Program Testing and Synthesis by Program Semantics

Lin Cheng
Western Michigan University, chenalin1109@gmail.com

Follow this and additional works at: https://scholarworks.wmich.edu/dissertations

Part of the Computer Sciences Commons

Recommended Citation
https://scholarworks.wmich.edu/dissertations/3583

This Dissertation-Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Dissertations by an authorized administrator of ScholarWorks at WMU. For more information, please contact wmu-scholarworks@wmich.edu.
Program Testing and Synthesis by Program Semantics

by

Lin Cheng

A dissertation submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
Computer Science
Western Michigan University
June 2020

Doctoral Committee:

Zijiang Yang, Ph.D., Chair
Steve Carr, Ph.D.
Chao Wang, Ph.D.
GUIs are event-driven applications where the flow of the program is determined by user actions such as mouse clicks and key presses. GUI testing is a challenging task not only because of the combinatorial explosion in the number of event sequences, but also because of the difficulty to cover the large number of data values. To tackle this problem, we propose two algorithms: GUICat and GUICat2.

GUICat is a white-box GUI testing tool that augments traditional sequence generation techniques with concolic execution. It also has a cloud-based parallel algorithm for mitigating both event sequence explosion and data value explosion, by distributing the concolic execution tasks over public clouds such as Amazon EC2. Our experiments show that GUICat significantly outperforms state-of-the-art GUI testing tools such as GUITAR.

GUICat2 is a new and efficient GUI testing framework to generate progressively longer event sequences while avoiding redundant sequences. GUICat2 identifies the redundancy among these sequences by a set of simple checks, whose reduction power matches and sometimes exceeds that of the classic techniques based on the theory of partial order reduction. Our experimental results show the new technique, while being sound and systematic, can achieve more than 10X reduction in the number of test sequences compared to the state-of-the-art GUI testing
tools.

Program synthesis enables people to program computers without training in coding by automatically generating programs given specifications. We propose two program synthesis algorithms: SQLSol and GRCNN.

SQLSol helps users to synthesize SQL queries from input-output (IO) examples. It is the first algorithm to encode the SQL synthesis problem into the constraint-solving problem. We propose an axiom that encodes the semantics of a SQL query into logic constraints, and decompose the SQL synthesis problem into two parts: problem-encoding and constraint-solving. Our algorithm supports multiple IO examples, and therefore users can add more examples to refine the solution until a correct one is found. Experimental results show that it efficiently solved 68% of the benchmarks in 3 seconds on average. For those benchmarks that SQLSol cannot solve, it terminated in 4 seconds on average.

GRCNN is a framework that synthesizes programs from images of flow charts that serve as accurate and intuitive specifications. In order doing so, we propose a deep neural network called GRCNN that recognizes graph structure from its image. GRCNN is trained end-to-end, which can predict edge and node information of the flow chart simultaneously. Experiments show that the accuracy to synthesize a program is 66.4%, and the accuracy to recognize edges and nodes are 94.1% and 67.9%, respectively. on average, it takes about 60 milliseconds to synthesize a program.
ACKNOWLEDGMENTS

I would like to thank Dr. Zijiang Yang for being my advisor and committee chair. I thank Dr. Yang for opening the gate to the academic world for me, providing funding support, technical discussion, and patience in my research.

Besides Dr. Zijiang Yang, I sincerely thank the other members of my dissertation committee members, Dr. Steve Carr and Dr. Chao Wang, for their support and advice on a variety of problems I encountered during my research.

I express my deepest gratitude to all my friends that accompanied and supported me in the period of my Ph.D. study.

Lastly, I thank God for all his blessings, and I hope God bless me in the future.
## Contents

### 1 Introduction
1.1 GUICat: GUI testing as a Service ........................................ 1
1.2 Systematic Reduction of GUI Test Sequences .......................... 3
1.3 SqlSol: an Accurate SQL Query Synthesizer ............................ 5
1.4 Synthesizing Programs from Flowchart Image Using Deep Convolutional Neural Networks ........................................ 9

### 2 Literature Review

### 3 GUICat: GUI testing as a Service
3.1 Motivating Examples ............................................................. 16
3.2 Architecture ............................................................................ 19
3.3 Experimental Evaluation ......................................................... 22
  3.3.1 The Ticket Seller ............................................................. 23
  3.3.2 The Workout Generator ................................................... 26
  3.3.3 Effect of Cloud Computing ............................................... 28
3.4 Conclusion .............................................................................. 29

### 4 Systematic Reduction of GUI Test Sequences
4.1 Motivation ............................................................................... 31
  4.1.1 Naive Solution ............................................................... 32
4.1.2 Our New Method .................................................. 34
4.1.3 Comparison to Existing Techniques .............................. 36
4.1.4 Stateless Implementation ...................................... 38
4.2 Preliminaries .......................................................... 39
4.2.1 GUI Application .................................................. 39
4.2.2 Event Flow Graph ................................................ 39
4.2.3 Dependency Relation ............................................ 40
4.3 Systematic Test Generation ........................................ 41
4.3.1 The Baseline Algorithm ........................................ 41
4.3.2 Partial Order Reduction ........................................ 42
4.4 The New Reduction Technique ..................................... 44
4.4.1 The New Algorithm ............................................. 44
4.4.2 Detecting Redundant States .................................... 46
4.4.3 Eliminating Redundant Sequences .............................. 49
4.4.4 Computing Causal Dependencies .............................. 51
4.5 Experiments ........................................................... 52
4.5.1 Comparison of Different Methods .............................. 53
4.5.2 Comparison of Individual Reduction Techniques .......... 55
4.5.3 Comparison of Test Execution Results ...................... 56
4.6 Conclusion ............................................................. 57

5 SqlSol: an Accurate SQL Query Synthesizer ...................... 58
5.1 Motivating Example .................................................. 58
5.1.1 Join Conditions .................................................. 60
5.1.2 Parameterized SQL query ..................................... 60
5.1.3 Auxiliary Columns ............................................... 61
5.1.4 Logic Constraints for SQL Synthesis Problem .............. 61
5.1.5 Solve .............................................................. 63
List of Tables

3.1 Results on the ticket seller (length = 2, node = 1). .................. 24
3.2 Results on ticket seller: the test cases generated in addition to GUI-TAR and the unique paths covered. ................................. 25
3.3 Results on workout generator (length=2/3, node=1). ................. 27
3.4 Results on workout generator: the test cases generated in addition to GUI-TAR and the unique paths covered. ................................. 28
4.1 Statistics of the GUI benchmark applications. .......................... 53
4.2 Comparison of event sequence reduction. ............................... 54
4.3 Effectiveness of different reduction techniques. ....................... 56
4.4 Comparison of the test execution results. ............................... 56
5.1 Statistics of numbers of solved (unsoloved) benchmarks, percentage, time usage, average time usage on SQLSynthesizer and Scythe benchmark set. Columns marked by sqlsyn, scythe, sqlsol1, sqlsol2, and sqlsol3 are the results of the algorithms SQLSynthesizer, Scythe, SQLSol with the number of predicates in the WHERE clause, wp, set to 1, 2, and 3. Timeout for Scythe is 120 seconds. .................. 73
6.1 Basic block architecture. .................................................. 78
6.2 Backbone network architecture. ......................................... 78
6.3 Edge network architecture. .............................................. 80
6.4 Node recognition network architecture. ........................................... 82

6.5 Accuracy in percentage of GRCNN. The columns are Edge, Sequence, Nodes and Graph accuracy, respectively. ............................................ 85

6.6 Time cost in milliseconds of GRCNN. The columns are the time cost of the backbone, edge, node detection, node recognition network and GRCNN, respectively. ......................................................... 86

6.7 Result of real-world program synthesis. The columns are program name, Graph accuracy, Edge accuracy, Nodes accuracy, number of nodes, correctly predicted nodes. ............................................. 87
List of Figures

1.1 Systematic generation and reduction of GUI tests. .......................... 4
1.2 Solution generated by Scythe. It has 3 levels of nested queries and 4 sub- queries. ................................................................. 7
1.3 The Architecture of SQLSol. .................................................. 8
1.4 Overview of GRCNN. The input is the flow chart of abs function. The middle is the adjacent matrix and text of each node generated by GRCNN. The output is the synthesized source code. ....................... 10

3.1 A GUI example and GUICat’s configuration file. ......................... 17
3.2 Code snippet for computing the ticket price. ............................. 17
3.3 The architecture of our GUICat tool. ........................................ 20
3.4 Architecture of GUICat’s distributed algorithm. ........................ 22
3.5 Communication from load balancer to workers. .......................... 22
3.6 Ticket Seller. ................................................................. 23
3.7 The buggy code snippet in ticket seller. .................................... 26
3.8 The ticket seller bug detected by GUICat. ................................ 26
3.9 Workout Generator. .......................................................... 26
3.10 Time reduction achieved by cloud computing. ........................... 29

4.1 The class ModifyImageWindow defines event handlers onCheckBox, onSlider, onSave, and onOK. ............................................. 32
4.2 Event flow graph, where CB, SL, SA, and OK denote onCheckBox, onSlider, onSave, and onOk, respectively. ................................................................. 32
4.3 The complete tree of 40 event sequences of length ≤ 3 for the GUI application in Fig. 4.1 and Fig. 4.2. ................................................................. 33
4.4 State transition graph, where each state is a valuation of variable convert and predicate (angle>0). ................................................................. 34
4.5 The reduced tree of event sequences of length ≤ 3: From 40 sequences to 13 sequences and then to 6 sequences. ........................................ 34
4.6 The number of sequences generated by Gazoo and our tool as the maximum sequence length increases. ......................................................... 37
4.7 Conditions for $s_n$ to be a redundant state. ........................................... 49

5.1 Hand-written solution for the motivating example. .............................. 59
5.2 Input-Output Tables for Motivation Example. ....................................... 59
5.3 Input-Output Tables After Join in Motivation Example. ....................... 60
5.4 Parameterized SQL Query for Motivation Example. ............................ 61
5.5 Syntax of the supported SQL subset in SQLSol: atom is a table name, or column name, or a cell value. ......................................................... 64
5.6 Time usage with increasing number of rows in input table. ................... 74
5.7 Single input. .................................................................................. 75
5.8 Multiple inputs. ............................................................................ 75
5.9 Statistics comparison. ...................................................................... 75

6.1 Architecture of GRCNN. .................................................................... 78
6.2 Samples of GRCNNprediction. The predicted bounding boxes are drawn in red lines. The predicted text is drawn above the bounding box in blue characters. On the right is the predicted adjacent matrix and synthesized source code. ..................................................... 87
Chapter 1

Introduction

In this era of information technology, software is ubiquitous in our life. Software testing is the task to investigate the quality of the software under test. It is important for stakeholders to provide high quality software and critical to too-expensive-to-fail projects such as spaceships. Program synthesis is the task to automatically generate software programs that satisfy given specifications. It enables people without programming skills to program computers.

In this dissertation, I present the result of my research on software testing and program synthesis. In particular, I present my work that automatically generates test cases for graphical user interface (GUI) software, and my work that synthesizes SQL queries given input-output tables as specifications, and my work that generates programs given an image of a flowchart.

1.1 GUICat: GUI testing as a Service

Graphical User Interfaces provide a convenient way for the user to interact with the computer. They are event-driven applications where the flow of the program is determined by user actions such as mouse clicks and key presses. In contrast to console applications whose only point of interaction is at the beginning, GUIs have a potentially large num-
ber of interaction points, each of which may be associated with a different state. These features often make traditional software testing techniques ineffective. Specifically, GUI testing has two significant challenges. First, covering all possible event sequences of a GUI application is difficult due to the combinatorial explosion, i.e., the number of possible ways of clicking $k$ buttons can be as large as $k!$. Second, GUI behaviors depend not only on the event sequence but also on the data values of widgets such as text-boxes, edit-boxes, and combo-boxes, thus leading to an extremely large input space. For example, covering all possible values of a $k$-character input string requires $26^k$ test cases.

Although existing GUI testing tools [56, 61, 68, 81] have addressed the challenge of generating high-quality event sequences, they have not addressed the challenge of simultaneously generating high-quality data values. As such, data-dependent GUI behaviors are often inadequately tested.

We propose GUICat, the first cloud-based GUI testing framework that simultaneously generates both high-quality event sequences and high-quality data values, by augmenting state-of-the-art event sequence generation techniques with concolic execution. The result is a white-box GUI testing tool that uniformly explores the event flow as well as the data flow. We also propose a parallel concolic execution algorithm for mitigating the data value explosion, by distributing the computation tasks over workers on private clusters as well as public clouds such as Amazon EC2 [5]. It provides an illusion of running GUICat on a powerful super computer and thus allows it to handle significantly larger applications than previously possible.

We have implemented GUICat based on a number of open-source tools, including GUITAR [61] for generating the initial event sequences, ASM $^1$ for Java bytecode instrumentation, Catg [74] for concolic execution, and JaCoCo $^2$ for computing code coverage. Unlike prior techniques, GUICat is fully automated in modeling GUI widgets. That is, it does not require developers to manually model these widgets. This is important because

---

$^1$http://asm.ow2.org
$^2$http://eclemma.org/jacoco/
manual modeling is not only labor intensive and error prone but also hard to sustain in the long run due to frequent widget updates.

We have evaluated GUICat by deploying it on Amazon EC2 to test a set of GUI testing benchmarks. GUICat achieves scalability through the distribution of symbolic execution tasks.

1.2 Systematic Reduction of GUI Test Sequences

Graphic user interface is an integral part of many software applications that monitor user actions such as keyboard and mouse events and respond by invoking listener functions. To test a GUI application, one must create tests to cover its input space, where each test is a finite sequence of events. Due to combinatorial blowup, the number of sequences can be astronomically large, e.g., up to $10^{10}$ for all length-10 sequences of 10 events, if these events are enabled all the time. Thus, the main problem is to generate a small subset of these event sequences while achieving the same testing effect as the complete set. Since manually crafting these sequences is labor-intensive, techniques have been developed to generate them automatically [7, 9, 62, 86, 88, 89]. Unfortunately, these existing techniques are neither systematic nor efficient, i.e., they often miss important event sequences and produce many redundant sequences.

To avoid these problems, we propose a new test generation tool to construct progressively longer event sequences. Our tool has the advantage that, during the sequence generation process, it eliminates an event sequence only if the sequence is guaranteed to be redundant, i.e., subsumed by some other sequences. This is accomplished by a new type of semantic reduction technique that differs from classic partial order reduction (POR) methods [28, 34, 35, 64, 77]. Our tool is also efficient in that it relies on a set of easily-checkable conditions to identify redundant sequences. These conditions are expressed in terms of the sequence of events as opposed to the concrete program states. Thus, they can
Fig. 1.1 shows the overall flow of our method. The input consists of Java byte-code of the GUI application and a bound on the sequence length. The output is a set of event sequences. Internally, our method goes through several steps. First, it leverages GUITAR [29] to reverse-engineer an event flow graph (EFG) of the application. The EFG shows the set of events enabled at any step of the execution, as well as the enabled events afterward. Then, our method leverages Soot [76] to perform static analysis of the Java byte-code to compute dependencies over the events. Next, it invokes our core algorithm, which takes the EFG, the dependencies and a bound as input and constructs the test sequences. Finally, the sequences are executed on the actual GUI application using Cobertura [22], which measures the coverage.

Compared to state-of-the-art GUI testing tools [7, 9, 62], our method has two advan-
tages. First, it is systematic, meaning that useful test sequences are not excluded in any ad hoc fashion: within the maximum sequence length, our method eliminates a sequence only if it is provably redundant; when in doubt, it retains the sequence. Second, our method is efficient in that it generates significantly fewer test sequences than prior techniques. For instance, when applied to the example in Fig. 4.1, prior techniques based on partial order reduction can only remove 11 redundant sequences, whereas our method removes 34 redundant sequences. Although Arlt et al. [9] proposed a reduction technique that goes beyond POR, their tool still generates significantly more sequences for Fig. 4.1: when the maximum length is set to 5, 7 and 9, it generates 33, 129 and 513 sequences, respectively, whereas our method generates only 6 sequences.

We have evaluated our method on 17 Java Swing applications consisting of 105,937 lines of code in total. The experimental results show our new reduction technique is more effective: it outperforms partial order reduction consistently and significantly. We also experimentally compared with state-of-the-art GUI testing tools including Gazoo [7, 9] and GUITAR [62]. Overall, our tool achieves more than 10X reduction in the number of test sequences compared to Gazoo and GUITAR, and significantly reduces the corresponding test execution time.

1.3 SqlSol: an Accurate SQL Query Synthesizer

Relational database is one of the most important data management infrastructures in the modern era of data technology. Structured query language (SQL) is the language that is used to manipulate relational databases. According to TIOBE index [4], SQL is one of the top 10 ranked programming languages, and the best-ranked database query language in the year 2018. However, writing SQL queries is difficult because of the high expressiveness of the language structs. Searching on Stack Overflow with the keyword SQL, we got over 30000 pages of results, each of which contains 15 questions. That is, more than 450K
questions about writing SQL queries were asked.

Programming By Example (PBE) has attracted research interests as a technique to help end-user programming computers by demonstrating concrete examples [27, 37, 40, 50]. Observing that users usually provide input-output (IO) tables as examples when raising questions, researchers have proposed different algorithms to synthesize SQL queries by examples [79, 91].

The state-of-the-art SQL synthesizers, SQLSynthesizer [91] and Scythe [79], use hand-crafted search algorithms to automatically generate SQL queries from input-output examples. SQLSynthesizer uses a parameterized SQL template, which is showed by a survey that it supports the mostly widely used SQL features, and uses a decision tree to decide the parameters, and uses an online database to validate the result. Scythe enlarges the supported SQL subset by allowing nested queries, and uses abstract SQL query to prune the search space, and manually implements SQL semantics to validate the result. Both SQLSynthesizer and Scythe support one IO example and use heuristics to rank solutions because there may be many SQL queries that satisfy the IO example.

Instead of manually creating search algorithms, we propose an axiom which encodes the semantics of a SQL query and a new algorithm that encodes the SQL synthesis problem into logic constraints and uses the off-the-shelf satisfiability modulo theory (SMT) solvers to solve for the solution. Nowadays, SMT solvers such as Z3 [24] and CVC4 [14] have become the corner stones of modern computer science. The rich features of modern SMT solvers allow us to express complicated datatypes, like table. For example, in this dissertation, we encode table as a datatype of Array(String Int Cell), where Cell is a datatype which can hold int, float, string or null values of a table cell.

Our algorithm enjoys the following benefits brought by the logic constraints based approach. First, it inherits the theoretical properties from the formal methods. That is, because the logic constraints are sound and complete, if the algorithm of SMT solver is sound and complete, the whole algorithm is sound and complete. Second, our algo-
Algorithm enjoys performance improvement, because the implementations of modern SMT solvers focus on performance and are battle-hardened [3]. Third, it allows our approach to support multiple IO examples, because our algorithm decomposes the SQL synthesis problem into two parts: logic encoding and logic solving. Actually, we only need to conjuncture the logic constraints for all IO examples. Therefore, instead of heuristically ranking solutions to match an IO example, our algorithm is able to synthesize the correct SQL query, i.e., captures the user intent. For example, in our motivating example in section ??, our algorithm generates the SQL query that matches the specification in natural words exactly. However, the state-of-the-art tool, Scythe, generates the solution, Figure 1.2, which is more complicate and semantically different because its predicate is score > 59 instead of average(score) > 59.

```
Select t2.student, t2.avg_score
From
(Select t4.student, t4.level, t4.avg_score, t5.student As student1, t5.level As level1, t5.course, t5.score
From ((Select t3.student, t3.level, Avg(t3.score) As avg_score
From (Select *
From input_table_0
Where input_table_0.score > 59.0) As t3
Group By t3.student, t3.level) As t4
Join (Select *
From input_table_0
Where input_table_0.score = 59.0) As t5)) As t2
Where t2.level = t2.level1;
```

Figure 1.2: Solution generated by Scythe. It has 3 levels of nested queries and 4 sub-queries.

Figure 1.3 is the overall flow of our tool, SQLSol. It starts with one IO example provided by the user, and synthesizes the JOIN conditions. Then, it determines the number and types of the unknowns in a parameterized SQL query, and encodes IO constrains, domain constraints, and semantics constraints. Then, it sends all constraints to an SMT solver to check for satisfiability. If the solver returns unsatisfiable, it declares failure. If the solver returns satisfiable, it fetches the model generated by the solver and compiles a SQL
query and asks the user to decide whether to accept or not. If yes, the algorithm returns; otherwise, the algorithm asks the user to add new IO examples to refine the solution until a solution is accepted or failure declared. Note that our algorithm joins input tables in the same way and supports the same SQL subset as SQLSynthesizer.

We evaluated SQLSol on two benchmark sets used by previous work, SQLSynthesizer and Scythe [79, 91]. Under a reasonable setting, on SQLSynthesizer benchmark set, SQLSol solved more problems than both SQLSynthesizer and Scythe. On Scythe benchmark set, although SQLSol solved fewer problems than Scythe because the supported SQL subset is smaller, SQLSol was faster for both solved and unsolved problems. Specially, for unsolved problems, SQLSol terminates in seconds, while Scythe only terminates after time out of 120 seconds. We evaluated the scalability of SQLSol and Scythe on another 25 hand-written benchmarks, and the result showed that SQLSol is 2X to 10X faster than Scythe. Our evaluation also showed that supporting multiple IO examples not only allows user to find the correct SQL query, but also can speedup the algorithm.
1.4 Synthesizing Programs from Flowchart Image Using Deep Convolutional Neural Networks

Program synthesis enables people to program computers without training in coding. It has been used in many domains such as data wrangling, graphs, and code repair [37]. A good example is FlashFill [38], which allows spreadsheet users to provide a few examples and generates a program that conforms to the examples.

To synthesize a program, specification must be provided. Specification in formal language can accurately represent the user intent and is used in deductive program synthesis [53]. However, very few users have the knowledge of formal language, so it cannot benefit most end users. Under-specification is used in programming by example [27, 38], and programming by demonstration [49]. Under-specification does not require language knowledge and is accessible by most end users. However, because there may be more than one program that satisfies the specification, how to choose the correct program that captures the user intent is still an open problem.

In this dissertation, we propose a new technique called GRCNN (Graph Recognition Convolutional Neural Network) that takes as input images of flow charts as accurate specification, and uses deep convolutional neural network (CNN) to analyze a given image, and compiles the obtained information into program code. GRCNN is an end-to-end network that shares the computation of a rich convolutional feature vector and predicts edge and node information simultaneously. A flow chart is a diagram that represents the workflow of a program. It is used widely in textbooks to teach coding and illustrate programs. Moreover, flow charts are intuitive, which allows users to focus on programming logic instead of language details. Thus they are also frequently used in the program designing stage. Figure 1.4 shows a flow chart representing the workflow of a function that computes the absolute value of an input. Because flow charts can accurately describe programs, synthesizing program from flow chart may precisely capture users’ intent.
Due to recent progress in deep learning, CNN is powerful enough to detect and recognize complicated information from image. Therefore, we can obtain a graph data structure from a flow chart with the help of deep CNN. Because neural networks are differentiable functions, our method does not suffer from the combinatorial explosion problem that plagues traditional program synthesis methods.

Figure 1.4 gives an overview of our approach. First, an image of flow chart is resized to a fixed size and fed to GRCNN. Then, GRCNN generates the graph information including an adjacent matrix for the edges and a list of strings for the nodes. Finally, we compile the graph information to source code.

```
if (x > 0)
    a = x
else
    a = -x
return a
```

Figure 1.4: Overview of GRCNN. The input is the flow chart of `abs` function. The middle is the adjacent matrix and text of each node generated by GRCNN. The output is the synthesized source code.

Our evaluation showed that it is feasible to share the convolutional vector between edge and node networks. Experiments on our synthetic test dataset show that the accuracy rate to synthesize a program is 66.4%, and the accuracy rates to predict edges and nodes are 94.1% and 67.9%, respectively. Experiments on another dataset, which is manually converted from a textbook, show that the accuracy to synthesize a program is 63.6%, and the accuracy to predict edges and nodes are 72.7% and 81.8%, respectively. The average time to synthesize a program is about 60 milliseconds.
Chapter 2

Literature Review

GUITAR [61] is the first framework capable of performing the whole process of test generation, execution, and result assessment for GUIs. Since its first publication there have been multiple improvements (e.g. [83]). This framework generates tests as event sequences up to a given bound. For emulating user input a specification based approach is adopted, i.e., using values from a prefilled database. Since GUITAR does not provide a mechanism for reasoning about input values for data widgets, GUICat offers complementary and more comprehensive testing.

The work closest to ours is Barad [30] that also exploits symbolic execution to compute input values for data widgets. It manually creates symbolic mirror of java GUI library, so its released source code contains a large symbolic java GUI library. Manual modeling is error prone and hard to sustain. In fact, we downloaded the tool but failed to make it work on our benchmarks. We employ a different test generation algorithm and symbolic analysis method for obtaining inputs. Another line of work is to apply model checking techniques. For example, jfp-awt [54] is an extension of the Java PathFinder for GUI applications.

Performance enhancement of GUI testing has traditionally focused on minimizing event sequences [56, 81]. Barad generates test cases as chains of event listener method
invocations and maps these chains to event sequences that force the execution of these invocations. Such approach prunes the event input space because it does not need to consider events where there are no event listeners. More recent work starts to apply event dependency analysis [8], program slicing [10] or partial order reduction [52] to improve the performance. Our performance improvement is obtained by exploiting massive hardware resource available on cloud. Therefore our approach is orthogonal to the existing algorithmic approaches.

GUI is an indispensable component of many software applications. Thus, there has been abundant research on improving the efficiency of GUI testing in various domains, including desktop [21, 88–90], mobile [6, 44, 46, 59], and web applications [11, 73, 85]. Although techniques proposed in this work were implemented in our tool, which is designed for testing desktop applications, the underlying principle may also be applied to other types of GUI applications and event-driven programs in general.

GUI testing is a complex process that requires efficient algorithms and implementation techniques in many aspects such as static program analysis, model extraction [58, 87], automated replay [36], and test case maintenance [85]. In this work, we focus on event sequence generation while relying on a number of existing tools such as GUITAR [62], Soot [76], and Cobertura [22] to offer an end-to-end solution. However, there is also room for improvement in these other aspects.

Beside the work mentioned so far, there are other GUI testing techniques [7, 9, 25, 31, 57, 69, 82]. For example, earlier works [69, 82] create models of the software code based on finite state machines, but as pointed out in [12], some of these techniques would not work well when the model does not accurately reflect the actual code. To avoid this problem, Yuan and Memon [89] propose to leverage feedback from the execution of a seed test suite to generate new test cases. Such approach depends on the quality of the seed as well as randomness during test execution.

Our method is closely related to state-space reduction techniques in model checking,
but with some important differences. In model checking, existing methods are either model-based [34, 35, 64, 77], e.g., relying on a state-transition system where values of state variables are available, or stateless [28] where the model checker does not maintain states but instead dynamically executes the software. In contrast, our method a hybrid approach that augments an abstract model (the EFG) with dependencies derived from the software statically. The EFG used in our work is more abstract than state-transition systems in model checking because it does not contain values of the program variables.

Test sequence reduction has also been studied in event-driven programs [7, 9, 16, 57] to reduce the test execution cost. In this context, partial order reduction (POR) [28, 34, 52] serves as a foundational technique for removing redundancy. However, as shown in Section ?? as well as the experiments, although POR is effective in identifying redundancy among sequences of the same length, it misses other redundant sequences. In comparison, our method is more effective since it also exploits redundancy among sequences of different lengths.

Beyond test sequence generation, an important problem is diversifying the input data. Several recent works have focused on this problem, e.g., by using model checking [55, 60, 73] and symbolic execution [6, 21, 46, 59]. However, scalability remains a problem and thus there is still room for improvement. We will consider it for future work.

Program Synthesis is the task that automatically generates programs that satisfy some high-level specifications. Our synthesis algorithm belongs to SMT-solver aided program synthesis, is particularly inspired by [39]. In [39], Gulwani etc. parameterized sequential programs by making the line numbers of a program symbolic variables, and encoded the syntax and semantics of a sequential program into logic constraints. Their algorithm lets the user refine the solution by providing more examples. Our algorithm differs from [39] in that it is the first to encode the syntax and semantics of SQL queries for inductive SQL query synthesis, to our knowledge.

Paper [79, 91] use search algorithms to synthesize SQL queries from IO examples.
Our work differs from those in that we delegate the searching algorithm to modern SMT solvers, so we can focus on a sound and complete logic encoding of the SQL semantics and at the same time enjoys the performance of state-of-the-art SMT solvers. Paper [78] proposes axiom system for SQL semantics to synthesize input tables from SQL queries. Our work solves the reverse problem that synthesizes SQL queries from input-output examples.

Traditional methodology of program synthesis is to construct a program space and design search algorithms to find a solution that satisfies the specification. The program space usually grows exponentially with the size of the target program. Different methods are proposed to speedup the search. For example, Flashfill, [38], synthesizes string-transforming programs given input-output examples. It uses dynamic programming to speedup the search. Morpheus, [80], enumerates nested queries and prunes by grouping programs with same input-output pairs. Sketch, [71], and Sqlsol, [20], encode the synthesis problem into logic constraints and delegate the searching algorithm to modern SMT solvers. This approach can boost the performance because modern SMT solvers are implemented for efficiency. Though different methods are proposed to speedup the synthesis process, the underlying complexity is unchanged and scalability is still an issue when it comes to large programs. Another issue is how to capture users’ intent. Typically, synthesis algorithms terminate when the first solution is found or the best-ranking solution is found. They may ask a user to provide more examples. However, there is no guarantee that the solution precisely captures the users’ intent.

Recently researchers propose to use machine learning to speed up the search for synthesized program. Morpheus [27] uses statistics to rank $R$ program sketches, and uses the rank to guide the search. DeepCoder [13] augments beam search with deep learning recommendation, and the speedup is significant. However, the underlying complexity is still unchanged.

Researchers proposed natural language based program synthesis techniques. SQLizer,
Locascio et al, [51], synthesizes regular expressions from natural language. —name differs from these approaches in that our input is different. Flow charts can accurately specify the users’ intent, while natural language is ambiguous.

Faster RCNN, [67], and LPRNet, [92], are closely related to our work. Two subnets of GRCNN are built following their ideas. Faster RCNN is a deep neural network for object detection. It slides a small window on a convolutional feature and generates box proposals relative to anchors at each position. The box proposal is used to crop the image for a classifier to detect the class of the object in it. Because of the shape and positioning of nodes in a flow chart are different from those in general object-detection jobs, —name chooses different anchors and methods to select the proposals. LPRNet is a license plate recognizing deep convolutional neural network. It reads an image of a license plate and generates a sequence that preserves the spatial order of the characters. A feature of LPR-Net is that it has only one deep convolutional network, while other work has both a CNN for feature extraction and a recurrent neural network for prediction. This feature helps to limit the number of subnets in —name, since it already has four subnets. Instead of taking a license plate as input, —name takes as input a crop of image whose boundary is predicted by another network.
Chapter 3

GUICat: GUI testing as a Service

This chapter describes GUICat, a tool that generates test cases for GUI testing to cover as many program paths as possible, in details. The chapter is organized as follows: Section 3.1 illustrates the main idea behind GUICat using motivating examples. Section 3.2 presents the algorithm. Section 3.3 describes the experimental results.

3.1 Motivating Examples

Figure 3.1 shows an example GUI for computing ticket fare based on user inputs, including Name, Age Level, Distance, and Coupons. Once the Buy button is clicked, the application computes and then displays ticket price, using a coefficient associated with the chosen age level. To allow GUICat to automatically generate test cases, the user must provide a configuration file that specifies the name and type of the symbolic variables as shown in Figure 3.1 (right). Each entry (line) of the configuration file consists of the widget name, widget type, method for obtaining user input (e.g., `getText`), type of user input, and the default value (e.g., `superman`). Here, `0:1` means the default value is of `enum` type with two values `0` and `1`.

Figure 3.2 shows the code for computing the ticket price. It has a bug that can lead to negative ticket prices. For example, if a user has three coupons, then purchasing a child
ticket for a distance of 60 miles would result in the price being -40 dollars. However, since a negative price requires a specific combination of widget values, such bug is difficult to detect using state-of-the-art GUI testing tools such as GUITAR [61]. This is because GUITAR focuses primarily on generating event sequences as opposed to generating a diverse set of widget values. Our new tool GUICat, in contrast, can quickly generate a combination of event sequences and widget values to expose this assertion failure.

```java
OnClickComputePrice() {
    int coupon = 0;
    String age = (String) ageComboBox.getSelectedItem();
    String sdistance = distanceTextField.getText();
    int distance = Integer.parseInt(sdistance);
    if (d100CheckBox.isSelected())
        coupon += 100;
    if (d200CheckBox.isSelected())
        coupon += 200;
    if (d400CheckBox.isSelected())
        coupon += 400;
    if (age.equals(Child)) {
        coeffienct = 1;
    } else {
        coeffienct = 2;
    }
    if (distance < 60) {
        price = 500;
    } else if (distance < 80) {
        price = 11 * distance * coeffienct - coupon;
    } else if (distance < 100) {
        price = 10 * distance * coeffienct - coupon;
    } else if (distance < 120) {
        price = 9 * distance * coeffienct - coupon;
    } else {
        price = 8 * distance * coeffienct - coupon;
    }
    assert (price > 0);
    infoField.setText(price);
}
```

Figure 3.2: Code snippet for computing the ticket price.

GUICat generates the test cases as follows. First, it uses GUITAR to generate the initial set of event sequences up to a bounded length. Then, for each event sequence, it creates
an instrumented GUI where some variables are marked as symbolic based on the configuration file. Next, it conducts symbolic execution of the instrumented GUI over the cloud. Finally, it uses JaCoCo to generate the coverage report. Now, we explain each step in more detail.

**Step 1.** We use GUITAR to generate event sequences of a bounded length. Assume the bound is 2, and events $e_1, \ldots, e_7$ denote `nameTextField`, `distanceTextField`, `AgeComboBox`, `100DollarCheckbox`, `200DollarCheckbox`, `400DollarCheckbox`, and `BuyButton`. After running GUITAR, we have the following seven length-2 event sequences: $(e_1, e_7)$, $(e_2, e_7)$, $(e_3, e_7)$, $(e_4, e_7)$, $(e_5, e_7)$, $(e_6, e_7)$, and $(e_7, e_7)$. In this example all the bounded sequences end in $e_7$ because otherwise no action can be taken at the end of the user interaction. Consider $(e_2, e_7)$ as an example. It means the user specifies a distance ($e_2$) before clicking the BuyButton ($e_7$). Although logically meaningless, this particular event sequence is feasible.

**Step 2.** For each event sequence produced by GUITAR, we generate more sequences by enumerate the values of widgets with `enum` types. For example, in $(e_3, e_7)$, we know `JComboBox` is associated with $e_3$ and it has an `enum` type. Therefore, we enumerate all possible values of `JComboBox` to produce the new sequences $(e_3^0, e_7)$ and $(e_3^1, e_7)$, where $e_3^i$ means the value $i$ is chosen for event $e_3$. We also enumerate the values of other widgets with `enum` types, including `AgeComboBox`, `100DollarCheckbox`, `200DollarCheckbox`, `400DollarCheckbox`. Since these selectable widgets have a limited number of data values, enumerating them is often more efficient than generating the values using concolic execution.

**Step 3.** For each event sequence generated in Step 2, we instrument the GUI application by marking certain variables as symbolic. Consider the `guicat-conf` file in Figure 3.1 (right), where the last four widgets are of `enum` types but the first two are not. Thus, we mark the first two widgets as symbolic. That is, we use `CATG.readString("superman")` to create a symbolic string and use `CATG.readString(100)` to create a symbolic integer. Here "superman" and 100 are the default concrete values for the symbolic variables.

**Step 4.** We use Catg to execute each instrumented GUI application symbolically fol-
lowing the specific event sequence. Consider \((e^1_3, e_7)\) again, where \(e_7\) is associated with branches of an \textit{if} statement. Symbolic execution will lead to the creation of new values for \textit{name} and \textit{distance}, e.g., \textit{name}=\textit{superman} and \textit{distance}=50. Mapping the values back to the event sequence will result in \((e^{\textit{superman}}_1, e^{50}_2, e^1_3, e_7)\), where the first two events are added to set the values of the widgets.

Step 5. After generating all the event sequences and data values using concolic execution, we use JaCoCo to compute the coverage report. JaCoCo is an open-source code coverage library for Java, whose output is formulated as an HTML page to show the coverage statistics.

3.2 Architecture

Figure 3.3 shows the architecture of GUICat. Given a GUI program \(P\) as input, GUICat first invokes \textit{GUITAR} to generate event sequences. Then, it instruments the program based on each sequence and the symbolic variables specified in \textit{guicat-conf}. Next, it invokes the distributed algorithm to conduct symbolic execution of the instrumented program on a cloud node. Finally, the test cases generated by all instrumented programs are collected and then used by JaCoCo to compute the coverage report.

In the distributed symbolic execution algorithm, \(N_0\) is the load balancer and \(N_1 \ldots N_k\) are the \(k\) workers on the cloud. \(N_0\) distributes the set \(E\) of instrumented GUI programs, one per event sequence, over the \(k\) workers. The workers then conduct symbolic execution on their share of tasks. Initially, each worker receives roughly the same number of tasks. However, since the cost of symbolic execution varies for each event sequence, some workers may finish their symbolic execution tasks sooner than others. \(N_0\) detects such imbalance and requests a worker with the largest workload to share its tasks with the idle worker. After all workers complete their tasks, the load balancer \(N_0\) collects the test cases.
Algorithm 1 shows the symbolic execution procedure for each individual worker. Initially, the set $E$ of tasks and test cases $T$ are both empty. Then, the worker keeps checking messages from $N_0$ and conducting local symbolic execution. If it receives a set $E'$ of tasks (from $N_0$ or another worker), the new tasks are added to the local set. Since the number of tasks is changed, it updates $N_0$ with its current number of tasks. This also occurs at Lines 8 and 16. With such updates, $N_0$ knows the number of tasks to be processed at the workers. If the worker receives a message from $N_0$ that requests more tasks on behalf of another worker $N_i$, it sends half of its tasks to $N_i$ (Line 7). A signal of termination is received if $N_0$ asks for its test cases, and in such case, the current worker sends the locally generated test cases and then terminates.

Symbolic execution is conducted for each individual event sequence $e \in E$. Essentially, it allows us to execute the event-driven application as if it is a sequential Java program. We leverage the Catg concolic execution tool, which maintains two execution stacks: one for concrete execution and the other for symbolic execution. When Catg executes an unknown method, for example, `Integer.parseInt()` , the symbolic execution stack would not be updated, we have modified Catg to handle unknown methods.

The distributed algorithm in GUICat has been implemented on Amazon EC2 as a
Algorithm 1 Symbolic Execution on a Worker.

1: \( E = T = \emptyset; \)
2: \( \text{while true do} \)
3: \( \quad \text{if receiving task set } E' \text{ then} \)
4: \( 
\quad E = E \cup E';
\)
5: \( \quad \text{send}(N_0, |E|); \)
6: \( \quad \text{else if requesting tasks on behalf of } N_i \text{ then} \)
7: \( \quad \text{send}(N_i, E[0..|E|/2]); \)
8: \( \quad \text{send}(N_0, |E|); \)
9: \( \quad \text{else if collecting test cases then} \)
10: \( \quad \text{send}(N_0, T); \)
11: \( \quad \text{terminate}; \)
12: \( \text{end if} \)
13: \( \text{for all } e \in E \text{ do} \)
14: \( 
\quad P_e = \text{instrument}(P, e, c f g) \)
15: \( 
\quad T_e = \text{Catg'.execute}(P_e); \)
16: \( \quad \text{send}(N_0, |E|); \)
17: \( \quad T = T \cup T_e; \)
18: \( \text{end for} \)
19: \( \text{end while} \)

Cloud service. We divide EC2 instances into the load balancer and the workers. The load balancer is a multi-processor EC2 instance that generates event sequences, distributes tasks to the workers, and collects the test cases generated by the workers. Each worker is a single-processor EC2 instance that symbolically executes an event sequence to generate test cases. GUICat allows the user to customize the Cloud service, such as the number of workers and their computation capabilities, based on customer requirements such as whether a budget-first testing is preferred over a speed-first testing, or whether branch coverage is preferred over instruction coverage.

To allow easy customization, we implement GUICat by following the star topology shown in Figure 3.4, where the load balancer generates and distributes event sequences to the workers, and the workers conduct concolic execution with respect to the event sequence in isolation, before sending test cases back to the load balancer. Figure 3.5 illustrates how event sequences are distributed to the workers and how test cases are collected from the workers. For now, distributed file system libraries are used to implement the transfer of event sequences and test cases between the load balancer and workers. The main advantage of this architecture is efficiency since there is no communication between the workers.
3.3 Experimental Evaluation

We have implemented GUICat using a number of open-source tools, including GUITAR for generating the initial event sequences, ASM for instrumenting the Java bytecode, Catg for implementing the distributed parallel concolic execution, and JaCoCo for computing the code coverage report.

We have evaluated GUICat on several GUI testing benchmarks. In all experiments, we have used the Amazon EC2 cloud computing infrastructure, where the load balancer is deployed as a multi-processor EC2 instance and each worker is deployed as a single-processor EC2 instance.

In the remainder of this section, we report the results of two case studies: a ticket seller and a workout generator. In each case study, our experiment consists of the following steps. First, we create a configuration file for the application under test. Then, we generate the event sequences using GUITAR [61]. Next, we distribute the event sequences from the load balancer to workers on Amazon EC2. The initial distribution is static and divides the tasks evenly to the EC2 instances. After receiving the event sequences, each worker conducts concolic execution using Catg; as a result, test cases are generated for these event
sequences. When all workers finished, the load balancer collects their test cases and then uses JaCoCo to compute the coverage report.

### 3.3.1 The Ticket Seller

Figure 3.6 shows the user interface of a more sophisticated ticket seller than the one shown in Figure 3.1. It allows the user to provide passenger information such as the Name, ID, start distance (From), end distance (To), Age Level, Class Level, and the Coupon. When the user clicks the Buy Ticket button, the application stores the passenger information to an object named TicketModel, checks for consistency using the method `checkModel()`, and computes the price using the method `computePrice()`.

![Figure 3.6: Ticket Seller.](image)

There are five different types of GUI widgets in Figure 3.6: four of `JTextField` type, one of `JComboBox` type, one of `JRadioButton` type, one of `JCheckBox` type, and two of `JButton` type.

The first two `JTextField` widgets collect the values of Name and ID by invoking `JTextField.getText()`, the combination of which may lead to buggy behaviors. We mark both fields as symbolic. That is, when loading the related Java class, we use the bytecode rewriting tool ASM to instrument the program on the fly, to replace invocations of `getText()` with invocations of `sGetText()`, a method that we develop to return a symbolic value. The symbolic values for Name and ID are used during the subsequent symbolic execution step.

The next two `JTextField` widgets collect the values of From and To by first invoking `JTextField.getText()` and then invoking `Integer.parseInt` to cast the strings into integers. Since
the symbolic execution engine Catg does not support such casting, we have modified Catg to convert these strings into integers before using them in the subsequent logic.

As for the selectable widgets JComboBox, and JCheckbox, we enumerate all possible values of the enum types. We choose enumeration over symbolic execution for these selectable widgets due to efficiency and ease of implementation. First, selectable widgets do not have many different values. Second, the existing symbolic execution engine often has trouble handling them. For example, JComboBox has two methods getSelectedIndex() and getSelectedItem() that return values of a customized Object type, which cannot be easily cast to strings or integers inside Catg.

To summarize, we used GUITAR to generate the initial event sequences together with the initial parameters/states. Then, we add the enumerated values for selectable widgets, before conducting symbolic execution to generate the values for widgets of other types. If the initial event sequence is too short to contain all widgets of JTextField type needed, we remove the stateless JTextField event and then append more stateful JTextField event to the beginning of the event sequence, thus increasing the length of the sequence.

We analyzed the ticket seller with several configurations. Table 3.1 shows the results of generating the length-2 test cases. Columns 1-2 show the tool name and the name of the button clicked. Columns 3-4 show the total number of branches and the percentage of branches covered. Columns 5-6 show the total number of instructions and the percentage of instructions covered. The results show that GUICat can achieve full branch and instruction coverage even with length-2 test cases, whereas GUITAR can only achieve 33.3% branch coverage and 53.5% instruction coverage for checkModel, 23.5% branch coverage and 30.8% instruction coverage for computePrice.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Button</th>
<th>Branches</th>
<th>Coverage</th>
<th>Instructions</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUITAR</td>
<td>checkModel</td>
<td>6</td>
<td>33.3%</td>
<td>56</td>
<td>53.5%</td>
</tr>
<tr>
<td></td>
<td>computePrice</td>
<td>34</td>
<td>23.5%</td>
<td>172</td>
<td>30.8%</td>
</tr>
<tr>
<td>GUICat</td>
<td>checkModel</td>
<td>6</td>
<td>100%</td>
<td>56</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>computePrice</td>
<td>34</td>
<td>100%</td>
<td>172</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 3.1: Results on the ticket seller (length = 2, node = 1).
Table 3.2 compares GUICat and GUITAR in terms of the number of paths covered. Column 1 shows the tool name. Column 2 shows the length of the test sequences. Column 3 shows the total number of test cases generated by GUITAR. Column 4 shows the number of additional test cases generated by GUICat after enumerating the values of selectable widgets. Column 5 shows the number of test cases generated by GUICat after concolic execution. Column 6 shows the path coverage.

Table 3.2: Results on ticket seller: the test cases generated in addition to GUITAR and the unique paths covered.

<table>
<thead>
<tr>
<th>Tool</th>
<th>length</th>
<th>Test Case (TC)</th>
<th>enum TC</th>
<th>concolic TC</th>
<th>paths covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUITAR</td>
<td>2</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>110</td>
<td>-</td>
<td>-</td>
<td>7</td>
</tr>
<tr>
<td>GUICat</td>
<td>2</td>
<td>11</td>
<td>13</td>
<td>286</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>110</td>
<td>156</td>
<td>3212</td>
<td>190</td>
</tr>
</tbody>
</table>

To accurately compute the number of paths covered, we manually added code into the program. Specifically, we used a vector named path, where each element was mapped to an if-statement. For example, the age combo-box corresponds to an if-statement where we set path[0]=0 in the then branch and path[0]=1 in the else branch. Each time the program terminates, we will obtain a unique vector path that acts as the path identifier. These vectors are stored and then used to compute the path coverage after GUICat terminates.

Our result shows that, overall, GUICat achieved significantly higher path coverage than GUITAR. For length-2 test sequences, in particular, GUICat had 96/4=24 times higher path coverage, whereas for length-3 test sequences, GUICat had 190/7=27 times higher path coverage.

GUICat also successfully detected two bugs in ticket seller. One bug is a NullPointerException caused by the race condition between clicking of the save button and clicking of the buy button as shown in Figures 3.7 and 3.8. The other bug is the failure of an assertion due to the computed price being less than zero.
buyButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ticketModel.name = nameInput.getText();
        if (ticketModel.checkModel()) {
            ticketModel.computePrice();
            assert ticketModel.price > 0 : @ Bug: price <= 0 ! ;
        }
    }
});
saveButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ... 
        ticketModel = null;
    }
});

Figure 3.7: The buggy code snippet in ticket seller.

[AWT-EventQueue-0] ERROR Uncaught exception
java.lang.NullPointerException
at examples.ticket.BaradTicket$1.actionPerformed(BaradTicket.java:177)
...

[AWT-EventQueue-0] ERROR Uncaught exception
java.lang.AssertionError: @ Bug: price <= 0!
at examples.ticket.BaradTicket$1.actionPerformed(BaradTicket.java:235)

Figure 3.8: The ticket seller bug detected by GUICat.

3.3.2 The Workout Generator

Figure 3.9 shows the user interface of the workout generator, which is a program taken from Barad [30]. It generates a workout plan based on the user input, including the Gender, Metabolism, Experience, Age, Height, and Weight. The computation starts when the user clicks the Generate button. Depending on the user information, the computation goes through different execution paths that use different cardio coefficients.

Figure 3.9: Workout Generator.

There are three JTextField widgets and three JComboBox widgets. The JTextField widgets return string values of Age, Height, and Weight using the method JTextField.getText(), before casting them to integers using Integer.parseInt(). Based on the configuration file provided
by the user, GUICat creates three symbolic variables for these three widgets, and replaces 
\textit{getText()} with \textit{sGetText(Object)} so it can return symbolic values. The \textit{JComboBox} widgets return values of \textit{Gender}, \textit{Metabolism}, and \textit{Experience}. Since they are selectable widgets, we use enumeration to create the value combinations.

Table 3.3 shows the results of applying GUICat to workout generator with the length of test sequences set to 1 and 3. Columns 1-2 show the tool name and the length of the test sequences. Columns 3-4 show the number of branches covered and the percentage of branches covered. Columns 5-6 show the number of instructions covered and the percentage of instructions covered. Again, GUICat significantly outperformed \textit{GUITAR} in terms of both path coverage and instruction coverage. Neither tool was able to reach full path and instruction coverage with length-2 test sequences, because there are three selectable widgets, which requires the length of test sequences being to be larger than 2.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Length</th>
<th>Branches</th>
<th>Coverage</th>
<th>Instructions</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUITAR</td>
<td>2</td>
<td>154</td>
<td>13.0%</td>
<td>2425</td>
<td>42.3%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>154</td>
<td>13.0%</td>
<td>2425</td>
<td>42.3%</td>
</tr>
<tr>
<td>GUICat</td>
<td>2</td>
<td>154</td>
<td>68.9%</td>
<td>2425</td>
<td>85.7%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>154</td>
<td>97.4%</td>
<td>2425</td>
<td>100%</td>
</tr>
</tbody>
</table>

With length-3 test sequences, GUICat was able to achieve 100% instruction coverage and 97.4% branch coverage. In contrast, \textit{GUITAR} did not show significant improvement in branch/instruction coverage. The reason is that \textit{GUITAR} only used the default values of the widgets, and thus cannot explore alternative paths even with a longer event sequence. In contrast, GUICat used both enumeration and symbolic execution to diversify the values of the widgets.

Table 3.4 compares GUICat and \textit{GUITAR} in terms of the number of paths covered. Column 1 shows the tool name. Column 2 shows the length of the test sequences used in the experiments. Column 3 shows the number of test cases generated by \textit{GUITAR}. Column 4 shows the number of additional test cases generated by GUICat after enumerating the values of selectable widgets. Column 5 shows the number of test cases generated by
GUICat after concolic execution. Column 6 shows the number of paths covered. Our result demonstrates that, overall, GUICat can achieve significantly higher path coverage than GUITAR. For length-2 test sequences, in particular, GUICat reached $24/1=24$ times higher path coverage, whereas for length-3 test sequences, GUICat reached $56/2=28$ times higher path coverage.

Table 3.4: Results on workout generator: the test cases generated in addition to GUITAR and the unique paths covered.

<table>
<thead>
<tr>
<th>Tool</th>
<th>length</th>
<th>Test Case (TC)</th>
<th>enum TC</th>
<th>concolic TC</th>
<th>paths covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUITAR</td>
<td>2</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>72</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>GUICat</td>
<td>2</td>
<td>9</td>
<td>14</td>
<td>69</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>72</td>
<td>182</td>
<td>909</td>
<td>56</td>
</tr>
</tbody>
</table>

We also observed that GUICat generated many more test cases than the paths covered. For example, with the length set to 3, GUICat generated 909 test cases to cover 56 unique paths, which means some of these test cases have led to the same paths. If we could, for example, identify and eliminate these redundant test cases, the performance of GUICat would be further improved. However, we leave the pruning of these redundant test cases for future work.

3.3.3 Effect of Cloud Computing

Figure 3.10 shows the effectiveness of distributing the testing of the ticket seller (a) and the workout generator (b) over Amazon EC2. The x-axis denote the number of workers, ranging from 1 to 16, and the y-axis denote the time usage in second. The solid, dashed and dotted lines represent the bounded length of 2, 3 and 4, respectively. Due to the inherent parallelism in symbolic execution of different event sequences, the speedup is almost linear.
3.4 Conclusion

We have present GUICat, the first cloud-based GUI testing tool for simultaneously generating high-quality event sequences as well as high-quality data values. Internally, GUICat leverages GUITAR to generate the initial set of event sequences, and then uses a combination of value enumeration and symbolic execution to generate data values of the widgets. GUICat also leverages the cloud computation infrastructure to speed up the test generation, by distributing independent concolic execution tasks to EC2 nodes. We have
implemented GUICat and evaluated it on a set of GUI testing benchmarks. Our experiments show that GUICat can significantly outperform GUITAR on standard GUI testing benchmarks in terms of both branch coverage and instruction coverage.
Chapter 4

Systematic Reduction of GUI Test Sequences

This chapter describes GUICat2, a tool that systematically reducts the GUI test sequences, in details. Section 4.1 uses a motivating example to illustrate the main idea behind our tool. Section 4.2, 4.3, and 4.4 describes preliminaries and our technique formally. Section 4.5 reports our experimental results.

4.1 Motivation

Consider the Java code in Fig. 4.1, which controls a window that allows the user to modify an image by clicking the check box, choosing an angle from the slider control, and clicking the OK button. Clicking the OK button closes the window and thus disables all event handlers. Optionally, the user may click the Save button to store the angle. Fig. 4.2 shows the event flow graph (EFG), where nodes are events and edges indicate the set of events enabled in each step. All four events are enabled initially. However, since clicking OK closes the window, the node labeled OK has no outgoing edges.

To test all possible behaviors, we must visit all reachable states and, from each state,
class ModifyImageWindow extends JFrame {
    boolean convert = false;
    int angle = 0;
    void onCheckBox() {
        int cbValue = checkBox.getValue();
        convert = (1 == cbValue) ? true : false;
    }
    void onSlider() {
        int sliderValue = slider.getValue();
        angle = sliderValue;
        print(convert, angle);
    }
    void onSave() {
        int anValue = angle;
        if (anValue > 0)
            UserSettings.RotationAngle = anValue;
        else
            assert(0); // BUG #1: Crash if reached
    }
    void onOK() {
        if (convert) {
            image.convertToGrayscale();
            image = null;
        }
        if (angle > 0)
            image.rotate(angle); // BUG #2: if image==null
        else
            image.draw(); // BUG #3: if image==null
    }
}

Figure 4.1: The class ModifyImageWindow defines event handlers onCheckBox, onSlider, onSave, and onOK.

Figure 4.2: Event flow graph, where CB, SL, SA, and OK denote onCheckBox, onSlider, onSave, and onOk, respectively.

invoke all enabled events at least once. Naively, this can be accomplished by enumerating all event sequences in the EFG up to a predefined length.

4.1.1 Naive Solution

For the example in Fig. 4.1, all possible states will be reached after invoking at most two events and then from each state, invoking all enabled events will cover the $(state \times event)$
combinations. When the maximum sequence length is set to 3, the total number of event sequences will be \((3 \times 3 \times 4 + 3 + 1 = 40)\) as shown in Fig. 4.3. The number is less than \((4 \times 4 \times 4 = 64)\) because clicking the OK button ends the execution.

However, some of these sequences are redundant. For example, \(\{SA, SA, OK\}\) covers the same behavior as \(\{SA, OK\}\): they visit the same states and, from these states, they execute the same events. This is because executing \(SA\) does not change the program state.

Here, states are value combinations of the variable \(convert\) and the predicate \((angle>0)\). Although \(angle\) is an integer variable, the only thing matters in this application is whether \((angle>0)\). Thus, there are four distinct states: 00, 01, 10, and 11, where 00 is the initial state. From each state, there are outgoing edges, corresponding to the enabled events SL, CB, SA, and OK.

Fig. 4.4 shows the state transition graph (STG) described above, where nodes are states and edges are events executed at the source states. Specifically, from the state 00, if we execute SL, the program goes to the state 01. From the state 01, if we execute CB, the program goes to the state 11. From the state 11, if we execute CB again, the program goes back to the state 01, because clicking CB twice, or any even number of times, reset the status of the check box. Finally, at any of these four states, if we click OK, the execution ends – in this sense, OK can only appear at the end of an event sequence.
Figure 4.4: State transition graph, where each state is a valuation of variable convert and predicate (angle>0).

4.1.2 Our New Method

As we have mentioned, ideally, we would like to execute each enabled event (CB, SL, SA, or OK) at every reachable state (00, 01, 10, and 11). Surprisingly, to achieve this goal, only a small subset of event sequences in the search tree of Fig. 4.3 need to be explored, as shown by the reduced tree in Fig. 4.5. The yellow states are irredundant states, solid blue lines are the irredundant sequences, while blue states represent the backtracking points because they match some previously explored states and thus do not need to be explored again.

Figure 4.5: The reduced tree of event sequences of length ≤ 3: From 40 sequences to 13 sequences and then to 6 sequences.

Initially, the program is at the state 00. The state 01 can be reached via the sequence \{SL\}, 10 can be reached via the sequence \{CB\}, and 11 can be reached via the sequence \{CB, SL\}. Thus, we have brought the GUI application to each of the four states. Next, we
execute each of the four enabled events at every one of the four reachable states.

The number of sequences is not 16, but 13, because some of the shorter sequences are subsumed by longer ones. Specifically, there is no need to execute \{CB\}, \{SL\}, or \{CB, SL\} from the initial state 00, because they are already part of the longer sequences \{CB, SL, OK\} and \{SL, OK\}.

However, there are redundancies even among these 13 event sequences. For example, since CB does not read from any of the two variables (convert and angle), executing CB from any of the four states would result in the same coverage within the Java code of the listener function for CB. Since CB is enabled at the initial state 00, there is no need to test CB again at any other state.

Similarly, since SA reads only from angle, executing SA from 00 and 10 (or 01 and 11) would result in the same coverage within SA. Although SL reads from both variables, it overwrites angle before reading it, and thus depends only on the value of convert. Due to this reason, executing SL from 00 and 01 (or 10 and 11) would result in the same coverage within SL. Finally, since CB depends on both variables, it has to be executed from all four reachable states.

Using this notion of semantic reduction, we will be able to generate the following 6 event sequences, while maintaining the same test coverage as the complete set of 40 sequences.

- $t_1 = \{CB, SL, OK\}$
- $t_2 = \{CB, OK\}$
- $t_3 = \{SL, SA\}$
- $t_4 = \{SL, OK\}$
- $t_5 = \{SA\}$
- $t_6 = \{OK\}$
4.1.3 Comparison to Existing Techniques

Our semantic reduction is different from techniques based on partial order reduction (POR), which is a widely used idea for performing state-space explosion, e.g., in model checking [35, 64, 77] and concurrency testing [28, 34]. The idea of POR has also been used to reduce the cost of testing event-driven programs [52]. That is, when two sequences are equivalent permutations of each other, only one of them will be tested. However, since POR relies solely on the theory of equivalent traces [34], it can only identify redundancy in event sequences of the same length.

In contrast, our reduction goes beyond sequences that are equivalent permutations; it also can identify redundancy in sequences of different lengths, e.g., as shown by \{SA, SA, OK\} and \{SA, 0K\}, which clearly are not permutations of each other. Indeed, applying POR to the example in Fig. 4.1 would produce 29 sequences, which are significantly more than the 6 sequences produced by our new method.

Compared to existing GUI testing tools such as GUITAR [62] and Gazoo [9], our method has two significant advantages: it does not skip useful test sequences and it often leads to fewer test sequences. Both GUITAR and Gazoo skip test sequences in a \textit{ad hoc} manner to reduce their computational overhead, which means they often miss important corner cases. For instance, below are the seven sequences generated by Gazoo for our running example:

- $t_1' = \{CB, SL, OK\}$ – same as our $t_1$
- $t_2' = \{CB, CB, OK\}$ – equivalent to a prefix of our $t_1 : \{CB, \ldots \}$ and $t_6: \{0K\}$
- $t_3' = \{SL, SL, OK\}$ – equivalent to our $t_4: \{SL, 0K\}$
- $t_4' = \{CB, CB, SL\}$ – equivalent to $t_3: \{SL, \ldots \}$
- $t_5' = \{CB, SL, SL\}$ – equivalent to $t_1: \{CB, SL, \ldots \}$
- $t_6' = \{SL, SL, SL\}$ – equivalent to $t_3: \{SL, \ldots \}$
- $t_7' = \{SL, SL, SA\}$ – equivalent to our $t_3: \{SL, SA\}$
Note that all sequences generated by Gazoo are subsumed by our sequences. Some are clearly redundant: both $t'_6$ and $t'_7$ are subsumed by our sequence $t_3$, and both $t'_1$ and $t'_5$ are subsumed by our sequence $t_1$. In addition, our sequences are not only fewer (6 versus 7) but also shorter, which translates to faster test execution. We will show in our experimental evaluation that, for larger applications, the test execution time of Gazoo is significantly longer than that of our tool.

Second, the seven sequences generated by Gazoo does not cover all behaviors manifested by our six sequences. In particular, they missed $t_2 = \{\text{CB, OK}\}$ and $t_5 = \{\text{SA}\}$. Both are useful test cases. For example, Bug#1 in Fig. 4.1 (Line 18) can be reached by $\{\text{SA}\}$ and Bug#3 (Line 28) can be reached by $\{\text{CB, OK}\}$. Since Gazoo failed to generate these two sequences, it missed these bugs.

A more severe problem of Gazoo and other existing techniques is that as the length increases, the number of sequences grows exponentially. For our running example, when the maximum sequence length is set to 5, 7, 9, $\ldots$, the number of sequences generated by Gazoo would be 33, 129, 513, $\ldots$, respectively, as shown in Fig. 4.6, whereas the number of sequences generated by our new method would remain 6. The reason is because our six sequences already cover all possible behaviors of this GUI application as shown in Fig. 4.5.

![Figure 4.6](image)

Figure 4.6: The number of sequences generated by Gazoo and our tool as the maximum sequence length increases.
4.1.4 Stateless Implementation

At this moment, one may have the impression that our new method relies on recording the states of the GUI application at run time, which is often how classic POR techniques are implemented in model checkers. However, this is not the case. Our main contribution in this work is using stateless techniques to identify the provably redundant event sequences without executing the GUI application. This is important because recording concrete states may be prohibitively expensive in practice.

Toward this end, we assume the state transition graph (STG) shown in Fig. 4.4 is not available to us. Instead, we rely on checking a set of sufficient conditions under which two sequences are guaranteed to result in the same state. Furthermore, we make sure that these conditions can be checked statically by inspecting only the event sequences.

For example, since SA does not write to any variable, executing it does not change the state. Therefore, if an event leading to the state s is SA (or any other event that does not modify state variables) we know s is an already-explored state. Consequently, we can skip all event sequences starting from s. Similarly, executing \{SL, SL\} would lead to the same state as executing \{SL\}, assuming that executing SL always returns the same set of non-0 values for angle.

In both cases, we need not access the values of program variables. Instead, we check, for each event, what are the set of variables it reads from or writes to, and for any two events, if they are causally dependent (details in Section 4.4).

These efficiently-checkable conditions are sufficient in that, if they hold, the corresponding sequences are guaranteed to be redundant. In this sense, our stateless reduction never removes useful sequences. However, these conditions are not necessary: we do not attempt to capture all redundant sequences because the runtime overhead would be prohibitively high. Inherently, this is a trade-off between the pruning capability and the computational overhead. Thus, it is important to assess how well our method perform in practice, e.g., does it reach or come close to the ideal reduction? In our experimental eval-
uation (Section 4.5), we will demonstrate that our carefully designed sufficient conditions work well in practice.

4.2 Preliminaries

4.2.1 GUI Application

A GUI application consists of (1) a set $C$ of containers, such as windows, panels and tabs, (2) a set $W$ of widgets, such as labels, buttons, and check boxes, and (3) a set $L$ of listener functions associated with the widgets. Widgets are grouped by containers that host them, and are elements of interaction allowing the user to interact with functional parts of the software code. GUI libraries such as Java AWT/Swing contain a large collection of widgets and the default program logic for manipulating them. Typically, each widget facilitates a specific type of user-computer interaction.

A listener $l \in L$ is a function that may be invoked to respond to a user event. Some listeners are built-in listeners provided by GUI libraries while others are custom-made: they are written by the application developers. For ease of presentation, we consider one listener per widget in the remainder of this paper. Thus, triggering an event means executing the listener associated with the widget.

4.2.2 Event Flow Graph

The mapping between widgets and listeners of a given application may be reverse engineered using tools such as GUITAR [29], which leverages dynamic execution and the Java accessibility feature to traverse an object and its children and execute their listeners. The widget-to-listener mapping obtained in this way is represented by an event flow graph (EFG), which shows the set of events enabled at every step of the execution. The EFG is one of the inputs to test sequence generation tools including ours.
Formally, an event flow graph is a directed graph $G_{EFG} = (E, T)$, where $E$ is a set of events and $T$ is a set of transitions between these events. Let $E_0 \subseteq E$ be the set of events enabled at the beginning of the GUI execution. From these initial events, each subsequent transition $(ev_i, ev_j) \in T$, where $ev_i, ev_j \in E$, represents the fact that executing $ev_i$ allows $ev_j$ to be executed in the next step.

Fig. 4.2 shows an example EFG consisting of four events CB, SL, SA, and OK, which corresponds to the four listener functions in Fig. 4.1. All four events are enabled initially, and after executing any of the first three events, all four events remain enabled. However, OK is different in that executing this event would end the execution: in the EFG, OK does not have any outgoing edges.

### 4.2.3 Dependency Relation

Since event listener functions may read from or write to shared variables, they may impose dependency over these events. In partial order reduction [34], two events are considered conflict-dependent (or simply dependent) if they access the same variable and at least one of the accesses is a write operation. Two events that are not dependent with each other are considered as independent. In Fig. 4.1, for example, OK is conflict-dependent on both CB and SL because it reads from convert and angle written by the other two events. Based on this notion of dependency, two event sequences are equivalent if they can be transformed to each other by repeatedly swapping the adjacent and independent events.

Although this dependency relation has been widely used in model checking and concurrency testing, it is often not accurate enough for dealing with events in GUI applications. For example, in Fig. 4.1, one SL event and another SL event have overlapping read-write and write-write sets – since they both read from and write to angle. However, the listener function of SL always overwrites the value of angle before reading from it, which means the behavior of the second SL event’s listener function does not depend on the value of angle written by the first SL event. In this sense, we say these two events are
not causally dependent.

**Causal Dependency.** Thus, we rely on a more refined notion of dependency, named the *causal-dependency*. More formally, two events \( ev_1, ev_2 \in E \) are causally dependent, denoted \((ev_1, ev_2) \in R_{CD}\), where \( R_{CD} \) is the dependency relation, if the execution of any one of them may affect the subsequent execution of the other. When two events are not causally dependent, we say they are *causally independent*.

Thus, causal-dependency is more accurate than conflict-dependency in that it reflects the actual impact of one event over another. In Section 4.4, we shall explain how a simple static analysis of the event listener functions can be leveraged to determine whether two events are causally dependent.

### 4.3 Systematic Test Generation

We first present the baseline algorithm for generating test sequences (with no reduction), and then discuss how to integrate POR-based reduction into the algorithm.

#### 4.3.1 The Baseline Algorithm

Given the EFG \( G = (E, T) \) and the causal dependency relation \( R_{CD} \), Algorithm 2 (excluding the blue colored lines 9 and 12) generates all possible event sequences up to a predetermined length. Following the well-established notation in stateless model checking [34], we use a state stack named \( S \) to store the sequence of (abstract) states. \( S \) contains the initial state \( s_0 \) at the beginning. For each state \( s \in S \), we use \( s\.enabled \) to denote the set of events enabled at \( s \), use \( s\.selEV \) to denote the event chosen to execute at \( s \), and use \( s\.done \) to denote the set of all previously chosen events at \( s \).

The procedure **EXPLOR**E first checks if the execution has ended, i.e., if \( S\.size > MAXLENGTH \) or \( s\.enabled = \emptyset \). If either condition is met, the while-loop will be skipped. Then, **OUT-PUT-SEQUENCE**\((S)\) will be invoked to print the event sequence stored in \( S \), provided that
S holds a complete execution (indicated by \( s.selEV = NULL \)) as opposed to the prefix of a longer execution.

Otherwise, it enters the while-loop to execute a previously unexplored event, sets \( s.selEV = event \), and recursively invokes \( \text{EXPLORE} \). After all events in \( s.enabled \) are explored, it exits the while-loop. At this moment, \( s.selEV \) will not be NULL, meaning \( S \) holds the prefix of a longer sequence (that has been printed). Thus, we do not invoke \( \text{OUTPUTSEQUENCE}(S) \).

Consider the running example in Fig. 4.1. Applying Algorithm 2 with \( \text{MAXLENGTH}=3 \) would explore the complete tree of 40 sequences as shown in Fig. 4.3. Clearly, some of these sequences are redundant and thus should be removed. Toward this end, we will present partial order reduction in the remainder of this section, as well as our new redundancy removal technique in Section 4.4.

For now, we would like to note that, compared to existing test generation tools such as GUITAR [62] and Gazoo [9], the main advantage of Algorithm 2 is that it captures all possible event sequences that the EFG can produce up to the predefined length. As such, it does not miss useful test sequences.

### 4.3.2 Partial Order Reduction

The idea of partial order reduction originated from explicit-state model checking [35, 64, 77], where the model checker needed to reduce the size of the state space to be searched. In this context, a large number of algorithms were developed, including the stubborn set method, ample set method, and persistent set method. For a comprehensive review of these classic methods, refer to Godefroid’s book [34]. Nevertheless, all these classic methods rely on the same principle, which is first classifying the execution traces of a concurrent systems into equivalence classes of permutations, and then exploring one representative from each of these equivalence classes. Since all traces in the same equivalence class lead to the same system behavior, covering all equivalence class is the same as cov-
In this context, two execution traces are equivalent if one trace can be transformed into another trace by repeated swapping the adjacent and independent events in this trace.

In Fig. 4.3, for example, the following two sequences meet this criterion and thus are considered equivalent: \{…, CB, SA, …\} and \{…, SA, CB, …\}. The reason is that CB only writes to convert and SA only reads from angle; thus, the execution order of these two events is immaterial. If \{…, CB, SA, …\} has been explored by the test generation algorithm, then \{…, SA, CB, …\} can be skipped.

In Algorithm 2, we have added Lines 9 and 12 to show a particular implementation of POR: the sleep-set based method [34]. Specifically, after an event (e.g., CB in Fig. 4.3) is explored from a state \(s\) (Line 11), it is put into the set \(s.\text{sleep}\) (Line 12). When we explore another event (e.g., SA) from \(s\), we check if the new event is independent of events in \(s.\text{sleep}\). If that is the case, we carry the sleep set over to the next state \(s'\). Otherwise, we drop all the dependent events from \(s'.\text{sleep}\). At any time, if the procedure EXPLORE attempts to execute a new event (e.g., CB) that already exists in the sleep set (e.g., as in \{…, SA, CB\}), we skip the event, because executing the event is guaranteed to reach a previously explored state.

**Algorithm 2** Baseline test generation procedure with POR.

```
1: Let StateStack \(S = \{s_0\}\), \(s_0.\text{enabled} = \text{initially-enabled events} \), and invoke EXPLORE\((S)\)
2: procedure EXPLORE\((S)\)
3: \hspace{1em} let \(s = S.\text{top}()\)
4: \hspace{2em} if \((S.\text{size}()) \leq \text{MAXLENGTH}) \text{ then} \hspace{1em}
5: \hspace{3em} let \(s.\text{done} = \emptyset\)
6: \hspace{4em} while \(\exists \text{event} \in (s.\text{enabled} \setminus s.\text{done} \setminus s.\text{sleep})\) do \hspace{1em}
7: \hspace{5em} add event to \(s.\text{done}\)
8: \hspace{6em} add event to \(s.\text{sleep}\) \hspace{1em}
9: \hspace{2em} let \(s'.\text{sleep} = \{e \in s.\text{sleep} | e \text{ and event are independent} \}\)
10: \hspace{1em} S.\text{push}(s')
11: \hspace{1em} EXPLORE\((S)\)
12: \hspace{1em} add event to \(s.\text{sleep}\)
13: end while
14: if \((s.\text{selEV} = \text{NULL}) \text{ then} \hspace{1em}
15: \hspace{2em} OUTPUTSEQUENCE\((S)\) \hspace{1em}/\!/ End trace: \forall s \in S, print \(s.\text{selEV}\)
16: end if
17: S.\text{pop}() \hspace{1em}
18: end procedure
```

Although POR has been used in testing event-driven programs [9, 52]), it has a limi-
tation. That is, POR can only identify redundant sequences that are permutations of each other, which implies that these sequences must have the exact same set of events to be considered equivalent. Thus, it will never be able to identify redundancy in sequences such as \{SA, SA, OK\} and \{SA, OK\} in Fig. 4.5 because these sequences have different lengths. Therefore, we need to develop a more powerful reduction technique.

4.4 The New Reduction Technique

We first explain the rationale behind our new reduction technique and then present our stateless implementation.

4.4.1 The New Algorithm

Algorithm 3 shows our method, which is the original algorithm augmented with two modifications shown at Lines 8 and 17. That is, prior to executing an event (Line 8), we check if the new state \(s'\) is a previously explored state by analyzing the events stored in \(S\) \((s.selEV \in S)\). Similarly, prior to printing an event sequence (Line 17), we check if it can be subsumed by some other (shorter) sequences. Both of these two additional checks are conservative in nature (sufficient conditions), which means if they return true, we can safely skip the corresponding states and sequences.

The subroutine REDUNDANTSTATE takes the current state stack \(S\) and the next event \(ev\) as an input. Recall that states in \(S\) are abstract states that do not have concrete values of the variables. Instead, we rely on the event sequence stored in the \(selEV\) field of each \(s \in S\) to check if the new state (to be executed) has been explored. Thus, our implementation of the check is stateless. Similarly, REDUNDANTSEQUENCE checks if the event sequence stored in \(S\) can be subsumed by some shorter sequences.

Consider our running example in Fig. 4.5. The new subroutine REDUNDANTSTATE would return true when the current sequence in \(S\) plus the next event, denoted \([S]::\)
Algorithm 3 New test generation procedure with reduction.

1: Let StateStack $S = \{s_0\}$, $s_0.enabled =$ initially-enabled events, and invoke $\text{EXPLORE}(S)$
2: \textbf{procedure} $\text{EXPLORE}(S)$
3: \hspace{1em} let $s = S.top()$
4: \hspace{1em} if ($s.size() \leq \text{MAXLENGTH}$) then
5: \hspace{2em} let $s.done = \emptyset$ and $s.printed = \emptyset$
6: \hspace{2em} while $\exists \text{event} \in (s.enabled \setminus s.done \setminus s.sleep)$ do
7: \hspace{3em} add event to $s.done$
8: \hspace{2em} \hspace{1em} if $\neg \text{REDUNDANTSTATE}(S, \text{event})$ then
9: \hspace{3em} \hspace{2em} let $s' = \text{NEXTSTATE}(s, \text{event})$ \hspace{1em} // Set $s.selEV = \text{event}$
10: \hspace{3em} \hspace{2em} let $s.sleep = \{e \in s.sleep \mid e \text{ and } \text{event} \text{ are independent}\}$
11: \hspace{3em} \hspace{2em} $S.push(s')$
12: \hspace{3em} $\text{EXPLORE}(S)$
13: \hspace{2em} \hspace{1em} add event to $s.sleep$
14: \hspace{2em} end while
15: \hspace{1em} end if
16: \hspace{1em} if $\neg \text{REDUNDANTSEQUENCE}(S, s)$ then
17: \hspace{2em} $\text{OUTPUTSEQUENCE}(S)$ \hspace{1em} // End trace: $\forall s \in S$, print $s.selEV$
18: \hspace{2em} end if
19: \hspace{1em} $S.pop()$
20: \textbf{end procedure}
21: \textbf{procedure} $\text{REDUNDANTSTATE}(S, \text{ev})$
22: \hspace{1em} if $\text{NoWrite}() \lor \text{SameWrite}() \lor \text{CovWrite}() \lor \text{GenCovWrite}()$ then
23: \hspace{2em} return true
24: \hspace{2em} else
25: \hspace{3em} return false
26: \hspace{2em} end if
27: \textbf{end procedure}
28: \textbf{procedure} $\text{REDUNDANTSEQUENCE}(S, s)$
29: \hspace{1em} if $s.selEV \notin s.printed$ \land $\text{IrrelevantTail}() \lor \text{extraSink}() \lor \text{CausalIndep}()$ then
30: \hspace{2em} return true
31: \hspace{2em} else
32: \hspace{3em} $\forall s \in S$, add $s.selEV$ to $s.printed$
33: \hspace{2em} return false
34: \hspace{2em} end if
35: \textbf{end procedure}
36: \textbf{end procedure}

\{\text{event}\}, contains the following sequences:

- $[\text{CB}] :: \{\text{CB}\} \to \text{state } c_1$
- $[\text{CB}, \text{SL}] :: \{\text{CB}\} \to \text{state } d_5$
- $[\text{CB}, \text{SL}] :: \{\text{SL}\} \to \text{state } d_6$
- $[\text{CB}, \text{SL}] :: \{\text{SA}\} \to \text{state } d_7$
- $[\text{CB}, \text{SL}] :: \{\text{OK}\} \to \text{state } d_8$ (not redundant sequence);
- $[\text{CB}] :: \{\text{OK}\} \to \text{state } c_3$
- $[\text{CB}] :: \{\text{OK}\} \to \text{state } c_4$ (not redundant sequence);
- $[\text{SL}] :: \{\text{CB}\} \to \text{state } c_5$
- $[\text{SL}] :: \{\text{SL}\} \to \text{state } c_6$
- $[\text{SL}] :: \{\text{SA}\} \to \text{state } c_7$ (not redundant sequence);
• [SL] :: {OK} → state $c_8$ (not redundant sequence);

• [SA] :: {CB} → state $c_9$.

• [SA] :: {SL} → state $c_{10}$.

• [SA] :: {SA} → state $c_{11}$.

• [SA] :: {OK} → state $c_{12}$.

• [SA] :: → state $b_3$ (not redundant sequence);

• {OK} → state $b_4$ (not redundant sequence);

That is, we backtrack as soon as reaching any of the following states: $c_1, d_{5-8}, c_{3-12},$ and $b_3-4$, because REDUNDANTSTATE proves they are already explored. In addition, except for six of these sequences, REDUNDANTSEQUENCE proves they are subsumed by some shorter sequences.

Among the six event sequences that are not redundant, {SA} :: → $b_3$ deserves further explanation since it is the only one that is a prefix of some longer sequences. Furthermore, none of these long sequences (extensions of {SA}) was printed (because they are redundant sequences themselves). Thus, upon reaching Line 17 of Algorithm 3, we invoke OUTPUTSEQUENCE to print it out.

To enable the printing of partial sequences such as {SA} :: → $b_3$, we add $s.printed$ to record the set of events printed at $s$. Initially, $s.printed$ is empty (Line 5). Every time REDUNDANTSEQUENCE returns false (which forces the current sequence to be printed), we add $s.selEV$ to $s.printed$ (Line 33). Thus, only if ($s.selEV$ $\notin$ $s.printed$) (Line 30), we allow REDUNDANTSEQUENCE to return true. Otherwise, the current sequence would have already been printed as part of a longer sequence.

### 4.4.2 Detecting Redundant States

Now, we present the sufficient conditions for detecting already explored states (Line 8 of Algorithm 3). Let
• $s_{n-1}$ and $s_n$ be the last two states in the state stack $S$,
• $ev_{n-1}$ be the event chosen (and executed) at $s_{n-1}$, and
• $ev$ be the event considered (but not yet executed) at $s_n$.

We start with special cases $NoWrite$ and $SameWrite$, which are easier to understand, before presenting the general cases.

$NoWrite()$. The first sufficient condition for $[S] :: \{ev\}$ to result in a redundant state is as follows:

$$(ev_{n-1}.write = \emptyset) \land (ev \in s_{n-1}.enabled)$$

Proof sketch: As shown in Fig. 4.7 (a), since $ev_{n-1}.write = \emptyset$, we know executing $ev_{n-1}$ does not change the state. Thus, $s_n = s_{n-1}$. Furthermore, since $ev \in s_{n-1}.enabled$, the sequence $\{ev_1, \ldots, ev_{n-2}, ev\}$ always exists, and is shorter than $[S] :: \{ev\} = \{ev_1, \ldots, ev_{n-1}, ev\}$. Thus, executing $ev$ from $S$ would not lead to new program behavior.

$SameWrite()$. The second sufficient condition for $[S] :: \{ev\}$ to result in a redundant state is as follows:

$$(ev_{n-1}.write \cap ev_{n-1}.read = \emptyset) \land (ev = ev_{n-1})$$

Proof sketch: First, since $ev = ev_{n-1}$, we know the condition $(ev \in s_{n-1}.enabled)$ holds as well. Furthermore, since $ev_{n-1}.read \cap ev_{n-1}.write = \emptyset$, the values read by $ev$ (and hence the values written by $ev$) do not depend on the values written by $ev_{n-1}$. In other words, executing $ev$ more than once results in the same state. Thus, executing $ev$ from $S$ would not lead to new program behavior.

For example, in Fig. 4.1, $\{\ldots, SL, SL\}$ satisfies this condition. Although SL reads from both convert and angle, the read of angle is dominated by its own write to angle. Thus, we do not consider angle as part of SL’s read-variable set. Consequently, SL does not depend on values written by the previous SL.

$CovWrite()$. This is a generalization of the two previous cases. In this case, the sufficient
condition for $S :: \{ev\}$ to result in a redundant state is as follows:

\[
(ev_{n-1}.write \subseteq ev.write) \land \\
(ev_{n-1}.write \cap ev.read = \emptyset) \land \\
(ev \in s_{n-1}.enabled)
\]

Proof sketch: The first condition means $ev$ overwrites all values written by $ev_{n-1}$, the second condition means $ev_{n-1}$ does not affect $ev$ via shared variables, and the third condition means $ev$ is enabled at $s_{n-1}$ as well. Therefore, the shorter sequence $\{ev_1, \ldots, ev_{n-2}\} :: \{ev\}$ would lead to the same state as the longer sequence $[S] :: \{ev\} = \{ev_1, \ldots, ev_{n-1}, ev\}$. Thus, we can safely skip the execution of $ev$ from $S$.

**GeneralizedCovWrite().** This is a further generalization of the previous case. Let $s_1, \ldots, s_i, \ldots, s_k, \ldots, s_n$ be the entire sequence of states currently in $S$, where $ev_i$ and $ev_k$ are events selected at $s_i$ and $s_k$, respectively, and $ev$ is the event selected (but not yet executed) at $s_n$. The sufficient condition for $[S] :: \{ev\}$ to result in a redundant state is as follows:

\[
\exists 1 \leq i < n . \ (ev_i.write \subseteq ev.write) \land \\
(ev_i.write \cap \bigcup_{i< k < n} ev_k.read = \emptyset) \land \\
(ev_i.write \cap ev.read = \emptyset) \land \\
(ev_{i+1} \in s_i.enabled)
\]

Proof sketch: As shown in Fig. 4.7 (b), the current event $ev$ overwrites all values written by $ev_i$. Furthermore, $ev_i$ does not affect any of the subsequent events including $ev$. Furthermore, since $ev_{i+1} \in s_i.enabled$, there is a sequence $\{ev_1, \ldots, ev_{i-1}, ev_{i+1}, \ldots, ev_n, ev\}$ that is shorter and results in the same state as $[S] :: ev$. In this case, executing $ev$ from $S$ will not lead to new behavior, because the next state can be reached by shorter sequence.
4.4.3 Eliminating Redundant Sequences

Now, we present our sufficient conditions for detecting redundant sequences (Line 17 of Algorithm 3). These reductions are complementary to POR because they consider a sequence as redundant if it is subsumed by some other sequences of shorter length.

Specifically, in Algorithm 3, prior to generating the event sequence (Line 17), we check if any of the following conditions is satisfied. If the answer is yes, we skip the sequence because the equivalent but shorter sequence would be generated.

**IrrelevantTail()**. The first sufficient condition for \([S]\) to be a redundant sequence is as follows. Let \(s_1, \ldots, s_i, \ldots, s_n\) be the sequence of states currently in \(S\), \(ev_i\) be the event selected at the state \(s_i\), and \(ev_n\) be the event selected (and executed) at the state \(s_n\). The sequence \([S]\) is redundant if the following condition is satisfied: \(\exists 1 \leq i < n\) such that

\[
(ev_n.read \cap \bigcup_{1 \leq k < n} ev_k.write) = \emptyset \land (ev_n \in s_i.enabled)
\]

Proof sketch: When the above condition is satisfied, the last event \(ev_n\) (selected and executed at \(s_n\)) is guaranteed not to depend on any value written by the preceding events \(ev_i, \ldots, ev_{n-1}\). In such case, \([S] = \{ev_1, \ldots, ev_n\}\) can be replaced by the two shorter sequences \(\{ev_1, \ldots, ev_{n-1}\}\) and \(\{ev_1, \ldots, ev_{i-1}, ev_n\}\), and thus can be skipped.

In our example, \(\{CB, SA\}\) has an irrelevant tail and thus can be replaced by the two shorter sequences \(\{CB\}\) and \(\{SA\}\).
**ExtraSink()**. Let \( s_1, \ldots, s_i, \ldots, s_j, \ldots, s_n \) be the state sequence in \( S \), \( ev_i \) and \( ev_j \) be the events selected at \( s_i \) and \( s_j \), respectively, and \( ev_n \) be the event selected (and executed) at the last state \( s_n \). The sequence \([S]\) is redundant if the following conditions are satisfied: \( \exists 1 \leq i < j \leq n \) such that (1) \( e_i \) and \( e_j \) do not enable or disable any event executed after them, and (2) the following condition is met:

\[
\begin{align*}
(e_i.\text{write} \cap \bigcup_{k < i \leq n} e_k.\text{read}) &= \emptyset \\
(e_j.\text{write} \cap \bigcup_{k < j \leq n} e_k.\text{read}) &= \emptyset
\end{align*}
\]

Proof sketch: The above condition means neither \( e_i \) nor \( e_j \) can affect any event executed after them in \( S \). Furthermore, skipping \( e_i \) or \( e_j \) does not enable/disable other events. Thus, \([S]\) can be replaced by the shorter sequences \([S] \setminus \{e_i\}\) and \([S] \setminus \{e_j\}\).

For example, in Fig. 4.5, \{CB, \ldots, SA, OK\} has two extra sinks SA and OK. Therefore, it can be replaced by the two shorter sequences \{CB, \ldots, SA\} and \{CB, \ldots, OK\}.

**CausalIndependentWrite()**. The third sufficient condition is related to causally-independent writes. Let \( s_1, \ldots, s_i, \ldots, s_n \) be the state sequence in \( S \), \( ev_i \) be the event selected at \( s_i \) and \( ev_n \) be the event selected (and executed) at \( s_n \). The sequence \([S]\) is redundant if the following conditions are satisfied: \( \exists 1 \leq i < n \) such that (1) \( ev_i \) does not enable/disable any event executed after it in \( S \) and (2) the following condition is met:

\[
(ev_n, ev_i) \not\in R_{CD}
\]

Proof sketch: First, the above condition means \( ev_n \) is not causally dependent on \( ev_i \). In other words, whether \( ev_i \) is executed at \( s_i \) does not affect the behavior of \( ev_n \). Furthermore, since \( ev_i \) does not enable or disable any event executed after it, there exist two shorter sequences \( \{ev_1, \ldots, ev_{n-1}\} \) and \( \{ev_1, \ldots, ev_{i-1}, ev_{i+1}, \ldots, ev_n\} \) that subsume \([S]\). Thus, the event sequence \([S]\) can be skipped.
**4.4.4 Computing Causal Dependencies**

Whether two events $e_{v_i}$ and $e_{v_j}$ are causally dependent, i.e., $(e_{v_i}, e_{v_j}) \in R_{CD}$, can be decided using a conservative static analysis of their listener functions. The analysis is conservative in that, if it says $e_{v_i}$ is not causally dependent on $e_{v_j}$, the behavior of $e_{v_i}$ is guaranteed not to be affected by $e_j$. However, the analysis may not identify all causally independent event pairs due to limitations of static analysis.

We use *Soot* to implement the static analysis. We mark each Java class member as `className.memberName` and consider all program variables. First, we compute, for each event listener function, the set of read variables and the set of write variables. The difference between our *read* and *write* variable sets and those computed by conventional techniques is that we exclude, from the *read* set, variables that are overwritten before they are read.

Specifically, for each event listener function, we parse the Java byte-code and initialize an empty *write* variable set. For each basic block through which the data flows, we take the union of the in-flow and out-flow. For each basic block where two data flows merge, we take the intersection. When we compute the *read* variable set, if the variables read by a basic block is included in the previously-computed flow set, we ignore them, because they have been overwritten by the method itself. Otherwise we add them to the *read* variable set.

After the aforementioned intra-procedural analysis is completed, we use an inter-procedural program slicer [43] similar to the one used by Gazoo [32] to compute the causal-dependency relation $R_{CD}$. Essentially, the program slicer recursively adds variables read or written by the listener function as well as functions invoked by the listener function.

As an example, consider the SL event in Fig. 4.1. Although both `angle` and `convert` are read by the listener function, since `angle` is overwritten before it is read, it is excluded from the *read* set. Thus, SL.read = \{`convert`\} and SL.write = \{`angle`\}, and we have
\((SL, SL) \notin R_{CD}\).

4.5 Experiments

We have implemented our method in a testing tool in which the following open source tools are used: GUITAR [29] for reverse engineering the event flow graph, Soot [63] for conducting static program analysis, and Cobertura [22] for executing test sequences on the GUI application to obtain the coverage report. Our core algorithm for generating test sequences was implemented in about 4,000 lines of Java code. For experimental comparison, we implemented the algorithm in such a way that individual reduction techniques can be enabled and disabled. Thus, we were able to compare the performance of the following configurations: (1) our baseline procedure as shown in Algorithm 2, (2) baseline with POR, (3) baseline with POR plus the individual reductions presented in Section 4.4, and (4) all reductions in Algorithm 3 combined. We also downloaded GUITAR [29, 62] and Gazoo [9, 32] and experimentally compared them with our tool on the same benchmark applications.

In both GUITAR and Gazoo, the test sequence generation is model-based. That is, they leverages the same EFG of a GUI application as in our method, but the difference is in how event sequences are constructed. In our method, the construction starts from the initial states and proceeds systematically, but in GUITAR and Gazoo, the construction may start from any node in the EFG. As such, their initial set of event sequences may not be feasible. To make them feasible—meaning they can be executed by the GUI application—GUITAR and Gazoo have to insert connecting events to these sequences. In contrast, our method can directly generate feasible event sequences.

Our experiments were designed to answer two questions:

- Can our new method, which soundly generates test sequences, outperform state-of-the-art GUI testing tools such as GUITAR and Gazoo?
• How effective is our semantic reduction technique and its stateless implementation in identifying and eliminating redundant sequences?

We used 17 benchmark applications written using Java Swing. Their statistics are shown in Table 4.1. Specifically, Columns 1 and 2 show the name of each application and the number of lines of code (LoC), respectively. Note that the LoCs of regextester and hashvcalc (marked with asterisks) are estimated by decompiling the Java byte-code due to lack of source code. Columns 3 and 4 show the size of the input EFGs including the number of nodes and the number of edges. Finally, Column 5 provides a brief description of the nature of each application. Together, the benchmark applications have 105,937 lines of Java code. We performed all experiments on a computer with a 3.3 GHz CPU and 8 GB RAM.

<table>
<thead>
<tr>
<th>Name</th>
<th>LoC</th>
<th>#Node</th>
<th>#Edge</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GazooV0.1</td>
<td>80</td>
<td>5</td>
<td>12</td>
<td>Example application taken from [7].</td>
</tr>
<tr>
<td>guess</td>
<td>126</td>
<td>4</td>
<td>16</td>
<td>Example crafted to illustrate scenarios similar to GazooV0.1</td>
</tr>
<tr>
<td>GazooV0.2</td>
<td>165</td>
<td>4</td>
<td>16</td>
<td>Example application taken from [9].</td>
</tr>
<tr>
<td>ticketseller</td>
<td>367</td>
<td>11</td>
<td>121</td>
<td>GUI application taken from [31]</td>
</tr>
<tr>
<td>hashvcalc</td>
<td>*376</td>
<td>17</td>
<td>162</td>
<td>Hash value calculator from sourceforge [41]</td>
</tr>
<tr>
<td>workout</td>
<td>526</td>
<td>9</td>
<td>81</td>
<td>Workout generator taken from [31]</td>
</tr>
<tr>
<td>payment</td>
<td>665</td>
<td>19</td>
<td>127</td>
<td>Payment form application from the example of squish [72]</td>
</tr>
<tr>
<td>regextester</td>
<td>*756</td>
<td>14</td>
<td>144</td>
<td>Java regular expression tester from sourceforge [66]</td>
</tr>
<tr>
<td>addressbook</td>
<td>1,267</td>
<td>17</td>
<td>71</td>
<td>Address book application from the example of squish [72]</td>
</tr>
<tr>
<td>jnotepad</td>
<td>1,378</td>
<td>45</td>
<td>649</td>
<td>Notepad application from [47]</td>
</tr>
<tr>
<td>crosswords</td>
<td>3,594</td>
<td>29</td>
<td>106</td>
<td>Dictionary application [23]</td>
</tr>
<tr>
<td>jgp</td>
<td>7,739</td>
<td>71</td>
<td>2,191</td>
<td>GNU plot front-end application from sourceforge [48]</td>
</tr>
<tr>
<td>calc</td>
<td>11,940</td>
<td>83</td>
<td>1,634</td>
<td>Student math calculator from sourceforge [18]</td>
</tr>
<tr>
<td>ce</td>
<td>14,027</td>
<td>98</td>
<td>1,361</td>
<td>Java class file editor from sourceforge [19]</td>
</tr>
<tr>
<td>terpspread</td>
<td>15,231</td>
<td>306</td>
<td>3,079</td>
<td>Java spreadsheet application [75]</td>
</tr>
<tr>
<td>rachota</td>
<td>18,852</td>
<td>148</td>
<td>1,347</td>
<td>Time management application from sourceforge; used in [9]</td>
</tr>
<tr>
<td>buddi</td>
<td>28,848</td>
<td>103</td>
<td>927</td>
<td>Financial management application [17]</td>
</tr>
</tbody>
</table>

4.5.1 Comparison of Different Methods

In this experiment, we compared the performance of our tool against GUITAR and Gazoo. Within our own tool, we also compared three configurations: Baseline, with POR, and with the full-blown reduction.
Table 4.2 shows the results. Specifically, Column 1 shows the name of each benchmark while the remaining columns are divided into the following groups: the baseline algorithm (denoted \textit{Baseline}), baseline with POR (denoted \textit{+POR}), baseline with POR plus our new reduction (denoted \textit{+AllNew}), and the two existing tools. For each method, we show the number of test sequences generated and the test generation time in seconds. The last row sums up the numbers in each column.

Table 4.2: Comparison of event sequence reduction.

<table>
<thead>
<tr>
<th>Name</th>
<th>Baseline</th>
<th>+POR</th>
<th>+AllNew</th>
<th>GUITAR [29]</th>
<th>Gazoo [9]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tests (s)</td>
<td>tests (s)</td>
<td>tests (s)</td>
<td>tests (s)</td>
<td>tests (s)</td>
</tr>
<tr>
<td>GazooV0.1</td>
<td>31</td>
<td>2</td>
<td>19</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>guess</td>
<td>64</td>
<td>3</td>
<td>40</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>GazooV0.2</td>
<td>40</td>
<td>2</td>
<td>29</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>ticketseller</td>
<td>1,331</td>
<td>17</td>
<td>418</td>
<td>8</td>
<td>23</td>
</tr>
<tr>
<td>hashcalc</td>
<td>326</td>
<td>6</td>
<td>198</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>workout</td>
<td>729</td>
<td>12</td>
<td>246</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>payment</td>
<td>1,009</td>
<td>14</td>
<td>214</td>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>regextester</td>
<td>1,470</td>
<td>24</td>
<td>334</td>
<td>12</td>
<td>25</td>
</tr>
<tr>
<td>addressbook</td>
<td>125</td>
<td>4</td>
<td>63</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>jnotepad</td>
<td>8,243</td>
<td>87</td>
<td>1,019</td>
<td>15</td>
<td>158</td>
</tr>
<tr>
<td>crosswords</td>
<td>417</td>
<td>9</td>
<td>140</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>lisp</td>
<td>70,335</td>
<td>728</td>
<td>38,189</td>
<td>599</td>
<td>10,613</td>
</tr>
<tr>
<td>calc</td>
<td>4,179</td>
<td>54</td>
<td>1,183</td>
<td>74</td>
<td>68</td>
</tr>
<tr>
<td>ce</td>
<td>21,484</td>
<td>233</td>
<td>5,191</td>
<td>232</td>
<td>755</td>
</tr>
<tr>
<td>terpspread</td>
<td>15,303</td>
<td>165</td>
<td>4,840</td>
<td>111</td>
<td>3,045</td>
</tr>
<tr>
<td>rachota</td>
<td>5,036</td>
<td>67</td>
<td>1,646</td>
<td>39</td>
<td>135</td>
</tr>
<tr>
<td>buddi</td>
<td>1,666</td>
<td>32</td>
<td>318</td>
<td>20</td>
<td>60</td>
</tr>
<tr>
<td>Total</td>
<td>131,788</td>
<td>1,459</td>
<td>54,087</td>
<td>1,144</td>
<td>15,009</td>
</tr>
</tbody>
</table>

Following Arlt et al. [9] we set the maximum sequence length to 3 for all methods. Note that Gazoo was not able to generate any test sequences for four applications because the tool was hard-wired to produce sequences of dependent events with length 2 or longer, but these four applications do not have sequences that meet the criterion.

Overall, there is a significant reduction (59%) in the number of test sequences from \textit{Baseline} to \textit{+POR}, and another significant reduction (72%) to \textit{+AllNew}. This means our method is different from and complementary to POR. Furthermore, there are fewer sequences generated by our method (\textit{+AllNew}) than GUITAR and Gazoo, and the reduction is more than an order of magnitude (14.3X over GUITAR and 11.9X over Gazoo). The re-
duction is obtained despite that our method is sound whereas GUITAR and Gazoo may miss important corner cases, as illustrated by our running example in Section 4.1.

The reason why Baseline had fewer test sequences than GUITAR and Gazoo was because, as we have mentioned, neither GUITAR nor Gazoo could guarantee their initial set of event sequences were feasible. Thus, they had to insert connecting events afterward, which means the final sequences might not be strictly bounded by the MAXLENGTH. There were also no easy fixes that could force them to strictly adhere to the bound. In contrast, the sequences generated by our method were guaranteed to be feasible and within the MAXLENGTH.

The time taken by all test generation methods are more or less the same. Since they all work on the EFG as opposed to executing the actual GUI application, the time is negligible compared to the time taken to execute the test sequence.

### 4.5.2 Comparison of Individual Reduction Techniques

In this experiment, we evaluated the effectiveness of the individual reduction techniques in our method. Table 4.3 shows our results, where Column 1 shows the name of each application, Columns 2-10 show the number of test sequences generated by each reduction technique, and the last column shows the number of test sequences generated by all reductions techniques combined. POR was used in conjunction of all the individual reduction techniques.

The results show that each technique is effective compared to the baseline with POR (denoted $+POR$) with improvement ranging from 0.2% to 60% (e.g., computed by $CauInd = 1 - 21311 / 54087 = 60.60\%$). Furthermore, when combined, they can achieve the largest reduction (72%). This not only means each reduction technique makes its own contribution, but also means they are complementary to each other. For brevity, we do not show the time taken by these individual methods but they are almost the same.
Table 4.3: Effectiveness of different reduction techniques.

<table>
<thead>
<tr>
<th>Name</th>
<th>+POR</th>
<th>Individual Reduction in Our Tool</th>
<th>AllNew</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+NoW</td>
<td>+SameW</td>
<td>+CovW</td>
</tr>
<tr>
<td>GazooV0.1</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>guess</td>
<td>40</td>
<td>40</td>
<td>34</td>
</tr>
<tr>
<td>GazooV0.2</td>
<td>29</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>ticketseller</td>
<td>418</td>
<td>418</td>
<td>311</td>
</tr>
<tr>
<td>hashcalc</td>
<td>198</td>
<td>145</td>
<td>161</td>
</tr>
<tr>
<td>workout</td>
<td>246</td>
<td>140</td>
<td>186</td>
</tr>
<tr>
<td>payment</td>
<td>214</td>
<td>116</td>
<td>153</td>
</tr>
<tr>
<td>regextester</td>
<td>334</td>
<td>153</td>
<td>235</td>
</tr>
<tr>
<td>addressbook</td>
<td>63</td>
<td>40</td>
<td>53</td>
</tr>
<tr>
<td>jnotepad</td>
<td>1,019</td>
<td>524</td>
<td>981</td>
</tr>
<tr>
<td>workout</td>
<td>140</td>
<td>69</td>
<td>101</td>
</tr>
<tr>
<td>ggp</td>
<td>38,189</td>
<td>31,296</td>
<td>36,868</td>
</tr>
<tr>
<td>calc</td>
<td>1,183</td>
<td>928</td>
<td>1,145</td>
</tr>
<tr>
<td>ce</td>
<td>5,191</td>
<td>3,664</td>
<td>4,902</td>
</tr>
<tr>
<td>terpspreads</td>
<td>4,840</td>
<td>4,498</td>
<td>4,744</td>
</tr>
<tr>
<td>rachota</td>
<td>1,646</td>
<td>897</td>
<td>1,417</td>
</tr>
<tr>
<td>buddi</td>
<td>318</td>
<td>109</td>
<td>266</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>54,087</strong></td>
<td><strong>43,085</strong></td>
<td><strong>51,605</strong></td>
</tr>
</tbody>
</table>

4.5.3 Comparison of Test Execution Results

Finally, we compare the test execution. Since running test sequences generated by all methods on all applications takes a long time, we only obtained results on four larger applications. Table 4.4 shows the results, including the name, the percentage of statements covered, and the number of test sequences. The results show all method achieved a similar coverage. The main difference is in the number of test sequences: it is 3,995 for our method, 93,446 for GUITAR, and 154,345 for Gazoo. For buddi, the reduction is 430X: it is 60 sequences for our tool compared to 28K sequences for Gazoo.

Table 4.4: Comparison of the test execution results.

<table>
<thead>
<tr>
<th>Name</th>
<th>our tool (new)</th>
<th>GUITAR [62]</th>
<th>Gazoo [9]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>coverage</td>
<td>tests</td>
<td>coverage</td>
</tr>
<tr>
<td>ce</td>
<td>33%</td>
<td>755</td>
<td>33%</td>
</tr>
<tr>
<td>terpspreads</td>
<td>56%</td>
<td>3,045</td>
<td>45%</td>
</tr>
<tr>
<td>rachota</td>
<td>64%</td>
<td>135</td>
<td>62%</td>
</tr>
<tr>
<td>buddi</td>
<td>36%</td>
<td>60</td>
<td>36%</td>
</tr>
</tbody>
</table>
4.6 Conclusion

We present an algorithm that systematically generate event sequences. Our reduction criteria are sufficient conditions that guarantee no useful event sequences are removed. Experiments show that our algorithm achieve more than 10X reduction compared to the state-of-the-art GUI testing tools.

However, we did not consider external dependencies imposed by remote network communication, database access, or the file IO. Therefore, our method may miss useful event sequences in the presence of these external dependencies. This limitation is shared by the other GUI testing tools as well. We did not consider the diversity of data input either. During our experiments, the data input was generated by GUITAR’s replayer using its default setting, to allow a fair comparison of all tools. However, it was also the reason why the testing coverage did not come close to 100%. Same as GUITAR and Gazoo, we focused on only one aspect of GUI testing, which is the diversity of event sequences. To improve further, fuzzing or symbolic execution techniques may be needed to diversify input data; we leave this for future work.
This chapter describes SQLSol, a tool that synthesizes SQL query from Input-Output (IO) examples, in details. The chapter is organized as follows: Section 5.1 uses a motivating example to demonstrate the algorithm step by step. Section 5.2 introduces the SQL subset. Section 5.3 presents the technique in details. Section 5.4 evaluates the algorithm and compare to the state-of-the-art SQL synthesizing tools.

5.1 Motivating Example

In this section, we use an example to demonstrate our algorithm. Consider the following SQL writing problem, which is taken from a classic database management textbook [65] and modified for the purpose of illustration.

*Find the name and average score of each senior student whose average score is greater than 59.*

Figure 5.1 is our hand-written solution of the problem. The SQL query first joins two tables on the columns student.id and grade.student_id. Then, it selects all senior students using the condition in the WHERE clause. Next, it computes the average score for
each student and selects those rows where the average scores are greater than 59 using
the condition in the **HAVING** clause. Finally, it projects the selected columns to the columns
name and average of the output table.

```sql
SELECT student.name AS name, average(grade.score) AS average
FROM student
JOIN grade
ON student.id = grade.s_id
WHERE student.level = 'senior'
GROUP BY student.name
HAVING average(grade.score) > 59.
```

Figure 5.1: Hand-written solution for the motivating example.

Figure 5.2 is an IO example we manually wrote for the input of the synthesis algo-
rithm. It includes two input tables, **Student** and **Grade**, and one output table, **Output**. The
column `s_id` of table **Grade** is a **foreign key** of the column `id` of table **Student**. The goal
of the SQL synthesizer is to automatically generate a SQL query which satisfies the IO
example. Ideally, it returns the exact solution, Figure 5.1, instead of other solutions like
Figure 1.2.

<table>
<thead>
<tr>
<th>Student</th>
<th>Grade</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>name</td>
<td>level</td>
</tr>
<tr>
<td>1</td>
<td>stu1</td>
<td>senior</td>
</tr>
<tr>
<td>2</td>
<td>stu2</td>
<td>senior</td>
</tr>
<tr>
<td>3</td>
<td>stu3</td>
<td>senior</td>
</tr>
<tr>
<td>4</td>
<td>stu4</td>
<td>junior</td>
</tr>
<tr>
<td>5</td>
<td>stu5</td>
<td>junior</td>
</tr>
<tr>
<td>6</td>
<td>stu6</td>
<td>senior</td>
</tr>
<tr>
<td>7</td>
<td>stu7</td>
<td>senior</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.2: Input-Output Tables for Motivation Example.

Our tool, SQLSol, joins the two input tables on the foreign key relation, and creates
a parameterized SQL query, Figure 5.4. Then, it encodes constraints for the IO example,
the domain of the unknowns, and the semantics of the SQL query, and then it sends the
constraints to a SMT solver, Z3 we used, to check satisfiability. After the SMT solver
returns satisfiable, it fetches the models of the unknowns from the solver and compiles a
SQL query, Figure 5.1. The user accepts the solution, and the algorithm terminates.
5.1.1 Join Conditions

For the purpose of simplifying the synthesis algorithm, without loss of information, we join the input tables using two heuristic rules same as SQLSynthesizer. The first rule is to match the column names and types; the second rule is to compare the constants in the columns. Figure 5.3 shows the IO example after the input tables are joined. The join condition is `student.id = grade.s_id`, which is a foreign key constraint of the schema of the two input tables.

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>level</th>
<th>course</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>stu1</td>
<td>senior</td>
<td>Math</td>
<td>70</td>
</tr>
<tr>
<td>1</td>
<td>stu1</td>
<td>senior</td>
<td>English</td>
<td>80</td>
</tr>
<tr>
<td>2</td>
<td>stu2</td>
<td>senior</td>
<td>Math</td>
<td>99</td>
</tr>
<tr>
<td>3</td>
<td>stu3</td>
<td>senior</td>
<td>English</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>stu4</td>
<td>junior</td>
<td>Math</td>
<td>70</td>
</tr>
<tr>
<td>5</td>
<td>stu5</td>
<td>junior</td>
<td>English</td>
<td>85</td>
</tr>
<tr>
<td>6</td>
<td>stu6</td>
<td>senior</td>
<td>English</td>
<td>60</td>
</tr>
<tr>
<td>7</td>
<td>stu7</td>
<td>senior</td>
<td>Computer</td>
<td>90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>stu1</td>
<td>75</td>
</tr>
<tr>
<td>stu6</td>
<td>60</td>
</tr>
<tr>
<td>stu7</td>
<td>90</td>
</tr>
</tbody>
</table>

Figure 5.3: Input-Output Tables After Join in Motivation Example.

5.1.2 Parameterized SQL query

We create a parameterized SQL query, Figure 5.4, for the motivating example. A parameterized SQL query is a SQL query skeleton that has unknowns regarding to the IO table schema and constants et al. The unknowns can be represented by uninterpreted variables when encoded into logic constraints.

In Figure 5.4, there are two unknowns, $??_{1s}$ and $??_{2s}$, for there are two columns in table Output. The tuple $(??_{wpop}, ??_{wpc}, ??_{wpv})$ is the predicate in the WHERE clause, and the unknowns stand for the logic comparator, column name in table Input and a constant. The tuple $(??_{hpop}, ??_{hpc}, ??_{hpv})$ is the predicate in the HAVING clause, and the unknowns stand for the logic comparator, column name of the aggregation column and a constant. We set the upper bound of length of the predicates in both WHERE and HAVING clause to be 1. Note that the constant $??_{hpv}$ is a constant in the aggregation column, and has to be computed.
before it can be used.

```
SELECT ??s AS name, ??s AS average FROM input
WHERE (??wpop ??wpc ??wpv) GROUP BY ??gc HAVING (??hpop ??hpc ??hpv)
```

Figure 5.4: Parameterized SQL Query for Motivation Example.

### 5.1.3 Auxiliary Columns

We add an auxiliary aggregation column, to encode the SQL query with an aggregation column. An aggregation column depends on three variables: the aggregation function ??af, the aggregation column ??ac, and the group column ??gc, which are all unknowns in our synthesis algorithm. The aggregation column can be computed with the unknowns in the SMT language, as presented in details in section 5.3.4.

### 5.1.4 Logic Constraints for SQL Synthesis Problem

The logic constraints for SQL synthesis problem fall into three categories: IO constraints, domain constrains, and semantics constraints. IO constraints encode the IO examples into logic constraints. However, because there is no Table data structure and no polymorphism support in SMT language, we define an innovative Table and Cell data structure, to effectively implement the constraints of IO examples in SMT language.

Domain constraints are the domains of the unknowns. In Figure 5.4, the unknowns ??s in the SELECT clause are column names or aggregation column names of table input. The unknown ??af is an aggregation function. The unknowns ??ac, ??wpc, ??hpc are column names. The unknowns ??wpop and ??hpop are logic comparators. The unknowns ??wpv and ??hpv are constants in the columns of ??wpc and ??hpc.

Let input.cols be the column names of table input, input.acols be the aggregation column names acol, afs be the set of aggregation functions, ops be the set of logic com-
The domain constraints for the unknowns are

\[ s_1, s_2 \in input.cols \cup input.acols \]
\[ a_{fs} \in afs \]
\[ a_{ac}, w_{pc}, g_{c} \in input.cols \]
\[ h_{pc} \in input.acols \]
\[ w_{pop}, h_{pop} \in ops \]
\[ w_{pv}, h_{pv} \in input.w_{pc} \]
\[ h_{pv} \in input.h_{pc} \]

The semantics constraints encode the semantics of a SQL query with respect to the IO example. For each row \( r \) in the table \( input \), the SQL query first checks the predicates in the \textit{WHERE} and \textit{HAVING} clause. If they are both true, the columns of the row in the \textit{SELECT} clause will form a row \( s \) in the table \( output \). Otherwise the row will be skipped. In the other direction, for a row \( s \) in the table \( output \), there exists a row in the table \( input \) that satisfies the predicates in the \textit{WHERE} and \textit{HAVING} clause. The semantic constraints for the
parameterized SQL query are

\[\forall 1 \leq r \leq 8, (??_{wpop} ??_{wpc} ??_{wpv}) \land (??_{hpop} ??_{hpc} ??_{hpv})\]

\[\implies \exists 1 \leq s \leq 3, (\text{input}(??_{c1}, r) = \text{output}(\text{name}, s))\]

\[\land (\text{input}(??_{c2}, r) = \text{output}(\text{average}, s))\]

\[\forall 1 \leq s \leq 3 \implies \exists 1 \leq r \leq 8, (??_{wpop} ??_{wpc} ??_{wpv}) \land (??_{hpop} ??_{hpc} ??_{hpv})\]

\[(\text{input}(??_{c1}, r) = \text{output}(\text{name}, s))\]

\[(\text{input}(??_{c1}, r) = \text{output}(\text{name}, s))\]

5.1.5 Solve

We sent the constraints above to an SMT solver, Z3, to check for satisfiability, and the solver returned satisfiable. Then, we fetched the model from the solver and substituted the unknowns in the abstract query with the model, and a concrete SQL query was generated. We manually checked and accepted the solution, since it is exactly the one that solves the question of this motivating example. Then the algorithm terminated. The computation took less than 1 second.

5.2 SQL Subset Syntax

Figure 5.5 shows the syntax of the standard SQL subset supported by our tool SQLSol, which is similar to the state-of-the-art tool, SQLSynthesizer [91]. This subset is designed to support the most widely used SQL features according to a survey by Zhang et al [91].

While mostly same, there are two differences between SQLSol and SQLSynthesizer. One difference is that the logic constraints in SQLSol cannot support DISTINCT and ORDER BY. That is because SQLSol models tables as sets, which is common in formal methods. For example, relational algebra is defined based on sets [1]; previous work, [78], models
Figure 5.5: Syntax of the supported SQL subset in SQLSol: atom is a table name, or column name, or a cell value.

...tables as sets. Fortunately, because both DISTINCT and ORDER BY are of arity 0, synthesis of them only needs to compare rows of the result table and the output table in the IO example, therefore can be done in a post-process step.

Another difference is that, in addition to the three logic comparators =, >, < supported by SQLSynthesizer, SQLSol also supports ≠, ≥, ≤.

5.3 **Technique**

In this section, we present the technique details of our algorithm.
5.3.1 Overview

Algorithm 4 is the high-level algorithm of SQLSol. Line 2 joins all input tables into one input table. Line 3 declares all unknowns in the parameterized SQL query as uninterpreted functions in SMT language. Line 4-6 add the domain constraints for all unknowns. Line 7-22 are the iteration process that adds one new example each time. For every example, line 8-9 encode it with our new Table datatype in SMT language; line 10 encodes the semantics of parameterized SQL query into constraints with respect to the input-output tables. In line 10-22, we send all constraints to the SMT solver to check for satisfiability. If the solver returns satisfiable, we fetch the model and compile a concrete SQL query, which satisfies the IO examples. Then, instead of stopping here, we let the user decide whether to accept the query or not: if yes, the algorithm returns with success; otherwise, the user can add a new IO example, and the loop continues. If the solver returns unsatisfiable, the algorithm declares failure.

Algorithm 4 SQLSol SQL Synthesizer.

1: Let \((I_1, O_1)\) be the Input Output example, \(X = \emptyset\) be the set of unknowns, \(B = \emptyset\) be the set of constraints, \(S\) be the SMT solver
2: \(I = \text{join}(I_1), O = O_1\)
3: \(X = \text{makeUnknowns}(I, O)\)
4: for \(x \in X\) do
5: \(B.add(\text{encodeDomain}(I, O, x))\)
6: end for
7: while True do
8: \(B.add(\text{encodeTable}(I))\)
9: \(B.add(\text{encodeTable}(O))\)
10: \(B.add(\text{encodeSemantics}(I, O, X))\)
11: if \(S.solve(B) == SAT\) then
12: \(m = S\\text{.model}()\)
13: if \(m.\text{accepted}()\) then
14: return // solution
15: else
16: \((I, O) = \text{addExample}()\)
17: \(I = \text{join}(I)\)
18: continue
19: end if
20: else
21: return // no solution
22: end if
23: end while
5.3.2 New SMT Datatype for Table Encoding and IO constraints

Satisfiability modulo theories (SMT) solve decision problems using background theories expressed in logic constraints. Though modern SMT solvers have theories of various data structures such as List, Array, Bit Vector, they do not have theories for Table. Veanes et al, [78], proposed a theory which uses a list of tuples to model Table. The unknowns in this theory are table cells, therefore the theory is able to synthesize input tables given a SQL query. However, because our algorithm synthesizes a SQL query given IO examples and the unknowns are elements in the SQL query, we can not simply use the theory.

We use a new Table SMT datatype to model input-output tables. The table datatype is a customized 2-dimension Array. Because we handle different types of table cells, including Int, Float, String, and Null, but SMT language does not support polymorphism, we work around by using the datatype feature of the latest SMT-lib standard [15]. We define a new datatype Cell and define the Table datatype as Array(String Int Cell), where the elements are the column name, row index, and cell value of the table, respectively.

With our new Table datatype, we can efficiently encode the IO constraints of the input-output tables in SMT language. The IO constraints of each table is the union of the constraints of all table cells. An example use of the table datatype is (assert (= (select table col 1) (String John))), which asserts that the cell value of table at column col and row 1 equals to John of String type.

5.3.3 Parameterize SQL Query and Domain Constraints

A parameterized SQL Query is a SQL query that has unknowns in the SELECT, WHERE, GROUP BY, HAVING clauses. In this section, we define the unknowns and their domain constraints.
The unknowns in the SELECT clause define the original or aggregated column names in the input table that are projected to the output table. The number of unknowns in the SELECT clause equal to the number of columns in the output table. Each unknown is of String type. The domain of the unknowns is the set of original and aggregated column names.

Let \( \text{input.cols} \) and \( \text{input.acols} \) be the set of original and aggregated column names of table \( \text{input} \). Let \( \text{noc} \) be the number of columns of table \( \text{output} \). Let \( ?i \) be the \( i - \text{th} \) unknown in the SELECT clause. The domain constraints of an unknown in the SELECT clause is the union of \( \text{input.cols} \) and \( \text{input.acols} \).

\[
?i \in \text{input.cols} \cup \text{input.acols}, \ i = 1, \cdots, \text{noc}
\] (5.1)

The predicate in the WHERE clause is a boolean expression which decides whether a row in the table \( \text{input} \) will be selected to the table \( \text{output} \). The boolean expression is comparison patterns connected by logic conjunctive connector \( \land \) or logic disjunctive connector \( \lor \). A comparison pattern is a basic pattern or logic constant \( \text{true} \) or \( \text{false} \).

Let \( \text{nwp} \) be the upper bound of the number of basic comparison patterns. Let \( ?wpop \) be the unknowns for logic comparators of Enumeration type, \( ?wpc \) be the unknowns for column names of String type, \( ?wpv \) be the unknowns for values of the columns of Cell type, \( ?wpm \) be unknowns of Boolean type, \( ?wpb \) be unknowns of Enumeration type \( \{\land, \lor\} \).

A basic comparison pattern is of form \( \text{bcp} = (??wpop ??wpc ??wpv) \), whose semantics is executing the comparator \( ?wpop \) with operands \( ?wpc \) and \( ?wpv \). A comparison pattern is defined as \( \text{cp} = (??m \text{ bcp}) \), whose semantics is when \( ??m \) is \( \text{true} \), it evaluates to \( \text{true} \), otherwise \( \text{bcp} \). The predicate \( \text{wp} \) is defined in such a way that it covers basic predicates
of length from 0 to \( nwp \). The definition of \( wp \) is \( wp = wp^{nwp} \), where \( wp^i \) is defined recursively:

\[
wp^i = \begin{cases} 
  \text{true} & i = 0 \\
  cp^1 & i = 1 \\
  (??^i_{wpb} cp^i wp^{i-1}) & i = 2, \ldots, nwp
\end{cases}
\]

Let \( ops \) be the set of logic comparators, \( input.??^i_{wpc} \) be the cell values of column \( ??^i_{wpc} \), \( cs \) be the set of logic connectors \( \{\land, \lor\} \). The domain constraints in the \( \text{WHERE} \) clause are

\[
??^i_{wpop} \in ops \\
??^i_{wpc} \in input.cols \\
??^i_{wpb} \in input.??^i_{wpc}, i = 1, \ldots, nwp \\
??^j_{wpb} \in cs, j = 2, \ldots, nwp
\]  \( (5.2) \)

**parameterize GROUP BY Clause.**

The unknowns in the \( \text{GROUP BY} \) clause define the columns by which the output table is grouped. In addition to original columns of the input table, we added special columns to the input table to support special groups. Particularly, we added two special group-by columns: \( \text{ucol} \) and \( \text{scol} \). The column \( \text{ucol} \) has values that are all unique. Grouping by \( \text{ucol} \) means every row is one group. Therefore, queries with \( \text{GROUP BY} \) clause generalize to queries without \( \text{GROUP BY} \) clause. The column \( \text{scol} \) has values that are all equal. Grouping by \( \text{scol} \) means all rows is one group.

Let \( ng \) be the upper bound of the number of group-by columns, \( input.cols \) and \( input.hcols \) be the original columns and added group-by columns. Let \( ??^i_{gc} \) be the unknown in the \( \text{GROUP BY} \) clause. The domain constraints for the \( \text{GROUP BY} \) clause are

\[
??^i_{gc} \in input.cols \cup \{input.hcols\}, i = 1, \ldots, ng
\]  \( (5.3) \)
parameterize HAVING Clause.

The predicate in the HAVING clause is the same as the predicate in the WHERE clause except that it only applies to the aggregation column, i.e., input.acols, while the predicate in the WHERE clause only applies to the original column, i.e., input.cols.

Let \( n_{hp} \) be the upper bound of the number of basic comparison pattern. Let \( ?^i_{hpop} \) be the unknowns for logic comparators of Enumeration type, \( ?^i_{hpc} \) be the unknowns for column names of String type, \( ?^i_{hpv} \) be the unknowns for values of the columns of Cell type, \( ?^i_{hpm} \) be unknowns of Boolean type, \( ?^i_{hpb} \) be unknowns of Enumeration type \( \{\wedge, \vee\} \).

Let a basic comparison pattern be \( bcp = (??^i_{hpop} ??^i_{hpc} ??^i_{hpv}) \), a comparison pattern be \( cp = (?^m_{hp}, bcp) \). The definition of predicate \( hp \) is \( hp = hp^{n_{hp}} \), where \( hp^i \) is defined recursively:

\[
hp^i = \begin{cases} 
\text{true} & i = 0 \\
cp^1 & i = 1 \\
??^i_{hp} cp^i hp^{i-1} & i = 2, \ldots, n_{hp}
\end{cases}
\]  
(5.4)

The domain constraints in the HAVING clause are

\[
??^i_{hpop} \in ops \\
??^i_{hpc} \in input.cols \\
??^i_{hpv} \in input, ??^i_{hpc}, i = 1, \ldots, n_{hp} \\
??^i_{hpb} \in cs, j = 2, \ldots, n_{hp}
\]  
(5.5)

5.3.4 Compute Aggregation Columns

Aggregation columns are computed before semantics constraints are encoded. Unlike explicit search algorithms enumerate all combinations of group-by columns and aggregation columns, our algorithm computes the aggregation columns only once with the unknowns defined before.
Let \( r \) be a row index, \( n \) be the number of rows, \( \text{ite} \) be the if-then-else struct, \( wp \) be the predicate in the \text{WHERE} clause, \( col \) be the columns in the \text{GROUP BY} clause, \( col(r) \) be the cell value of column \( col \) at row \( r \).

**Compute Aggregation COUNT.**

Let \( count \) be the aggregation column for aggregation function \text{COUNT}. The formula to compute \( count \) is

\[
\text{count}(r) = \sum_{1 \leq i \leq n} \text{ite}(wp(r) \wedge (col(i) = col(r)), 1, 0)
\]  
(5.6)

**Compute Aggregation SUM.**

Let \( acol \) be the column to apply the aggregation function \text{SUM} on, \( sum \) be the aggregation column for aggregation function \text{SUM}. The formula to compute \( sum \) is

\[
\text{sum}(r) = \sum_{1 \leq i \leq n} \text{ite}(wp(r) \wedge (col(i) = col(r)), acol(r), 0)
\]  
(5.7)

The aggregation column \text{AVERAGE} is computed as the division of \text{SUM} and \text{COUNT}.

**Compute Aggregation MAX.**

Let \( acol \) be the column to apply the aggregation function \text{MAX} on, \( max \) be the aggregation column for aggregation function \text{MAX}. The formula to compute the aggregation column \( max \) is recursive:

\[
\text{max}(r) = \text{ite}((wp(r) \wedge (acol(r) > max(r - 1))), \\
acol(r), \text{max}(r - 1))
\]  
(5.8)

**Compute Aggregation MIN.**

Let \( acol \) be the column to apply the aggregation function \text{MIN} on, \( min \) be the aggregation column for aggregation function \text{MIN}. The formula to compute the aggregation column
is recursive:

\[
min(r) = \text{ite}((wp(r) \land (acol(r) > min(r - 1))),
acol(r), min(r - 1))
\]  

(5.9)

5.3.5 Encode Semantics Constraints

In this section, we introduce the axiom that models the semantics of SQL queries in logic language. On one hand, the axiom considers the direction from input to output. For each row in the table \text{input}, if it satisfies the predicate \(wp\) in the \text{WHERE} clause and the predicate \(hp\) in the \text{HAVING} clause, it should be selected into table \text{output}, i.e., there exists a row in the table \text{output} that contains the selected columns. On the other hand, the axiom considers the direction from output to input. For each row in the table \text{output}, there exists a row in the table \text{input} which satisfies the predicate \(wp\) and \(hp\) and the selected column equals to the column in the output row.

Let \(nrow\) be the number of rows of a table, let \(wp\) be the predicates in the \text{WHERE} clause, let \(hp\) be the predicates in the \text{HAVING} clause, let \(PC\) be set of the selected columns. The axiom for the semantics of the SQL query is:

\[
\forall r \in \text{input.nrow}, wp(r) \land hp(r) \implies \\
\exists s \in \text{output.nrow}, \forall c \in PC, \text{input}(c, r) = \text{output}(c, s) \\
\forall s \in \text{output.nrow} \implies \\
\exists r \in \text{input.nrow}, \forall c \in PC, wp(r) \land hp(r) \land \text{input}(c, r) = \text{output}(c, s)
\]  

(5.10)

5.4 Evaluation

We implemented our algorithm, SQLSol, in Java. We use Z3, [24], as the backend SMT solver. In this section, We present our evaluation of SQLSol.
We set the upper bound of GROUP BY columns be 1, the upper bound of aggregation column be 1, the upper bound of basic predicates in the WHERE clause be 1, 2, or 3. the upper bound of basic predicates in the HAVING clause be 1. The evaluation was conducted on a quad-core Intel Core i7 3.3 GHz CPU with 8 GB memory.

5.4.1 Experiments on SQLSynthesizer and Scythe Benchmarks

We evaluated SQLSol on two open-source benchmark sets: SQLSynthesizer benchmark set and Scythe benchmark set. The benchmarks were downloaded from the site [2] of the open-source project Scythe. All benchmarks in both benchmark sets have one IO example. The SQLSynthesizer benchmark set contains 28 benchmarks, including 23 benchmarks collected from the classic database textbook [65], and 5 ones collected from forums. The average number of table cells of an IO example in SQLSynthesizer benchmark set is 57. The Scythe benchmark set has three folders: dev-set, top-rated, recent-posts. We combined all the benchmarks in the three folders, and removed some empty files. The final benchmark set has 143 benchmarks. The average number of table cells in one benchmark in the final set is 29. We downloaded Scythe from its github page [2]. However, we could not obtain an effectively working software of SQLSynthesizer, so the data of SQLSynthesizer was from its paper [91].

Table 5.1 shows the result of evaluation. On the SQLSynthesizer benchmark set, SQLSol solved up to 5, 18%, more problems than SQLSynthesizer and Scythe, while the average time usage is comparable to SQLSynthesizer, but smaller than Scythe. On the Scythe benchmark set, for the solved benchmarks SQLSol is over 2X faster than Scythe; for the unsolved case, SQLSol is 44X faster than Scythe. Overall, SQLSol is over 8X times faster than Scythe.

Note that on Scythe benchmark set, the number of problems solved by SQLSol is less than by Scythe. We manually checked the unsolved problems, and found that all are not in our SQL subset, mostly are nested queries. We leave nested queries support for future
work.

### 5.4.2 Scalability Comparison of SQLSol and Scythe

We created 25 benchmarks, each of which is an IO example to solve the problem in the motivating example in section 5.1. The number of rows in the input tables in the benchmarks increases from 5 to 30, and each row contains 5 constants. Constants are unique except those in the first 5 rows. The output tables in all benchmarks are the same, containing 3 rows.

We tested Scythe with two settings. One is Scythe with all constraints provided: constants senior, 59, and the aggregation function average. The other is Scythe with no constraints provided. The timeout is set to be 300 seconds. In SQLSol, the \( w_p \) is set to be 2.

Figure 5.6 plots the result. The \( x \)-axis is the number of rows in the input table, and the \( y \)-axis is the time usage. We can see that Scythe without constants cannot solve any problem before timeout. SQLSol performs better than Scythe at every input size and the speedup is between 2X to 10X.
5.4.3 Benefits from Multiple IO Examples Support

Given an IO example, there are many SQL queries that satisfy them, but only one is the user intention. In this section, we first evaluate how many IO examples are needed to find the user intention. We picked 6 benchmarks, whose solutions fall into our SQL subset, from the SQLSynthesizer benchmark set and manually wrote solution for them. The numbers of IO examples that SQLSol took to synthesize the solution are: 5, 4, 1, 1, 1, 1.

Another benefit from multiple IO examples support is that multiple examples contain less constants than single example, therefore can speed up the algorithm. For example, consider the synthesis problem: `SELECT id, name FROM input where id = 2`. Figure 5.7 is the input table of one IO example, and Figure 5.8 is two IO examples derived from Figure 9. Although the output of SQLSol are the same, but the number of constants in Figure 5.7 and Figure 5.8 are different: 7 in single input, 4 in multiple inputs. We ran SQLSol with the two cases and fetched the statistics data from Z3 solver. The result, Figure 11, shows that in the number of added equations, the number of decisions, the memory used (in MB), the time used (in mini-seconds), multiple IO examples require less and perform better.
5.5 Conclusion

We present an algorithm, SQLSol, which encodes the semantics of a SQL query into logic constraints, and leverages SMT solvers to synthesize SQL queries from IO examples. The evaluation shows that SQLSol outperforms the state-of-the-art tools, SQLSynthesizer and Scythe. Furthermore, by supporting multiple examples, SQLSol is able to find the user-intended solution and improve the speed at the same time.
Chapter 6

Synthesizing Programs from Flowchart Image Using Deep Convolutional Neural Networks

This chapter describes GRCNN, a tool that synthesizes program from the image of its flow chart, in details. The chapter is organized as follows: Section 6.1 describes the architecture and training details of GRCNN. Section 6.2 describes the experimental evaluation of GRCNN.

6.1 Network

In this section, we describe the overall network of GRCNN and its subnets. Then, we describe the loss function and training details.

The input of GRCNN is the image of a flow chart. A flow chart is a graph diagram that represents the work flow of a program [70]. Standard flow charts have several shapes for nodes. In this work, we consider all shapes of nodes are rectangles. Because we use heuristic algorithms to generate ground truth data for bounding boxes, designing
and implementing algorithms for every shape requires considerable engineering effort. However, given ground truth data for bounding boxes, our algorithm can be trained the same way to handle other shapes.

The input image of the flow chart is resized to a fixed size (400 pixels for height and 200 pixels for width) before being fed to the neural network. If both height and width is less then the fixed size, we pad the image by zeroes. Otherwise, we interpolate the image to the fixed size. The output of GRCNN is a graph representation of the flow chart, including an adjacent matrix for the edge representation and a list of text for content in nodes. In addition to the original text, we insert an id to the text to match the text with the $id$-th row of adjacent matrix. In this project, we use an alphabet of 50 characters including English characters in lower case, digit characters, arithmetic operators and other symbols. The source code that our algorithm produces supports sequential statements and control structure including IF-ELSE, WHILE loop, DO-WHILE loop.

Figure 6.1 is the overview of GRCNN. GRCNN has four parts: backbone network, edge detection network, node detection network, and node recognition network. The backbone network takes as input the raw image of the flow chart and outputs a feature vector. The feature vector is fed to both the edge detection to produce the adjacent matrix of the flow chart, and the node detection network to generate the bounding boxes of each node, which is used to crop the node from the original image. Then, the crop of each node is fed to the node recognition network to generate the text in the node.

### 6.1.1 Backbone Network

The backbone network is a deep CNN that takes as input an image of size $C_{in} \times H \times W$. The output is a rich feature vector, which is later used as input to the edge network and the node detection network. The backbone network is a sequence of four basic blocks. Table 6.1 shows the architecture of a basic block, which takes as input a feature vector of $C_{in}$ channels and outputs a feature vector of $C_{out}$ channels and the same height and
width. Each convolutional layer in the basic block is followed by a batch normalization layer and a ReLU activation layer. Each basic block is followed by a max-pooling layer, except the third one. During training, a dropout layer (p=0.1) is added after each pooling layer.

Table 6.1: Basic block architecture.

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$C_{in} \times H \times W$</td>
</tr>
<tr>
<td>Conv2D</td>
<td>$C_{out}/4, 3 \times 3$</td>
</tr>
<tr>
<td>Conv2D</td>
<td>$C_{out}/4, 3 \times 3$</td>
</tr>
<tr>
<td>Conv2D</td>
<td>$C_{out}/4, 3 \times 3$</td>
</tr>
<tr>
<td>Conv2D</td>
<td>$C_{out}, 3 \times 3$</td>
</tr>
</tbody>
</table>

Table 6.2: Backbone network architecture.

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$C \times H \times W$</td>
</tr>
<tr>
<td>Basic Block</td>
<td>$15, 3 \times 3$</td>
</tr>
<tr>
<td>Max Pool</td>
<td>$2 \times 2$</td>
</tr>
<tr>
<td>Basic Block</td>
<td>$50, 3 \times 3$</td>
</tr>
<tr>
<td>Max Pool</td>
<td>$2 \times 2$</td>
</tr>
<tr>
<td>Basic Block</td>
<td>$200, 3 \times 3$</td>
</tr>
<tr>
<td>Basic Block</td>
<td>$400, 3 \times 3$</td>
</tr>
<tr>
<td>Max Pool</td>
<td>$2 \times 2$</td>
</tr>
</tbody>
</table>

Optionally, the backbone network can be made a residual learning network (ResNet), [42], by modifying the basic block as follows. We perform a down-sampling on the input
vector with $1 \times 1$ convolutional layer to $C_{out}$ channels, and add the result to the original output as new output.

### 6.1.2 Edge Network

The edge network takes as input the feature vector generated by the backbone network and outputs an adjacent matrix, which is the edge representation of the flow chart. The element of the adjacent matrix at row $i$ and column $j$ has three possible values: 0, 1, 2, which encode no edge, normal edge or YES branch of a decision node, NO branch of a decision node from node $i$ to node $j$, respectively. Because the number of nodes in the flow chart may vary, we pad the adjacent matrix to a fixed size $PAD$ by zeroes. We set $PAD = 6$ in this work. We encode the three values in the adjacent matrix with one-hot-vectors of length 3. Therefore, the edge network outputs a vector of $PAD \times PAD \times 3$ scores.

Table 6.3 shows the architecture of the edge network. It first performs a convolutional layer activated by a ReLU function. Then, a max pooling layer is performed followed by two linear layers activated by Tanh function. We observed the training converges significantly faster when using Tanh activation function in the linear layers than using other activation functions.

The loss function to train the edge network is the multi-class multi-classification hinge loss. Equation 6.1 is the formula of the loss function, where $x$ is the input vector and $y$ is the target class indices.

$$l_e(x, y) = \sum_{i,j} \max(0, 1 - \frac{x[y[j]] - x[i])}{x.size(0)})$$

(6.1)
Table 6.3: Edge network architecture.

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv2D</td>
<td>400, 3 × 3</td>
</tr>
<tr>
<td>ReLu</td>
<td></td>
</tr>
<tr>
<td>Max Pool</td>
<td>3 × 3</td>
</tr>
<tr>
<td>Linear</td>
<td>400 × 16 × 7 × 400</td>
</tr>
<tr>
<td>Tanh</td>
<td></td>
</tr>
<tr>
<td>Linear</td>
<td>400 × 6 × 6 × 3</td>
</tr>
<tr>
<td>Tanh</td>
<td></td>
</tr>
</tbody>
</table>

6.1.3 Node Detection Network

The node detection network takes as input the convolutional feature vector from the backbone network and outputs a set of rectangular node proposals, each with an objectness score.

The network slides a small network over the input convolutional layer. The output of the window is fed into two sibling layers: a box-regression layer and a box-classification layer. Because the small networks work in the sliding window fashion, they are naturally implemented with convolutional networks.

At each sliding window, we predict one region proposal which encodes the four coordinates of a box and one score which estimates the probability of whether the proposal is a node or not. Each region proposal is parameterized relative to a reference box, called anchor. We parameterize the coordinates of the bounding boxes following [33]:

\[
t_x = (x - x_a) / w_a, \quad t_y = (y - y_a) / h_a,
\]

\[
t_w = \log(w / w_a), \quad t_h = \log(h / h_a)
\]

\[
t_x^* = (x^* - x_a) / w_a, \quad t_y^* = (y^* - y_a) / h_a
\]

\[
t_w^* = \log(w^* / w_a), \quad t_h^* = \log(h^* / h_a),
\]

where \(x, y, w,\) and \(h\) is the box’s center point and width and height. Variables \(x, x_a\) and
$x^*$ are for the predicted box, anchor box, and ground-truth box respectively (likewise for $y, w, h$). This parameterization converts large integers of bounding box coordinates to variables close to interval $[-1, 1]$, and therefore improves the numerical performance of the algorithm.

To train the node detection network, we assign a binary class label of being a node or not to each anchor. We assign a positive number to two kinds of anchors: (1) The anchor with the highest Intersubsection-over-Union (IoU) with a ground-truth box, or (2) an anchor that has an IoU higher than 0.9 with any ground-truth box. We assign a negative number to an anchor if its IoU is less than 0.3 for all ground-truth boxes. An anchor that is neither positive nor negative does not contribute to the train objective. Because the number of positive labels and negative anchors may be different and therefore the train may be biased toward one direction, we sample from the more to ensure the same size of positive and negative labels during training.

We apply binary cross entropy loss to the objectness, and smooth-l1 loss to the region proposal. The final loss is the sum of the objectness loss and region proposal loss over all anchors.

For prediction, we choose top 50 anchors with the highest scores, and group them by the condition that anchors with IoU less than 0.5 are in the same group. Then, we choose from each group the highest score as the final prediction.

Nevertheless, our network differs from the faster RCNN in two ways. (1) Because the nodes in flow charts have similar size and shape, we use one anchor, instead many anchors with different size and ratio, to save computing power. (2) Because the nodes in flow charts do not overlap, we consider proposals in the same group if one has IoU over a threshold with any other one in the group. Meanwhile, fast RCNN considers proposals in the same group if one has IoU over a threshold with the one of highest score.

The architecture of the node detection network is as follows. The intermediate layer is a Conv2D layer with 3 X 3 kernel and 400 output channels. The classifier is a Conv2D
layer with $3 \times 3$ kernel and 1 output channel. The regressor is a Conv2D layer with $3 \times 3$ kernel and 4 output channels. Each Conv2D layer is activated by ReLU function.

6.1.4 Node Recognition Network

The node recognition network takes as input the crop of each node and outputs a vector of size $LENVOC \times 1 \times W$, where $LENVOC$ is the vocabulary size.

Table 6.4 is the architecture of node recognition network, where the base block shares the same architecture as the basic block in the backbone network, Table 6.1,

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$C \times H \times W$</td>
</tr>
<tr>
<td>Basic Block</td>
<td>$64, 3 \times 3$</td>
</tr>
<tr>
<td>Max Pool</td>
<td>$3 \times 3$</td>
</tr>
<tr>
<td>Basic Block</td>
<td>$128, 3 \times 3$</td>
</tr>
<tr>
<td>Max Pool</td>
<td>$3 \times 3$</td>
</tr>
<tr>
<td>Basic Block</td>
<td>$256, 3 \times 3$</td>
</tr>
<tr>
<td>Max Pool</td>
<td>$3 \times 3$</td>
</tr>
<tr>
<td>Basic Block</td>
<td>$LENVOC, 3 \times 3$</td>
</tr>
</tbody>
</table>

The output of the node recognition network can be interpreted as probability distribution over the vocabulary at each position along the width direction. CTC loss is a function for training sequences problems such as handwriting recognition or speech recognition. CTC loss does not attempt to learn the character boundaries, and can be applied if the input is a sequence with some order. By adoption CTC loss, we do not need to use another recurrent neural network to predict the text. Instead, we directly predict from the output of the node recognition network.

When prediction, we use greedy search to decode text from the output vector.

Optionally, a spatial transformer network (STN), [45], can be inserted before the node recognition network to further adjust the boundary of the crop. STN is the network that can be inserted into existing convolutional architectures, giving neural networks the
ability to actively spatially transform feature maps.

### 6.1.5 Train and Implementation

The whole network is trained end-to-end. The lost function is the sum of loss of all sub-networks:

\[
loss = loss_{\text{edge}} + loss_{\text{nec}} + loss_{\text{ndr}} + \sum_{\text{node}} loss_{\text{nr}},
\]

(6.2)

where \(loss_{\text{edge}}\), \(loss_{\text{nec}}\), \(loss_{\text{ndr}}\) and \(loss_{\text{nr}}\) is the loss of edge network, classifier and regressor of node detection network and node recognition network, respectively.

We use SGD method to train the network. Learning rate is 0.02, and halved when the error plateaus. The total epoch is 200.

### 6.2 Experiment

We designed experiments to answer the following research questions: (1) what is the accuracy of GRCNN? (2) what is the inference performance of GRCNN? (3) Is GRCNNable to synthesize real-world program?

The experiments are conducted on a desktop with Geforce 1070 GPU, Intel i7 CPU, and 16GB memory. GRCNN is implemented with PyTorch.

We tested the following networks.

- GRCNN: which is our basic network
- Separated GRCNN: not sharing the backbone network. See Section 6.2.4.
- GRCNN with STN: add option STN network. See Section 6.1.4.
- GRCNN with ResNet and STN.
6.2.1 Dataset Generation

We created a synthetic dataset to train and test the GRCNN, since there are no existing datasets we can use.

A data sample includes a flow chart image in PNG format and a text file containing the ground truth which is used to train the network. The ground truth includes the adjacent matrix which represents the edge information, and the bounding box and text content for each node.

The dataset contains flow charts with 3 to 6 nodes, and 0 to 2 decisions. The text in each node contains 3 to 9 random characters from an alphabet of 50 characters, including the lower case English characters, digit characters, arithmetic operators, parenthesis et al. Note that the alphabet can be chosen freely without changing the essential network architecture. The train and test dataset contains 9960 and 2490 data samples, respectively.

We draw the flow chart using Graphviz, a popular graph-drawing tool. We set the maximal width to be 200 pixels and maximal height to be 400 pixels.

When drawing the flow chart, we randomize to cover wide range of data samples. The width of lines, including nodes boundaries and edges, is randomly chosen between 1 to 5 pixels. The font size of characters is randomly chosen between 20 to 30. The font color of characters is random RGB color. We draw nodes in rectangles and design a heuristical algorithm to effectively compute high accuracy coordinates of bounding boxes.

6.2.2 Accuracy

We measure the Edge Accuracy, Sequence Accuracy, Nodes accuracy and Graph Accuracy of GRCNN prediction. Edge accuracy is the percentage of images whose edge is correctly predicted. We say the edge is correctly predicted if the predicted adjacent matrix is exactly the same as the ground truth. Sequence accuracy is the percentage of nodes whose text content is correctly predicted among all nodes in all images. This item shows how well the
node recognition network works for individual nodes. Nodes accuracy is the percentage of images whose nodes are all correctly predicted. This item shows how well the tool predicts the nodes as a whole. Graph accuracy is the percentage of images whose edge and nodes are all correctly predicted.

The accuracy of GRCNN and its subnets are described in the first row of Table 6.5. The Edge, Sequence, Nodes and Graph accuracy is 94.1%, 90.6%, 67.9% and 66.4%, respectively. Note that the Graph accuracy is the result of joint probability of all edges and all nodes, so it is lower than Edge and Sequence accuracy.

<table>
<thead>
<tr>
<th></th>
<th>Edge</th>
<th>Sequence</th>
<th>Nodes</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRCNN</td>
<td>94.1</td>
<td>90.6</td>
<td>67.9</td>
<td>66.4</td>
</tr>
<tr>
<td>+STN</td>
<td>90.0</td>
<td>93.8</td>
<td>71.3</td>
<td>64.4</td>
</tr>
<tr>
<td>+ResNet</td>
<td>91.9</td>
<td>91.3</td>
<td>64.8</td>
<td>60.3</td>
</tr>
<tr>
<td>+STN+ResNet</td>
<td>90.9</td>
<td>93.2</td>
<td>70.0</td>
<td>63.2</td>
</tr>
<tr>
<td>Separated</td>
<td>93.7</td>
<td>91.1</td>
<td>68.2</td>
<td>65.8</td>
</tr>
</tbody>
</table>

We conducted ablation study to identify how well the optional enhancements are. In Table 6.5, row 2, 3 and 4 shows the accuracy of GRCNN with different enhancements. We find that the edge accuracy and graph accuracy decreased with those enhancements. Our explanation is that the extra trainable weights in the enhancements caused the network biased toward the nodes detection and recognition networks, but caused the decrease in edge accuracy, and therefore caused decrease in the graph accuracy. We also find all STN enhancements improved the accuracy of node detection and recognition networks.

### 6.2.3 Performance

Table 6.6 shows the performance of GRCNN and its subnets. The overall time cost of GRCNN is about 60 milliseconds, and the performance of other networks are close to the performance of GRCNN. Among the subnets, the time cost of node detection network is
the major part, 70.1%, of the overall cost. The rest is the node recognition network (23.5%),
backbone network (5.8%), and edge network (1%).

Table 6.6: Time cost in milliseconds of GRCNN. The columns are the time cost of the backbone,
edge, node detection, node recognition network and GRCNN, respectively.

<table>
<thead>
<tr>
<th></th>
<th>BB</th>
<th>Edge</th>
<th>ND</th>
<th>NR</th>
<th>GRCNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRCNN</td>
<td>3.5</td>
<td>0.4</td>
<td>42.4</td>
<td>14.2</td>
<td>60.5</td>
</tr>
<tr>
<td>+STN</td>
<td>3.5</td>
<td>0.4</td>
<td>43.3</td>
<td>16.6</td>
<td>63.8</td>
</tr>
<tr>
<td>+ResNet</td>
<td>3.5</td>
<td>0.4</td>
<td>44.8</td>
<td>14.0</td>
<td>79.1</td>
</tr>
<tr>
<td>+STN+ResNet</td>
<td>3.6</td>
<td>0.5</td>
<td>47.0</td>
<td>16.7</td>
<td>84.3</td>
</tr>
<tr>
<td>Separated</td>
<td>3.5</td>
<td>0.4</td>
<td>42.8</td>
<td>15.1</td>
<td>61.8</td>
</tr>
</tbody>
</table>

6.2.4 End-to-End vs Separated Network

Because edges and nodes information do not depend on each other, it is natural to con-
sider to use two separate networks to predict edges and nodes. We designed experiments
to see how well both ideas work.

We made two clones of GRCNN and modified as follows. For one clone, we disable
the node detection network and node recognition network. For the other clone, we dis-
able the edge detection network. We trained the two networks separately with the same
dataset and hyper parameters.

The row Separated in Table 6.5 and 6.6 shows the accuracy and performance of the
separated networks. The performance are close to GRCNN, therefore it is feasible to
share the computation of the backbone network. The sharing saves the 3.5 milliseconds,
which is 5.8% of the time cost.

6.2.5 Real-world Program Synthesis

In order to see how well GRCNN synthesizes real-world programs, we created a test
dataset of 11 programs, which are selected from the problems in a programming text-
Table 6.7 is the testing result. It shows that 70% of programs are correctly predicted, 70% edges are correctly predicted, 93.6% individual nodes are correctly predicted.

Table 6.7: Result of real-world program synthesis. The columns are program name, Graph accuracy, Edge accuracy, Nodes accuracy, number of nodes, correctly predicted nodes.

<table>
<thead>
<tr>
<th>program</th>
<th>Graph</th>
<th>Edge</th>
<th>Nodes</th>
<th>#Nodes</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>swap</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>max</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>sum</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>max3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>log</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>radius</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>poly</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>factorial</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>quadrant</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>cntpos</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>sum</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>53</td>
<td>49</td>
</tr>
<tr>
<td>percentage</td>
<td>63.6</td>
<td>72.7</td>
<td>81.8</td>
<td>-</td>
<td>92.5</td>
</tr>
</tbody>
</table>

In Figure 6.2, we draw two samples to visually demonstrate the input and output of GRCNN. The first one is the function that finds the max of three numbers. The second one is the factorial function.

Figure 6.2: Samples of GRCNN prediction. The predicted bounding boxes are drawn in red lines. The predicted text is drawn above the bounding box in blue characters. On the right is the predicted adjacent matrix and synthesized source code.
6.3 Conclusion

We presented GRCNN, a deep convolutional neural network that parses graph data structure from an image of flow chart, and automatically generates source code that matches the flow chart. GRCNN predicts the edge information and nodes information simultaneously. Experiments show that we can share the computation of the feature vector. GRCNN achieves 66.4%, 94.1%, 90.6% for graph, edge and node accuracy, respectively, on our test dataset, and close accuracy on a real-world dataset.
Bibliography


