid, the symbolic execution rules for each μ-Dalvik instruction, are standard and quite straightforward, as μ-Dalvik is so compact. Section 2.2 fits essentially all of the main rules on a couple of pages of text, and those rules correspond directly to our implementation, which comprises approximately 17K lines of OCaml code. Of course, in addition to modeling the bytecode itself, SymDroid also needs to provide models of the platform, including the system libraries (many of which contain native code, and hence cannot be directly executed by SymDroid) and the Android control framework (which is quite complex). As a prototype tool, currently SymDroid implements just enough of these to support a range of apps. (Section 2.3 describes our Android platform model in more detail.)

We evaluated SymDroid in two ways. First, we used it to run the Android Compatibility Test Suite (CTS) [7], which tries to thoroughly exercise Dalvik bytecode and platform functionality. We found that SymDroid passed 26 out of 92 CTS tests. It failed the remaining tests not because of errors in instruction handling, but because we do have not yet implemented all the Java and Android libraries used by CTS. Thus, SymDroid passes all the tests that it could be expected to pass. We also measured SymDroid's performance, and found it is roughly 2x slower than the Dalvik virtual machine, and roughly 2x faster than a Java virtual machine. Note that in these experiments there was no symbolic computation—all values were concrete. Thus, these results suggest that SymDroid is likely fast enough in practice, especially since in our experience symbolic executors spend much of their time in the SMT solver.

Second, we used SymDroid to discover under what conditions certain privileged operations were used in PickContact, an Activity from the API demonstration app supplied with the Android SDK. This problem is a good fit for symbolic execution as the interaction between the user and the system is complex on Android, and determining whether a call is privileged can depend on subtle semantics. We ran SymDroid on the PickContact and found it was able to discover the correct conditions under which the READ_CONTACTS permission was used. (Section 2.4 describes our experimental results.)

In summary, the contributions of this chapter are

- A clean and concise core bytecode language, μ-Dalvik, to which Dalvik can be easily translated (Section 2.1), and which has a simple semantics (Section 2.2).
- A discussion of the issues of modeling the Android platform and other challenges in building SymDroid, a symbolic executor for μ-Dalvik (Section 2.3).
- Experiments demonstrating the correctness of SymDroid and suggesting how

it may be useful in practice (Section 2.4).

2.1 μ-Dalvik

Dalvik bytecode is designed to run in a resource constrained environment, namely on mobile devices. Among others, Dalvik is carefully designed to reduce the overall size of applications and for performance [15]. In contrast, we are interested in performing more expensive, off-device analyses, in particular symbolic execution. For easier maintenance and improvement, we also want to have as simple and concise a semantics as possible. μ-Dalvik represents our attempt to achieve these aims.

μ-Dalvik has three main differences compared to Dalvik:

- Dalvik includes many instruction variants that differ only in the number of bits reserved for operands. For example, consider three Dalvik move instructions, `move vA, vB`; `move/from16 vAA, vBBBB`; and `move/16 vAAAA, vBBBB`. These instructions all move values of the same size among registers; the only difference between them is how many bits they use to represent registers indices (`vA`, `vAA`, and `vAAAA` require 4, 8, and 16 bits, respectively). Since we are not constrained in terms of bytecode space representation, we instead always use 32-bit indices to refer to registers.

- Many Dalvik instructions encode their operand type in the operator. For example, to read an instance field, Dalvik includes opcodes `iget` (read an integer or float instance field), `iget-object` (read an Object instance field), `iget-boolean` (read a boolean instance field), `iget-byte` (read a byte instance

$$
\begin{array}{rcll}
P & ::= & \langle cls^*, fld^*, mtd^*, str^* \rangle & \text{DEX binary} \\
cls & ::= & \texttt{class}\ @s < @c\ \texttt{imp}\ @c^*\ \{@f^*\ @m^*\} & \text{Class definition} \\
fld & ::= & \texttt{field}\ @s : @c & \text{Field definition} \\
mtd & ::= & \texttt{method}\ @s : @c^* \rightarrow @c\ \{b\} & \text{Method definition} \\
b & ::= & \cdot \mid s\ ;\ b & \text{Method body}
\end{array}
$$

Figure 2.1: μ-Dalvik syntax (program).

field), *etc.* From the perspective of an analysis tool, we prefer to have one generic instruction of each kind, but allow the operand type to vary.

- Dalvik includes some complex instructions that μ-Dalvik desugars to simpler instruction sequences. For example, the `filled-new-array(/range)` and `fill-array-data` instructions fill the given array with the supplemental data. In SymDroid, these instructions are desugared into a sequence of μ-Dalvik instructions that copy constant bytes into the array.

2.1.1 μ-Dalvik Syntax

Figures 2.1 presents the syntax of μ-Dalvik programs, which are made up of definitions of classes, fields, and methods, and also contain a *string pool* mapping integer indices to string values. In full Dalvik, the string pool exist to make the bytecode compact by reusing strings across the entire codebase of an app; even such strings as class names, method names, and types are shared in the string pool. We maintain this indirect representation, and thus, in μ-Dalvik, all strings are accessed via their indices, which we write as $@c$ (class index), $@f$ (field index), $@m$ (method index), and $@s$ (program string index).

Class definitions contain the class name, its superclass, its implemented inter-

$s ::=$	$\texttt{goto}\ pc$	Unconditional branch
$\mid$	$\texttt{if}\ r \olessthan r\ \texttt{then}\ pc$	Conditional branch
$\mid$	$lhs \leftarrow rhs$	Move
$\mid$	$r \leftarrow r \oplus r$	Binary operation
$\mid$	$r \leftarrow \odot\ r$	Unary operation
$\mid$	$r \leftarrow \texttt{new}\ @c$	New instance
$\mid$	$r \leftarrow \texttt{newarray}\ @c[r]$	New array
$\mid$	$r \leftarrow (@c)\ r$	Type cast
$\mid$	$r \leftarrow r\ \texttt{instanceof}\ @c$	Instance of
$\mid$	$r.@m(argv)$	Dynamic method invocation
$\mid$	$@m(argv)$	Static method invocation
$\mid$	$\texttt{return}$	Method return
$\mid$	$r \leftarrow \texttt{sym}$	New symbolic variable
$\mid$	$\texttt{assert}\ r$	Assertion

Figure 2.2: μ-Dalvik syntax (statements).

faces, and its fields and methods. Field definitions are comprised of the field name and type. Finally, method definitions include the method name, argument types, return type, and method body.

A method body is a sequence of statements, which are defined in Figure 2.2. As execution progresses we maintain the *program counter pc*, which is the index of the currently executing statement in the sequence. (Note that in Dalvik, the program counter is a pointer to the bytecode instruction's offset, which can be slightly different as different bytecodes have different numbers of operands, and hence use different numbers of bytes.) As in many imperative languages, we distinguish the left- and right-hand side operands of move statements, whose definitions are described in Figure 2.3. On the left-hand side we allow a single register name; an array access; and instance and static field access with field index $@f$. Right-hand side operands of a move statement can have any left-hand side operands as well as constants.

$\oplus$	$::=$	$+ \mid - \mid \times \mid \div \mid \cdots$	Binary operators
$⧀$	$::=$	$< \mid > \mid \cdots$	Comparison operators
$\odot$	$::=$	$- \mid \neg \mid \cdots$	Unary operators
lhs	$::=$	r	Register
	$\mid$	$r[r]$	Array access
	$\mid$	$r.@f$	Instance fields
	$\mid$	$@f$	Static fields
rhs	$::=$	*lhs*	
	$\mid$	c	Constants
argv	$::=$	$\cdot \mid r,\ argv$	Arguments
c	$::=$	n	Integers
	$\mid$	$@s$	String indexes
	$\mid$	`true` | `false`	Booleans
	$\mid$	`null`	Null

Figure 2.3: μ-Dalvik syntax (operators, *etc.*).

Next, μ-Dalvik includes binary and unary operations. `new` and `newarray` statements create a new instance of $@c$ and an array of class $@c$, respectively. For array allocation, a register containing the array size is also required. μ-Dalvik also includes type cast and `instanceof`. Method calls refer to method index $@m$, and all arguments must be in registers; dynamically dispatched method calls also include a receiver object. Finally, μ-Dalvik includes a special statement to insert symbolic variables and an `assert` statement that checks a property of interest.

2.1.2 Translation from Dalvik to μ-Dalvik

Translating Dalvik bytecode into μ-Dalvik is a fairly straightforward process. Figure 2.4 illustrates the translation process from Java source code (left column) into Dalvik (middle column) and then into μ-Dalvik (right column). For the sake of clarity, we label key statements to represent program counters. The example code includes method call and return, array initialization, and various instructions that

Java source code	Dalvik instructions	μ-Dalvik instructions
`static byte foo(int x) {`	*parameter* `x = v2`	
	`const/16 v0 1000`	$r_0 \leftarrow 1000$
`if(x > 1000) {`	`if-le v2 v0 +9`	`if` $r_2 \leq r_0$ `then` ℓ_2
	`rem-int/lit16 v0 v2 1000`	$r_0 \leftarrow r_2\%1000$
`byte y = foo(x % 1000);`	`invoke-static v0` $@m_0$	$@m_0(r_0)$
	`move-result v0`	$r_0 \leftarrow r_{ret}$
`return y;`	`return v0`	$\ell_1 :$ $r_{ret} \leftarrow r_0$
		`return`
`}`	`const/4 v0 2`	$\ell_2 :$ $r_0 \leftarrow 2$
`byte [] data = {7, 9};`	`new-array v0 v0` $@c_0$	$r_0 \leftarrow$ `newarray` $@c_0[r_0]$
	`fill-array-data v0 +8`	$r_t \leftarrow 0; r_0[r_t] \leftarrow 7$
		$r_t \leftarrow 1; r_0[r_t] \leftarrow 9$
`byte z = data[x % 2];`	`rem-int/lit8 v1 v2 2`	$r_1 \leftarrow r_2\%2$
	`aget-byte v0 v0 v1`	$r_0 \leftarrow r_0[r_1]$
`return z;`	`goto -11`	`goto` ℓ_1
`}`	`[0: 7]`	*See fill-array-data*
	`[1: 9]`	*translation above*

$@c_0$ = byte array $@m_0$ = foo()

Figure 2.4: Translation example.

can be translated into simpler μ-Dalvik instructions. Note that μ-Dalvik's `return` statement does not have any operands; instead, there is a special register r_{ret} for holding method return values, so return values must be copied into r_{ret} before `return`, as depicted in the figure.

This example demonstrates all three of μ-Dalvik's key differences from Dalvik. First, we can see that `const/16` and `const/4`, which both load constant values into registers, are translated into the same μ-Dalvik instruction, and similarly for `rem-int/lit16` and `rem-int/lit-8`. Second, we can see that the `aget-byte` instruction is translated into μ-Dalvik's generic array access instructions; the other variants, such as `aget`, `aget-boolean`, *etc.*, would be translated similarly. Finally, this example shows how SymDroid translates the complex `fill-array-data` instruction, which loads an array appended to the end of the code segment, into a

$$
\begin{array}{rcll}
\ell & \in & \textit{Heap locations} & \\
x & \in & \textit{Symbolic variables} & \\
\pi, \phi & ::= & x \mid c \mid \odot\,\pi \mid \pi \oplus \pi \mid \pi \olessthan \pi & \text{Symbolic expressions} \\
\upsilon & ::= & c \mid \ell \mid \pi & \text{Values} \\
R & ::= & \{r \mapsto \upsilon, \ldots\} & \text{Register file} \\
L & ::= & pc, b, R & \text{Local state} \\
C & ::= & L \mid L :: C & \text{Call stack} \\
o & ::= & \langle @c;\ \{@f \mapsto \upsilon, \ldots\}\rangle & \text{Objects} \\
\alpha & ::= & @c[\upsilon, \ldots] & \text{Arrays} \\
\beta & ::= & o \mid \alpha & \text{Memory block} \\
H & ::= & \{\ell \mapsto \beta, \ldots\} & \text{Heap} \\
S & ::= & \{@f \mapsto \upsilon, \ldots\} & \text{Static field state} \\
\Sigma & ::= & \langle C, \phi, H, S\rangle & \text{Program state}
\end{array}
$$

Figure 2.5: Semantic domains.

sequence of multiple μ-Dalvik move instructions.

2.2 Symbolic Execution

In this section, we present a formalism for symbolic execution of μ-Dalvik and discuss our implementation in detail.

2.2.1 Domains

Figure 2.5 summarizes the domains used by our symbolic executor. There are three basic kinds of *values* υ used in the semantics: constants (defined in Figure 2.3), *heap locations* ℓ, and *symbolic expressions* π or ϕ, which are comprised of symbolic variables and constants combined with unary, binary, and relational operators.

As the symbolic executor runs, it maintains a *program state* Σ, which includes a call stack C, path condition ϕ, heap H, and static field state S. The *call stack* is a list of *local states* comprising a program counter, method body, and register file mapping registers to values. (Notice that each method gets its own registers, and

$\boxed{@c \leq_P @d}$

$$\textsc{SUBrefl}\ \frac{}{@c \leq_P @c} \qquad \textsc{SUBtrans}\ \frac{@b \leq_P @c \qquad @c \leq_P @d}{@b \leq_P @d}$$

$$\textsc{SUBsuper}\ \frac{}{@c \leq_P \mathsf{superclass}(P, @c)} \qquad \textsc{SUBitf}\ \frac{@d \in \mathsf{interface}(P, @c)}{@c \leq_P @d} \qquad \textsc{SUBarr}\ \frac{@c \leq_P @d}{@c\ \mathsf{array} \leq_P @d\ \mathsf{array}}$$

Figure 2.6: Subtyping.

hence these are used for local variables.) The top of the call stack is on the left, and represents the state of the currently executing method.

The state also contains a *path condition* ϕ, which records which conditional branches have been taken thus far. (For clarity, we will use ϕ to denote a symbolic expression that is a path condition, and π for other symbolic expression.)

The *heap* maps locations to *memory blocks* β, which are either *objects* o, which record their class and field values, and *arrays* α, which record the array type and the values in the array. Finally, the *static field state* is a mapping from static field names to their values.

In what follows, we will write $\Sigma.x$ for the x component of Σ, *e.g.*, $\langle C', \phi', H', S' \rangle.H = H'$. When we write $\Sigma.pc$, $\Sigma.b$, or $\Sigma.R$, we will mean those components of the current (top-most) local state in $\Sigma.C$. Similarly, we write $o.@c$ and $\alpha.@c$ for the class of an object or array type, respectively, and refer to object fields and array elements via $o[@f]$ and $\alpha[i]$, respectively. We also write Σ^+ to mean state Σ but with the program counter of the current local state incremented by one.

Figure 2.6 defines the usual Java subtype relation, which is the reflexive, transitive closure of the superclass and interface relations defined in the program. Note

$\boxed{\Sigma[\![rhs]\!] = v}$

$$\frac{\textsc{EReg}}{\Sigma[\![r]\!] = \Sigma.R[r]} \qquad \frac{\textsc{EStt}}{\Sigma[\![@f]\!] = \Sigma.S[@f]} \qquad \frac{\textsc{EConst}}{\Sigma[\![c]\!] = c}$$

$$\textsc{EArr}\ \frac{\ell = \Sigma[\![r_a]\!] \qquad \alpha = \Sigma.H[\ell] \qquad i = \Sigma[\![r_i]\!]}{\Sigma[\![r_a[r_i]]\!] = \alpha[i]} \qquad \textsc{EObj}\ \frac{\ell = \Sigma[\![r_o]\!] \qquad o = \Sigma.H[\ell]}{\Sigma[\![r_o.@f]\!] = o[@f]}$$

Figure 2.7: Evaluation of right-hand sides.

that Java allows covariant subtyping on arrays (SUBArr). This is statically unsound, and so Java dynamically tracks the type of each array and forbids writes of objects that are not subtypes of the contents type.

Finally, Figure 2.7 defines a convenience relation $\Sigma[\![rhs]\!] = v$ for evaluating the right-hand side of a move expression. These rules are straightforward: constants are evaluated to themselves, and registers, static fields, array accesses, and field accesses are evaluated based on the state Σ.

2.2.2 Semantics

Figures 2.8 and 2.9 give the symbolic semantics for μ-Dalvik statements, which prove judgments of the form $\langle \Sigma, s \rangle \Downarrow_P \Sigma'$, meaning in program P, starting in state Σ, statement s updates the state to Σ'. The rules are mostly standard.

The rule SEgoto updates the program counter unconditionally. Rules SEif-true and SEif-false model conditional branches. Here SAT(ϕ) asserts that ϕ is satisfiable. In SEif-true, we evaluate the guard and conjoin it with the current path condition. If the resulting path condition is satisfiable, it means the true branch is feasible, so we can branch to the specified program counter, and we update

$\boxed{\langle \Sigma, s \rangle \Downarrow_P \Sigma'}$

$$\textsc{SEgoto}\quad \frac{}{\langle \Sigma, \texttt{goto}\ pc' \rangle \Downarrow_P \Sigma[pc \mapsto pc']}$$

$$\textsc{SEif-true}\quad \frac{\pi = (\Sigma[\![r_1]\!] \olessthan \Sigma[\![r_2]\!]) \qquad \phi_t = \pi \wedge \Sigma.\phi \qquad \mathrm{SAT}(\phi_t)}{\langle \Sigma, \texttt{if}\ r_1 \olessthan r_2\ \texttt{then}\ pc_t \rangle \Downarrow_P \Sigma[\phi \mapsto \phi_t,\ pc \mapsto pc_t]}$$

$$\textsc{SEif-false}\quad \frac{\pi = \neg\,(\Sigma[\![r_1]\!] \olessthan \Sigma[\![r_2]\!]) \qquad \phi_f = \pi \wedge \Sigma.\phi \qquad \mathrm{SAT}(\phi_f)}{\langle \Sigma, \texttt{if}\ r_1 \olessthan r_2\ \texttt{then}\ pc_t \rangle \Downarrow_P \Sigma^+[\phi \mapsto \phi_f]}$$

$$\textsc{SEmove-reg}\quad \frac{v = \Sigma[\![rhs]\!] \qquad R' = \Sigma.R[r \mapsto v]}{\langle \Sigma, r \leftarrow rhs \rangle \Downarrow_P \Sigma^+[R \mapsto R']} \qquad \textsc{SEmove-static-fld}\quad \frac{v = \Sigma[\![rhs]\!] \qquad S' = \Sigma.S[@f \mapsto v]}{\langle \Sigma, @f \leftarrow rhs \rangle \Downarrow_P \Sigma^+[S \mapsto S']}$$

$$\textsc{SEmove-inst-fld}\quad \frac{v = \Sigma[\![rhs]\!] \qquad l = \Sigma[\![r_o]\!] \qquad o = \Sigma.H[l] \qquad H' = \Sigma.H[l \mapsto o[@f \mapsto v]]}{\langle \Sigma, r_o.@f \leftarrow rhs \rangle \Downarrow_P \Sigma^+[H \mapsto H']}$$

$$\textsc{SEmove-arr}\quad \frac{\begin{array}{c} v = \Sigma[\![rhs]\!] \qquad l = \Sigma[\![r_a]\!] \qquad \alpha = \Sigma.H[l] \\ \alpha.@c = @c'\ \mathsf{array} \qquad v.@c \leq @c' \qquad i = \Sigma[\![r_i]\!] \qquad H' = \Sigma.H[l \mapsto \alpha[i \mapsto v]] \end{array}}{\langle \Sigma, r_a[r_i] \leftarrow rhs \rangle \Downarrow_P \Sigma^+[H \mapsto H']}$$

$$\textsc{SEbop}\quad \frac{v = (\Sigma[\![r_{s_1}]\!] \oplus \Sigma[\![r_{s_2}]\!]) \qquad R' = \Sigma.R[r_d \mapsto v]}{\langle \Sigma, r_d \leftarrow r_{s_1} \oplus r_{s_2} \rangle \Downarrow_P \Sigma^+[R \mapsto R']} \qquad \textsc{SEuop}\quad \frac{v = \odot\,(\Sigma[\![r_s]\!]) \qquad R' = \Sigma.R[r_d \mapsto v]}{\langle \Sigma, r_d \leftarrow \odot\ r_s \rangle \Downarrow_P \Sigma^+[R \mapsto R']}$$

Figure 2.8: Symbolic semantics for μ-Dalvik statements.

the path condition. SEif-false is similar, permitting fall-through if the guard is satisfiable. Notice that these rules may be simultaneously valid, hence we have non-determinism in the semantics. As is standard in symbolic execution, we can choose whatever heuristics we like to decide whether to explore zero, one, or both possible branches.

Rules SEmove-reg evaluates the right-hand side subexpression and then updates the current register file. Rules SEmove-arr, SEmove-inst-fld, and SEmove-static-field are analogous, updating the appropriate array element, instance field, or static field. Rule SEmove-arr checks whether the given value is a subtype of the contents type, as mentioned earlier. Rules SEbop and SEuop compute a binary or unary expression and store the results in the appropriate register.

Rule SEnew-obj allocates a new object in the heap, giving it the appropriate class and an empty set of fields. Note that we do not call a constructor here—Dalvik bytecode will contain an explicit call to method <init> to initialize any object fields. Rule SEnew-arr is analogous, initializing the array elements with null values. Notice here we require that the type passed to `newarray` is an array type, which is also required in Dalvik [8]. Rules SEcast and SEinstance-of check subtype relations defined in Figure 2.6, and either allow the cast or return the appropriate boolean value. Note that, for simplicity, we do not model exceptions in these semantics; hence a failed cast is simply not allowed, rather than raising an exception.

Rules SEcall-static, SEcall-dyn and SEreturn model method call and return. Both method call rules look up the appropriate method, in the dynamic case from the receiver object. We omit the definition of lookup, which is standard. The Dalvik virtual machine conforms to the ARM architecture's calling convention, in which the caller and callee share part of their register files; thus, the caller passes arguments by setting the appropriate range of registers. We do the same in μ-Dalvik, to make the translation from Dalvik to μ-Dalvik simple. We assume here the lookup function returns the first register r_i that should be set as a parameter. In dynamic dispatch, that first register is set to the receiver object. In both cases we advance the current program counter (so that return will continue at the correct instruction)

$\boxed{\langle \Sigma, s \rangle \Downarrow_P \Sigma'}$

$$\textsc{SEnew-obj}\ \frac{o = \langle @c, \emptyset \rangle \qquad \ell \notin dom(\Sigma.H) \qquad H' = \Sigma.H[\ell \mapsto o] \qquad R' = \Sigma.R[r_o \mapsto \ell]}{\langle \Sigma, r_o \leftarrow \texttt{new}\ @c \rangle \Downarrow_P \Sigma^+[H \mapsto H'][R \mapsto R']}$$

$$\textsc{SEnew-arr}\ \frac{\begin{array}{c} j = \Sigma[\![r_i]\!] \qquad \alpha = \langle @c\ \mathsf{array}, [\overbrace{\texttt{null}, \ldots}^{j}] \rangle \\ \ell \notin dom(\Sigma.H) \qquad H' = \Sigma.H[\ell \mapsto \alpha] \qquad R' = \Sigma.R[r_a \mapsto \ell] \end{array}}{\langle \Sigma, r_a \leftarrow \texttt{newarray}\ @c\ \mathsf{array}[r_i] \rangle \Downarrow_P \Sigma^+[H \mapsto H'][R \mapsto R']}$$

$$\textsc{SEcast}\ \frac{\beta = \Sigma[\![r_s]\!] \qquad \beta.@c \leq_P @c' \qquad R' = \Sigma.R[r_d \mapsto \beta]}{\langle \Sigma, r_d \leftarrow (@c')\ r_s \rangle \Downarrow_P \Sigma^+[R \mapsto R']}$$

$$\textsc{SEinstance-of}\ \frac{\beta = \Sigma[\![r_s]\!] \qquad R' = \Sigma.R[r_d \mapsto (\beta.@c \leq_P @c')]}{\langle \Sigma, r_d \leftarrow r_s\ \texttt{instanceof}\ @c' \rangle \Downarrow_P \Sigma^+[R \mapsto R']}$$

$$\textsc{SEcall-static}\ \frac{\begin{array}{c} b_m, r_i = \mathsf{lookup}(P, @m) \\ R' = \{r_i \mapsto \Sigma[\![r_1]\!], \ldots, r_{i+n-1} \mapsto \Sigma[\![r_n]\!]\} \\ C' = \langle 0, b_m, R' \rangle :: \Sigma.C \end{array}}{\langle \Sigma, @m(r_1, \ldots, r_n) \rangle \Downarrow_P \Sigma^+[C \mapsto C']} \qquad \textsc{SEreturn}\ \frac{\begin{array}{c} C :: C' = \Sigma.C \\ R' = C'.R[r_{ret} \mapsto \Sigma[\![r_{ret}]\!]] \end{array}}{\langle \Sigma, \texttt{return} \rangle \Downarrow_P \Sigma[C \mapsto C'][R \mapsto R']}$$

$$\textsc{SEcall-dyn}\ \frac{\begin{array}{c} \ell = \Sigma[\![r_{this}]\!] \qquad o = \Sigma.H[\ell] \qquad b_m, r_i = \mathsf{lookup}(P, @m, o) \\ R' = \{r_i \mapsto \ell, r_{i+1} \mapsto \Sigma[\![r_1]\!], \ldots\} \qquad C' = \langle 0, b_m, R' \rangle :: \Sigma.C \end{array}}{\langle \Sigma, r_{this}.@m(r_1, \ldots, r_n) \rangle \Downarrow_P \Sigma^+[C \mapsto C']}$$

$$\textsc{SEsym}\ \frac{\mathit{fresh}(x) \qquad R' = \Sigma.R[r \mapsto x]}{\langle \Sigma, r \leftarrow \texttt{sym} \rangle \Downarrow_P \Sigma^+[R \mapsto R']} \qquad \textsc{SEassert}\ \frac{\neg \mathrm{SAT}(\neg \Sigma[\![r]\!])}{\langle \Sigma, \texttt{assert}\ r \rangle \Downarrow_P \Sigma^+}$$

Figure 2.9: Symbolic semantics for μ-Dalvik statements (cont'd).

and push another frame onto the call stack. Rule SEreturn models return, which copies the value from a special return register r_{ret} from the callee back to the caller, and pops the call stack.

Finally, the last two rules are for symbolic execution. The rule SEsym introduces a fresh symbolic variable, and SEassert checks that the argument assert is always true (*i.e.*, that its negation is not satisfiable).

$$\textsc{SEseq}\quad \frac{\langle \Sigma, s\rangle \Downarrow_P \Sigma_s \qquad s' = \Sigma_s.b[\Sigma_s.pc] \qquad \langle \Sigma_s, s'\rangle \Downarrow_P \Sigma_n}{\langle \Sigma, s\rangle \Downarrow_P \Sigma_n}$$

$$\textsc{SEprogram}\quad \frac{L = 0, b_{drv}, \emptyset \qquad \Sigma_{init} = \langle L, \mathtt{true}, \emptyset, \emptyset\rangle \qquad s = b_{drv}[0] \qquad \langle \Sigma_{init}, s\rangle \Downarrow_P \Sigma_f}{\vdash P \Rightarrow \Sigma_f}$$

Figure 2.10: Symbolic execution for μ-Dalvik

Based on the rules in Figures 2.8 and 2.9, rules in Figure 2.10 define how to execute μ-Dalvik program symbolically. The rule SEseq executes the current statement s, and moves to the next designated statement. Finally, the rule SEprogram explains how to execute μ-Dalvik program symbolically. Note that, unlike general programs, Android applications do not have an explicit entry point, such as `main` function. Thus, we will execute the app via "driver" codes that literally drive the app as desired. After initializing all the elements inside the state, the rule SEprogram starts the symbolic execution by running driver codes.

Additional Instructions. Our formalism includes almost every feature in our implementation (and thus almost every feature of Dalvik), except for two. First, we omitted Dalvik's `array-length` instruction; SymDroid includes the same instruction, and its semantics is straightforward to implement. (Using a separate instruction rather than making it a unary operator was an arbitrary choice.)

Second, we omitted exception handling and propagation from our formalism, but these can be supported with some minor tedium: First, we need to attach exception handlers to method definitions, and add a rule that searches for an appropriate

```
module IntMap = Map.Make(int)
type value_ =                                       (* υ *)
  | Const of const                                  (* c *)
  | Loc   of loc                                    (* ℓ *)
  | Sym   of SMT.exp                                (* π *)
type regs = value_ IntMap.t                         (* R *)
type l_state = pc * instr list * regs               (* L *)
type c_stack = l_state list                         (* C *)
type block =                                        (* β *)
  | Obj of (id_c * value_ IntMap.t)                 (* o *)
  | Arr of (id_c * value_ IntMap.t)                 (* α *)
type heap = block IntMap.t                          (* H *)
type static = value_ IntMap.t                       (* S *)
type state = c_stack * SMT.exp * heap * static      (* Σ *)
```

Figure 2.11: Implementation of semantic domains.

handler and changes the control-flow accordingly when an exception is raised. Second, for the case when an exception is raised but there is no handler, we need a rule to propagate that exception to the caller. SymDroid includes both of these features and the corresponding instruction `throw`. Recall that, compared to more than 200 Dalvik bytecode instructions, μ-Dalvik has just 16 instructions, which are made up of 14 instructions shown in the syntax as well as `array-length` and `throw` instructions.

2.2.3 Implementation

Figures 2.11, 2.13, and 2.14 sketch our implementation, which follows the formal system very closely. Figure 2.11 shows the OCaml definitions matching the formal semantic domains from Figure 2.5; we omit some definitions of primitive types such as `pc` *etc.* Notice that the representation of symbolic expressions comes from the SMT solver (type `SMT.exp`).

```
val deref_H : state → loc → block
val adv_pc : state → state
val upd_pc : state → pc → state
val upd_R : state → reg → value_ → state
val upd_H : state → loc → block → state
val upd_o : block → id_f → value_ → block
```

Figure 2.12: Interfaces of utility functions.

Figure 2.12 lists the types of some utility functions whose names are self explanatory. For instance, deref_H is used to retrieve a memory block in the heap; adv_pc advances the program counter; and upd_R updates the register file.

Then, Figure 2.13 gives a partial definition of the step function, which corresponds to the $\langle \Sigma, s \rangle \Downarrow_P \Sigma'$ relation in Figures 2.8 and 2.9. The input to step is a Dalvik bytecode file (of type dex), a program state, and a bytecode instruction, and the output is a pair containing a state and a state option; the latter is None in all cases except at a conditional branch when both branches are possible.

We give code for a few of the instruction handlers, for illustration. The first case, for Mu_move, evaluates the right-hand side and then updates the state appropriately depending on whether the left-hand side is a Register (SEmove-reg), an instance field (SEmove-inst-fld), and so on. The second case, for Mu_sym, gets a fresh symbolic variable from the SMT solver and updates the corresponding register. The last case, Mu_if, checks satisfiability of the guard and the negated guard cojoined with the current path condition, and then returns the state updated with the new pc(s). Notice in the code for Mu_if, the last match case, when both branches are unsatisfiable, should never occur unless there is a bug in SymDroid.

```
let step (p: dex) (st: state): state * state option = function
  | Mu_move (lh, rh) →
    let v = eval st rh in
      ( match lh with
        | Register r → adv_pc (upd_R st r v)
        | InstFld (ro, f) →
          let l = eval st ro in
          let o = deref_H st l in
          let o' = upd_o o f v in
            adv_pc (upd_H st l (Obj o')), None
        | ... )
  | Mu_sym r →
    let x = SMT.fresh_var () in
      adv_pc (upd_R r (Sym x)), None
  | Mu_if (r1, cmp, r2, pc) →
    let v1::v2::[] = List.map (fun r → eval st r) [r1; r2] in
    let pi_t :: pi_f ::[] = ... in
    let sat :: n_sat ::[] = List.map (fun pi → SMT.query pi) [pi_t; pi_f] in
      ( match sat, n_sat with
        | true, true → upd_pc st pc, Some (adv_pc st)
        | true, false → upd_pc st pc, None
        | false, true → adv_pc st, None
        | false, false → raise Infeasible )
  | ...
```

Figure 2.13: Implementation of symbolic semantics.

Finally, Figure 2.14 shows the function vm that orchestrates the whole symbolic execution process. It maintains a (mutable) queue worklist of states to explore. After adding the initial state (which sets the pc to the beginning of the code passed as drv), the vm function repeatedly picks a state off the worklist, single-steps it, and then updates the worklist with the resulting state(s).

Notice that this implementation explores *all* possible program paths; in practice we must carefully limit the use of the Mu_sym instruction so that full path exploration is feasible. On the other hand, it would be very easy to modify this driver to include heuristics for exploring a subset of paths [17, 34, 55].

Note also one other important design decision here: The state of the symbolic

```
val worklist : state Queue.t

let vm (p: dex) (drv: mu_instr list ) =
  let local_st = 0, drv, IntMap.empty in
  let init_st =
     local_st , SMT.true, IntMap.empty, IntMap.empty in
  Queue.add init_st worklist ;
  while not (Queue.is_empty worklist ) do
    let st = Queue.pop worklist in
    let ins = Dex.get_ins dx st.pc in
    let st1, so = step st ins in
    Queue.add st1 worklist ;
    match so with Some st2 → Queue.add st2 worklist | _ → ()
  done
```

Figure 2.14: Implementation of symbolic execution driver.

executor is fully captured in state, which is a purely functional data structure. This makes path exploration very easy, since we can explore executions in any order. In contrast, symbolic executors that actually run the program under test on the underlying system [33, 75] must be careful that side effects from different executions do not interfere with each other (see Section 3.10 for more discussion).

2.3 Manual Model of Android

To symbolically execute Android apps, we not only need to model each bytecode instruction, but we also need to model the platform that apps run on top of. Modeling the platform is challenging even for C programs [17, 88], but in our opinion it is even harder for Android, as there are many system libraries; the platform itself is quite large and complex; apps have several different entry points; and the interaction with Android is quite involved, with various layers of callbacks.

Thus, in our preliminary work, we manually implemented only as much of a model as we need to carry out our particular case study (Section 2.4). The challenge of more fully modeling Android is the motivation of synthesizing a model of frameworks (Chapter 3). To motivate that challenge, we briefly explain the three main portions of our current model: system libraries, system services and views, and the component lifecycle. Full details are discussed in a technical report [43].

2.3.1 System Libraries

On Android, third-party libraries are statically linked with apps, but system libraries and the Java standard libraries are loaded at run time to reduce app size. Thus, SymDroid includes the ability to add “hooks” in for invocation rules that are invoked when the target method body is not in the app code (and thus it must be dynamically linked, assuming type safety). These hooks then transfer control to our manual model of Android that implements the desired functionality.

We found that two of the most important system classes to model are Intent and Bundle, which are used to pass information between the system and an app, and between components of an app; SymDroid includes special internal support for both classes. In more detail, a Bundle is essentially a mapping from arbitrary string keys to values, and it is up to the sender and receiver of a Bundle to agree on the meaning of any particular element in the mapping. SymDroid stores Bundle keys and values in the field map for the Bundle object. Intents are used to specify component names to launch. Intents may also include extra Bundle-style key-value mappings, *e.g.*, added with intent.putExtra(“aa”, v_1). As with Bundles, we add those mappings directly to

```
String name = Context.LOCATION_SERVICE;
LocationManager lm =
   (LocationManager)getSystemService(name);
Location l = lm.getLastKnownLocation(...);
```

(a) System services.

```
setContentView(R.layout.start);
Button b =
   (Button)findViewById(R.id.startButton);
b.setOnClickListener(...);
```

(b) View object via findViewById.

Figure 2.15: Code snippets that retrieve runtime instances.

the field set of an Intent object. In addition to the above two classes, SymDroid has a partial model for the basic components of Android, including Context, View, and Activity. Additionally, SymDroid currently includes partial support for several commonly used Java libraries, including String, StringBuilder, Object, Class, and Integer. For those methods that are not modeled yet, SymDroid returns symbolic values with the return type of the invoked method. (Although method bodies for system libraries are not included in the bytecode, their signatures are declared in it, thus we can retrieve return types.)

2.3.2 Runtime Instances

In the process of building a model of the Android platform, we found that several key methods in Android return a variety of different object types, depending on their arguments. We also found that such instance retrievals from the environment have a common pattern: returning an instance of generic types and downcasting before using it. Figure 2.15 shows such usage patterns for system services and runtime View objects on the screen. System services are associated with unique service names, and View objects are distinguishable by their distinct ids. Thanks to these features, we can easily support such on-demand runtime instances: we extend state Σ to have two additional components for system services and View objects; generate an instance on demand; and make a mapping from a service name or a resource id to the newly generated instance. SymDroid keeps consistency by returning the same instance when it is accessed later.

2.3.3 Component Lifecycle

Android apps run under quite a different model than standard desktop applications. Rather than have a `main` method at which execution begins, Android apps instead declare (in an XML "manifest" file) which components respond to which Intents, and apps begin execution at these points when the system's ActivityManager receives a corresponding Intent. These Intents could be sent from another app (*e.g.*, apps often use this feature to launch the web browser to show a particular web page) and are even sent when starting an app from the home screen: tapping an app's icon sends an Intent to the app's launcher activity.

Moreover, even once an app is launched, apps are largely event-driven. Apps dynamically register various event handlers (*e.g.*, for GUI events or for handling additional Intents), and control flow alternates between app code and the system's event dispatch loop. This is again in contrast to more standard, non-reactive systems.

For symbolic execution purposes, we need to model all of this behavior. We chose to use *client-oriented specifications* [39] (co-spec) to model the system side of an app's execution. It is up to the developer to write such specifications so that they drive the app under test as desired. In our preliminary work, we manually wrote

```
67 class Driver {
68     public static void main(String [] args) {
69         String comp = "Lcom/.../PickContact;";
70         Object o = Mock.__new__(comp);
71         Mock.__invoke__(o, "onCreate", null );
72         Mock. __click_rand__ ();
73     }
74     // android.app. Activity . startActivityForResult (...)
75     public static void startActivityForResult (Object receiver , Intent i , int req) {
76         ... // the designated Activity is invoked
77         Object res = Mock.__new_sym__(''res'' );
78         Mock.__invoke__( receiver , " onActivityResult" , req , res , i );
79     }
80     // android.content.ContentResolver.query (...)
81     public static Cursor query(Object receiver , Uri uri , ...) {
82         String contacts = "content://com.android.contacts";
83         assert (! uri .getPath(). startsWith(contacts ));
84         ... // invokes the corresponding system API
85 }   }
86 class Mock {
87     public static Object __new__(String ty) { }
88     public static Object __invoke__ (Object _this , String mtd, Object ... args) { }
89     public static void __click_rand__ () { }
90     public static Object __new_sym__(String var) { }
91      ...
92 }
```

Figure 2.16: Example client-oriented specification.

drivers as well as properties of interest, and we propose to automatically generate them via program synthesis in Section 5.1.1 and Section 5.1.2, respectively.

For example, consider the specification code in Figure 2.16. This code defines a class Driver with a main() method; SymDroid uses this as the entry point for symbolic execution. The main() method first launches the PickContact activity of the app under test (lines 69–71) by calling its onCreate method [10]. In turn, this method registers several callbacks for button clicks, and then passes control back to the system.

Now SymDroid continues with Driver.main(), which clicks on a non-deterministically chosen button (line 72). This is a symbolic execution branch point, where symbolic execution will fork for all possible button clicks. A simulated button click turns into a callback to the handler that was registered for that button.

Afterward, system-supplied methods will be treated specially by SymDroid. If those methods are re-defined in Driver, *e.g.*, startActivityForResult() and query() in Figure 2.16, SymDroid passes control to Driver code, where we can, again, simulate the component lifecycle (line 78) or check properties of interest, *e.g.*, that the particular query was not for contacts (line 83). Otherwise, as we discussed earlier, the control goes to the modeled APIs inside SymDroid.

Notice that in the example, Driver performs invocations using class Mock, which has various unimplemented methods. This class is specially recognized by SymDroid, which ignores Mock method bodies and instead performs the action specified by the method name, *e.g.*, creating a new instance of the given type string, invoking a method, *etc.* We use Mock rather than calling app methods directly because doing the latter would require linking against the app code, which would be complicated because we expect SymDroid may often be used without direct access to app source code.

2.4 Preliminary Experiments

We performed two kinds of experiments to evaluate SymDroid. First, we ran SymDroid against the Android Compatibility Test Suite (CTS) [7], which tests whether a Dalvik virtual machine implementation is correct. Our results suggest that Sym-

Droid's translation to μ-Dalvik and semantics thereof are correct. Second, we used SymDroid to determine the conditions under which certain privileged system calls would be invoked by a chosen activity in a target app. This case study, while preliminary, demonstrates how SymDroid might be used in practice.

2.4.1 Compatibility Test Suite

We ran SymDroid against the Compatibility Test Suite version 4.0, which contains 93 test cases. We found that SymDroid passes 26 of the test cases. We manually inspected the failing test cases and concluded that all of them were due to unimplemented system libraries (recall we only implemented as much of Android as needed for our case study). Thus, despite the seemingly low coverage, SymDroid passed all of the CTS tests it could be expected to pass without a complete system model. We leave implementing the remaining libraries (including reflection, various I/O Streams and Buffers, the System class, and several others) as future work.

Next, we compared the performance of SymDroid (compiled to native code with OCaml version 3.12.1) to a Java virtual machine (Java 1.6.0_33) and a Dalvik virtual machine (the Dalvik VM from the Android source branch 4.0.4 as of July 2, 2012). Figure 2.17 summarizes the results for the 26 test cases that passed. For each test case, the figure lists its size (in terms of its Java source code), the size of the corresponding Dalvik bytecode file, and its number of Dalvik bytecode instructions. The next three columns report the test case's running time on the DVM, SymDroid, and JVM. The reported performance is the average of ten runs on a 1.8 GHz Intel

		DEX	#	Time (s)		
Name	LoC	(B)	Ins	DVM	SymDroid	JVM
005-args	20	2,004	121	0.066	0.139 (2.1x)	0.257 (3.9x)
006-count10	8	720	10	0.072	0.124 (1.7x)	0.261 (3.6x)
007-exceptions	25	1,232	26	0.068	0.133 (2.0x)	0.249 (3.7x)
008-instanceof	63	2,684	102	0.070	0.122 (1.7x)	0.292 (4.2x)
009-instanceof2	59	2,380	64	0.069	0.154 (2.2x)	0.259 (3.8x)
012-math	78	2,696	382	0.062	0.120 (1.9x)	0.263 (4.2x)
013-math2	10	940	15	0.064	0.128 (2.0x)	0.261 (4.1x)
015-switch	80	2,576	217	0.065	0.126 (1.9x)	0.249 (3.8x)
017-float	14	1,212	53	0.065	0.126 (1.9x)	0.260 (4.0x)
019-wrong-array-type	13	960	18	0.066	0.124 (1.9x)	0.249 (3.8x)
022-interface	52	2,080	50	0.077	0.121 (1.6x)	0.302 (3.9x)
026-access	14	952	15	0.063	0.115 (1.8x)	0.248 (3.9x)
029-assert	12	1,276	29	0.066	0.121 (1.8x)	0.255 (3.9x)
034-call-null	14	1,188	28	0.072	0.128 (1.8x)	0.279 (3.9x)
038-inner-null	24	1,680	31	0.066	0.123 (1.9x)	0.251 (3.8x)
040-miranda	58	2,612	151	0.063	0.125 (2.0x)	0.289 (4.6x)
043-privates	33	1,816	105	0.061	0.123 (2.0x)	0.247 (4.0x)
047-returns	46	1,868	83	0.065	0.121 (1.9x)	0.263 (4.0x)
052-verifier-fun	90	2,276	80	0.067	0.124 (1.9x)	0.262 (3.9x)
056-const-string-jumbo	6	1,158K	17	0.069	3.207 (46x)	0.248 (3.6x)
076-boolean-put	20	1,580	31	0.063	0.118 (1.9x)	0.251 (4.0x)
081-hot-exceptions	23	1,688	45	0.066	0.129 (2.0x)	0.284 (4.3x)
085-old-style-inner-class	25	2,120	87	0.067	0.121 (1.8x)	0.255 (3.8x)
090-loop-formation	31	1,488	94	0.067	0.494 (7.1x)	0.280 (4.0x)
091-deep-interface-hierarchy	48	5,396	10	0.070	0.122 (1.7x)	0.319 (4.6x)
095-switch-MAX_INT	9	964	12	0.067	0.121 (1.8x)	0.259 (3.9x)

Figure 2.17: Results for Android compatibility test suite.

Core i7 with 2 GB RAM, running 64-bit Ubuntu 12.04.

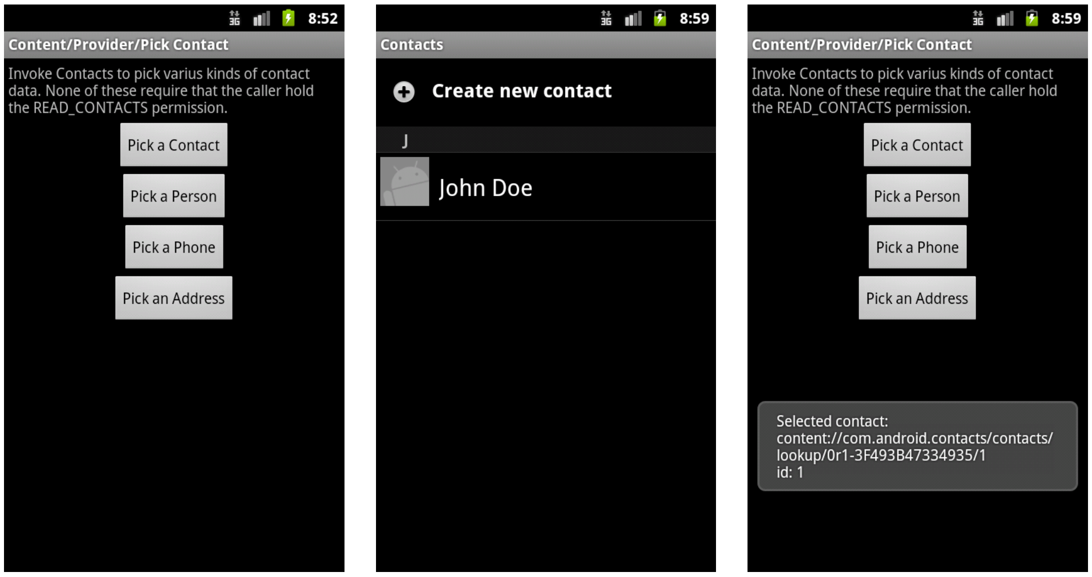
In almost every case, the DVM is the fastest, followed by SymDroid (about twice as slow), followed by the JVM (another factor of two slower). The one exception to this trend is 056-const-string-jumbo, for which SymDroid is dramatically slower than either the DVM or JVM. We investigated this further, and found that SymDroid's core is very fast in this case, and what is taking most of the time is unpacking the apk (which is extremely large). The DVM and JVM take a .jar file as input, and apparently need not pay the same cost. Nonetheless, SymDroid is surprisingly fast, and we expect its performance to be adequate in practice, especially as SymDroid will be run on desktop machines that are much more powerful than the mobile devices the DVM would more typically be run on.

2.4.2 Case Study: Finding Privileged Calls

There are many possible ways to use SymDroid, as the literature on symbolic execution in general suggests [53, 14, 41, 66, 33, 75, 17, 34, 70]. To get a sense for how SymDroid might be used in practice, we applied it to the problem of discovering under what conditions various privileged system calls could be made.

In more detail, Android's security model includes *permissions* that protect sensitive platform APIs, such as access to the Internet, telephony, GPS, and so on. At app installation time, the user is presented with the set of permissions requested by an app. The user can then decide to proceed with installation, in which case all permissions are granted to the app; or the user can abort installation. While this model shows the user *what* permissions apps request, it does not explain *why* those permissions are needed, and under what circumstances they will be used. With SymDroid, however, we can find this information out.

For purposes of this initial study, we decided it was particularly convenient to analyze an app whose source code was available. Thus, we elected to study the Android API demonstration app, which is included in the Android SDK [9]. We looked in detail at one of this app's activities (an Activity essentially corresponds to a screen shown to the user): PickContact, which lets the user select a single contact from the contacts database on the phone.



Invoke Contacts to pick varius kinds of contact data. None of these require that the caller hold the READ_CONTACTS permission.

Selected contact:
content://com.android.contacts/contacts/lookup/0r1-3F493B47334935/1
id: 1

Figure 2.18: Sequence of screens in the PickContact Activity.

PickContact Figure 2.18 shows a sequence of screenshots from PickContact.[1] On the left is the initial screen displayed when PickContact is launched within the demo app. The user is presented with four choices to filter the set of contacts that will be shown—any contact, those that are for a person, those with a phone number, or those with an address. In this case, we clicked on the *Pick a Contact* button. The app then sends an Intent to the standard Android contacts app, which launches that app (if it is not already running) and brings up the contact picker window, shown in the middle screenshot. We click to select a contact, and then control passes back to PickContact, which displays the URI for the selected contact on screen.

We wanted to use SymDroid to investigate under what conditions PickContact's READ_CONTACTS permission was used in this sequence of events. Somewhat confusingly, it is *not* used when the contact picker is launched, as that is done in a

[1]The misspellings in the screenshots are that way in the app source code.

```
public class PickContact extends Activity {
  class ResultDisplayer implements OnClickListener {
    String mMimeType;
    ResultDisplayer (String msg, String mimeType) {
      mMimeType = mimeType;
    }
    public void onClick(View v) {
      Intent intent = new Intent(Intent.ACTION_GET_CONTENT);
      intent .setType(mMimeType);
      ...
      startActivityForResult (intent , 1);
  } }
  @Override
  protected void onCreate (Bundle saved) {
    ((Button)findViewById(R.id. pick_contact )). setOnClickListener (
        new ResultDisplayer (‘‘ Selected contact’’ ,
            ContactsContract.Contacts.CONTENT_ITEM_TYPE));
    // set three more call−back listeners
  }
  @Override
  protected void onActivityResult (int req , int res , Intent data) {
    if (data ≠ null) {
      Uri uri = data.getData();
      if (uri ≠ null) {
        Cursor c = getContentResolver().query (...);
        ...
} } } }
```

Figure 2.19: PickContact source code (excerpt).

different app on the phone, which runs in its own process and has its own separate set of permissions. Thus, for example, if the user gets to the contact picker but then clicks the back button, PickContact will not try to read any contact information. The permission will only be used if the user actually selects a contact, in which case PickContact will query the contacts database. (The id returned from querying the contacts database is shown in the right screenshot in Figure 2.18.)

Figure 2.19 gives a portion of the source code for the PickContact activity. Recapping some of the earlier discussion, when this activity is started, its onCreate()

method on line 106 is called. This method sets callbacks for the four buttons shown in the left screenshot in Figure 2.18; the code for setting one callback is shown on lines 107–109. In this case, the callbacks are instances of the ResultDisplayer class parameterized by the mime type of the contacts to select.

When a button is clicked, the corresponding callback is invoked, in this case calling the onClick() method on lines 99–104. This method then creates an Intent for the contact picker app (the Intent kind is specified on line 100) and launches it on line 103. When this returns, the system automatically calls onActivityResult() of the Intent sender (line 113), which then performs the query call (line 115) that actually requires the READ_CONTACTS permission. Notice that this call will not occur if no contact is selected (*e.g.*, if the user clicked the back button), as in that case data will be null. (The uri null check is an extra sanity check; that should always be non-null.)

We ran SymDroid against this program using the co-spec in Figure 2.16. Recall that in this case, there are four symbolic variables: three for onActivityResult parameters (req, res, data) and one for information retrieved from another symbolic variable (uri). SymDroid explored a total of 16 different paths, and 4 of them included a privileged call that used READ_CONTACTS:

```
privilege call :
  android.content.ContentResolver→ query
    requires READ_CONTACTS
    where NOT(sym3 = 0x0) AND NOT(sym3.getData = 0x0)
```

We can see that the only path triggering the condition is along the path when neither the data (corresponding to sym3) nor uri (corresponding to sym3.getData, as it was derived by calling getData on sym3) fields are null. This corresponds to the case when the user did not close the contact picker without selecting a contact, and the contact they picked does indeed exist in the phone's database. The path condition does *not* include the check indicating that asserts the path begins with the URI specific to the contacts database. However, while this path condition is asserted, the co-spec uses only concrete instances of strings, rather than symbolic strings (as SymDroid currently does not support symbolic strings). We verified manually that this is the correct set of path conditions leading to privileged calls in this example.

Over all paths, SymDroid executed a total of 4,462 μ-Dalvik instructions, which included 54 system calls that were hooked specially by SymDroid. The average of ten runs on the same machine on which CTS tests were conducted is 30.93 seconds. This running time shows, again, that SymDroid is fast enough to analyze real apps.

2.5 Literature Review

Concolic vs. Pure. Symbolic executors can roughly be divided into two kinds. The first kind, so-called *concolic* executors, perform symbolic execution at program run time by shadowing underlying concrete system values with symbolic expressions [33, 75]. When faced with a call to unavailable (library or system) code, concolic executors can simply call the actual external code with the underlying concrete values, extending the path condition to constrain the corresponding symbolic expression to equal the concrete value.

The second kind of symbolic executors are "pure" in the sense that they do not directly execute the subject program on the underlying platform. KLEE [17], Otter [70], and SymDroid are examples of this kind of symbolic executor. The main drawback to this approach is the significant effort required to model the underlying system. To address this limitation, we propose to automate the process of generating a model of framework via program synthesis in Chapter 3.

Search Strategy. Orthogonally to the type of symbolic executor, another key research area has been search strategies to allow symbolic executors to find "interesting" executions to explore, since in practice symbolic execution cannot cover all paths. KLEE [17] uses a round-robin-based heuristic that attempts to reach the closest uncovered nodes in the control-flow graph. SAGE [34] maintains a coverage-guided worklist to explore execution paths in a generational order. Otter [55] explored shortest-distance symbolic execution and call-chain-backward symbolic execution to target particular lines of interest. Researchers have also begun exploring how to symbolically execute multi-threaded programs [88, 16] As many Android apps include some threading (although typically not in the main part of the app, which is single-threaded), these techniques could be useful for developing a symbolic executor for Android.

Applications to Android. The most closely related work is ACTEve [5], a concolic executor for Android apps. The key contribution of ACTEve is mimicking user interactions by automatically generating event sequences. SymDroid, in contrast, requires the user to write a driver to reflect app usage. ACTEve uses ded [28, 64] to translate from Dalvik to Java bytecode, and then performs symbolic execution inside of Soot. It is unclear how ACTEve deals with Android framework code, as it does not run on top of Android. Another closely related study is a model of Android libraries [62] in Java PathFinder (JPF) [71], which includes stub classes to resolve incompatibility with Java and mock classes for Android's own components. Similar to SymDroid, it also use drivers to steer JPF toward program points of interest.

Chapter 3: Framework Synthesis for Symbolic Execution

In this chapter, we take a first step toward *automatically synthesizing* framework models by introducing PASKET ("*Pa*ttern *sket*cher") [48], 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, consider an Android 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. The button click callback mentioned above is an instance of the *Observer* pattern [29]. 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.

3.1 Overview

Figure 3.1 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

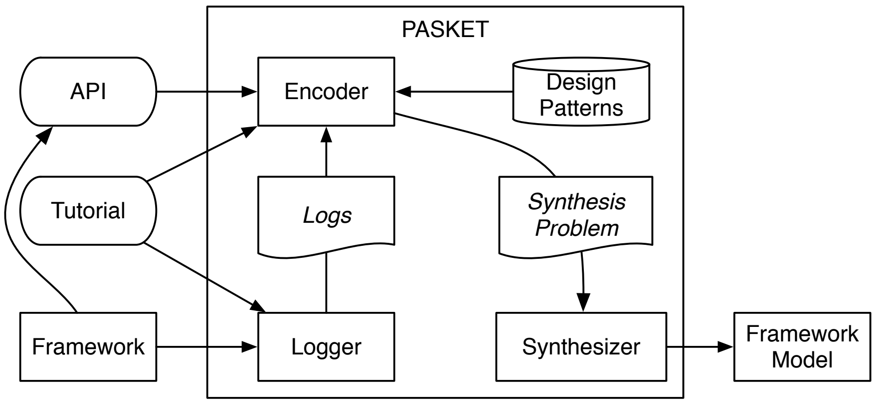


Figure 3.1: Pasket architecture.

designed to be used with tutorial programs that each exercises a small part of the framework, and Pasket then combines the information from each tutorial into a full model. In the case of Swing, for example, Oracle provides tutorials for buttons, checkboxes, and similar components, which are ideal for this purpose [65].

The framework API information can be extracted from the JAR or AAR files of the framework, although some user input is needed to select the parts of the framework API that should be modeled. This API provides the skeleton of the expected model. Pasket's goal is to generate code for that skeleton—insert the bodies of constructors and methods—to yield a model that can be used to analyze the tutorial programs and that, ideally, will also generalize to larger programs that use the same parts of the framework.

Behavioral Constraints. As a first step in the model creation process, the *logger* component inside Pasket executes the tutorial programs (perhaps requiring user interaction with the tutorial) and logs the method names, arguments, and return values that cross the boundary between the tutorial code and the framework (Calls internal to the framework are omitted from the log). For instance, in the Swing button callback example, the user would run the application and press the button while the logger records the execution. The log would therefore capture the button creation, registration, and callback, including the precise identities of the objects, so it captures the fact that the registered object is the one being called back when the button is clicked.

These captured logs serve as a behavioral specification for the synthesis process: the synthesizer aims to produce a model that achieves *log conformity* with the original program, meaning if the application were to run using the model code in place of the framework code under the same user inputs, we would observe the exact same sequence of calls as in the original log. Section 3.3 details this.

Structural Constraints. To produce a model, the log conformity requirement must be combined with a *structural hypothesis* to limit the space of possible models. In Pasket, this structural hypothesis comes from Pasket's internal knowledge of design patterns. The idea is that by limiting the search to models that implement design patterns we know to be used by the actual framework, we increase the likelihood the synthesized model will generalize and behave correctly with other applications. Pasket currently supports four main design patterns: Observer, Accessor, Adapter, and Singleton. Section 3.4 explains how these patterns are instantiated to match the given API and produce models satisfying log conformity.

Implementation. Pasket uses the Sketch synthesis system to search for log-conforming instantiations of the design patterns (hence the "sketcher" part of the name Pasket) [80]. Sketch's input is a *sketch* that describes a space of programs and a set of semantic constraints, usually given as assertions the synthesized program must satisfy. Sketch uses a symbolic search procedure to find a program in that space that satisfies the constraints. Section 3.5 discusses Pasket's Encoder component, which translates the client app, logs, framework API, and design pattern information into a sketch whose solution solves the Pasket synthesis problem.

The encoded synthesis problems are quite challenging due the large number of possible design pattern instantiations as well as the difficulty of reasoning about dynamic dispatch. Despite this, the problems are made tractable using recent research on combining constraint-based synthesis with explicit search, together with a careful encoding that allows the synthesizer to efficiently rule out large numbers of incorrect solutions.

Evaluation. We used Pasket to produce a model of the Java Swing GUI framework and the Android framework. For Swing, we used ten tutorials distributed by Oracle. Synthesis took just a few minutes, and in the end produced a model consisting of 95 different classes and 2,676 lines of code, making it one of the largest pieces of code ever synthesized using constraint-based synthesis. For Android, we used three tutorials gathered from the web. Synthesis took a few seconds and produced a model consisting of 50 different classes and 1,419 lines of code.

We validated the models in three ways. First, we ran the Swing tutorials against the synthesized Swing model and checked that they match the original logs. Second, we ran the Swing tutorials under Java PathFinder [71] (JPF). We found we could successfully execute eight of the ten tutorials (two tutorials are not supported by JPF's event generating system), while JPF's own model failed due to some missing methods. Finally, we selected eight code examples from O'Reilly's Java Swing, 2nd Edition [54] that use the same part of the framework and verified that they run under JPF using our merged model. We also selected two Android code examples and verified they run under SymDroid, a Davlik bytecode symbolic executor [43], using our merged model. (Section 3.7 describes our experiments.)

Scope and Limitation. Pasket's main focus is to generate a symbolically executable model whose control-flow behaviors conform to the original framework. As a specification of control-flow behaviors, Pasket inputs logs composed of call–return sequences, where we can observe implicit control-flows between apps and the framework. To dramatically reduce the search space of candidate model implementations, Pasket exploits its knowledge about design patterns and reduces the framework synthesis to finding proper instantiations of design patterns.

Pasket's approach will work well on certain features of the framework if the corresponding behaviors are observable from outside the framework and underlying structures can be expressed as any code patterns. On the other hand, Pasket's approach will not work properly if the feature of interest has side effects (*i.e.*, not observable via logs) or if the structural hypothesis should be described in very low-level expressions. An example of the former case is hardware abstraction, such as socket or file. An example of the latter case is utility functions in libraries, such as HashSet, where the detailed implementation might be too specific to be a general code pattern. (In that case, specifying structural hypothesis requires just same efforts as writing a corresponding model by hand.)

Properties of Synthesized Models. A model synthesized by PASKET is *abstract* in the sense that some features of the framework could be safely discarded as long as those are not related to symbolic execution. At the same time, a model is neither sound nor complete: a model is unsound as it could encompass wrong behaviors that still conform to the logs, the observed behavior while running tutorials; and a model is incomplete as it may miss some important features of the framework if corresponding tutorials were not provided. Nevertheless, the main benefit of using PASKET is that we can easily resynthesize an improved model, thanks to PASKET's semi-automated process. (Section 3.8 discusses the abstraction, soundness and completeness of synthesized models.)

Contributions. In summary, this chapter makes the following contributions:

- 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 3.3 and 3.4)
- We show how to encode the synthesis problem as a SKETCH synthesis problem.

```
class ButtonDemo implements ActionListener {
    public ButtonDemo() {
        b1 = new JButton("Disable middle button", ...);
        b1.setActionCommand("disable");
        b2 = new JButton("Middle button", ...);  ...
        b3 = new JButton("Enable middle button", ...);  ...
        b1.addActionListener(this); b3.addActionListener(this);
        add(b1); add(b2); add(b3);
    }
    public void actionPerformed(ActionEvent e) {
        if ("disable".equals(e.getActionCommand())) {
            ...
    }   }
    private static void createAndShowGUI() {
        JFrame frame = new JFrame("ButtonDemo");
        ButtonDemo newContentPane = new ButtonDemo(); ...
        frame.setContentPane(newContentPane); ...
}   }
```

Figure 3.2: ButtonDemo source code (simplified).

(Sections 3.5 and 3.6)

- 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 3.7)
- We discuss the abstraction, soundness and completeness of synthesized models in detail. (Section 3.8)

3.2 Running Example

As a running example, we show how Pasket synthesizes a Java Swing framework model from the tutorial program in Figure 3.2, which is a simplified extract from one of the tutorials for Java Swing.

Here the main method (not shown) calls createAndShowGUI (line 14), which instantiates a new window and adds a new instance of ButtonDemo to it. The ButtonDemo constructor (line 2) creates and initializes button objects b1 through b3, each of which are labeled (line 4). The code then registers this as an observer for clicks to b1 and b3 (line 7) and then adds the buttons to the window. When either button is clicked, Swing calls the actionPerformed method of the registered observer (line 10), whose behavior depends on the label of the button that was clicked (line 11).

In addition to the tutorial, the second input to Pasket is the framework API, consisting of classes, methods and types. The API is then completed by Pasket to produce a complete model like the Swing model that is partially shown in Figure 3.3. The black text in the figure corresponds to the original API given as input; package names are omitted for space reasons. The rest of the code (highlighted in blue) is generated by Pasket given a log from a sample run of ButtonDemo. For example, Pasket discovers that AbstractButton is a *subject* in the observer pattern—thus it has a list olist of observers, initialized in the constructor—and its *attach* method is addActionListener. The handle and handle_1 methods are introduced entirely by the synthesizer to model the way in which the AbstractButton invokes the actionPerformed methods in its registered listeners. In this model, the runtime posts events into the EventQueue and dispatches them by calling run. The model then propagates those events to any listeners that have been registered with a button. Pasket also discovers that EventObject, AWTEvent, and ActionEvent participate in the *accessor* pattern, with a field set via their constructor and retrieved via getSource in the case

```
class EventDispatchThread {
  private EventQueue q;
  void run() {
    EventObject e;
    while ((e = q.getNextEvent()) != null) q.dispatchEvent(e);
   }... }
class EventQueue {
  private Queue<EventObject> q;
  void postEvent(EventObject e) { q.add(e);  }
  void dispatchEvent(EventObject event) {
    if (event instanceof ActionEvent) {
      AbstractButton btn = (AbstractButton)event.getSource();
      btn.handle((ActionEvent)event);
    } ...
  }  ... }
class AbstractButton extends JComponent {
  private List<ActionListener> olist;
  AbstractButton() {olist = new LinkedList<ActionListener>();}
  void addActionListener( ActionListener  l) {olist.add(l);}
  void setActionCommand(String actionCommand) {// empty}
  void handle(ActionEvent event) { handle_1(event); }
  void handle_1(ActionEvent event) {
    int i=0; while(0¡=i && i¡olist.size && (o=olist.get(i) != null))
                { l.actionPerformed(event); i=i+1;}
  }  ... }
class JButton extends AbstractButton {
  JButton(String text, Icon icon) { // empty } }
class JFrame extends Frame { ... }
class EventObject {
  private Object source;
  EventObject(Object source) {this.source = source;}
  Object getSource() {return source;}
}
class AWTEvent extends EventObject { ... }
class ActionEvent extends AWTEvent {
  priavte String command;
  ActionEvent(Object source, int id, String command) {
    super(source, id); this.command = command;
  }
  String getActionCommand() {return command;}
}
```

Figure 3.3: Framework API to be modeled (partial). Highlighted code produced by synthesis.

of EventObject.

Notice that Pasket abstracts several constructors and methods to have empty bodies, because this particular tutorial program does not rely on their functionality. For example, the argument to the JButton constructor is never retrieved. Thus, the tutorials control Pasket's level of abstraction. Unneeded framework features can be omitted and then they will not be synthesized, and framework features can be added by introducing tutorials that exercise them.

3.3 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 3.4 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 65 has a call to addActionListener with arguments JButton@8 and ButtonDemo@9. Subsequently, on line 69 an ActionEvent associated with this button is created and immediately posted into the EventQueue;

```
60 ButtonDemo.main()
61   ButtonDemo.createAndShowGUI()
62     ButtonDemo.ButtonDemo@9()
63       JButton.setActionCommand(JButton@8, ‘‘disable’’)
64       JButton.setEnabled(JButton@4, false)
65       JButton.addActionListener(JButton@8, ButtonDemo@9)
66       JButton.addActionListener(JButton@4, ButtonDemo@9)
67     JFrame.setContentPane(JFrame@8, ButtonDemo@9)
68 ...
69 ActionEvent.ActionEvent@7(JButton@8, 0, ”disable”)
70 EventQueue.postEvent(EventQueue@1, ActionEvent@7)
71 EventDispatchThread.run(EventDispatchThread@0)
72   ButtonDemo.actionPerformed(ButtonDemo@9, ActionEvent@7)
73     ActionEvent.getActionCommand(ActionEvent@7)
74     return ” disable”
75 ...
76 ActionEvent.ActionEvent@5(JButton@4, 0, ”enable”)
77 EventQueue.postEvent(EventQueue@1, ActionEvent@5)
78 EventDispatchThread.run(EventDispatchThread@0)
79   ButtonDemo.actionPerformed(ButtonDemo@9, ActionEvent@5)
80     ActionEvent.getActionCommand(ActionEvent@5)
81     return ”enable”
82 ...
```

Figure 3.4: Sample output log from ButtonDemo.

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 72 corresponds to the framework's call to the actionPerformed method in the application. It will be up to Pasket to infer that this sequence of log entries is part of the observer design pattern. Pasket will then use its knowledge of this pattern to infer the contents of postEvent, run, and all the other functions that were invoked inside the framework to eventually call actionPerformed.

As another example, line 73 shows getActionCommand returning the string "disable", which was set in the setter on line 63. Thus, again given PASKET's library of design patterns, these log elements must be part of an accessor pattern.

The log conformity constraint is that a correct framework model, run against the same tutorial program under the same inputs, should produce the same log as the actual framework. In reactive frameworks such as Swing or Android, however, events such as button clicks are relayed by the runtime system to the framework, and the framework interacts with the application in response to these events. For such a reactive framework, these events are what constitute the "inputs" to the framework/application pair, so to check log conformity, the system needs to check that the combined framework model and application react to these events in the same way as the original framework and application did.

One subtle point is that the actual calls from the runtime system to the framework are likely to operate at a much lower level of abstraction than what we want to capture in the model. Our solution is to treat every top-level entry in the log (*i.e.*, every entry that appears without any previous entry in the stack) as if it is coming from the runtime system. So for example, the call to the ActionEvent constructor in line 69 might actually be coming from some code deep inside the framework in response to an operating system event, but from the model viewpoint, we can assume that the operating system is directly creating the ActionEvent and passing it to the EventQueue, and the framework and application are reacting to those actions.

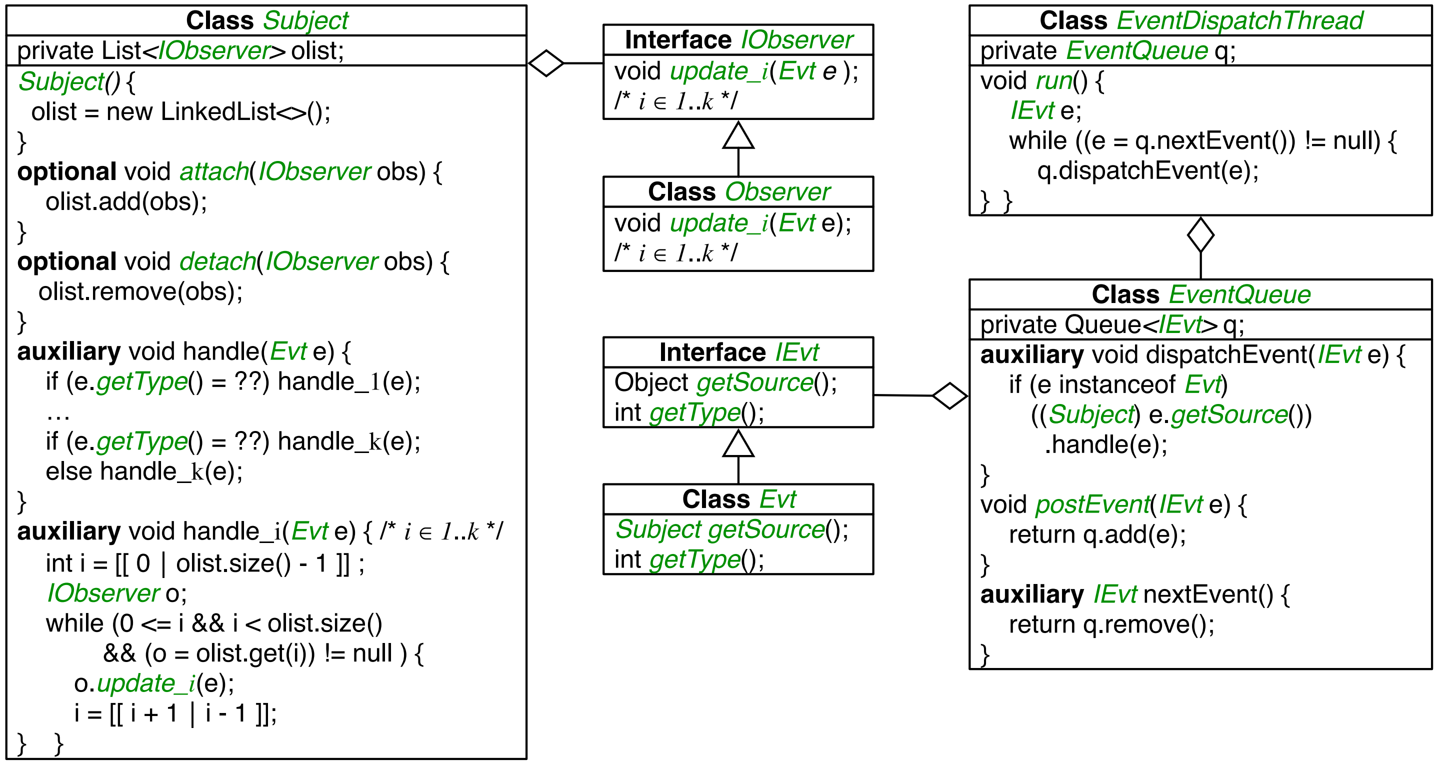
Another subtle aspect of the log conformity constraint is that the objects created when running against the real framework will have different ids from those created when running against the model, so the log conformity check must allow for the renaming of objects of the same type when comparing the logs for the two executions.

In the next section, we discuss Pasket's design patterns, and then in Section 3.5 we show how to combine the API, logs, and design pattern knowledge to synthesize a framework model using Sketch.

3.4 Design Pattern Instantiation

Pasket synthesizes the code in Figure 3.3 by *instantiating* design patterns. To understand the synthesis process, consider Figures 3.5 and 3.6, which show 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 3.5 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*



```
Class Subject
private List<IObserver> olist;
Subject() {
  olist = new LinkedList<>();
}
optional void attach(IObserver obs) {
    olist.add(obs);
}
optional void detach(IObserver obs) {
   olist.remove(obs);
}
auxiliary void handle(Evt e) {
    if (e.getType() = ??) handle_1(e);
    ...
    if (e.getType() = ??) handle_k(e);
    else handle_k(e);
}
auxiliary void handle_i(Evt e) { /* i ∈ 1..k */
    int i = [[ 0 | olist.size() - 1 ]] ;
    IObserver o;
    while (0 <= i && i < olist.size()
            && (o = olist.get(i)) != null ) {
        o.update_i(e);
        i = [[ i + 1 | i - 1 ]];
}   }
```

```
Interface IObserver
void update_i(Evt e );
/* i ∈ 1..k */
```

```
Class Observer
void update_i(Evt e);
/* i ∈ 1..k */
```

```
Interface IEvt
Object getSource();
int getType();
```

```
Class Evt
Subject getSource();
int getType();
```

```
Class EventDispatchThread
private EventQueue q;
void run() {
    IEvt e;
    while ((e = q.nextEvent()) != null) {
        q.dispatchEvent(e);
} }
```

```
Class EventQueue
private Queue<IEvt> q;
auxiliary void dispatchEvent(IEvt e) {
    if (e instanceof Evt)
      ((Subject) e.getSource())
        .handle(e);
}
void postEvent(IEvt e) {
    return q.add(e);
}
auxiliary IEvt nextEvent() {
    return q.remove();
}
```

Figure 3.5: Observer pattern in PASKET.

method. Thus to create 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 handler *handle_i*, the synthesizer is left to choose in what direction to iterate over the observer list.

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 discuss the problem of synthesizing multiple patterns simultaneously. We selected this set of patterns to support the experiments in Section 3.7, 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 3.5. The *Subject* maintains a list of *IObserver*'s, initialized in the constructor. Observers can be *attach*ed or *detatch*ed 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 3.2, 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 3.3, where the synthesizer instantiated handle to just call handle_1 and handle_1 to iterate forward through olist while calling the *update* method actionPerformed.

The body of handle_i includes a *generator* of the form $[[e_1 \mid \ldots \mid e_n]]$, which indicates the synthesizer must choose one of the expressions e_i as a solution.[1] In this case, there are two generators, one to determine whether i starts at the beginning or end of olist, and the other to determine whether i is incremented or decremented.

The right half of the figure shows the design pattern for an *event queue*, which actually dispatches events to the subject. This pattern is instantiated in conjunction with the observer pattern. Here the class *EventQueue* has an internal queue of events, which can be added to by *postEvent*. There are also methods to get the next event and to dispatch on an event by invoking the *Subject*'s handle method.

Singletons. Figure 3.6a shows the singleton pattern, used for classes that may only have a single instance. Such a class contains a private, static field *ins* storing the

[1]We use $[[\cdot]]$ for generators, rather than SKETCH's standard $\{| \cdot |\}$ notation, because the former is more readable in these figures.

```
Class Singleton
private static Singleton ins;
private Singleton (void);
public static Singleton getIns() {
      if (ins == null)
         ins = new Singleton();
      return ins;
}
```

(a) Singleton pattern.

```
Class Accessor
private Ti fi;   /* i ∈ 1..k */
Accessor(T1 o1, …, Tj oj) {   /* j <= k */
  if ([[ true | false ]]) super( [[ o1 | … | oj ]]* );
  f1 = o1; … ; fj = oj;
  if ([[ true | false ]]) fj+1 = [[ new cls() | ?? ]];
  …
  if ([[ true | false ]]) fk = [[ new cls() | ?? ]];
}
Ti get_fi(void) { return [[ f1 | … | fk ]]; } /* i ∈ 1..r, r <= k */
void set_fi(Ti v) { [[ f1 | … | fk ]] = v; }  /* i ∈ 1..s, s <= k */
```

(b) Accessor pattern.

```
Class Adapter
private T fld_i;   /* i ∈ 1..k */
void method(T1 arg1, …, Tj argj) {
  T adaptee = [[ fld_1 | … | fld_k ]];
  adaptee.other_method(arg1, …, argj);
}
```

(c) Adapter pattern.

Figure 3.6: Other patterns in PASKET.

instance; a private constructor (so no other instances can be created); and method *getIns* to get the instance (which is created on the first call to *getIns*). Notice that PASKET solves for the name of the class and the name of its *getIns* method. The name of *ins* is private, and thus PASKET can choose it arbitrarily, similarly to *olist* from the observer pattern.

Accessors. Figure 3.6b 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_f*1 through *get_fr* and s getter methods *set_f*1 through *set_fs*. Each getter method *get_fi* retrieves the value of a field chosen from f1 through fk; similarly, each setter method updates a field chosen from f1 through fk with the input v.

The *Accessor* class also has a single constructor that accepts j arguments, for some $j \leq k$. The i-th argument is used to initialize the i-th field fi, respectively. This incurs no loss of generality since PASKET can choose to enumerate the fields in any order. For those fields beyond fj, *i.e.*, fields f$j + 1$ through fk, PASKET may opt to initialize some of them implicitly 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 $[[o1 \mid \ldots \mid oj]]^*$. For example, in Figure 3.3 we see that ActionEvent's constructor passes only two parameters to its superclass AWTEvent, which in turn passes only one parameter to its superclass EventObject. Finally, the constructor initializes the fields appropriately.

Adapters. Figure 3.6c shows the adapter pattern, used to delegate method calls to another object. In this pattern, there is a *method* with j arguments. When called, it retrieves an object adaptee from one of its k fields and calls one of its methods with the same arguments. Here we assume the adapted method returns void, which was always the case in our experiments. In practice, we allow multiple methods to be adapted at once, but we show only one method here for simplicity. For example, in Swing, InvocationEvent is an adapter with a protected field runnable of class Runnable as its adaptee. When InvocationEvent's dispatch() method is called, it simply calls runnable.run().

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 instances 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 3.6.

Since the same class might be involved in multiple patterns, the design patterns in Figures 3.5 and 3.6 should be taken as minimal specifications of classes—PASKET always allows classes to contain additional fields and methods than are 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 fireActionPerformed instantiate an observer pattern, and its methods getActionCommand and setActionCommand instantiate an accessor pattern. Currently PASKET requires that each method body be instantiated from at most one pattern.

3.5 Framework Sketching

Pasket uses Sketch to discover how to instantiate the design patterns from Section 3.4 into the method bodies in Figure 3.3 to satisfy log conformity.

3.5.1 From Java to Sketch

Background. The input to Sketch is a space of programs in a C-like language. The space is represented as a program with choices and assertions. The choices can include unknown constants, written ??, as well as explicit choices between alternative expressions, written [[e_1 | ... | e_n]]. The goal of Sketch is to find a program in the space that satisfies the assertions [81]. For example, given a program

```
void double(int x) { int t = [[ x | 0 ]] * ??; assert t = x + x; }
```

Sketch will choose 2 for the constant ?? and x for the choice. Full details about Sketch can be found elsewhere [80, 81].

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.

Class Hierarchy. The first issue we face is that SKETCH's language is not object-oriented. To solve this problem, PASKET follows a similar approach to [79] and encodes objects with 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:

```
struct V_Object {
  int class_id ; fields-from-C1 ... fields-from-Cm
}
```

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 belongsTo[m] to be its class id; argNum[m] to be its number of arguments; and argType[m][i] to be the type of its i-th argument. We model the inheritance hierarchy using a two-dimensional array subcls such that subcls[i][j] is true if class i is a subclass of class j.

Encoding Names. When we translate the class hierarchy into PASKET, we also flatten out the namespace, and we need to avoid conflating overridden or overloaded method names, or inner classes. Thus, we name inner classes as *Inner_Outer*, where *Inner* is the name of the nested class and *Outer* is the name of the enclosing class. We also handle anonymous classes by assigning them distinct numbers, *e.g.*, *Cls_1*.

To support method overriding and overloading, methods are named *M_C_Ps*, where *M* is the name of the method, *C* is the name of the class in which it is declared, and *Ps* is the list of parameter types. For example, in the Swing APIs shown in Figure 3.3, JButton inherits method addActionListener from AbstractButton, hence the method is named addActionListener_AbstractButton_ActionListener(V_Object self, V_Object l) in SKETCH. The first parameter represents the callee of the method.

Dynamic Dispatch. We simulate the dynamic dispatch mechanism of Java in SKETCH. For each method name M (suitably encoded, as above), we introduce a function dyn_dispatch_M(V_Object self, ...) that dispatches based on the class_id field of the callee:

```
void dyn_dispatch_M(V_Object self,  ...)  {
  int cid = self.class_id;
  if (cid == R0_id) return M_R0_P(self,  ...);
  if (cid == R1_id) return M_R1_P(self,  ...);
  ...
  return;
}
```

Note that if M is static, the self argument is omitted.

JSketch. Although we devised aforementioned Java-to-C translation for our own purpose—framework synthesis atop SKETCH—we later realized that such translation itself can be a standalone tool. We separated it from PASKET; named it JSKETCH [47]; and added a few more general-purpose features, such as class-level *generators*. Full details of JSKETCH's general features are elaborated in Appendix A.

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.

3.5.2 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 3.4, we see three items that come from "outside" both the client app and the framework: the initial call to `main` (line 60) and the user inputs on lines 69 and 76. The driver is responsible for triggering these events, which it does by calling the appropriate (hard-coded) method names in Figure 3.5 for the event queue (or the appropriate names for Android if applying Pasket to that domain).

Figure 3.7 shows the driver for our running example. The code begins by creating a global array containing all the other log elements (the ones that are "inside" the client app and framework) and a global counter (code not shown). Next, the code (which is specific to Swing) begins by getting the system event queue and calling the `main` method of `ButtonDemo`. Then it performs the button click, mimicking Swing closely: The button click event object is created, added to the event queue, removed from the event queue, and then dispatched. (Recall from Figure 3.5 that this last call will trigger any subjects for this event.) The code for the second button click is similar.

Notice that since the driver simulates events that are external to the app and

```
void driver () {
  // (code not shown) create global array of other log elements
  V_Object t = getDefaultToolkit ();
  V_Object q = getSystemEventQueue(t);

  /* launch the client app*/
  main_ButtonDemo();

  /* perform the first button click */
  V_Object e0 = ActionEvent(get_JButton(0), 0, "disable");
  e0.kind_AWTEvent = 0;
  postEvent(q, e0); /* Add event to queue */
  V_Object evt1 = getNextEvent(q);
  dispatchEvent(q, evt1); /* And dispatch event right away */

  /* perform the second button click */
   ...
}
```

Figure 3.7: SKETCH driver code for ButtonDemo.

framework, we are forced to hard-code some method names here—hence also the hard-coded method names. We also need to know which items in the logs are events that should be created in the driver. Currently, we simply assume any instance of a class ending in Event is an event, and we generate one call to dispatchEvent for each of these.

One subtlety in the driver is that an event sometimes refers to objects created in the tutorial code. In our example, driver needs to refer to the button object created inside main_ButtonDemo, but that object is not in scope in driver. To address this problem, PASKET maintains a mapping of objects to ids and provides functions get_C(i) to retrieve object number i of class C. We assign number to the objects based on the order in which they are created, so here we call get_JButton(0) to get the first button that was created. The correct object to retrieve is determined by examining the `@` object ids in the log.

3.5.3 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 3.5) will be a variable in the framework sketch:

```
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. This approach helps greatly reduce Sketch's search space, compared to initializing the variables with unconstrained integers. (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 3.8 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

```
assert Subject ≠ Observer;

assert subcls[Subject][belongsTo[attach]];
assert subcls[Subject][belongsTo[detach]];
assert argNum[attach] == 1;
assert argNum[detach] == 1;
assert argType[attach][0] == IObserver;
assert argType[detach][0] == IObserver;
assert retType[attach][0] == VOID;
assert retType[detach][0] == VOID;
assert subcls[Observer][IObserver];

assert attach ≠ detach;
```

Figure 3.8: Constraints on design pattern variables (partial).

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 3.9 depicts the framework sketch code corresponding to addActionListener (Figure 3.3). The first statement (line 129) 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 130), 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.

```
void addActionListener_AbstractButton_ActionListener(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 3.9: Framework sketch (partial).

(Note that void returns are included in the actual log though we omitted them from Figure 3.4.)

Putting this all together, the check_log assertions will only allow this method to be called at appropriate points in the trace, specifically lines 65 and 66 of Figure 3.4. SKETCH will determine that *attach* is 19, hence the *attach* method code will be called in the function body.

There is one additional nit in the encoding of design patterns into SKETCH functions: Some patterns involve calls to methods whose names are determined during solving, *e.g.*, o.*update_i*(e) in Figure 3.5. Since we are solving for these method names, we cannot translate this as a direct function call in the framework sketch. Instead, similarly to ObserverPatternMethod above, PASKET translates this as a call to a function call_indirect(int method_id, ...) that invokes a method based on its id. The number of arguments to the method determines both the arguments to call_indirect and the possible methods it can dispatch to. For the framework model for our running example, the function looks like:

```
void call_indirect (int method_id, V_Object rcv, V_Object arg) {
  if (mtd_id == 40) actionPerformed(rcv, arg);
  else if ...
   ...
}
```

3.5.4 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 3.3, 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.

3.6 Implementation

We implemented Pasket 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 front-end of ANTLR v3.1.3 [67] 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 [26] 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 [6] using `javassist` [24]. 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 [44], 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`).

Currently, we manually ran instrumented apps to collect logs. Nonetheless, it is relatively easy to explore all possible behaviors of simple tutorials. To make the entire process of Pasket's model generation fully *automatic*, running instrumented apps and collecting logs should be automated as well, *e.g.*, by using script-based testing tool [42]. We leave it as a future work.

Java Libraries. Among many other Java libraries, parameterized collections, such as List<E>, Map<K,V>, Queue<E>, *etc.*, are heavily used in tutorials, acting like built-in types. Even design pattern knowledge in PASKET relies on them, *e.g.*, the observer *list* in the subject, event *queue* to asynchronously react to events, *etc.* PASKET supports some of those collections and APIs by defining low-level data structures and encoding functionalities. Recall that all class hierarhcy is merged into one big V_Object struct, and thus PASKET can easily support generic types.

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.

Optimization. The sketches passed to SKETCH are relatively large compared to sketches produced by other applications; they typically contain thousands of lines of code and some very large arrays. Because of their size and complexity, they exposed two performance issues we had to address.

First, SKETCH transforms a program into a formula by inlining all function calls up to a per-function bound, which is provided as a command-line argument. This caused problems for indirectly recursive functions like ObserverPatternMethod shown earlier that are indirectly recursive and call many candidate functions, as this causes the simple inlining heuristic to fail. We solved this by using an explicit counter in our sketches to force inlining to stop at a (small) recursive depth. Nevertheless, the depth-bounded inlining still blowed up the formula in SKETCH, due to the huge branching factor for those reflexive functions like call_indirect, as they call virtually all possible methods declared in the API. Our solution is to leverage the call site information to reduce the branching factor. For example, if call_indirect is called in a call site where the second argument rcv is an instance of JButton, we immediately know the indirect call must go to a method that can be accessed by JButton. Hence instead of calling a all-in-one call_indirect function, we call a specialized function call_indirect_for_JButton, which enumerates JButton's methods only, reducing the branching factor by two orders of magnitude.

Second, SKETCH treats constant-sized arrays as collections of scalar variables, under the assumption that such arrays are generally small. However, the arrays generated by PASKET, representing Java type information, have tens of thousands of entries. To solve this issue, we modified SKETCH to use its implementation of the theory of arrays when dealing with large constants.

Multi-pattern Synthesis. Recall from Section 3.4 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.)

3.7 Experiments

We evaluated Pasket by using it to separately synthesize a Swing framework model and an Android framework model from tutorial programs. Table 3.1 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 3.1, 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 3.2, 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.

	Tutorial			Sketch		w/ **AC**			**Patterns**				**Java**			
	Name	**LoC**	**Log**	**LoC**	**Std(s)**	||	**Tm(s)**	**Tot(s)**	**O**	**Ac**	**Ad**	**S**	**LoC**	**C**	**M**	$\emptyset$
Swing	ButtonDemo	150	90	8,785	64	358	8	60	1	4	1	1	2,636	95	296	30
	CheckBoxDemo	235	90	8,758	139	375	9	65	1	3	1	1	2,636	95	296	30
	ColorChooserDemo	116	40	8,466	15	336	5	56	1	3	1	1	2,626	95	296	30
	ComboBoxDemo	147	38	8,540	16	256	4	42	1	3	1	1	2,629	95	296	30
	CustomIconDemo	233	82	8,837	69	449	9	80	1	4	1	1	2,636	95	296	30
	FileChooserDemo	183	58	8,706	33	380	11	69	1	4	1	1	2,633	95	296	30
	MenuDemo	276	150	9,481	764	488	67	190	2	5	1	1	2,643	95	296	30
	SplitPaneDividerDemo	134	46	8,699	236	428	8	67	1	3	1	1	2,627	95	296	30
	TextFieldDemo	244	40	8,728	`OOM`	400	39	104	3	5	1	1	2,656	95	297	30
	ToolBarDemo	199	78	8,751	135	428	13	72	1	4	1	1	2,645	95	296	30
	Model						*(merging)*	14	5	9	1	1	2,676	95	297	30
Android	UIButton	50	46	5,258	8	113	1	16	1	10	0	5	1,412	50	169	10
	UICheckBox	96	82	5,455	25	209	7	33	1	10	0	5	1,419	50	169	10
	Telephony	86	54	5,131	6	30	1	14	0	9	0	5	1,412	50	169	10
	Model						*(merging)*	1	1	10	0	5	1,419	50	169	10

Table 3.1: Pasket results. LoC stands for lines of code; Log indicates number of log entries; Std(s) is the median running time under the standard version of Sketch; || shows the median number of parallel processes forked to find a solution; Tm(s) is the median running time of a single process that found a solution; Tot(s) is the median total running time; O(bserver), Ac(cessor), Ad(apter), and S(ingleton) are the number of instantiations of each design pattern; C and M are the number of synthesized classes and methods; and $\emptyset$ is the number of empty methods.

To synthesize an Android model, we used three tutorial apps, listed in the Android group of Table 3.1. Two of them, UIButton and UICheckBox, were examples in a 2014 Coursera class on Android programming. The third tutorial app, Telephony, is from an online tutorial site.[2] Table 3.1 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 LoC lists the lines of code 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 $95^{21} \times 263^{47}$. 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, which

[2]`http://www.javatpoint.com/android-telephony-manager-tutorial`

will be discussed in Chapter 4), 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 lines of code and the number of classes, methods, and empty methods (*i.e.*, those that are essentially abstracted away) in the synthesized model.

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 symbolic executor (discussed in Chapter 2). 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 [59] that ships as part of Java PathFinder [71] (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.

```
import gov.nasa.jpf.awt.UIActionTree;
import gov.nasa.jpf.util.event.Event;

public class TestEvent extends UIActionTree {
  @Override
  public Event createEventTree() {
    return sequence(
        click("$Disable", true),
        click("$Enable", true)
      );
} }
```

Figure 3.10: JPF driver for ButtonDemo.

Similar to SymDroid's driver in Figure 2.16, we need JPF drivers so as to run JPF on tutorials. Those drivers are designed to simulate the same user interactions under which we ran the tutorials to collect logs. For example, Figure 3.10 shows the JPF driver for ButtonDemo: as all identifiers are self explanatory, this driver creates an event tree that simulates user's clicks on buttons with labels "Disable" and "Enable" in sequence. 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.

Name	LoC	Tutorials
ToolbarFrame2	76	ToolBarDemo
ToolbarFrame3	156	ToolBarDemo + CustomIconDemo
JButtonEvents	40	ButtonDemo + CheckBoxDemo
JToggleButtonEvents	43	ButtonDemo + CheckBoxDemo
SimpleSplitPane	45	SplitPaneDividerDemo + FileChooserDemo
ColorPicker	35	ColorChooserDemo + ButtonDemo
ColorPicker3	72	ColorChooserDemo + ButtonDemo
SimpleFileChooser	94	FileChooserDemo

Table 3.2: Examples from O'Reilly's Java Swing, 2nd Edition.

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 [54] that use the same part of the framework as the Oracle tutorials we used. Table 3.2 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 3.3. 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 [60]) that looks up a phone number and/or device ID from the TelephonyManager, depending

[3] `http://developer.android.com/sdk/installing/adding-packages.html`

Name	LoC	Tutorials
Visibility	114	UIButton + UICheckBox
“Bump”	50	UIButton + UICheckBox + Telephony

Table 3.3: Example apps for Android.

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 [60], 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 synthesized framework. 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.

3.8 Properties of Synthesized Models

Although the experimental results look promising, it is worth discussing several properties of synthesized models. Among many others, in this section, we will discuss the abstraction, soundness and completeness of synthesized models.

3.8.1 Abstraction

Judgement about Abstraction. Pasket aims to synthesize an *abstract* model of the framework whose control-flow behaviors are the same as the observed behaviors of the framework yet amenable to symbolic execution. That is, only parts of the framework that are necessary to reproduce the same control-flows will be reserved, whereas unneeded parts will be discarded. The key evidence the synthesized models are abstract is *empty* methods. Recall empty methods shown in Figure 3.3 and the number of empty methods in Table 3.1. Those methods could be left as blank because the input tutorial does not depend on their functionality.

Origin of Abstraction. There are two phases where a model is abstracted: slicing the framework APIs, *i.e.*, selecting parts of the APIs, and avoiding synthesizing unimportant parts of the input APIs. Not surprisingly, the first phase compresses synthesized models more than the second phase does, and Pasket currently takes advantage of that fact. The main reason is that the first phase dramatically reduces the search space for design pattern instantiations.

If the (partial) framework APIs given to Pasket have C classes and M methods, and the design patterns Pasket searches for have uc class-role unkonwns and um method-role unknowns, then the search space is size $C^{uc} \times M^{um}$. Of course, a smaller size of APIs (thanks to slicing) yields a smaller base, resulting in a much smaller search space.

For example, our partial Android APIs have 50 classes and 153 methods. The numbers of unkonwns Pasket searches for are 15 class roles and 27 methods roles. Hence the search space of size $50^{15} \times 153^{27}$. The number of APIs that are newly added to Android 5.0 is over 2000,[4] and of course the total number of APIs is much larger than that. If we add those new methods for the same design patterns, the search space will be 31 orders of magnitude larger, a totally different scale.

The efficiency of the second phase depends on the given logs, *i.e.*, test scenarios for selected tutorials. If the functionality of certain methods is not exercised, there will be no log conformity constraints for those methods, allowing Pasket to unmodel them. In contrast, if logs utilize all the input APIs, all of them must be modeled somehow. Indeed, we intentially selected a minimal set of APIs, but only 10 out of 153 methods for Android were left as unmodeled.

3.8.2 Soundness and Completeness

Soundness. A model synthesized by Pasket may not be sound, *i.e.*, it may not be correct, since the model only conforms to the input logs, the observed behaviors of the framework. In other words, the model could have incorrect implementations as long as the reproduced behaviors still conform to the logs. To validate the synthesized models, we used tutorials to check log conformity and ran other applications from other source. Of course, these are still insufficient, and the best solution for checking the correctness of a synthesized model is to use the framework's own regression tests, such as CTS [7] for Android.

[4]`http://developer.android.com/about/versions/android-5.0.html`

Completeness. It is intractable to provide a finite set of tutorials that cover all aspects of the framework, and thus a model synthesized by PASKET may not be complete. That is, the model may miss some important parts of the framework if appropriate tutorials were not given. On the flip side, if we can identify which aspects of the framework are missing, and which tutorials can cover those parts, we can easily regenerate an enhanced model by taking advantage of PASKET's high-degree of automation.

Therefore, instead of the completeness of the model, the actual questions we should address are: how to determine which aspects of the framework is missing and how to find a *right* set of tutorials for those missed aspects. For addressing the first question, we need co-simulation of a synthesized model and the original framework. Here the framework acts as an *oracle*, the expected solution. By executing a set of apps step by step, we might be able to capture noticeable behavior differences, which will in turn pinpoint where PASKET missed to model. Several testing techniques, such as Gray-Box testing [52], could be used in order to drive the framework to unexplored paths. Lastly, addressing the second question is straightforward: syntactically searching for API usages in candidate tutorials.

3.9 Future Work

PASKET introduced a high-degree of automation to the process of generating framework models, and the synthesized models made existing symbolic executors more effective and efficient. However, there is still room for improvement. In particular, the whole process is not fully automated, such as selecting, running, and logging tutorials. Also, in addition to framework models, there are more artifacts that could be automatically generated via program synthesis, such as drivers, properties of interest, or search strategies. To support more design patterns or even arbitrary programming idioms, we could devise a way to express code patterns in general. We will discuss the aforementioned future directions in Chapter 5.

3.10 Literature Review

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 [59] 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 3.7, the model is missing several methods.

There are some studies that attempted to automatically create models of Swing [19] and Android [90] 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 [18] ran backward data-flow analysis over the Android source code to find implicit flows. Modelgen [25] 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 [44]. Both of these systems target information flow, which is insufficient to support symbolic execution.

Given an app, Droidel [13] 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 [40] 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.

Some other authors researched variants of control-flows as models. AVERROES [2] generates an over-approximate call graph of a library, and then uses it to create a model whose calling behavior mimics the call graph. AVERROES thus enables static analyses to conduct whole-program analysis even when missing the actual code to that library. However, AVERROES is not guaranteed to be sound with respect to any state within the library. Yang et al. [94] introduce *callback control-flow graph* (CCFG) that focuses on the same implicit flows as EDGEMINER. Their CCFG construction inputs a client program; performs graph-reachability analysis using platform-specific knowledge about callbacks; and outputs CCFG for that client program, which could be used by other static analyses. Again, neither of these tools is designed to support symbolic execution.

Samimi et al. [72] propose automatically generating mock objects for unit tests, using manually written pre- and postconditions. 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 Pnueli and Rosner [68], 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 [82, 84, 85]. The idea of encoding a richer type system as a single `struct` type with a type id was also used in the Autograder work [79]. Component-based synthesis techniques [36, 51] 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 [32] propose a synthesis algorithm that can efficiently produce bit-vector circuits for processor instructions, based on smart sampling. Storyboard [78] is a programming platform that can synthesize low-level data-structure-manipulating programs from user-provided abstract I/O examples. Transit [89], 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.

Several researchers have explored synthesis to aid Java programmers. Prime [63] takes partial programs with holes for call sites and aims to fill those holes with appropriate API calls. To do so, Prime uses information gathered from tutorials and code snippets from repositories such as GitHub. Automata-based synthesis, proposed by Alur et al. [3], aims to algorithmically synthesize a dynamic interface for Java classes that satisfies a given safety requirement. Such dynamic interfaces can help program analysis tools check if client code interacts with a library correctly. In contrast to both of these approaches, Pasket aims to produce a symbolically executable framework model.

Design Patterns. In their original form, design patterns [29] 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 [61] 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. [49] 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 an 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. [12] 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.

Chapter 4: Adaptive Concretization for Parallel Program Synthesis

Program synthesis aims to construct a program satisfying a given specification. One popular style of program synthesis is *syntax-guided synthesis*, which starts with a structural hypothesis describing the shape of possible programs, and then searches through the space of candidates until it finds a solution. Recent years have seen a number of successful applications of syntax-guided synthesis, ranging from automated grading [79], to programming by example [35], to synthesis of cache coherence protocols [89], among many others [23, 73, 85].

Despite their common conceptual framework, each of these systems relies on different synthesis procedures. One key algorithmic distinction is that some use *explicit search*—either stochastically or systematically enumerating the candidate program space—and others use *symbolic search*—encoding the search space as constraints that are solved using a SAT solver. The SyGuS competition has recently revealed that neither approach is strictly better than the other [4].

In this chapter, we propose *adaptive concretization*, a new approach to synthesis that combines many of the benefits of explicit and symbolic search while also parallelizing very naturally, allowing us to leverage large-scale, multi-core machines. Adaptive concretization is based on the observation that in synthesis via symbolic search, the unknowns that parameterize the search space are not all equally important in terms of solving time. In Section 4.1, we show that while symbolic methods can efficiently solve for some unknowns, others—which we call *highly influential* unknowns—cause synthesis time to grow dramatically. Adaptive concretization uses explicit search to concretize influential unknowns with randomly chosen values and searches symbolically for the remaining unknowns. We have explored adaptive concretization in the context of the Sketch synthesis system [80], although we believe the technique can be readily applied to other symbolic synthesis systems, such as Brahma [50] or Rosette [87].

Combining symbolic and explicit search requires solving two challenges. First, there is no practical way to compute the precise influence of an unknown. Instead, our algorithm estimates that an unknown is highly influential if concretizing it will likely shrink the constraint representation of the problem. (Section 4.2 describes the influence estimation.)

Second, because influence computations are estimates, even the highest influence unknown may not affect the solving time for some problems. Thus, our algorithm uses as a series of trials, each of which makes an independent decision of what to randomly concretize. This decision is parameterized by a *degree of concretization*, which adjusts the probability of concretizing a high influence unknown. At degree 1, unknowns are concretized with high probability; at degree ∞, the probability drops to zero. (Section 4.3 introduces the degree of concretization and concretization probability functions.)

The degree of concretization poses its own challenge: a preliminary experiment showed that there is a different optimal degree for almost every benchmark. Since there is no fixed optimal degree, the crux of adaptive concretization is to estimate the optimal degree online. Our algorithm begins with a very low degree (*i.e.*, a large amount of concretization), since trials are extremely fast. It then exponentially increases the degree (*i.e.*, reduces the amount of concretization) until removing more concretization is estimated to no longer be worthwhile. Since there is randomness across the trials, we use a statistical test to determine when a difference is meaningful. Once the exponential climb stops, our algorithm does binary search between the last two exponents to find the optimal degree, and it finishes by running with that degree. At any time during this process, the algorithm exits if it finds a solution. Adaptive concretization naturally parallelizes by using different cores to run the many different trials of the algorithm. Thus a key benefit of our technique is that, by exploiting parallelism on big machines, it can solve otherwise intractable synthesis problems. (Section 4.4 discusses pseudocode for the adaptive concretization algorithm.)

We implemented our algorithm for SKETCH and evaluated it against 26 benchmarks from a number of synthesis applications including automated tutoring [79], automated query synthesis [23], and high-performance computing, as well as benchmarks from the SKETCH performance benchmark suite [80] and from the SyGuS'14 competition [4]. (Section 4.5 elaborates our selection of benchmarks.)

The original adaptive concretization algorithm involved a few arbitrary design decisions. We perform a systematic evaluation of the design space to better understand the tradeoffs. We find that the algorithm is robust to changes in the original design decisions, but nevertheless the systematic exploration of the design space made our algorithm simpler and intuitive while also putting our initial design choices on a solid footing. (Sections 4.6.1 to 4.6.4 present a series of studies on design choices.)

By running our algorithm over twelve thousand times across all benchmarks, we are able to present a detailed assessment of its performance characteristics. We found our algorithm outperforms SKETCH on 22 of 26 benchmarks, sometimes achieving significant speedups of 3× up to 18×. In one case, adaptive concretization succeeds where SKETCH runs out of memory. We also ran adaptive concretization on 1, 4, and 32 cores, and found it generally has reasonable parallel scalability. Finally, we compared adaptive concretization to the winner of the SyGuS'14 competition on a subset of the SyGuS'14 benchmarks and found that our approach is competitive with or outperforms the winner. (Sections 4.6.5 and 4.6.6 discuss our performance and scalability results in detail.)

4.1 Combining Symbolic and Explicit Search

To illustrate the idea of influence, consider the following SKETCH example:

```
int foo(int x) implements spec{
  if(??){
      return x & ??; // unknown m1
  }else{
      return x | ??; // unknown m2
} }
```

```
int spec(int x){
    return minus(x, mod(x, 8));
}
```

The specification on the right asserts that the synthesized code must compute (x − (x mod 8)). Here the symbol ?? represents an unknown constant whose type is automatically inferred. Thus, the ?? in the branch condition is a boolean, and the other ??'s, labeled as unknowns m1 and m2, are 32-bit integers. It is worth emphasizing that a higher-level language may have more forms to represent unknowns, *e.g.*, even SKETCH has regular expressions, but when everything is taken into consideration, it is about solving for constants.

The sketch above has 65 unknown bits and 2^{33} unique solutions, which is too large for a naive enumerative search. However, the problem is easy to solve with *symbolic search*. Symbolic search works by symbolically executing the template to generate constraints among those unknowns, and then generating a series of SAT problems that solve the unknowns for well-chosen test inputs. Using this approach, SKETCH solves this problem in about 50ms, which is certainly fast.

However, not all unknowns in this problem are equal. While the bit-vector unknowns are well-suited to symbolic search, the unknown in the branch is much better suited to explicit search. In fact, if we incorrectly concretize that unknown to false, it takes only 2ms to discover the problem is unsatisfiable. If we concretize it correctly to true, it takes 30ms to find a correct answer. Thus, enumerating concrete values lets us solve the problem in 32ms (or 30ms if in parallel), which is 35% faster than pure symbolic search. For larger benchmarks this can make the difference between solving a problem in seconds and not solving it at all.

The benefit of concretization may seem counterintuitive since SAT solvers also make random guesses, using sophisticated heuristics to decide which variables to guess first. To understand why explicit search for this unknown is beneficial, we need to first explain how SKETCH solves for these unknowns. First, symbolic execution in Sketch produces a predicate of the form $Q(x, c)$, where x is the 32-bit *input* bit-vector and c is a 65-bit *control* bit-vector encoding the unknowns. $Q(x, c)$ is true if and only if foo(x)=x−(x mod 8) for the function foo described by c. Thus, Sketch's goal is to solve the formula $\exists c. \forall x. Q(x, c)$. This is a doubly quantified problem, so it cannot be solved directly with SAT [30].

Sketch reduces this problem to a series of problems of the form $\wedge_{x_i \in E} Q(x_i, c)$, *i.e.*, rather than solving for all x, Sketch solves for all x_i in a carefully chosen set E. After solving one of these problems, the candidate solution c is checked symbolically against all possible inputs. If a counterexample input is discovered, that counterexample is added to the set E and the process is repeated. This is the Counter-Example Guided Inductive Synthesis (CEGIS) algorithm, and it is used by most published synthesizers (*e.g.*, [50, 87, 89]).

Sketch's solver represents constraints as a graph, similar to SMT solvers, and then iteratively solves SAT problems generated from this graph. The graph is essentially an AST of the formula, where each node corresponds to an unknown or an operation in the theory of booleans, integer arithmetic, or arrays, and where common sub-trees are shared (see [80] for more details). For the simple example above, the formula $Q(x, c)$ has 488 nodes and CEGIS takes 12 iterations. On each iteration, the algorithm concretizes x_i and simplifies the formula to 195 nodes. In contrast, when we concretize the condition, $Q(x, c)$ shrinks from 488 to 391 nodes, which simplify to 82 nodes per CEGIS iteration. Over 12 iterations, this factor of two in the size of the problem adds up. Moreover, when we concretize the condition to the wrong value, Sketch discovers the problem is unsatisfiable after only one counterexample, which is why that case takes only 2ms to solve.

In short, unlike the random assignments the SAT solver uses for each individual sub-problem in the CEGIS loop, by assigning concrete values in the high-level representation, our algorithm significantly reduces the sub-problem sizes across *all* CEGIS loop iterations. It is worth emphasizing that the unknown controlling the branch is special. For example, if we concretize one of the bits in m1, it only reduces the formula from 488 to 486 nodes, and the solution time does not improve. Worse, if we concretize incorrectly, it will take almost the full 50ms to discover the problem is unsatisfiable, and then we will have to flip to the correct value and take another 50ms to solve, thus doubling the solution time. Thus, it is important to concretize only the most influential unknowns.

Putting this all together yields a simple, core algorithm for concretization. Consider the original formula $Q(x, c)$ produced by symbolic execution over the sketch. The unknown c is actually a vector of unknowns c_i, each corresponding to a different hole in the sketch. First, rank-order the c_i from most to least influence, $c_{j0}, c_{j1}, \cdots$. Then pick some threshold n smaller than the length of c, and concretize $c_{j0}, \cdots, c_{jn}$ with randomly chosen values. Run the previously described CEGIS algorithm over this partially concretized formula, and if a solution cannot be found, repeat the process with a different random assignment. Notice that this algorithm parallelizes trivially by running the same procedure on different cores, stopping when one core finds a solution.

This basic algorithm is straightforward, but three challenges remain: How to estimate the influence of an unknown, how to estimate the threshold of influence for concretization, and how to deal with uncertainty in those estimates. We discuss these challenges in the next three sections.

4.2 Influence Estimation

An ideal measure of an unknown's influence would model its exact effect on running time, but there is no practical way to compute this. As we saw in the previous section, a reasonable alternative is to estimate how much we expect the constraint graph to shrink if we concretize a given node. However, it is still expensive to actually perform substitution and simplification.

Our solution is to use a myopic measure of influence, focusing on the immediate neighborhood of the unknown rather than the full graph. Following the intuition from Section 4.1, our goal is to assign high influence to unknowns that select among alternative program fragments (*e.g.*, used as guards of conditions), and to give low influence to unknowns in arithmetic operations. For unknown n we define

$$\mathit{influence}(n) = \sum_{d \in \mathit{children}(n)} \mathit{benefit}(d, n)$$

where $\mathit{children}(n)$ is the set of nodes that depend directly on n. Here $\mathit{benefit}(d, n)$ is a crude measure of how much the overall formula might shrink if we concretize the parent node n of node d. The function is defined by case analysis on d:

- *Choices.* If d is an ite node,[1] there are two possibilities. If n is d's guard ($d = \mathrm{ite}(n, a, b)$) then $\mathit{benefit}(d, n) = 1$. This is a high-influence position, so 1 is our baseline for the ratio between high and low influence. If n corresponds

[1] $\mathrm{ite}(a, b, c)$ corresponds to **if** (a) b **else** c, as in SMT-LIB.

to one of the choices ($d = \text{ite}(c, n, b)$ or $d = \text{ite}(c, a, n)$), then *benefit*$(d) = 0$, since replacing n with a constant has no effect on the size of the formula.

- *Boolean nodes.* If d is any boolean node except negation, its benefit should be some fraction B of the baseline benefit 1. In our first attempt, we set B to be 0.5, so that ite nodes are two times as important as boolean nodes. Our intuition was that boolean nodes are often used in conditional guards, but sometimes not. The choice of B will be empirically evaluated in Section 4.6.4.

  If $d = \neg(n)$, then *benefit*(d, n) equals *influence*(d), since the benefit in terms of formula size of concretizing n and d is the same.

- *Choices among constants.* SKETCH's constraint graph includes nodes representing selection from a fixed sized array. If d corresponds to such a choice that is among an array of constants, then *benefit*$(d, n) =$ *influence*(d), *i.e.*, the benefit of concretizing the choice depends on how many nodes depend on d.

- *Arithmetic nodes.* If d is an arithmetic operation, *benefit*$(d, n) = -\infty$. The intuition is that these unknowns are best left to the solver. For example, given `??+in`, replacing `??` with a constant will not affect the size of the formula.

Note that while the above definitions may involve recursive calls to *influence*, the recursion depth will never be more than two due to prior simplifications. Before settling on this particular influence measure, we tried a simpler approach that attempted to concretize holes that flow to conditional guards, with a probability based on the degree of concretization. However, we found that a small number of conditionals have a large impact on the size and complexity of the formula. Thus, having more refined heuristics to identify high influence holes is crucial to the success of the algorithm.

4.3 Degree of Concretization

The next step is to decide the threshold for concretization. We hypothesize the best amount of concretization varies—we will test this hypothesis in Section 4.6.2. Moreover, since our influence computation is only an estimate, we opt to incorporate some randomness, so that (estimated) highly influential unknowns might not be concretized, and (estimated) non-influential unknowns might be.

Thus, we parameterize our algorithm by a *degree of concretization* (or just *degree*). For each unknown n in the constraint graph, we calculate its estimated influence $N = \mathit{influence}(n)$. Then we concretize the node with certain probability p, which will be determined by both degree d and influence N of the node. We opt to design probability functions, where unknowns with high influence (*i.e.*, with large N) are assigned a higher probability, as the overall probability of concretization decreases as d increases. (So, if d is 0, many unknowns are concretized, and if it is ∞, then none are.)

4.3.1 Discontinuous Probability Function

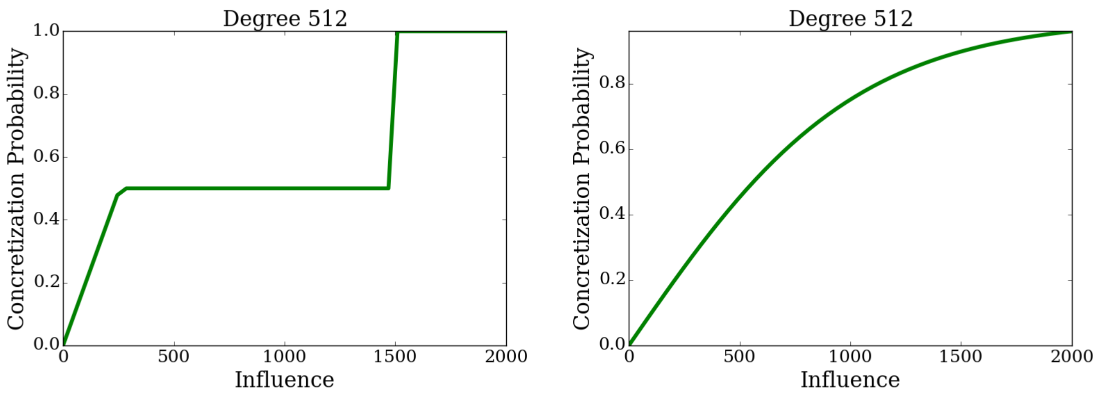
In our first attempt, we computed the probability p of concretization using the following formula, which we refer to as the *discontinuous probability function*:

$$p = \begin{cases} 0 & \text{if } N < 0 \\ 1.0 & \text{if } N > 1500 \\ 1/(\max(2, d/N)) & \text{otherwise} \end{cases}$$

To understand this function, ignore the first two cases, and consider what happens when d is low, *e.g.*, 10. Then any node for which $N \geq 5$ will have a 1/2 chance of being concretized, and even if N is just 0.5—the minimum N for an unknown not involved in arithmetic—there is still a 1/20 chance of concretization. Thus, low degree means many nodes will be concretized. In the extreme, if d is 0 then all nodes have a 1/2 chance of concretization. On the other hand, suppose d is high, *e.g.*, 2000. Then a node with $N = 5$ has just a 1/400 chance of concretization, and only nodes with $N \geq 1000$ would have a 1/2 chance. Thus, a high degree means fewer nodes will be concretized. There are also two special cases: Nodes of influence less than 0 are never concretized, and nodes of influence greater than 1500 are always concretized.

Figure 4.1a draws the discontinuous probability function at degree 512. There is a linear slope from 0 until the ceiling of 0.5, followed by a straight line until the degree cutoff 1500, and then the probability becomes 1.0.

While this function worked well, it is unsatisfying for a few reasons. First, the choices of probability ceiling 0.5 and influence cut off 1500 are ad hoc, based on what worked well for a subset of our benchmarks. Second, it has two large discontinuities as shown in the figure. The one is a straight line; depending on the degree, its length, which corresponds to the range of somewhat ambiguous influences, might be



(a) Discontinuous (b) Smooth

Figure 4.1: Probability functions at degree 512.

too long; *e.g.*, for degree 512, a node with 256 has the same 0.5 probability as a node with influence 1499 has. The other one is the discontinuous jump at influence 1500; a variable with influence 1499 is concretized with probability at most 0.5, whereas a variable with influence 1500 is always concretized. Such discontinuity is why we call this the *discontinuous* probability function.

4.3.2 Smooth Probability Function

To address these issues, we developed a new, *smooth* probability function:

$$p = \left(\frac{1}{1+e^{-N/d}} - 0.5\right) \times 2$$

Like the discontinuous function, the smooth function is parametrized only by degree d and influence N of node n; the larger N is, the more likely node n is concretized; and the larger d is, the less likely concretization is overall.

However, the smooth function addresses all the aforementioned issues. First, it does not include any ad hoc constants. In addition to base e, there are two extra constants in the formula, 0.5 and 2, but they are only used to ensure the output lies between 0 and 1. Second, it does not have any discontinuity. To visually compare both functions, Figure 4.1b depicts the smooth function at the same degree 512. As clearly shown in the figure, the smooth function has neither a straight line, where nodes with quite different influences may have the same concretization probability, nor discontinuous jumps at any points.

In Section 4.6.1, we empirically compared both probability functions. The experimental results show that both functions behave similarly, hence we choose the smooth function, which eliminates "magic constants" yet has the features we want. In Section 4.6.2, we conducted a preliminary experiment to test whether the optimal degree varies with subject program.

4.4 Adaptive, Parallel Concretization

Figure 4.2 gives pseudocode for adaptive concretization. The core step of our algorithm, encapsulated in the **run_trial** function, is to run SKETCH with the specified degree. If a solution is found, we exit the search. Otherwise, we return both the time taken by that trial and the size of the concretization space, *e.g.*, if we concretized n bits, we return 2^n. We will use this information to estimate the time-to-solution of running at this degree.

Since SKETCH solving has some randomness in it, a single trial is not enough to provide a good estimate of time-to-solution, even under our heuristic assumptions. For a practical algorithm, we cannot fix a number of trials, lest we run either too

```
run_trial(degree)
  run SKETCH with specified degree
  if solution found then
    raise success
  else
    return (running time,
            concretization space size)

compare(deg_a, deg_b)
  dist_a ← ∅
  dist_b ← ∅
  while |dist_a| ≤ Max_dist ∧
      wilcoxon(dist_a, dist_b) > T do
    dist_a ∪← run_trial(deg_a)
    dist_b ∪← run_trial(deg_b)
  if wilcoxon(dist_a, dist_b) > T then
    return tie
  elsif avg(dist_a) < avg(dist_b) then
    return left
  else
    return right
```

```
climb()
  low, high ← 0, 1
  while high < Max_exp do
    case compare($2^{low}$, $2^{high}$) of
      left: break
      right:
        low ← high
        high ← high + 1
      tie: high ← high + 1
  return (low, high)

bin_search(low, high)
  mid ← (low + high) / 2
  case compare(low, mid) of
    left: return bin_search(low, mid)
    right: return bin_search(mid, high)
    tie: return mid

main()
  (low, high) ← climb()
  deg ← bin_search($2^{low}$, $2^{high}$)
  while (true) do run_trial(deg)
```

Figure 4.2: Search Algorithm using Wilcoxon Signed-Rank Test.

many trials (which wastes time) or too few (which may give a non-useful result).

To solve this issue, our algorithm uses the *Wilcoxon Signed-Rank Test* [92] to determine when we have enough data to distinguish two degrees. We assume we have a function **wilcoxon**(dist_a, dist_b) that takes two equal-length lists of (time, concretization space size) pairs, converts them to distributions of estimated times-to-solution, and implements the test, returning a p-value indicating the probability that the means of the two distributions are different.

Expected Running Time. Due to randomness introduced by hole concretization, a hybrid approach that places somewhere between those two extreme algorithms should follows the geometric distribution[2]: it fails $k-1$ times with probability $1-p$ and then succeeds at k-th trial with probability p. Since the expected value of the geometric distribution is $1/p$, expected running time is t/p, where t is measured running time, and p is empirical success rate. We use the same calculation in this algorithm, except we need a different way to compute p, since the success rate is always 0 until we find a solution, at which point we stop. Thus, we instead calculate p from the search space size. We assume there is only one solution, so if the search space size is s, we calculate $p = 1/s$.[3]

Comparing Degrees. Next, **compare** takes two degrees as inputs and returns a value indicating whether the **left** argument has lower expected running time, the **right** argument does, or it is a **tie**. The function initially creates two empty sets of trial results, dist_a and dist_b. Then it repeatedly calls **run_trial** to add a new trial to each of the two distributions (we write x $\cup\leftarrow$ y to mean adding y to set x). Iteration stops when the number of elements in each set exceeds some threshold Max_dist, or the **wilcoxon** function returns a p-value below some threshold T. Once the algorithm terminates, we return **tie** if the threshold was never reached, or **left** or **right** depending on the means.

In our experiments, we use $3 \times max(8, |cores|)$ for Max_dist. Thus, **compare** runs at most three "rounds" of at least eight samples (or the number of cores, if that is larger). This lets us cut off the **compare** function if it does not seem to be

[2]`http://en.wikipedia.org/wiki/Geometric_distribution`

[3]Notice we can ignore the size of the symbolic space, since symbolic search will find a solution if one exists for the particular concretization.

finding any distinction. In our first attempt, we use 0.2 for the threshold T. This is higher than a typical p-value (which might be 0.05), but recall our algorithm is such that returning an incorrect answer will only affect performance and not correctness. We conduct a more systematic analysis about T in Section 4.6.3.

Searching for the Optimal Degree. Given the **compare** subroutine, we can implement the search algorithm. The entry point is **main**, shown in the lower-right corner of Figure 4.2. There are two algorithm phases: an *exponential climbing* phase (function **climb**) in which we try to roughly bound the optimal degree, followed by a binary search (function **bin_search**) within those bounds.

We opted for an initial exponential climb because binary search across the whole range could be extremely slow. Consider the first iteration of such a process, which would compare full concretization against no concretization. While the former would complete almost instantaneously, the latter could potentially take a long time (especially in situations when our algorithm is most useful).

The **climb** function aims to return a pair low, high such that the optimal degree is between 2^{low} and 2^{high}. It begins with low and high as 0 and 1, respectively. It then increases both variables until it finds values such that at degree 2^{high}, search is estimated to take a longer time than at 2^{low}, *i.e.*, making things more symbolic than low causes too much slowdown. Notice that the initial trials of the **climb** will be extremely fast, because almost all variables will be concretized.

To perform this search, **climb** repeatedly calls **compare**, passing in 2 to the power of low and high as the degrees to compare. Then there are three cases.

If **left** is returned, 2^{low} has better expected running time than 2^{high}. Hence we assume the true optimal degree is somewhere between the two, so we return them. Otherwise, if **right** is returned, then 2^{high} is better than 2^{low}, so we shift up to the next exponential range. Finally, if it is a **tie**, then the range is too narrow to show a difference, so we widen it by leaving low alone and incrementing high. We also terminate climbing if high exceeds some maximum exponent Max_exp. In our implementation, we choose Max_exp as 14, since for our subject programs this makes runs nearly all symbolic.

After finding rough bounds with **climb**, we then continue with a binary search. Notice that in **bin_search**, low and high are the actual degrees, whereas in **climb** they are degree exponents. Binary search is straightforward, maintaining the invariant that low has expected faster or equivalent solution time to high (recall this is established by **climb**). Thus each iteration picks a midpoint mid and determines whether low is better than mid, in which case mid becomes the new high; or mid is better, in which case the range shifts to mid to high; or there is no difference, in which case mid is returned as the optimal degree.

Finally, after the degree search has finished, we repeatedly run SKETCH with the given degree. The search exits when **run_trial** finds a solution, which it signals by raising an exception to exit the algorithm. (Note that **run_trial** may find a solution at any time, including during **climb** or **bin_search**).

Parallelization. Our algorithm is easy to parallelize. The natural place to do this is inside **run_trial**: Rather than run a single trial at a time, we perform parallel trials.

More specifically, our implementation includes a worker pool of a user-specified size. Each worker performs concretization randomly at the specified degree, and thus they are highly likely to all be doing distinct work.

Timeouts. Like all synthesis tools, SKETCH includes a timeout that kills a search that seems to be taking too long. Timeouts are tricky to get right, because it is hard to know whether a slightly longer run would have succeeded. Our algorithm exacerbates this problem because it runs many trials. If those trials are killed just short of the necessary time, it adds up to a lot of wasted work. At the other extreme, we could have no timeout, but then the algorithm may also waste a lot of time, *e.g.*, searching for a solution with incorrectly concretized values.

To mitigate the disadvantages of both extremes, our implementation uses an adaptive timeout. All worker threads share an initial timeout value of one minute. When a worker thread hits a timeout, it stops, but it doubles the shared timeout value. In this way, we avoid getting stuck rerunning with too short a timeout. Note that we only increase the timeout during **climb** and **bin_search**. Once we fix the degree, we leave the timeout fixed.

4.5 Experimental Design

We evaluated adaptive concretization across 26 benchmarks collected from five categories of synthesis problems. Each category stems from a distinct application domain and varies in complexity, amount of symmetry, *etc.* We briefly describe each category below, and Table 4.1 lists each benchmark, along with its lines of code and brief description.

- PASKET. The first three benchmarks, beginning with `p_`, come from the application that inspired this work: PASKET (discussed in Chapter 3), a tool that aims to synthesize a framework model for symbolic execution. PASKET's sketches are some of the largest that have ever been tried, and we developed adaptive concretization because they were initially intractable with plain SKETCH.

  Note that `p_button`, `p_color`, and `p_menu`, correspond to ButtonDemo, ColorChooserDemo, and MenuDemo in Table 3.1, respectively. These sketches were generated by an earlier version of PASKET, where we used fewer design patterns. Thus, their sizes and running times will be different from what we have reported in Chapter 3.

- *Data Structure Manipulation.* The second set of benchmarks is from a project aiming to synthesize provably correct data-structure manipulations [69]. Each synthesis problem consists of a program template and logical specifications describing the functional correctness of the expected program.

- *Invariants for Stencils.* The next set of benchmarks, beginning with `a_mom_`, are from a system that synthesizes invariants and postconditions for scientific computations involving stencils. In this case, the stencils come from a DOE Miniapp called Cloverleaf [31]. These benchmarks involve primarily integer arithmetic and large numbers of loops.

- *SyGuS Competition.* The next set of benchmarks, beginning with `ar_` and `hd_`, are from the first Syntax-Guided Synthesis Competition [4], which compared syn-

Benchmark		Description
Name	**LoC**	
`p_button`	3,436	aims to synthesize a model of JButton and ActionListener
`p_color`	3,194	aims to synthesize a model of JColorChooser
`p_menu`	4,099	aims to synthesize a model of JMenu and JMenuItem
`l_prepend`	708	accepts a sorted singly linked list L and prepends a key k
`l_min`	795	traverses a singly linked list via a while loop and returns the smallest key in the list
`a_mom_1`	229	stencil 1
`a_mom_2`	231	stencil 2
`ar_s_4`	313	array search SyGuS benchmark
`ar_s_5`	334	larger array search benchmark
`ar_s_6`	337	larger array search benchmark
`ar_s_7`	322	larger array search benchmark
`ar_sum`	328	array sum SyGuS benchmark
`hd_13_d5`	310	hackers delight bit-vector SyGuS benchmark
`hd_14_d1`	304	another bit-vector SyGuS benchmark
`hd_14_d5`	329	another bit-vector SyGuS benchmark
`hd_15_d5`	329	another bit-vector SyGuS benchmark
`deriv2`	1,444	automatically grades Python code to compute a derivative
`deriv3`	1,410	different automated grading Python benchmark
`deriv4`	1,410	different automated grading Python benchmark
`deriv5`	1,410	different automated grading Python benchmark
`s_cg`	124	conjugate gradient benchmark from SKETCH benchmark suite
`s_log2`	49	computes the logarithm base two of a bit vector
`s_logcnt`	30	counts the number of ones in a bit-vector in $\log n$ steps
`s_rev`	136	reverses a list
`q_noti`	262	SQL Query synthesis benchmark 1
`q_serv`	2,005	SQL Query synthesis benchmark 2

Table 4.1: Benchmarks.

thesizers using a common set of benchmarks. We selected nine benchmarks that took at least 10 seconds for any of the solvers in the competition, but at least one solver was able to solve it.

• SKETCH. The last group of benchmarks, beginning with `s_`, `deriv`, and `q_`, are from SKETCH's performance test suite, which is used to identify performance regressions in SKETCH and measure potential benefits of optimizations.

Throughout this chapter, all performance reports are based on 13 runs on a server equipped with forty 2.4 GHz Intel Xeon processors and 99 GB RAM, running Ubuntu 14.04.3 LTS. (We used the same machine for the experiments in Section 3.7.) For the pure SKETCH runs only, performance is also on 13 runs with a 2-hour timeout and 32 GB memory bound.

4.6 Experimental Evaluation

First, we consider the two "magic constants" in *degree of concretization*: degree cut-off 1500 and ceiling of probability 1/2 in the discontinuous probability function. We simply avoid these constants by introducing the new, smooth probability function whose shape and trend are similar to the previous function. In Section 4.6.1, we empirically show that both functions indeed behave similarly.

One assumption underlying our algorithm is that if we make a graph with the degree on the x-axis and the expected time to find a solution on the y-axis, the graph forms a "vee" shape with a low point at the optimal degree. This justifies adaptive concretization's degree search process, which uses a combination of exponential hill climbing and binary search. In Section 4.6.2, we empirically demonstrate on a subset of the benchmarks that the "vee" shape indeed occurs under the smooth probability function, hence justifying adaptive concretization's search process.

Next, we consider the last two "magic constants" in adaptive concretization. First, during degree search, the algorithm uses the Wilcoxon Signed-Rank Test [92] to compare the mean expected synthesis time from two sets of trials at two different degrees. That test returns a p-value indicating the probability any observed difference in the mean is due to random chance. Once the p-value exceeds a threshold T, adaptive concretization determines one degree is better than the other and then continues searching at a different pair of degrees. In our original algorithm, we fixed T at 0.2. In Section 4.6.3, we use a simulation of adaptive concretization to compare five different T values ranging from 0.001 to 0.5. We find that choosing a T anywhere between 0.05 and 0.2 seems to yield similarly good results.

Second, adaptive concretization's original influence calculation assigns arbitrary boolean unknowns 0.5 times the influence of unknowns in guard positions of if-then-else nodes. In Section 4.6.4, we empirically evaluate a range of different values for this ratio, B, ranging from 1/8 to 2. Surprisingly, we find no meaningful differences among this wide range of choices, suggesting the influence calculation is not sensitive to the choice of B. Cumulatively, our results put adaptive concretization on a much firmer foundation by demonstrating that the algorithm is robust to a wide range of design decisions.

Finally, in Sections 4.6.5 and 4.6.6, we empirically evaluated adaptive concretization against a range of benchmarks with various characteristics. Compared to regular Sketch (*i.e.*, pure symbolic search), we found our algorithm is substantially faster in many cases; competitive in most of the others; and slower on a few benchmarks. We also compared adaptive concretization with concretization fixed at the final degree chosen by the adaption phase of our algorithm (*i.e.*, to see what would happen if we could guess this in advance), and we found performance is reasonably close, meaning the overhead for adaptation is not high. We measured parallel scalability of adaptive concretization of 1, 4, and 32 cores, and found it

Bench	**Discontinuous**			**Smooth**			speedup
mark	*d*	||	Tm (s)	*d*	||	Tm (s)	(D/S)
p_button	4,864	597	52 $_{9}$	2,048	592	**46** $_{10}$	1.130
p_color	3,072	462	36 $_{12}$	640	336	**21** $_{2}$	1.714
p_menu	212	590	**70** $_{14}$	3,072	601	72 $_{28}$	0.972
l_prepend	32	88	**13** $_{1}$	256	151	17 $_{1}$	0.765
l_min	128	204	41 $_{12}$	256	204	**34** $_{10}$	1.206
a_mom_1	256	306	248 $_{20}$	1,024	222	**198** $_{12}$	1.253
a_mom_2	4,096	355	1,130 $_{144}$	2,048	219	**848** $_{98}$	1.333
ar_s_4	32	11	**4** $_{0}$	16	3	5 $_{0}$	0.800
ar_s_5	16	18	**5** $_{0}$	16	11	5 $_{1}$	1.000
ar_s_6	32	38	9 $_{2}$	16	15	**9** $_{0}$	1.000
ar_s_7	64	106	49 $_{10}$	16	37	**40** $_{10}$	1.225
ar_sum	16	15	**40** $_{6}$	32	8	55 $_{32}$	0.727
hd_13_d5	16	14	**8** $_{0}$	32	15	8 $_{1}$	1.000
hd_14_d1	32	70	**16** $_{6}$	52	107	22 $_{6}$	0.727
hd_14_d5	32	14	265 $_{62}$	32	10	**237** $_{70}$	1.118
hd_15_d5	32	13	**130** $_{48}$	32	12	178 $_{56}$	0.730
s_cg	64	141	13 $_{1}$	32	73	**11** $_{2}$	1.118
s_log2	64	110	141 $_{156}$	128	109	**136** $_{227}$	1.037
s_logcnt	32	110	27 $_{8}$	32	37	**25** $_{46}$	1.080
s_rev	128	164	**40** $_{13}$	128	118	44 $_{18}$	0.909
deriv2	16	20	**7** $_{2}$	16	25	8 $_{1}$	0.875
deriv3	32	15	**7** $_{2}$	32	20	8 $_{2}$	0.875
deriv4	16	17	6 $_{0}$	32	18	**5** $_{0}$	1.200
deriv5	32	19	6 $_{0}$	32	9	**5** $_{2}$	1.200
q_noti	32	115	**7** $_{0}$	64	125	7 $_{2}$	1.000
q_serv	16	9	**21** $_{4}$	16	5	22 $_{4}$	0.955

Table 4.2: Comparing AC with discontinuous vs. smooth probability function.

generally scales well. We also compared against the winner of the SyGuS'14 competition on a subset of the benchmarks and found that adaptive concretization is better than the winner on 6 of 9 benchmarks and competitive on the remaining benchmarks.

4.6.1 Concretization Probability

The first question we address in this section is:

Research Question 1 *How does the smooth probability function compare to the discontinuous probability function?*

Table 4.2 compares both functions on our full benchmark suite, running on 32 cores. For each benchmark, we list its lines of code, followed by the results under the discontinuous probability function, the smooth probability function, and the speedup, which is the ratio of the running time under the smooth function to the time under the discontinuous function.

For each probability function, we list the median of the final degrees chosen by adaptive concretization (column d), the median number of calls to **run_trial** (column $||$), and the median running time. The columns that include running time are greyed for easy comparison, with the semi-interquartile range (SIQR) in a small font. We boldface the fastest time in each row.

Overall, the degrees chosen by both functions are very similar in the sense that they usually are within a factor of two, which indicates that the climbing phase ended in about the same range. The two probability functions are about the same in terms of performance. Indeed, each function outperforms the other one for half of the benchmarks. The median speedup is 1.0, the average is 1.039, and the variance is 0.05.

We applied *Mann-Whitney U test* [56], which tests whether one of given sample sets is consistently better than the other, to the performance results under the two probability functions. Notice that this test is different from Wilcoxon signed-rank test [92] that we used to compare two degrees during the adaptive concretization.

The main difference is the category of input samples: Wilcoxon signed-rank test is applicable to repeatable samples on the single same benchmark, whereas Mann–Whitney U test is applicable to two performance result samples from the whole benchmark set.

According to the statistical test, we cannot reject the alternative hypothesis that either performance set is exceeding the other, due to a very poor confidentiality: 0.88. Therefore, from a statistical point of view, it is difficult to identify which one strictly outperforms the other, hence it is fairly safe to choose either function. The smooth probability function is preferable, since it is much intuitive due to the lack of design choices, *i.e.*, magic numbers. In the remainder of the chapter, all experiments use the new, smooth probability function.

Although there are no noticeable outliers, in the next subsection we investigated some cases where the smooth function performs better and some cases where the discontinuous function performs better, to get a better understanding of the algorithm.

In the following discussion, we refer to unknowns as *holes*, which is SKETCH's internal terminology for unknowns. Holes are named by prefixing their unique id with `H__`, *e.g.*, `H__26`. Sometimes the same syntactic hole may appear multiple times in the SAT formula due to inlining a function call or unrolling a loop. In this case, the hole name is appended with addition unique IDs, *e.g.*, `H__26_22` and `H__26_23` are two instances of the same original syntactic hole `H__26`.

While investigating the experimental results, we found the benchmarks can be divided into three general categories: those with many influential holes; those

Degree 16	**Discontinuous**		**Smooth**	
Success Rate	0 / 592		0 / 559	
Success Time (ms)	N/A		N/A	
Fail Time (ms)	2,598		343	
Max Search Space	5.37e8		2.75e12	
Concretization Histogram				
H__26_23	321	(0)	559	(0)
H__26_22	312	(0)	559	(0)
H__27_19_35	36	(0)	16	(0)
H__8_23_25	18	(0)	17	(0)
...	...		...	

Degree 512	**Discontinuous**		**Smooth**	
Success Rate	0 / 624		0 / 320	
Succeeded Tm(ms)	N/A		N/A	
Failed Tm(ms)	10,243		9,485	
Max Search Space	131,072		4,194,304	
Concretization Histogram				
H__26_23	301	(0)	140	(0)
H__26_22	322	(0)	160	(0)
H__27_27_35	56	(0)	11	(0)
H__30_27_33	49	(0)	7	(0)
...	...		...	

Degree 2048	**Discontinuous**		**Smooth**	
Success Rate	1 / 523		37 / 320	
Success Time (ms)	764,183		757,921	
Fail Time (ms)	64,292		109,731	
Max Search Space	32,768		1,024	
Concretization Histogram				
H__29_26_29	264	(1)	81	(6)
H__30_26_35	276	(1)	76	(9)
H__28_26_33	239	(1)	69	(3)
H__27_26_35	286	(0)	80	(7)
...	...		...	

Figure 4.3: Hole concretization histograms for `a_mom_2`.

with few influential holes; and those with a lot of symmetry. The smooth probability function tends to work better for the first two, and the discontinuous function better for the last one. We discuss each category in depth next, using concrete examples.

Many Influential Holes. Suppose there are many holes that are influential, but not above the cutoff of 1500 hard-coded into the discontinuous function. Then the smooth function tends to do better because it gives these holes a much higher probability of concretization, whereas the discontinuous function caps the probability at

0.5.

As an example, Figure 4.3 shows hole concretization statistics and histograms for `a_mom_2` at low, medium, and high degrees: 16, 512, and 2048, respectively. In each table, for each probability function, we list the success rate (how often we find a solution out of how many trials); the median time for a successful trial; the median time for a failed trial; and the maximum search space size.

We also give a partial histogram of the most often concretized holes, where the number indicates the count of times a hole was concretized in the trials, and the parenthesized number indicates in how many of those times synthesis was successful when that hole was concretized. For example, under the discontinuous function at degree 2048, hole `H__29_26_29` was concretized 264 times (out of 523 trials), and 1 concretization (out of 264) was in a trial that succeeded.

Looking at the table for degree 16, we see that under the discontinuous function, even the most influential holes (`H__26_23` and `H__26_22`) are concretized in at most half the trials, whereas they are always concretized under the smooth function. As a result, the maximum search space under the smooth function is four orders of magnitude larger. However, the failed trials also speed up, here by a factor of seven, thus leading the smooth function to give up sooner at this low degree. Under both functions the probability of success is extremely low: no successful trials.

While the search algorithm climbs up to degree 2048, however, the two functions behave differently. Under the discontinuous function, those influential holes still have a 0.5 probability of concretization. In contrast, under the smooth function, a concretization probability of those holes have dropped through 0.5 to around 0.16.

		Discontinuous		**Smooth**	
Degree 16	Success Rate	0 / 536		8 / 323	
	Success Time (ms)	N/A		39,600	
	Fail Time (ms)	3,792		6,674	
	Search Space	2.028e+31		2.882e+17	
	Concretization Histogram				
	H__17_10_1_0	164	(0)	50	(2)
	H__17_10_2_1_0	160	(0)	51	(1)
	H__0_10_2_1_1_2	49	(0)	21	(1)
	H__13_10_1_0_4_1	24	(0)	15	(1)
	H__1_10_0_4_2_2	24	(0)	8	(1)
	(... and more than 1,200 similar holes)				

Figure 4.4: Hole concretization statistics and histogram for `ar_s_7`.

Thus the overall level of concretization is much smaller, as is the concretization search space (1,024 for smooth versus 32,768 for discontinuous).

In sum, many holes in this benchmark are equally influential, which makes it hard to find an optimal degree as well as a right amount and subset of holes for concretization. Our new smooth function achieves slightly better performance, for two reasons. First, when degrees are low, it can quickly climb up, thanks to faster individual trials caused by aggressive concretization. Second, when a degree is high enough, it can balance the amount of concretization and the running time of individual trials by generously lowering the concretization probability of influential holes.

Few Influential Holes. If there are few influential holes, then it is better to use symbolic search rather than explicit search. We observed that under the smooth function, holes with smaller influence tend to have a lower probability of concretization; thus the smooth function yields more symbolic search, which in turn achieves slightly better performance.

		Discontinuous	**Smooth**
Degree 16	Success Rate	10 / 110	6 / 60
	Success Time (ms)	37,540	83,740
	Fail Time (ms)	4,586	12,359
	Search Space	7.556e+22	6.872e+11

Figure 4.5: Hole concretization statistics for `ar_sum`.

For example, Figure 4.4 shows the concretization statistics and histogram for `ar_s_7` at degree 16. This benchmark is very exceptional in that it has many equally unimportant holes—more than 1,200 of them. Since their influence is small, the probability of concretizing them under the smooth function is lower than under the discontinuous function. For example, the concretization probability of the most influential hole, `H__17_10_1_0`, is around 0.3 under the discontinuous function, whereas under the smooth function it is around 0.15. As a result, fewer holes are concretized under the smooth function, and the search space is sixteen orders of magnitude smaller.

Symmetry. If the synthesis problem has a lot of symmetry, then concretization helps in general, because there's a high probability of finding a solution. In this case, the discontinuous function does better because it tends to concretize more.

For example, Figure 4.5 shows the concretization statistics for `ar_sum`. This benchmark is very similar to `ar_s_7` in that it has many low-influence holes. However, it also has many solutions, and thus even aggressive concretization under the discontinuous function has a relatively good chance of concretizing correctly.

In this particular example, the discontinuous function has a search space that is 11 orders of magnitude larger. This huge search space makes both successful

Bench mark	**Degree**								
	16	32	64	128	256	512	1024	2048	4096
`p_button`	7	6	5	5	6	6	7	8	18
`p_menu`	1	1	2	163	196	196	196	196	196
`l_prepend`	>1M	>1M	>1M	908,067	474	19	6	4	4
`l_min`	1	1	0	4	52	337	597	142	150
`a_mom_1`	7,717	7,324	4,198	1,832	2,403	1,610	228	103	61
`a_mom_2`	>1M	>1M	>1M	>1M	>1M	>1M	8,697	1,571	641
`hd_14_d5`	23,142	10,010	4,184	N/A	N/A	N/A	N/A	N/A	N/A
`hd_15_d5`	5,299	29	0	0	N/A	N/A	N/A	N/A	N/A
`s_log2`	918	626	472	399	∞	∞	N/A	N/A	N/A
`s_rev`	>1M	>1M	>1M	39,290	13,111	146	N/A	N/A	N/A

Table 4.3: Expected running time (s) using empirical success rate. Fastest time in dark grey, second-fastest in light grey, ∞ if all failed trials exceed the 2-hour timeout, and N/A if there are no failed cases.

and failed trials around twice as fast as the ones under the smooth function. But the empirical success rates are similar (10/110 vs. 6/60). Thus, the discontinuous function, which has faster individual trials, outperforms the smooth function.

4.6.2 Degree/Time Tradeoff Curve

A critical hypothesis underlying adaptive concretization is that there exists an optimal degree such that the farther away from the optimal degree, the slower the running time (*i.e.*, the running time forms a "vee" around the optimal degree). We should confirm this before investigating other research questions:

Research Question 2 *Under the smooth probability function, do the expected running times across all concretization degrees form a vee shape around the optimal degree?*

To answer this question, we created a database of individual trials of Sketch run on a particular benchmark under a given degree (*i.e.*, the run on Sketch in

run_trial in Fig. 4.2). We gathered the data from the 13 runs used to generate Table 4.2, and we also ran extra trials for various benchmark/degree combinations to gather more information (more details below). We used a 2-hour timeout for the extra trials. For each trial the database records whether it succeeded or failed, the running time t, and the size of the concretization space n.

We assume the running is single-threaded, one trial after another, and compute the expected time to success as $t * n$, where t is the median running time of failed trials, and n is search space of the failed trials after random concretization. Then we can group the trials in the database by their benchmark and concretization degree, and compute each benchmark/degree pair's median expected running time. Table 4.2 summarizes the results. Here we give data for the two longest-running (according to Table 4.2) benchmarks from each category.

There are a few exceptional cases in *SyGuS* and SKETCH benchmarks. For *SyGuS* benchmarks, `hd_*_d5`, starting from degrees 128 or 256, randomly concretizing influential holes always succeeds in finding a solution, thanks to the low level of concretization as well as the symmetry in those benchmarks. For `s_log2`, at degrees 256 and 512, all the failed trials exceed the 2-hour timeout, hence ∞ expected running time. Starting from degree 1024, similar to *SyGuS* benchmarks, random concretization always succeeds as well. The other SKETCH benchmark, `s_rev`, has the same behavior at the same degrees (from 1024 to 4096). In case of no failed cases, we cannot *expect* the running time of the random concretization because the (empirical) success rate is 1. (Such cases are labeled as N/A.) In terms of the optimal degree of the adaptive concretization, those degrees are out of scope, since the adaptive concretization eventually settles earlier on other beneficial degrees.

Except for those cases, we can generally see that the running times indeed form a “vee” around the optimal degree, *i.e.*, performance gets worse the farther away from optimal in either direction. This matches our previous result for the discontinuous function, and it suggests that hill climbing and binary search can successfully find an optimal degree. The table also shows that the optimal degree varies across all benchmarks; indeed, all degrees except 1024 are optimal for at least one benchmark. This confirms our assumption that there is no fixed optimal degree, and necessitates our adaptive search algorithm.

In addition, we also use the resampling method [27] to check if our sample size is reasonable large to give us a reliable answer to Research Question 2. Specifically, for each benchmark and for each degree, if there are n trials in the database we resample from these trials n times, with replacement, and call the collected trials a bootstrap sample. Then based on these bootstrap samples, we compute the median expected running time for each benchmark/degree again, using the same formulation we set forth above. Although the expected running times are slightly different from those in Table 4.3 for most cases, bootstrap samples still show us the same “vee”, with the same optimal degree. This experiment shows that what we reported in Table 4.3 is solid enough and reliable.

4.6.3 Wilcoxon Test Threshold

Now that we have data about the optimal degree of each benchmark, we can ask whether adaptive concretization actually finds it. Recall that the algorithm is parameterized by a threshold T for the p-value of the Wilcoxon test. Thus, we actually want to ask:

Research Question 3 *How is adaptive concretization's search affected by the threshold T?*

We could try to answer this by running adaptive concretization many times, but since we already have a database of trials, there is a better way: We can perform a *simulation* in which we run the algorithm, but instead of running SKETCH itself, we randomly (with replacement) pick an appropriate benchmark/degree trial result from the database and return that from **run_trial**.

More concretely, we replace **run_trial** with a new function **sample_trial** in Figure 4.6. Here global variable `t` simulates the wall-clock time for the whole algorithm. Each time we retrieve a trial from the database, we add its time to the running wall-clock time and then return the estimated time to success.

We also simulate parallelized version of the algorithm using a single thread. Specifically, if there are n workers in the pool and the manager dispatches m trials at a time, the sample_trial function will sample failed trials m times with replacement, and get their running time Δ_1 through Δ_m. Then assuming that every worker is fully utilized in the long rung and $n \leq m$, sample_trial can simply return the running

```
t ← 0 /* “wall-clock” time measurement */
sample_trial (degree)
  sample a failed trial with specified degree, and
  get the running time $\Delta$ and concretization space size $S$.
  ($\Delta$, S) ← sample_from_database()
  t ← t + $\Delta$
  return ($\Delta$/S)
```

Figure 4.6: Sampling trials in the database in lieu of actual SKETCH runs.

results from the m trials, and advance the global time by $\big(\sum_{i\in[1,m]} \Delta_i\big)/n$.

Using this approach, we simulated 32-core adaptive concretization runs with T set at five thresholds: 0.001, 0.05, 0.1, 0.2 and 0.5. (Recall that a small number means we need more trials before reaching that significance level.) For each benchmark b and for each threshold p, we run AC on b with p as the p-value, for 301 times. For each benchmark/threshold combination, we counted the most often found degree and the median time taken by adaptive concretization to find a fixed degree.

Note that the sampled trials might be insufficient for the Wilcoxon test to produce a small enough p-value, and resampling more trials from the database won't help. In that case, we run extra trials with the current concretization degree, and add them to the database and restart the simulation from scratch. This iteration continues until the mode degree becomes obvious, *i.e.*, the mode degree won't change no matter what the unsettled runs' results would be. For each simulation that is still stuck on comparing degrees d_1 and d_2 due to the insufficient number of samples for d_1 or d_2 in the database, we assume every candidate degree (multiple of 16) within $[d_1, d_2]$ may be chosen with the same probability. For example, if a simulation stop

Bench	**Optimal**		$T = 0.001$			$T = 0.05$		
mark	**Degree**		Degree		Tm	Degree		Tm
p_button	608	128	2056	2064	99	2064	2056	99
p_menu	30	16	2056	3076	251	44	40	231
l_prepend	2048	4096	4096	104	4199	48	4096	2310
l_min	64	32	2048	N/A	79K	2048	52	70K
a_mom_1	3328	4096	3328	2560	69K	3072	3328	56K
a_mom_2	4096	2048	3328	3320	289K	3328	3712	299K
hd_14_d5	64	32	64	N/A	224K	64	16	216K
hd_15_d5	64	32	64	N/A	110K	64	16	107K
s_log2	128	64	512	16	135K	512	16	135K
s_rev	512	256	512	48	81K	512	48	74K

Bench	$T = 0.1$			$T = 0.2$			$T = 0.5$		
mark	Degree		Tm	Degree		Tm	Degree		Tm
p_button	2080	2064	95	16	2080	90	16	48	N/A
p_menu	44	44	165	16	40	131	16	32	97
l_prepend	48	4096	2175	48	4096	1697	48	16	981
l_min	2048	16	64K	2048	16	52K	2048	16	41K
a_mom_1	3072	3584	50K	3072	16	47K	4096	16	26K
a_mom_2	3328	3712	301K	3328	3712	292K	32	16	270K
hd_14_d5	64	16	217K	64	16	208K	64	16	181K
hd_15_d5	64	16	107K	64	16	102K	64	16	94K
s_log2	512	48	133K	512	48	125K	512	16	N/A
s_rev	512	48	73K	512	48	69K	512	48	71K

Table 4.4: Simulating 32-core adaptive concretization as T varies: Mode degree found and Median time taken. Most often found degrees in large text and second most often found degrees in small text.

at $[16, 48]$, we count each degree from $\{16, 32, 48\}$ as being chosen 1/3 time.

Table 4.4 summarizes our simulation results and compares them with the optimal degree based on all the trials in the database. The mode degrees are shown in large text and the second most often found degrees are shown in small text. Similarly, we show the optimal degrees in large text and the second-to-optimal degree in small text.

Experiments also show that the random nature of the adaptive concretization algorithm makes it robust with moderate thresholds ($T = 0.05$, 0.1 or 0.2): the

Bench mark	Influence Weights B													
	1/8		**1/4**		**1/2**		**3/4**		**1**		**4/3**		**2**	
`p_button`	42	4	35	8	46	10	41	6	38	10	42	6	50	8
`p_menu`	71	22	87	37	72	28	62	13	55	15	60	18	91	37
`l_prepend`	18	2	19	2	17	1	17	2	17	2	17	2	18	2
`l_min`	29	7	26	4	34	10	27	10	35	17	29	7	43	20
`a_mom_1`	211	32	204	24	198	12	188	18	185	37	199	30	214	26
`a_mom_2`	1,004	153	699	122	848	98	811	80	889	126	905	137	822	98
`hd_14_d5`	313	125	175	48	237	70	286	110	231	92	197	63	211	68
`hd_15_d5`	300	157	188	141	178	56	227	28	284	40	257	40	149	105
`s_log2`	222	161	444	461	136	227	159	271	137	310	390	212	84	70
`s_rev`	52	51	69	20	44	18	70	32	64	22	116	59	52	34

Table 4.5: Comparing influence weights of boolean nodes.

found degrees are very similar, and lower thresholds usually take just slightly less time to find a degree than higher thresholds take. However, extreme thresholds ($T = 0.001$ or 0.5) are clearly not good choices: on the one hand, when the threshold is extremely high, the Wilcoxon test usually cannot conclude which degree is better, and then algorithm tends to climb to very high degrees for most benchmarks; on the other hand, when the threshold is extremely low, the Wilcoxon test's results can be easily affected by random noises, and the algorithm will stop at very low degrees too often. In summary, the simulation results suggest that any threshold between 0.05 and 0.2 is reasonable.

4.6.4 Influence Computation

To evaluate the choice of B, the ratio between choice nodes and other booleans, we ran a subset of our benchmarks on seven ratios: 1/8, 1/4, 1/2 (our previous choice), 3/4, 1, 4/3, and 2. Notice the last two ratios weigh arbitrary booleans as more important than choice nodes. We used the same subset of benchmarks as

Tables 4.3 and 4.4 in Section 4.6.2. We ran each benchmark/B combination thirteen times on 32 cores. Table 4.5 shows the results. As in Table 4.2, the columns show median running time, with the SIQR in a small font, and we highlight fastest and second-fastest times in each row.

From these results, the fastest running times appear across all degrees except for 1/8, though the second-smallest weight, 1/4, typically has many fastest running times. This reinforces our intuition that choice nodes should be more influential than boolean nodes. However, the performance differences are not that huge. In order to see whether there exists a degree that outperforms everything else, we applied Mann-Whitney U test again to all possible pairs of ratios. The confidentiality from those tests ranges from 0.35 to 0.96, which simply implies that there is no best ratio at all. This rather indicates that our influence computation is not sensitive to the ratio between choice nodes and other booleans.

4.6.5 Performance Results

The right columns of Table 4.6 show our results. The columns that include running time are greyed for easy comparison, with the semi-interquartile range (SIQR) in a small font. (We only list the running times SIQR to save space.) The median is ∞ if more than half the runs timed out, while the SIQR is ∞ if more than one quarter of the runs timed out. The first grey column lists SKETCH's running time on one core. The next group of columns reports on adaptive concretization, run on 32 cores. The first column in the group gives the median of the final degrees chosen

Bench	SKETCH		**Adaptive**				**Non-Adaptive**		
mark	Time (s)		*d*	||	Time (s)		||	Time (s)	
`p_button`	60	14	2,048	592	**46**	10	205	23	2
`p_color`	**12**	4	640	336	21	2	61	12	4
`p_menu`	`OOM`		3,072	601	**72**	28	103	22	6
`l_prepend`	19	4	256	151	**17**	1	53	12	2
`l_min`	135	29	256	204	**34**	10	30	20	2
`a_mom_1`	**164**	18	1,024	222	198	12	130	246	3
`a_mom_2`	**700**	102	2,048	219	848	98	88	1,153	112
`ar_s_4`	**4**	0	16	3	5	0	4	4	0
`ar_s_5`	6	2	16	11	**5**	1	20	5	0
`ar_s_6`	10	2	16	15	**9**	0	46	9	1
`ar_s_7`	47	4	16	37	**40**	10	142	41	26
`ar_sum`	277	129	32	8	**55**	32	8	61	24
`hd_13_d5`	47	11	32	15	**8**	1	30	8	2
`hd_14_d1`	111	54	52	107	**22**	6	164	16	3
`hd_14_d5`	1,296	323	32	10	**237**	70	3	194	35
`hd_15_d5`	447	206	32	12	**178**	56	6	213	39
`s_cg`	12	2	32	73	**11**	2	259	16	2
`s_log2`	424	360	128	109	**136**	227	4	62	14
`s_logcnt`	447	889	32	37	**25**	46	71	18	8
`s_rev`	209	101	128	118	**44**	18	13	57	24
`deriv2`	18	3	16	25	**8**	1	89	12	8
`deriv3`	22	4	32	20	**8**	2	14	7	1
`deriv4`	11	2	32	18	**5**	0	9	5	1
`deriv5`	12	2	32	9	**5**	2	23	6	1
`q_noti`	13	4	64	125	**7**	2	38	6	1
`q_serv`	82	32	16	5	**22**	4	19	23	2

Table 4.6: Comparing SKETCH, adaptive, and non-adaptive concretization.

by adaptive concretization. The next column lists the median number of calls to **run_trial**. The last column lists the median running time. Lastly, the right group of columns shows the performance of our algorithm on 32 cores, assuming we skip the adaptation step and jump straight to running with the median degree shown in the table. For example, for `p_button`, these columns report results for running starting with degree 2,048 and never changing it. We again report the number of trials and the running time.

Comparing Sketch and adaptive concretization, we find that adaptive concretization typically performs better. In the figure, we boldface the fastest time between those two columns. We see several significant speedups, ranging from 18× for `s_logcnt`, 6× for `hd_14_d5`, and 5× for `hd_14_d1`, `ar_sum`, and `s_rev` to 4× for `l_min` and `q_serve` and 3× for `s_log2` and `hd_15_d5`. For `p_menu`, Sketch reliably exceeds our 32GB memory bound and then aborts, whereas our algorithm succeeds, mostly around one minute. Overall, adaptive concretization performed better in 22 of 26 benchmarks, and about the same on one benchmark.

On the remaining benchmarks (`p_color`, `a_mom_1`, `a_mom_2`, and `ar_s_4`), adaptive concretization's performance was within about a factor of two. Comparing other similarly short-running benchmarks, such as `ar_s_5` and `ar_s_6`, where the final degree (16) was chosen very early, the degree search process needed to spend more time to reach bigger degree, resulting in the slowdown.

Next we compare adaptive concretization to non-adaptive concretization at the final degree. In ten cases, the adaptive algorithm is actually faster, due to random chance. In the remaining cases, the adaptive algorithm is either about the same as non-adaptive or is at worst within a factor of approximately three.

4.6.6 Parallel Scalability and Comparison to SyGuS Solvers

We next measured how adaptive concretization's performance varies with the number of cores, and compare it to the winner of the SyGuS competition. Table 4.7 shows the results. The first two columns are the same as Table 4.6. The next three

Bench	SKETCH		**# Cores** (Time (s))						**Enum**	
mark	Time (s)		**1**		**4**		**32**		Time(s)	
`p_button`	60	14	201	∞	**43**	12	46	10		
`p_color`	**12**	4	53	8	22	3	21	2		
`p_menu`	`OOM`		∞		707	∞	**72**	28		
`l_prepend`	19	4	38	8	**15**	2	17	1		
`l_min`	135	29	369	185	48	38	**34**	10		
`a_mom_1`	**164**	18	812	200	279	36	198	12		
`a_mom_2`	**700**	102	1,860	150	1,071	270	848	98		
`ar_s_4`	4	0	5	0	**2**	0	5	0	1,804	44
`ar_s_5`	6	2	7	2	**3**	0	5	1	∞	
`ar_s_6`	10	2	10	2	**6**	0	9	0	∞	
`ar_s_7`	47	4	93	38	**32**	9	40	10	∞	
`ar_sum`	277	129	63	76	**45**	38	55	32	∞	
`hd_13_d5`	47	11	13	9	10	2	**8**	1	8	0
`hd_14_d1`	111	54	66	14	**19**	4	22	6	8	0
`hd_14_d5`	1,296	323	623	403	238	102	**237**	70	201	1
`hd_15_d5`	447	206	765	402	304	55	**178**	56	424	13
`s_cg`	12	2	**10**	1	**10**	0	11	2		
`s_log2`	424	360	467	342	663	694	**136**	227		
`s_logcnt`	447	889	340	342	**25**	26	**25**	46		
`s_rev`	209	101	304	89	86	121	**44**	18		
`deriv2`	18	3	26	6	10	2	**8**	1		
`deriv3`	22	4	35	17	**8**	1	**8**	2		
`deriv4`	11	2	18	4	**5**	1	**5**	0		
`deriv5`	12	2	18	4	7	0	**5**	2		
`q_noti`	13	4	13	4	**7**	2	**7**	2		
`q_serv`	82	32	26	7	**21**	6	22	4		

Table 4.7: Parallel scalability of adaptive concretization.

columns show the performance of adaptive concretization on 1, 4, and 32 cores. We discuss the rightmost column shortly. We boldface the fastest running time among SKETCH, 1, 4, and 32 cores.

The results show that, in the one-core experiments, adaptive concretization performs better than regular SKETCH in 9 of 26 cases. Although adaptive concretization is worse or times out in one case, its performance improves with the number of cores. The 4-core runs are consistently close to or better than 1-core runs; in one case, benchmarks that time out on 1 core succeed on 4 cores. At 32 cores, we see the best performance in 15 of the 26 cases, with a speedup over 4-core runs ranging up to 10×.

SyGuS Benchmarks and Solvers. Finally, the rightmost column of Table 4.7 shows the performance of the Enumerative CEGIS Solver, which won the SyGuS'14 Competition [4]. As the Enumerative Solver does not accept problems in Sketch format, we only compare on benchmarks from the competition (which uses the SyGuS-IF format, which is easily translated to a sketch). We should note that the enumerative solver is not parallelized and may be difficult to parallelize.

Adaptive concretization is faster for 6 of 9 benchmarks from the competition. It is also worth mentioning the Enumerative Solver actually won on the four benchmarks beginning with `hd_`. Our results show that adaptive concretization outperforms it on one benchmark and is competitive on the others.

4.7 Literature Review

There have been many recent successes in sampling-based synthesis techniques. For example, Schkufza et al. use sampling-based synthesis for optimization [73, 74], and Sharma et al. use similar techniques to discover complex invariants in programs [76]. These systems use Markov Chain Montecarlo (MCMC) techniques, which use fitness functions to prioritize sampling over regions of the solution space that are more promising. This is more sophisticated sampling technique than what is used by our method. We leave it to future work to explore MCMC methods in our context. Another alternative to constraint-based synthesis is explicit enumeration of candidate solutions. Enumerative solvers often rely on factoring the search space, aggressive pruning and lattice search. Factoring has been very successful for programming by example [35, 38, 77], and lattice search has been used in synchronization of concurrent data structures [91] and autotuning [11]. However, both factoring and lattice search require significant domain knowledge, so they are unsuitable for a general purpose system like SKETCH. Pruning techniques are more generally applicable, and are used aggressively by the enumerative solver compared against in Section 4.6.

Recently, some researchers have explored ways to use symbolic reasoning to improve sampling-based procedures. For example, Chaudhuri et al. have shown how to use numerical search for synthesis by applying a symbolic smoothing transformation [22, 21]. In a similar vein, Chaganty et al. use symbolic reasoning to limit the sampling space for probabilistic programs to exclude points that will not satisfy a specification [20]. We leave exploring the tradeoffs between these approaches as future work.

Finally, there has been significant interest in parallelizing SAT/SMT solvers. The most successful of these combine a portfolio approach—solvers are run in parallel with different heuristics—with clause sharing [37, 93]. Interestingly, these solvers are more efficient than solvers like PSATO [95] where every thread explores a subset of the space. One advantage of our approach over solver parallelization approaches is that the concretization happens at a very high-level of abstraction, so the solver can apply aggressive algebraic simplification based on the concretization. This allows our approach to even help a problem like `p_menu` that ran out of memory on the sequential solver. The tradeoff is that our solver loses the ability to tell if a problem is UNSAT because we cannot distinguish not finding a solution from having made incorrect guesses during concretization.

Chapter 5: Future Work

In this dissertation, we have shown that program synthesis can enhance the effectiveness of symbolic execution by automatically creating a framework model. Although the results are promising, there is still room for improvement, and this chapter will discuss the future work.

5.1 Towards Synthesis-Aided Symbolic Execution

In particular, the process of generating drivers and properties of interest can be automated via program synthesis, too. A synthesized driver (Section 5.1.1), together with synthesized properties to check (Section 5.1.2), will make it easier to utilize symbolic execution. We can also exploit program synthesis to derive search strategies that can effectively drive symbolic execution towards interesting program points (Section 5.1.3).

5.1.1 Synthesizing Drivers

In general, SymDroid drivers, as shown in Figure 2.16, can be divided into two parts: 1) events, *e.g.*, user interactions (line 72 in the figure), which drive SymDroid towards desired program points, and 2) properties to check at such desired points (line 83). Since events typically induce implicit calls, we can automatically generate the event sequence as long as we can identify potential implicit control-flows from the call graph of the app. One solution is to learn callback relations from the API documentation, *e.g.*, method onClick of the registered OnClickListener instance will be invoked to handle the user's button click.

The latter, however, is harder to automate than the former because it depends on problems users want to verify via symbolic execution. We propose to apply program synthesis to generate drivers, especially inferring properties of interest. As a starting point, we could manually write as many drivers as possible that explore a set of apps and discover interesting behaviors, *e.g.*, under what conditions those apps access privacy-sensitive user data. Those hand-written drivers can be useful not only to summarize app behavior from the viewpoint of privacy but also to learn how to derive properties to check from examples.

5.1.2 Synthesizing Properties of Interest

Recall that path conditions are used to filter out infeasible paths. Likewise, during symbolic execution, properties comprised of such symbolic expressions can be checked by a SMT solver. The question is, what properties should we check and where should we check them.

We propose to synthesize properties of interest from examples so that even end-users can easily articulate properties. We will again use constraint-based, inductive synthesis, using samples to learn properties to be synthesized. Similar to Pasket, samples will be logged behavior that users want to detect. The key idea behind our proposal is that no explicit templates are required to synthesize properties of interest, assuming an app's bytecode is available. Using the input call traces as well as the bytecode, we can extract candidate expressions that appear along the traces, and those expressions will be used to compose properties of interest. A similar idea was explored by `PINS` [86], where its goal is to synthesize an inverse program that reuses the structure of the target program with small tweaks.

Recall the running example, where the app can read user contacts right after it launches because of the default option settings. Suppose a user wants to prevent an app from reading her contacts before she explicitly allows it, aside from the consent at installation. In this context, she will provide the sample log in Figure 5.1a, which reflects the aforementioned behavior: reading user's contacts right after launching (on line 3).

Figure 5.1b depicts how to extract candidate expressions from the bytecode. Since the bytecode is much simpler than source code, we need to keep track of values in the registers, along with invoked methods that could be used as arbitrary predicates. For example, in the event handlers that appear along the sample trace, we can extract p1.getItemId and p0.getId. In a similar way, many other candidate expressions can be collected as well.

Figure 5.1c shows how to generate a sketch template that explores all possible combinations of candidate expressions. We will regard all program structures as integers by assigning unique class/method/field ids as in Section 3.5. Using Sketch generator we can encode a generic function that tries a certain number of mathematic

```
Splash.onCreate(Splash@1, null)
  Settings.sync(Splash@1)
    ContentResolver.query (...)
     ...
Splash.onResume(Splash@1)
Splash.onPause(Splash@1)
...
KeyEvent(..., KEYCODE_MENU)
KeyEvent(..., "Setting", ...)
Home.onMenuItemClick(...)
...
```

(a) Example log.

```
Home.onMenuItemClick:
  invoke-virtual {p1}, ...;. getItemId()
  move-result v2    // v2 → p1.getItemId
  if-eqz v2, 0002d // (p1.getItemId == 0)
   ...

Settings.onClick:
  invoke-virtual {p0}, ...;. getId()
  move-result v0  // v0 → p0.getId
  if-eqz v0, 00fd // (p0.getId == 0)
   ...
```

(b) Candidate expressions from the bytecode.

```
generator int gen_property(int v1, int v2) {
  int t = ??;
  if (t == 0) return v1;
  else if (t == 1) return v2;
  else if (t == 2) return v1 * v2;
   ...  }
harness void main() {
   ... x ...  // encoding sample trace
  int v1 = getItemId(m1.p1);
  int v2 = getId(m2.p0);
  assert gen_property(v1, v2) == x; }
```

(c) Template using candidate expressions.

Figure 5.1: Sample and template for property synthesis.

equations for inputs. Sample traces are encoded and in turn used to enforce the shape of the property to be synthesized. SKETCH will resolve all the constraints and find a solution (the hole variable t) that will inform us what the property looks like (only choosing the first expression in this example).

5.1.3 Synthesizing Search Strategy

Currently, SymDroid runs symbolic execution to completion, but this is impractical. Most symbolic executors include *search strategies* that visit a subset of the program paths that are heuristically likely to be "interesting." Notice that search strategies are different from drivers because search strategy determines where to proceed at branching points during execution, whereas drivers imitate user interactions. We will experiment with several different strategies for SymDroid, such as round-robin [17], generational order [34], shortest-distance and call-chain-backward execution [55].

Besides, as a first step, we propose to represent a search strategy as a mapping from testing predicates to search directions, which is very different from typical search strategies. For example, it could look like: $(\mathsf{mtd.name} = \mathsf{onActivityResult} \wedge \mathsf{type(rA)} = \mathsf{Intent}) \rightarrow \mathtt{T}$, which means that if the current method is onActivityResult and the type of the first operand is Intent, the true branch will be taken because that that method is likely invoked with a result Intent and thus the corresponding null check is likely bogus. Such predicates will be evaluated at analysis time, and we can incrementally extend the expressiveness of those predicates.

5.2 Towards Full Automation of Framework Synthesis

The whole process of framework synthesis in Pasket is not fully automated. It still requires human efforts to find a right set of tutorials and design patterns that are sufficient to generate a preferred framework model. Besides, running the selection of tutorials and collecting logs are also performed by hand. In this section, we propose to introduce a much higher degree of automation to aforementioned tasks: running tutorials using coverage-based test automation, generating artificial tutorials using internal knowledge about the target framework, and presenting design patterns or programming idioms via pattern template.

5.2.1 Gray-Box Testing

Recall that, when we collected logs, we observed only calls that cross the edge between tutorials and the framework. In this setting, we regard both tutorials and the framework as black box, *i.e.*, we do not use their internal structures and workings, assuming their source code is not available. However, that assumption is not always true. In particular, tutorials are publicly accessible for almost all cases, to encourage correct and effective uses of the framework. Also, the source code of frameworks is sometimes available, *e.g.*, Android is open-sourced. We can certainly improve the quality of gathered logs, the process of collecting logs, and the process of selecting (or generating) appropriate tutorials.

Increasing the Coverage in Tutorials. Tutorials used by Pasket evaluation were quite small, and thus only one or two user interactions, such as button clicks, were good enough to explore all possible paths in those tutorials. However, in general, manual testing might miss unexpected cases. By adopting Automated Gray-Box Testing [52], we can automatically explore multiple paths in a tutorial while increasing the coverage of the tutorial. The key idea is similar to symbolic execution in the sense that the test automation maintains path conditions to cover both feasible branches at branching points.

Generating Artificial Tutorials. Once we recognize certain features of the framework that are never captured by existing tutorials, we try to generate our own test code that exercises those unexplored parts in the framework. If the source of the framework is available, we can still utilize Gray-Box Testing, but in this case, the goal is to increase the coverage of the corresponding part of the framework. Then, we can regard the generated test code as an artificial tutorial, and repeat the same process in Pasket.

5.2.2 Pattern Templates

Currently, there is a gap between design pattern representations (Figures 3.5 and 3.6) and its corresponding encoding in Sketch (Figures 3.8 and 3.9). Also, we may want to search for code patterns that are not well-known design patterns. We propose to generalize design patterns as well as the translation step so that we can more easily support additional patterns or general programming idioms.

```
public class $Singleton {
  private $static Singleton ins;
  private $Singleton() {
  }
  public static $Singleton $getIns() {
    if (ins == null) {
      ins = new $Singleton();
    }
    return ins;
  }
}
```

Figure 5.2: Pattern template for the singleton pattern.

As a first step, we will extend an existing language so as to express nondeterministic roles in code patterns. For example, the singleton pattern shown in Figure 3.6a can be represented as a *pattern template* in Figure 5.2. Here, identifiers that start with $ sign are nondeterministic *role variables*, which will be in turn translated to *unknowns* in a synthesizer, *e.g.*, ?? in SKETCH. Similar to PASKET's encoding, finding correct answers for those unknowns corresponds to finding an instantiation of that code pattern.

This pattern template has several benefits. First, since it is extended from an existing language, it is easy and intuitive for end-users to represent code patterns. Second, since the template is written using a programming language, we can reuse features in that language as-is. For example, the template in Figure 5.2 uses access control (*e.g.*, a `private` constructor or a `public` method as a getter) and modifiers (*e.g.*, `static` method to retrieve a singleton instance without regard to the calling context). Third, users can represent not only structures, but also expected behaviors. In addition to the shape of the singleton pattern, method role `$getIns`'s body in Figure 5.2

articulates how to create a singleton and retrieve it.

Lastly, we can mechanize the translation from pattern template to synthesis problem. The code shape itself can be easily translated to structural constraints. For instance, from the pattern template in Figure 5.2, we can learn that method role $getIns belongs to class role $Singleton; that $getIns does not receive any arguments; and that the return type of $getIns is $Singleton. These can be encoded as follows:

```
int Singleton = ??; int getIns = ??;

assert subcls[Singleton][belongsTo[getIns]];
assert argNum[getIns] == 0;
assert retType[getIns] == Singleton;
```

Method bodies in method roles, *e.g.*, singleton creation and retrieval in $getIns, can be directly translated to the synthesizer's language (*e.g.*, using JSKETCH, which is described in Appendix A).

Note that initial values for role variables are set as general unknowns. Of course, additional syntax-guided code search might reduce the search space by replacing general holes with regular expressions over candidate classes or methods, *e.g.*, Singleton = {| 1 | 2 | $\cdots$ |}. Another technical challenge is to find candidate methods and insert guarded simulation of the pattern instantiation, such as Figure 3.9. The same syntax-guided code search would be helpful to filter out infeasible instantiations.

Chapter 6: Conclusion

In this dissertation, we claim that we can leverage *program synthesis* to introduce a high-degree of automation into the process of generating a framework model for symbolic execution. To support this thesis, we have presented three pieces of work.

SymDroid. First, we presented SymDroid, a symbolic executor for Dalvik bytecode. SymDroid actually operates on μ-Dalvik, a language with far fewer instructions than Dalvik, and to which Dalvik can be easily translated. In addition to modeling bytecode instructions, SymDroid includes limited support for system libraries. Since Android apps are event-driven, we use client-oriented specifications to model the system and drive the app under test in the desired ways. Running SymDroid against the Android Compatibility Test Suite, we found it passed all test cases that did not require more system modeling, and was only about twice as slow as the Dalvik VM running on the same machine. We also used SymDroid to discover the conditions under which the PickContact activity in the API demonstration app actually used contacts. These results suggest that, while still a prototype, SymDroid is a promising first step in direct, precise analysis of Android apps.

Pasket. Second, 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 used by three 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.

Adaptive Concretization. Lastly, we introduced *adaptive concretization*, a program synthesis technique that combines explicit and symbolic search. Our key insight is that not all unknowns are equally important with respect to solving time. By concretizing high *influence* unknowns, we can often speed up the overall synthesis algorithm, especially when we add parallelism. Since the best *degree of concretization* is hard to compute, we presented an online algorithm that uses exponential hill climbing and binary search to find a suitable degree by running many trials. We implemented our algorithm for Sketch and ran it on a suite of 26 benchmarks across several different domains. We found that adaptive concretization often outperforms

Sketch, sometimes very significantly. We also found that the parallel scalability of our algorithm is reasonable.

In addition, we empirically evaluated several of the key design choices in adaptive concretization, which we previously introduced. First, we introduced a new function to assign a concretization probability to an unknown based on its influence and the degree. Our new function assigns probability in a smooth, continuous manner, eliminating both heuristic constants and a discontinuity in the original function. We showed that both functions behave similarly. We also showed that when graphed against expected running times, the degree forms a "vee" around the optimal point, justifying adaptive concretization's degree search process.

We also explored a range of values for T, the threshold at which adaptive concretization decides that the p-value returned by the Wilcoxon Signed-Rank Test is significant enough to distinguish two degrees; and B, the ratio of the influence of arbitrary boolean unknowns versus those in guards of if-then-else nodes. We showed that many different choices for T and B work equally well, including the choices in our original algorithm.

Overall, our empirical study makes adaptive concretization simpler by introducing a new, smooth concretization probability function, and we showed that our algorithm is robust to a wide range of design decisions.

Appendix A: JSketch: Sketching for Java

When we articulate the framework synthesis, we had two design choices: encoding it directly to SAT formulae and solving them via a SAT solver versus encoding it using an existing synthesizer and then decoding the synthesizer's solution. Due to the log conformity constraints (as discussed in Section 3.3), we chose the latter because we eventually require program semantics. We opted to use Sketch, an *out-of-the-box* synthesis tool based on a C-like imperative language. Since our target frameworks, such as Swing and Android, are written in Java, we need non-trivial translations from Java to C (and vice versa). Such translation was initially baked in Pasket, and then separeted to be a standalone tool, along with a few more general-purpose features.

This chapter presents JSketch, a tool that makes sketch-based synthesis directly available to Java programmers. JSketch is built as a frontend on top of the Sketch synthesis system, a mature synthesis tool based on a simple imperative language that can generate C code [83]. JSketch allows Java programmers to use many of the Sketch's synthesis features, such as the ability to write code with unknown constants (*holes* written ??) and unknown expressions described by a *generator* (written {| e^* |}). In addition, JSketch provides a new synthe-

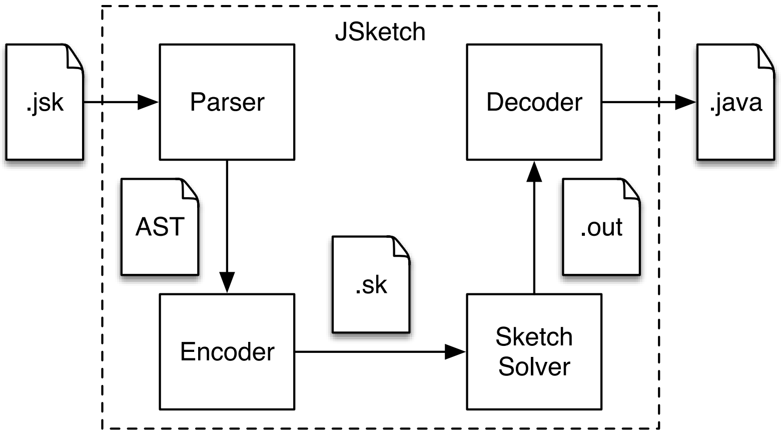


Figure A.1: JSketch Overview.

sis feature—a class-level generator—that is specifically tailored for object oriented programs. Section A.1 walks through JSketch's input and output, along with a running example.

As illustrated in Figure A.1, JSketch compiles a Java program with unknowns to a partial program in the Sketch language and then maps the result of Sketch synthesis back to Java. The translation to Sketch is challenging because Sketch is not object oriented, so the translator must model the complex object-oriented features in Java—such as inheritance, method overloading and overriding, anonymous/inner classes—in terms of the features available in Sketch.

A.1 Overview

We begin our presentation with two examples showing JSketch's key features and usage.

A.1.1 Basics

The input to JSKETCH is an ordinary Java program that may also contain unknowns to be synthesized. There are two kinds of unknowns: *holes*, written ??, represent unknown integers and booleans, and *generators*, written {| e^* |}, range over a list of expressions. For example, consider the following Java sketch[1], similar to an example from the SKETCH manual [81]:

```
class SimpleMath {
    static int mult2(int x) { return (?? * {| x , 0 |}); }
}
```

Here we have provided a template for the implementation of method mult2: The method returns the product of a hole and either parameter x or 0. Notice that even this very simple sketch has 2^{33} possible instantiations (32 bits of the hole and one bit for the choice of x or 0).

To specify the solution we would like to synthesize, we provide a *harness* containing assertions about the mult2 method:

```
class Test {
    harness static void test () { assert(SimpleMath.mult2(3) == 6); }
}
```

Now we can run JSKETCH on the sketch and harness.

```
$ ./jsk.sh SimpleMath.java Test.java
```

The result is a valid Java source file in which holes and generators have been replaced with the appropriate code.

```
$ cat result/java/SimpleMath.java
class SimpleMath { ...
static public int mult2 (int x) {
return 2 * x;
}
}
```

[1]https://github.com/plum-umd/java-sketch/blob/master/test/benchmarks/t109-mult2.java

A.1.2 Finite Automata

Now consider a harder problem: suppose we want to synthesize a finite automaton given sample accepting and rejecting inputs.[2] There are many possible design choices for finite automata in an object-oriented language, and we will opt for one of the more efficient ones: the current automaton state will simply be an integer, and a series of conditionals will encode the transition function.

Figure A.2a shows our automaton sketch. The input to the automaton will be a sequence of `Token`s, which have a `getId` method returning an integer (line 8). An `Automaton` is a class—ignore the `generator` keyword for the moment—with fields for the current `state` (line 9) and the number of states (line 10). Notice these fields are initialized to holes, and thus the automaton can start from any arbitrary state and have an arbitrary yet minimal number of states (restricted by SKETCH's `minimize` function on line 11). The class includes a `transition` function that asserts that the current state is in-bounds (line 13) and updates `state` according to the current state and the input `Token`'s value (retrieved on line 14).

Here we face a challenge, however: we do not know the number of automaton states or tokens, so we have no bound on the number of transitions. To solve this problem, we use a feature that JSKETCH inherits from SKETCH: the term `minrepeat { e }` expands to the minimum length sequence of `e`'s that satisfy the harness. In this case, the body of `minrepeat` (line 16) is a conditional that encodes an arbitrary

[2]Of course, there are many better ways to construct finite automata—this example is only for expository purposes.

transition—if the guard matches the current state and input token, then the state is updated and the method returns. Thus, the transition method will be synthesized to include however many transitions are necessary.

Finally, the Automaton class has methods transitions and accept; the first performs multiple transitions based on a sequence of input tokens, and the second one determines whether the automaton is in an accepting state. Notice that the inequality (line 21) means that states 0 up to some bound will be accepting; this is fully general because the exact state numbering does not matter, so the synthesizer can choose the accepting states to follow this pattern.

Class Generators. In addition to basic SKETCH generators like we saw in the mult2 example, JSKETCH also supports *class generators*, which allow the same class to be instantiated differently in different superclass contexts. In Figure A.2a, the generator annotation on line 8 indicates that Automaton is such a class. (Class generators are analogous to the the function generators introduced by SKETCH [81].)

Figure A.2b shows two classes that inherit from Automaton. The first class, DBConnection, has an inner class Monitor that inherits from Automaton. The Monitor class defines two tokens, OPEN and CLOSE, whose ids are 1 and 2, respectively. The outer class has a Monitor instance m that transitions when the database is opened (line 34) and when the database is closed (line 35). The goal is to synthesize m such that it acts as an inline reference monitor to check that the database is never opened or closed twice in a row, and is only closed after it is opened. The harnesses in TestDBConnection in Figure A.3 describe both good and bad behaviors.

```
interface Token{ public int getId (); }
generator class Automaton {
    private int state = ??;
    static int num_state = ??;
    harness static void min_num_state() { minimize(num_state); }
    public void transition (Token t) {
        assert 0 ≤ state && state < num_state;
        int id = t.getId ();
        minrepeat {
            if (state == ?? && id == ??) { state = ??; return; }
    } }
    public void transitions ( Iterator <Token> it) {
        while ( it .hasNext()) { transition ( it .next ()); }
    }
    public boolean accept() { return state ≤ ??; }
}
```

(a) Automaton sketch.

```
class DBConnection {
    class Monitor extends Automaton {
        final static Token OPEN =
            new Token() { public int getId () { return 1; } };
        final static Token CLOSE =
            new Token() { public int getId () { return 2; } };
        public Monitor() { }
    }
    Monitor m;
    public DBConnection() { m = new Monitor(); }
    public boolean isErroneous() { return ! m.accept(); }
    public void open() { m. transition (Monitor.OPEN); }
    public void close () { m. transition (Monitor.CLOSE); }
}
class CADsR extends Automaton { ...
    public boolean accept(String str ) {
        state = init_state_backup ;
        transitions ( convertToIterator ( str ));
        return accept ();
} }
```

(b) Code using Automaton sketch.

Figure A.2: Finite automata with JSketch.

```
class TestDBConnection {
    harness static void scenario_good() {
        DBConnection conn = new DBConnection();
        assert ! conn.isErrorneous();
        conn.open();  assert ! conn.isErroneous();
        conn.close(); assert ! conn.isErroneous(); }
    // bad: opening more than once
    harness static void scenario_bad1() {
        DBConnection conn = new DBConnection();
        conn.open(); conn.open(); assert conn.isErroneous(); }
    // bad: closing more than once
    harness static void scenario_bad2() {
        DBConnection conn = new DBConnection();
        conn.open();
        conn.close(); conn.close(); assert conn.isErroneous();
} }
class TestCADsR {
    // Lisp-style identifier : c(a|d)+r
    harness static void examples() {
        CADsR a = new CADsR();
        assert ! a.accept("c");   assert ! a.accept("cr");
        assert a.accept("car");   assert a.accept("cdr");
        assert a.accept("caar"); assert a.accept("cadr");
        assert a.accept("cdar"); assert a.accept("cddr");
} }
```

Figure A.3: Automata use cases.

The second class in Figure A.2b, CADsR, adds a new (overloaded) accept(String) method that converts the input String to a token iterator (details omitted for brevity), transitions according to that iterator, and then returns whether the string is accepted. The goal is to synthesize an automaton that recognizes `c(a|d)+r`. The corresponding harness TestCADsR.examples() in Figure A.3 constructs a CADsR instance and makes various assertions about its behavior. Notice that this example relies critically on class generators, since Monitor and CADsR must encode different automata.

Output. Figure A.4 shows the output produced by running JSketch on the code in Figures A.2 and A.3. We see that the generator was instantiated as Automaton1, inherited by DBConnection.Monitor, and Automaton2, inherited by CADsR. Both automata are equivalent to what we would expect for these languages. Two things were critical for achieving this result: minimizing the number of states (line 11) and having sufficient harnesses (Figure A.3).

We experimented further with CADsR to see how changing the sketch and harness affects the output. First, we tried running with a smaller harness, *i.e.*, with fewer examples. In this case, the synthesized automaton covers all the examples but not the full language. For example, if we omit the four-letter inputs in Figure A.3 the resulting automaton only accepts three-letter inputs. Whereas going to four-letter inputs constrains the problem enough for JSketch to find the full solution.

Second, if we omit state minimization (line 11), then the synthesizer chooses large, widely separated indexes for states, and it also includes redundant states (that could be merged with a textbook state minimization algorithm).

Third, if we manually bound the number of states to be too small (*e.g.*, manually set num_state to 2), the synthesizer runs for more than half an hour and then fails, since there is no possible solution.

Of these cases, the last two are relatively easy to deal with since the failure is obvious, but the first one—knowing that a synthesis problem is underconstrained—is an open research challenge. However, one good feature of synthesis is that, if we do find cases that are not handled by the current implementation, we can simply add those cases and resynthesize rather than having to manually fix the code (which

```
class Automaton1 {
  int state = 0; static int num_state = 3;
  public void transition (Token t) { ...
    assert 0 ≤ state && state < 3;
    if (state == 0 && id == 1) { state = 1; return; } // open@
    if (state == 1 && id == 1) { state = 2; return; } // open 2x
    if (state == 1 && id == 2) { state = 0; return; } // (init)@
    if (state == 0 && id == 2) { state = 2; return; } // close 2x
  }
  public boolean accept() { return state ≤ 1; } ...
}
class DBConnection{ class Monitor extends Automaton1 { ... } ...}
class Automaton2 {
  int state = 0; static int num_state = 3;
  public void transition (Token t) { ...
    assert 0 ≤ state && state < 3;
    if (state == 0 && id == 99) { state = 1; return; } // c
    if (state == 1 && id == 97) { state = 2; return; } // ca
    if (state == 1 && id == 100) { state = 2; return; } // cd
    if (state == 2 && id == 114) { state = 0; return; } // c(a|d)+r@
  }
  public boolean accept() { return state ≤ 0; } ...
}
class CADsR extends Automaton2 { ... }
```

Figure A.4: JSketch Output (partial).

could be quite difficult and/or introduce its own bugs). Moreover, minimization—trying to ensure the output program is small—seems to be a good heuristic to avoid overfitting to the examples.

Bibliography

[1] H. Albin-amiot, Y. gaël Guéhéneuc, and R. A. Kastler. Meta-Modeling Design Patterns: Application to Pattern Detection and Code Synthesis. In *Workshop Automating OOSD Methods*, pages 01–35, 2001.

[2] K. Ali and O. Lhoták. Averroes: Whole-Program Analysis without the Whole Program. In *ECOOP 2013*, pages 378–400, 2013.

[3] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL*, pages 98–109, 2005.

[4] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–17, 2013. URL `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679385`.

[5] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *FSE*, page 59. ACM, 2012.

[6] T. R. Andersen. Add Logging at Class Load Time, Apr. 22 2008. `https://today.java.net/article/2008/04/22/add-logging-class-load-time-java-instrumentation`.

[7] Android. Compatibility Test Suite, . `http://source.android.com/compatibility/cts-intro.html`.

[8] Android. Bytecode for the Dalvik VM, . `http://source.android.com/tech/dalvik/dalvik-bytecode.html`.

[9] Android. Android SDK, . `http://developer.android.com/sdk/index.html`.

[10] Android. Application Fundamentals, . `http://developer.android.com/guide/components/fundamentals.html`.

[11] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. P. Amarasinghe. Opentuner: an extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 303–316, 2014. doi: 10.1145/2628071.2628092. URL `http://doi.acm.org/10.1145/2628071.2628092`.

[12] M. Antkiewicz, T. T. Bartolomei, and K. Czarnecki. Automatic extraction of framework-specific models from framework-based application code. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 214–223, 2007.

[13] S. Blackshear, A. Gendreau, and B.-Y. E. Chang. Droidel: A general approach to android framework modeling. In *SOAP*, pages 19–25. ACM, 2015.

[14] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT–a formal system for testing and debugging programs by symbolic execution. In *International Conference on Reliable Software (ICRS)*, pages 234–245, 1975.

[15] P. Brady. Anatomy & Physiology of an Android. `https://sites.google.com/site/io/anatomy--physiology-of-an-android`.

[16] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 183–198, 2011.

[17] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI '08, pages 209–224, 2008.

[18] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS '15*, 2015.

[19] M. Ceccarello and O. Tkachuk. Automated generation of model classes for java pathfinder. *SIGSOFT Softw. Eng. Notes*, 39(1):1–5, Feb. 2014.

[20] A. Chaganty, A. V. Nori, and S. K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2013, Scottsdale, AZ, USA, April 29 - May 1, 2013*, pages 153–160, 2013. URL `http://jmlr.org/proceedings/papers/v31/chaganty13a.html`.

[21] S. Chaudhuri and A. Solar-Lezama. Smooth interpretation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 279–291, 2010. doi: 10.1145/1806596.1806629. URL `http://doi.acm.org/10.1145/1806596.1806629`.

[22] S. Chaudhuri, M. Clochard, and A. Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 207–220, 2014. doi: 10.1145/2535838.2535859. URL `http://doi.acm.org/10.1145/2535838.2535859`.

[23] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 3–14, 2013. doi: 10.1145/2462156.2462180. URL `http://doi.acm.org/10.1145/2462156.2462180`.

[24] S. Chiba. Load-Time Structural Reflection in Java. In *ECOOP*, pages 313–336, 2000.

[25] L. Clapp, S. Anand, and A. Aiken. Modelgen: Mining explicit information flow specifications from concrete executions. In *ISSTA*, pages 129–140. ACM, 2015.

[26] A. Demaille, R. Levillain, and B. Sigoure. TWEAST: A Simple and Effective Technique to Implement Concrete-syntax AST Rewriting Using Partial Parsing. In *SAC*, pages 1924–1929, 2009.

[27] B. Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1):1–26, 1979.

[28] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security Symposium*, 2011.

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

[30] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[31] W. Gaudin, A. Mallinson, O. Perks, J. Herdman, D. Beckingsale, J. Levesque, and S. Jarvis. Optimising hydrodynamics applications for the cray xc30 with the application tool suite. *The Cray User Group*, pages 4–8, 2014.

[32] P. Godefroid and A. Taly. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *PLDI*, pages 441–452, 2012.

[33] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, 2005.

[34] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network & Distributed Security Symposium*, NDSS '08, 2008.

[35] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330, 2011. doi: 10.1145/1926385.1926423. URL `http://doi.acm.org/10.1145/1926385.1926423`.

[36] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of Loop-free Programs. In *PLDI*, pages 62–73, 2011.

[37] Y. Hamadi, S. Jabbour, and L. Sais. Manysat: a parallel SAT solver. *JSAT*, 6(4):245–262, 2009. URL `http://jsat.ewi.tudelft.nl/content/volume6/JSAT6_12_Hamadi.pdf`.

[38] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 317–328, 2011. doi: 10.1145/1993498.1993536. URL `http://doi.acm.org/10.1145/1993498.1993536`.

[39] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster. Specifying and verifying the correctness of dynamic software updates. In *Proceedings of the 4th international conference on Verified Software: theories, tools, experiments*, VSTTE'12, pages 278–293, 2012.

[40] S. Heule, M. Sridharan, and S. Chandra. Mimic: Computing models for opaque code. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE 2015, pages 710–720. ACM, Sep 2015.

[41] W. E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.

[42] J. Jeon and J. S. Foster. Troyd: Integration Testing for Android. Technical Report CS-TR-5013, Department of Computer Science, University of Maryland, College Park, Aug 2012.

[43] J. Jeon, K. K. Micinski, and J. S. Foster. SymDroid: Symbolic Execution for Dalvik Bytecode. Technical Report CS-TR-5022, Department of Computer Science, University of Maryland, College Park, Jul 2012.

[44] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 3–14, Raleigh, NC, USA, October 2012.

[45] J. Jeon, X. Qiu, A. Solar-Lezama, and J. S. Foster. An Empirical Study of Adaptive Concretization for Parallel Program Synthesis. Oct 2015. Under submission.

[46] J. Jeon, X. Qiu, A. Solar-Lezama, and J. S. Foster. Adaptive Concretization for Parallel Program Synthesis. In *Computer Aided Verification (CAV)*, volume 9207 of *Lecture Notes in Computer Science*, pages 377–394, Jul 2015.

[47] J. Jeon, X. Qiu, A. Solar-Lezama, and J. S. Foster. JSKETCH: Sketching for Java. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, Bergamo, Italy, Sep 2015.

[48] J. Jeon, X. Qiu, J. Fetter-Degges, J. S. Foster, and A. Solar-Lezama. Synthesizing Framework Models for Symbolic Execution. In *38th International Conference on Software Engineering (ICSE '16)*, May 2016. To appear.

[49] S.-U. Jeon, J.-S. Lee, and D.-H. Bae. An automated refactoring approach to design pattern-based program transformations in Java programs. In *Asia-Pacific Software Engineering Conference*, pages 337–345, 2002.

[50] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806833. URL `http://doi.acm.org/10.1145/1806799.1806833`.

[51] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224, 2010.

[52] N. Kicillof, W. Grieskamp, N. Tillmann, and V. Braberman. Achieving both model and code coverage with automated gray-box testing. In *Proceedings of the 3rd International Workshop on Advances in Model-based Testing*, A-MOST '07, pages 1–11, 2007.

[53] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7): 385–394, July 1976.

[54] M. Loy, R. Eckstein, D. Wood, J. Elliott, and B. Cole. Java swing, 2nd edition: Code examples, 2003. `http://examples.oreilly.com/jswing2/code/`.

[55] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *Proceedings of the 18th international conference on Static analysis*, SAS '11, pages 95–111, 2011.

[56] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[57] Z. Manna and R. Waldinger. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, Jan. 1980.

[58] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, Mar. 1971.

[59] P. C. Mehlitz, O. Tkachuk, and M. Ujma. JPF-AWT: Model checking gui applications. In *ASE'11*, pages 584–587. IEEE, 2011.

[60] K. Micinski, J. Fetter-Degges, J. Jeon, J. S. Foster, and M. R. Clarkson. Checking Interaction-Based Declassification Policies for Android Using Symbolic Execution. In *European Symposium on Research in Computer Security (ESORICS)*, Vienna, Austria, Sep 2015.

[61] T. Mikkonen. Formalizing Design Patterns. In *ICSE*, pages 115–124, 1998.

[62] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6): 1–5, Nov. 2012.

[63] A. Mishne, S. Shoham, and E. Yahav. Typestate-based Semantic Code Search over Partial Programs. In *OOPSLA*, pages 997–1016, 2012.

[64] D. Octeau, S. Jha, and P. McDaniel. Retargeting Android Applications to Java Bytecode. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, FSE-20, 2012.

[65] Oracle Corporation. Using swing components: Examples, 2015. `https://docs.oracle.com/javase/tutorial/uiswing/examples/components/`.

[66] L. J. Osterweil and L. D. Fosdick. Program testing techniques using simulated execution. In *Symposium on Simulation of Computer Systems (ANSS)*, pages 171–177, 1976.

[67] T. Parr and K. Fisher. LL(*): The Foundation of the ANTLR Parser Generator. In *PLDI*, pages 425–436, 2011.

[68] A. Pnueli and R. Rosner. On the Synthesis of an Asynchronous Reactive Module. In *ICALP*, pages 652–671, 1989.

[69] X. Qiu and A. Solar-Lezama. Synthesizing Data-Structure Manipulations with Natural Proofs. Under submission.

[70] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 445–454, 2010.

[71] N. Rungta, P. C. Mehlitz, and W. Visser. Jpf tutorial, ase 2013, 2013. URL `http://babelfish.arc.nasa.gov/trac/jpf/raw-attachment/wiki/presentations/start/ASE13-tutorial.pdf`.

[72] H. Samimi, R. Hicks, A. Fogel, and T. Millstein. Declarative mocking. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 246–256, 2013.

[73] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 305–316, 2013. doi: 10.1145/2451116.2451150. URL `http://doi.acm.org/10.1145/2451116.2451150`.

[74] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 9, 2014. doi: 10.1145/2594291.2594302. URL `http://doi.acm.org/10.1145/2594291.2594302`.

[75] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, 2005.

[76] R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 88–105, 2014. doi: 10.1007/978-3-319-08867-9_6. URL `http://dx.doi.org/10.1007/978-3-319-08867-9_6`.

[77] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 634–651, 2012. doi: 10.1007/978-3-642-31424-7_44. URL `http://dx.doi.org/10.1007/978-3-642-31424-7_44`.

[78] R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *FSE*, pages 289–299, 2011.

[79] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated Feedback Generation for Introductory Programming Assignments. In *PLDI*, pages 15–26, 2013.

[80] A. Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013.

[81] A. Solar-Lezama. *The Sketch Programmers Manual*, 2015. Version 1.6.7.

[82] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294, 2005.

[83] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 404–415, 2006.

[84] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

[85] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *PLDI*, pages 136–148, 2008.

[86] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-Based Inductive Synthesis for Program Inversion. In *PLDI*, pages 492–503, June 2011.

[87] E. Torlak and R. Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 54, 2014. doi: 10.1145/2594291.2594340. URL `http://doi.acm.org/10.1145/2594291.2594340`.

[88] J. Turpie, E. Reisner, J. S. Foster, and M. Hicks. MultiOtter: Multiprocess Symbolic Execution. Technical report, CS-TR-4982, Department of Computer Science, University of Maryland, College Park, Aug 2011.

[89] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur. TRANSIT: Specifying Protocols with Concolic Snippets. In *PLDI*, pages 287–296, 2013.

[90] H. van der Merwe, O. Tkachuk, B. van der Merwe, and W. Visser. Generation of library models for verification of android applications. *SIGSOFT Softw. Eng. Notes*, 40(1):1–5, Feb. 2015.

[91] M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. *SIGPLAN Not.*, 43(6):125–135, 2008. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1379022.1375598.

[92] F. Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[93] C. M. Wintersteiger, Y. Hamadi, and L. de Moura. A concurrent portfolio approach to SMT solving. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 715–720, 2009. doi: 10.1007/978-3-642-02658-4_60. URL `http://dx.doi.org/10.1007/978-3-642-02658-4_60`.

[94] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *ICSE*, volume 1, pages 89–99. ACM, May 2015.

[95] H. Zhang, M. P. Bonacina, and J. Hsiang. Psato: A distributed propositional prover and its application to quasigroup problems. *J. Symb. Comput.*, 21(4-6):543–560, June 1996. ISSN 0747-7171. doi: 10.1006/jsco.1996.0030. URL `http://dx.doi.org/10.1006/jsco.1996.0030`.