CHAPTER ONE

Reverse-Engineering Software Behavior

Neil Walkinshaw
Department of Computer Science, The University of Leicester, Leicester, UK

Contents

1. Introduction 2
2. Background 4
   2.1 Modeling Software Behavior 6
      2.1.1 Modeling Data 6
      2.1.2 Modeling Sequential Behavior 7
      2.1.3 Combining Data and Control 8
   2.2 The Reverse-Engineering Process 11
      2.2.1 Summary 12
3. Static Analysis 13
   3.1 Why do we Need Behavioral Models if we have Source Code? 13
   3.2 Intermediate Representations—Source Code as a Graph 15
      3.2.1 The Control Flow Graph 15
   3.3 Answering Questions about Behavior with Static Analysis 19
      3.3.1 Dominance Analysis, Dependence Analysis, and Slicing 20
   3.4 Building behavioral Models by Static Analysis 25
      3.4.1 Collecting "Static Traces"—Identifying Potential Program Executions 25
      3.4.2 Deriving Models from Static Traces 29
   3.5 Limitations of Static Analysis 30
4. Dynamic Analysis 31
   4.1 Tracing and the Challenge of Inductive Inference 31
      4.1.1 Tracing 32
      4.1.2 The Essential Challenge of Dynamic Analysis 34
   4.2 Practical Trace Collection Approaches 35
      4.2.1 Selecting Inputs by the Category Partition Method 35
      4.2.2 Selecting Inputs by Exercising the Surrounding System 37
      4.2.3 Random Input Selection 38
   4.3 Data Function Inference 39
      4.3.1 Using General-Purpose Data Model Inference Algorithms 39
      4.3.2 Inferring Pre-Conditions, Post-Conditions, and Invariants 42
   4.4 State Machine Inference 43
   4.5 Limitations of Dynamic Analysis 44

Advances in Computers, Volume 91
ISSN 0065-2458, http://dx.doi.org/10.1016/B978-0-12-408089-8.00001-X

5. Evaluating Reverse-Engineered Models 46
5.1 Evaluating Syntactic Model Accuracy 47
5.1.1 Measuring the Difference 49
5.2 Comparing Model Behavior 49
6. Conclusions and Outstanding Challenges 51
6.1 Accommodating Scale 51
6.1.1 Static Analysis Technique 51
6.1.2 Dynamic Analysis Technique 52
6.2 Factoring in Domain Knowledge 53
6.3 Concurrency and Time-Sensitivity 54
References 55

Abstract

Software systems are large and intricate, often constituting hundreds of components, where the source code may or may not be available. Fully understanding the runtime behavior of such a system is a daunting task. Over the past four decades, a range of semi-automated *reverse-engineering* techniques have been devised to fulfill (or assist with the fulfillment) of this goal. This chapter provides a broad overview of these techniques, incorporating elements of source code analysis, trace analysis, and model inference.

1. INTRODUCTION

This chapter presents a broad introduction to the challenge of reverse-engineering software behavior. In broad terms, the challenge is to be able to derive useful information about the runtime dynamics of a system by analyzing it, either in terms of its source code, or by observing it as it executes. The nature of the reverse-engineered information can be broad. Examples include constraints over the values of variables, the possible sequences in which events occur or statements are executed, or the order in which input is supplied via a GUI.

As a research topic, the challenge of reverse-engineering behavioral models is one of the longest-established in Software Engineering. The first paper (to the best of the authors' knowledge) on reverse-engineering state machines (by dynamic analysis) was Moore's 1956 paper on Gedanken Experiments [39]. The paper that has perhaps had the biggest impact in the area and remains among the most cited is Biermann's 1972 paper on the *k*-Tails state machine inference algorithm [3].

The topic is popular because it represents a fundamental, scientifically interesting challenge. The topic of predicting software behavior is beset by negative results and seemingly fatal limitations. If we want to analyze the

source code, even determining the most trivial properties can be generally undecidable [45]. If we want to analyze software executions, there are an infinite number of inputs, and we are confronted by the circular problem that we need to know about the program behavior before we can select a suitable sample of executions to analyze.

However, the topic is not only popular because it is scientifically interesting. It also addresses a fundamentally important practical problem at the heart of software-engineering. In practice, software systems are often developed and maintained on an ad hoc basis. Although there might exist an initial model or statement of requirements, these rapidly become outdated as requirements change and the system evolves. Ultimately, a system can reach a point in its development where there is no point of reference about how the system behaves or is supposed to behave, which has obvious implications for crucial tasks such as testing and quality assurance.

This chapter aims to provide an overview of some of the key topics that constitute this broad field. It discusses the basic goals (models of software behavior), static and dynamic analysis techniques that aim to (at least in part) achieve them, methods to evaluate the resulting models, and some of the open challenges. It assumes no prior knowledge of source-code analysis, dynamic analysis, and only a limited knowledge about the possible modeling formalisms used to capture program behavior.

Clearly, given the breadth of the field, it is impossible to provide an exhaustive analysis and set of references for each of these subtopics within a single chapter. Instead this chapter aims to serve as an introduction. It aims to provide a useful overview that, while avoiding too much detail, manages to convey (at least intuitively) the key problems and research contributions for the main topics involved. Although the chapter seeks to avoid technical detail, where possible it uses applied examples and illustrations to provide the reader with a high-level picture of how the various techniques work. The rest of the chapter is structured as follows:

- **Section 2: Background.** This starts off by providing an overview of some of the main modeling languages that can be used to capture software behavior, providing small illustrative examples for each case. This is followed by a high-level overview of the reverse-engineering process.
- **Section 3: Static analysis.** Introduces some of the key source code analysis approaches that can be used to gain insights into program behavior. It also discusses why automated techniques are necessary at all (why manual inspection of source code alone is impractical), and finishes with a

discussion of some of the key limitations of static analysis, explaining why it is generally very difficult to relate software syntax to runtime behavior.

- **Section 4: Dynamic analysis.** Introduces the general problem of inferring models from program traces. Many of the fundamental limitations and concepts are rooted in the Machine Learning area of inductive inference, so the section starts by highlighting this relationship. This is followed by some practical approaches to collect execution traces (inspired by software testing), an overview of the key reverse-engineering techniques, and some of the key limitations that hamper dynamic analysis in general.
- **Section 5: Evaluation.** Considers how to interpret and evaluate reverse-engineered models. The usefulness of a model depends on its trustworthiness. This is by definition difficult to establish if there is a lack of prior knowledge about the system. This section shows how models can be evaluated if there exists some "gold standard" model against which to compare the model, and considers possible techniques that can be adopted from Machine Learning research to assess the accuracy of a model when there is no existing model to draw upon.
- **Section 6: Future work.** Concludes by providing an overview of some of the key outstanding challenges. It considers ongoing problems such as scalability, and the inability to factor in domain knowledge. It also looks toward the key looming challenge of being able to factor in concurrency and time-sensitivity.

2. BACKGROUND

The term "software behavior" can assume a broad range of different meanings. In this chapter, it is defined as broadly as possible: *The rules that capture constraints or changes to the state of a program as it executes*. There are numerous different ways in which such rules can be represented and captured. This subsection presents an overview of the two main facets of software behavior: data and control. It presents an informal introduction to the key modeling formalisms that can capture behavior in these terms, and provides a high-level overview of the generic reverse-engineering process.

To illustrate the various types of modeling languages, we adopt a simple running example of a bounded stack. As can be seen from Fig. 1, it has a conventional interface (the push, pop, and peek functions), with the additional feature that the bound on its size can be specified via its constructor.

Throughout this section, several notions and modeling formalisms will be introduced. Since the purpose is merely to provide an intuition of the

```
public class Driver{

  public static void main(String[] args){
    BoundedStack bs = new BoundedStack(Integer.valueOf(5));
    bs.push("object1");
    bs.push("object2");
    bs.push("object3");
    bs.push("object4");
    bs.push("object5");
    bs.pop();
    bs.pop();
    bs.pop();
    bs.peek();
    bs.push("object6");
    bs.push("object7");
    bs.push("object8");
    bs.push("object9");
    bs = new BoundedStack(Integer.valueOf(2));
    bs.push("object1");
    bs.push("object2");
    bs.push("object3");
    bs.push("object4");
    bs.push("object5");
    bs.pop();
    bs.pop();
    bs.pop();
    bs.peek();
    bs.push("object6");
    bs.push("object7");
    bs.push("object8");
    bs.push("object9");
    bs = new BoundedStack(Integer.valueOf(100));
    bs.push("object1");
    bs.push("object2");
    bs.push("object3");
    bs.push("object4");
    bs.push("object5");
    bs.pop();
    bs.pop();
    bs.pop();
    bs.peek();
    bs.push("object6");
    bs.push("object7");
    bs.push("object8");
    bs.push("object9");
  }
}
```

```
import java.util.Stack;

public class BoundedStack {
  private int lim;
  private Stack<Object> s;

  public BoundedStack(int limit){
    lim = limit;
    s = new Stack<Object>();
  }

  public boolean push(Object o){
    if(s.size()<lim){
      s.push(o);
      return true;
    }
    else
      return false;
  }

  public Object pop(){
    return s.pop();
  }

  public Object peek(){
    return s.peek();
  }
}
```

Fig. 1. Code for BoundedStack.

different perspectives on software behavior, they will be introduced in an informal, intuitive manner. Although notions such as state machines are commonly associated with formal, detailed definitions in conventional research texts, in our context these definitions are deemed redundant, and are omitted.

The rest of the section is structured as follows. Section 2.1 introduces an overview of some of the main modeling techniques, covering control, data, and the combination of the two. This is followed by Section 2.2, which provides an overview of the general reverse-engineering process.

2.1 Modeling Software Behavior

Modeling languages for software systems can be crudely split into three types: (1) languages to model the data state of the system, (2) languages to model the sequential control within the system, and (3) languages that attempt to combine the two. This subsection provides a brief overview of these modeling techniques (in the same order).

2.1.1 Modeling Data

Data models capture the data state of a system or component. In other words, they model the possible values of data variables at given points during the execution of a program, and potentially the possible data-transformations that lead from one state to another. In practice, such models are highly popular, and are widely used. Developers can easily use them to encode their assumptions into the system (e.g., via assertions), and testing frameworks such as JUnit use them to encode test-oracles.

Languages for specifying data behavior are commonly founded upon Hoare Logic [23]. This provides a framework where specifications consist of three parts: (1) a *pre-condition*, (2) the function itself, and (3) a *post-condition*. Pre- and post-conditions are predicates over the variables of the program that must hold for the behavior of the function to be correct.

Popular examples of modeling languages include Z [59], Alloy [26], and the Object Constraint Language (OCL) [40]. Ultimately, these provide linguistic constructs that can be used to express constraints over the variables of a program. They tend to differ in their expressivity, suitability for automated analysis, and target user-base. Most specification languages are associated with automated reasoning tools, which can check that the generated specifications are internally consistent.

Ultimately, regardless of the chosen formalism, a specification sets out rules covering specific relationships between variables that must hold at certain points during a program execution. As an example, we provide a post-condition for the `push` method in our stack example, expressed in OCL:

```
context: stack.push(Object):void
pre notNull: s!= null
post notEmpty: s.isEmpty() == false
post notOverUpperLimit: s.size <= lim.
```

The specification is relatively self-explanatory. The first line describes the method in question. The prefixes `pre` and `post` denote pre- and post-conditions respectively. This is followed by an identifier for the assertion, and then the assertion itself. It is evident that these rules can be mapped

directly to source-code assertions, where pre-conditions appear before the method body, and post-conditions are checked before the exit point of the method.

2.1.2 Modeling Sequential Behavior

Data models are concerned purely with the data states; the values of variables at various points during execution. They ignore the sequential order in which events occur that might cause these state changes, such as the provision of inputs or the order in which functions are called. This facet of software behavior is captured by "sequential models." These are commonly represented either as state machines or as message sequence charts (StateCharts or Sequence Diagrams in UML [40]).

2.1.2.1 State Machines

State machines represent a broad family of models that impose a partial order on a sequence of events in the system. Essentially, a state machine has five components: (1) a set of states, (2) a set of transitions between the states, (3) a set of labels that annotate the transitions, and (4) a single initial state. Depending on the type of system, states may also be associated with flags to denote them as "final" or "accepting," indicating that execution terminates when these states are reached. Conventionally, transition labels are taken to

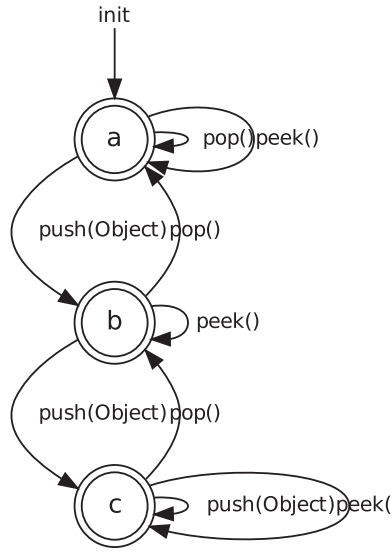


Fig. 2. State machine for BoundedStack where the limit is set to 3.

represent different events or signals in the software system. A state machine can thus be used to model the potential sequencing of these events.

An example of a state machine for the bounded stack source in Fig. 1 is shown in Fig. 2. Here, labels denote input-output pairs (where the output is in response to an input). All of the states are possible final states (denoted by the double-circle). Note that, as state machines cannot incorporate data, we have to presume that the limit of the size of the stack is fixed to some given value (here we choose 3). If we wanted to produce an FSM that could represent the inputs for *any* limit, it would require an infinite number of states and transitions, or would need to incorporate data (more sophisticated models to do this will be introduced later).

2.1.2.2 Message Sequence Charts and Sequence Diagrams

Whereas state machines serve to lay out *every* possible sequence of events, Message sequence charts (MSCs) [25]—or their UML variants Sequence Diagram [40] capture the order of particular sequences of events ("messages"). These are primarily used to show typical sequences of method calls between objects within an object-oriented system, but can also be used more generally, e.g., to model signals sent between processes in concurrent systems.

Sequences might correspond to a particular scenario, or describe the activity that underpins a particular software feature. The entities that send or receive messages (e.g., objects or processes) are given a box at the top of the diagram. A vertical line below the box signifies the life-span of the object (where time runs from top to bottom). The messages are added as arrows from one line to the other.

Figure 3 shows an example of a possible message sequence diagram for the BoundedStack example. Each box corresponds to a class in the system, and each arrow corresponds to a method call. As with state machines, there are several notational variants of this notation that can incorporate particular characteristics of paradigms or types of systems (c.f., Live Sequence Charts [12]).

2.1.3 Combining Data and Control

One of the weaknesses of the specification methods considered so far is the fact that they only partially capture software behavior. Data models capture data states, but fail to capture the sequences of events that lead from one data state to the other, and fail to show explicitly how data states affect the possible sequencing of events during execution. FSMs and MSCs capture the possible sequencing of events within or between components, but fail

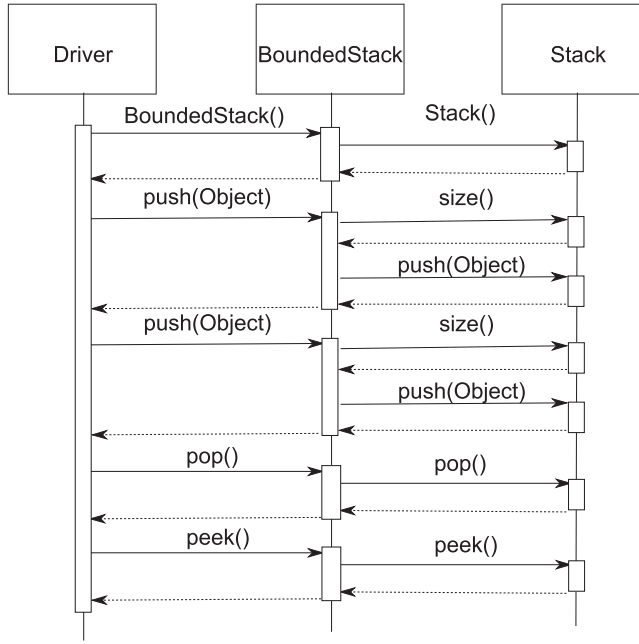


Fig. 3. Sequence diagram for a BoundedStack scenario.

to track the data-state of the system throughout these events. To address this problem, a range of extensions have been developed to existing modeling techniques, that enable the two facets of data and control to be combined. Such combined notations are generally *Turing-complete*. In other words, these notations can be used to capture any sequential program that can be encoded as a Turing Machine.

2.1.3.1 EFSMs

Several such notations have been developed, which extend the conventional FSM in such a way that states and transitions can be decorated with guards and data operations [19, 6, 24]. Extended Finite State Machines (EFSMs) [6] represent one particularly popular variant. These extend conventional state machines in three ways: (1) by adding a *memory* to store data variables, (2) by enabling transitions to have data-guards (expressed in terms of conditions on the memory contents), and (3) by associating transitions with functions that can transform the memory contents.

An example of an EFSM model of the BoundedStack example is shown in Fig. 4. For the memory we use the two variables that correspond to the

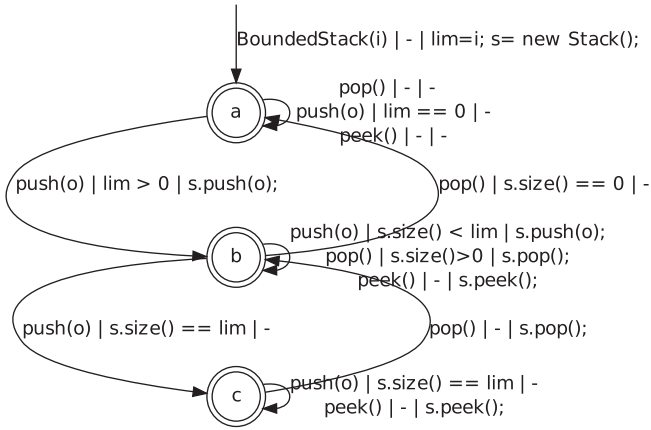


Fig. 4. An EFSM for BoundedStack.

data members in the source code: `s` and `lim`. The labels consist of three parts (separated by a "|"). The first part is the input (in our case the method call), the second part is the guard condition on the memory (a boolean predicate on `s` and `lim`), and the third part is any operation on the memory that is executed along with the state transition. For example, if we look to state `a`, there are two outgoing transitions for the function `push`. If `lim==0`, nothing can be pushed onto the stack, and a push has no effect on the data state (thus looping to the same state). If `lim>0` however, a push leads from state `a` to `b`, and results in an actual push of `o` onto the stack.

As a whole, the three states distinguish behavior from what happens when the stack is empty (state `a`), the stack is not empty, but also not full (state `b`), and the stack is full (state `c`). It is important to note that there can be lots of valid potential EFSMs for the same system, which may differ in the number of states and the nature of the guards.

Models such as EFSMs are better able to capture software behavior because they are "richer" in terms of their syntax and semantics. However, an important downside is that this adds complexity, which makes them less appealing to developers. For example, simple assertions in the source code are much more widely used than EFSMs, because they can be readily produced during development. Generating richer models such as EFSMs incurs an overhead; the developer has to invest at least as much effort into generating the EFSM as they might have to invest into the source code of the system itself. This of course is a key reason for wanting to use reverse-engineering

techniques—to generate such specifications when the developer has not had the time to, or when existing specifications have become outdated.

2.1.3.2 Algebraic Models

Algebraic specifications characterize a system in terms of the relationships between its operations over the underlying data structure. The output of each operation is captured in declarative terms, often recursively. As the number of function definition expands, they can be used to define or characterize each other's behavior. A chapter on the practical application and value of algebraic specifications can be found in Sommerville's Software Engineering book [49].

With respect to our stack example, some simple properties might be as follows:

```
pop(push(x)) -> x
pop(new BoundedStack(y)) -> Exception
peek(push(y). push(z)) -> z
```

These are self-explanatory. Popping a stack where an `x` has been previously pushed onto it should yield an `x`. Popping an empty stack should raise an exception. Peeking on a stack that has been constructed by `push(y)` followed by `push(z)` should yield a `z`. This concise, declarative format is precisely what makes algebraic specifications so popular. Such models are especially relevant to functional programming languages (where the core functionality is defined in a similar way), such as Scala, Erlang, and Prolog.

2.2 The Reverse-Engineering Process

The key conceptual steps involved in the general reverse-engineering process are shown in Fig. 5. Reverse engineering was defined by Chikofsky and Cross as *"the process of analyzing a system to . . . create representations of the system in another form, or at a higher level of abstraction."* [7]. Although this chapter is concerned with a more specific definition ("another form" is restricted to "a model of software behavior"), the general process discussed here applies more generally.

Reverse-engineering starts by a process of "information gathering," which may involve analysis of source code, observation of program executions, or some combination of the two. This information often has to be abstracted—it has to be recoded in such a way that it captures the relevant points of interest in the program at a useful level of detail, so that the final result is readable and relevant. Finally, the collected information has to be

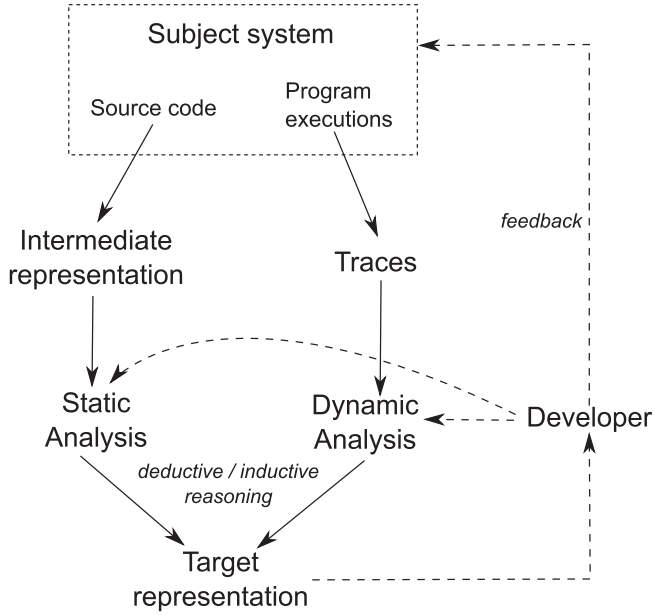


Fig. 5. Schematic diagram of the key conceptual steps in the reverse-engineering process.

used to infer or deduce a behavioral model. This step is often automated, but may involve human intervention (e.g., to incorporate domain knowledge).

All reverse-engineering procedures that produce behavioral models can be characterized in terms of these steps. Some solely rely on static analysis, others on dynamic analysis, and some combine the two. Some incorporate input from developers, and some are iterative—gradually refining the target representation by soliciting additional traces, or additional inputs from the developer at each iteration.

This schematic view of the reverse-engineering process provides a useful reference point for the rest of this chapter. It shows where static and dynamic analysis fit in, and how the developer can play a role in each case. It also shows how techniques can be iterative. The final model can be fed back to the reverse-engineer, who might seek to refine the result by considering additional sets of traces, or by integrating other forms of source code analysis.

2.2.1 Summary

This section has presented shown some of the core languages with which software behavior can be modeled. It has also presented an overview of the general reverse-engineering process: the task of analyzing a software system

and deriving approximations of these abstract behavioral models. The next two sections will look at some of the core tools for this process. The next section will look at static analysis techniques, and how these can obtain information about software behavior from source code. This will be followed by a similar overview of dynamic analysis, showing how software behavior rules can be extracted from observations program executions.

3. STATIC ANALYSIS

Static analysis is concerned with the analysis of the source code syntax (i.e., without executing the program). In principle, source code can be seen as the ultimate behavioral model of a software system. It encodes exactly those instructions that will be executed at runtime; it imposes an order on *when* events can occur, and governs the possible data values at different points during the execution.

This section seeks to provide a relatively self-contained introduction to the role of static analysis in reverse-engineering models of software behavior. It starts by covering the basics—showing how source code can be interpreted as a graph that captures the flow of control between individual expressions and the associated flow of data values. These are presented in relative detail here, not only because they form the basis for all more complex source code analysis techniques, but also because they are of considerable value in their own right. These representations alone provide a complete (albeit low level) overview of the order in which statements can be executed, and how they can pass data among each other.

This is followed by an overview of the analysis techniques that build upon these notions of control and data flow to extract information about behavior (the ability to reason about variable values at given points during the software execution). Finally, the section concludes with an overview of the various factors that limit the accuracy of static analysis. The descriptions of static analysis techniques attempt to present an intuitive and informal introduction. For more formal definitions, along with algorithms to compute the various relations within the code, there are several comprehensive overviews [2].

3.1 Why do we Need Behavioral Models if we have Source Code?

So why not simply refer to the source code in place of a behavioral model? Why are models necessary, and what can code analysis provide? There are three key reasons for this:

Firstly, there is the problem of *feature location*. Source code that pertains to a single facet of functionality can, depending on the language paradigm, be spread across the system. Whereas the specific element of behavior that we seek to reverse-engineer might only cover a small fraction of the source code base (e.g., the code in a large drawing package that is responsible for loading a drawing from a file), the code in question may not be localized to a particular module or component in the system. High-level software behavior often arises from extensive interaction between different modules or packages [34], especially where Object-Oriented systems are concerned [56]. So, although source code contains all of the relevant information, this can be difficult to locate because it contains an overwhelming amount of irrelevant information too.

Secondly, there is the related problem of *abstraction*. Once the relevant source code has been localized, there is the challenge of interpreting its functional behavior from its implementation. The way in which a particular unit of functionality is encoded in source code depends on several factors, such as the choice of programming language, paradigm, architecture, external libraries, etc. Thus, a program that achieves a particular functionality in one language or paradigm can differ substantially from another (the reader is referred to the Rosetta Code web repository[1] for some illustrative examples of this).

Thirdly, source code is a static representation, whereas its behavior is intrinsically *dynamic*. The challenge of discerning program behavior therefore necessarily requires the developer to "mentally execute" the relevant source code, to keep track of the values of various relevant variables, and to predict the output. For any non-trivial source code and non-trivial functionality, this can incur an intractable mental overhead [5].

In summary, attempting to develop a complete understanding of software behavior from source code alone is rarely practical. Leaving aside the basic problem that the source code might not be available in its entirety, the essential challenges listed above can be summarized as follows: There is (1) the "horizontal" complexity of trying to discern the source code that is relevant to the behavioral feature in question, there is (2) the "vertical" complexity of interpreting the relevant behavior at a suitable level of abstraction, and finally there is (3) the problem of mentally deducing the dynamic behavior of the system, and how its state changes from one execution point to the next.

[1] http://www.rosettacode.org.

3.2 Intermediate Representations—Source Code as a Graph

Most static analysis techniques can be discussed and implemented in graph-theoretical terms. From this perspective, source code is interpreted in terms of relations between statements. These can convey various types of information about the program, with respect to the sequence in which instructions are executed (the *control*), and the values of variables at various points (the *data*).

This subsection will focus on the low-level code representations that are especially useful as a basis for extracting knowledge about program behavior. To be clear, this section does not constitute a section on "reverse-engineering" in its own right, as the representations produced do not constitute the sort of behavioral models that one would usually aim for. However, the representations nonetheless provide valuable insights into software behavior, and form the basis for most static reverse-engineering techniques, which will be described in the following subsection. In any case, an understanding of these basic analyses provides a useful insight into the general nature of the information that can feasibly be reverse-engineered, without tying us down to specific techniques.

3.2.1 The Control Flow Graph

Individual procedures or routines in the source code can be represented as a *control-flow* graph (CFG) [2]. This is a directed graph, where nodes correspond to individual instructions, and edges represent the flow of control from one instruction to the next. Each graph has an entry point (indicating the point where the program starts), predicates such as `if` and `while` statements correspond to branches, and return statements or other terminating points in the program correspond to the exit nodes of the graph. Every possible path through the graph corresponds to a potential program execution (where loops in the graph indicate a potentially infinite number of executions).

It is important to note the generality of this representation. CFGs can be used to represent control flow for programs written in any programming language. Whether the program in question is written in BASIC, Java, or C, or is the Assembly code post-compilation, it will contain decision points and instructions that are executed according to some predefined sequence, which can always be represented in terms of a CFG.

To illustrate how a CFG is constructed we build a CFG for a simple BASIC routine (written by David Ahl [1][2]). The code, shown on the left

[2]Reproduced here with his kind permission.

```
1 DIM T(20)
2 INPUT "TIME INCREMENT (SEC)";S2
3 INPUT "VELOCITY (FPS)";V
4 INPUT "COEFFICIENT";C
5 PRINT "FEET"
6 S1=INT(70/(V/(16*S2)))
7 FOR I=1 TO S1
    8 T(I)=V*C^(I-1)/16
    9 NEXT I
10 FOR H=INT(-16*(V/32)^2+V^2/32+.5) TO 0 STEP -.5
    11 IF INT(H)<>H THEN 13
    12 PRINT H;
    13 L=0
    14 FOR I=1 TO S1
        15 FOR J=0 TO T(I) STEP S2
            16 L=L+S2
            17 IF ABS(H-(.5*(-32)*J^2+V*C^(I-1)*J))>.25 THEN 19
            18 PRINT TAB(L/S2);"0";
            19 NEXT J
        20 J=T(I+1)/2
        21 IF -16*J^2+V*C^(I-1)*J<H THEN 23
        22 NEXT I
    23 PRINT
    24 NEXT H
25 PRINT TAB(1);
26 FOR I=1 TO INT(L+1)/S2+10
    27 PRINT ". ";
    28 NEXT I
29 PRINT
30 PRINT "0";
31 FOR I=1 TO INT(L+.9995)
    32 PRINT TAB(INT(I/S2));I;
    33 NEXT I
34 PRINT
35 PRINT TAB(INT(L+1)/(2*S2)-2);"SECONDS"
36 PRINT
37 END
```

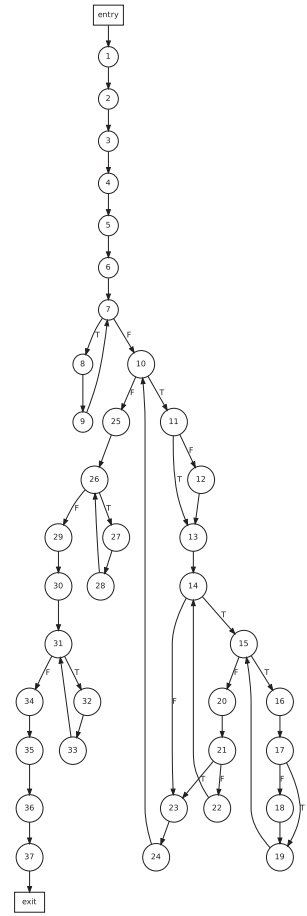


Fig. 6. Bounce.bas BASIC routine to plot the bounce of a ball, and the corresponding CFG.

in Fig. 6, plots out the trajectory of the bounce of a ball (shown in Fig. 7), given a set of inputs denoting time-increment, velocity, and a coefficient representing the elasticity of the ball.

The CFG is shown on the right of the figure. It plots the possible paths through the function (the possible sequences in which statements can be executed). It starts and ends with the *entry* and *exit* nodes, which do not correspond to actual statements, but merely serve to indicate the entry and exit points for the function. Predicate-statements (e.g., `if` or `while` statements) are represented by branches in the CFG, where the outgoing edges denote the true or false evaluation of the predicate. Statements at the end of a loop include an edge back to the predicate.

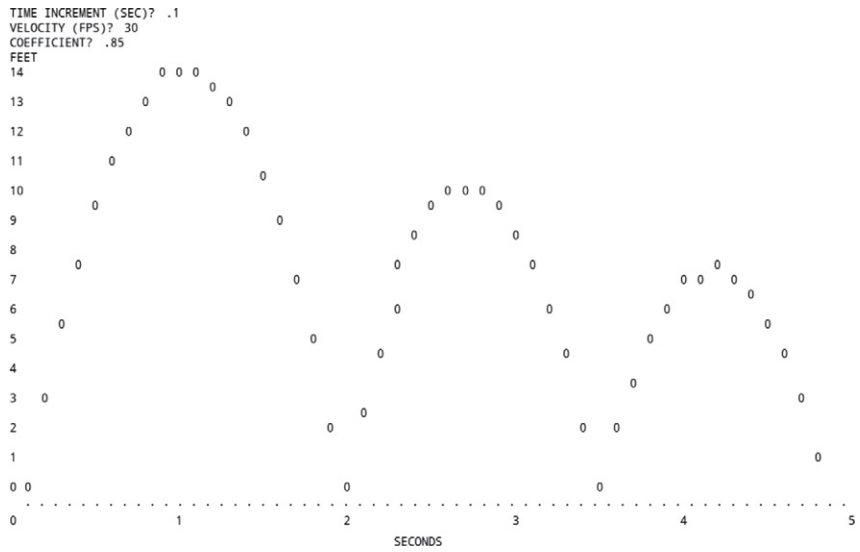


Fig. 7. Output from bounce.bas.

3.2.1.1 Data Flow

The CFG in itself only conveys information about the possible order(s) in which statements can be executed. To add information about the possible flow of variable values from one statement to the other, we start by defining two types of relationship. For a given node s in a CFG, the function $def(s)$ identifies the set of variables that are assigned a value (i.e., defined) at s. The function $use(s)$ identifies the set of variables that are used at s.

The above *def* and *use* relations can, along with the CFG, be used to compute the *reaching definitions* [2]. For a given definition of a variable in the code, the reaching definitions analysis computes the subsequent points in the code where that specific definition is used. To make it slightly more formal, each node s in the CFG is annotated with a set of variable-node pairs $\langle v, n \rangle$ for each $v \in use(s)$. These are computed such that the value of v is defined at n, and there is a definition-free path with respect to v from n to s. These reaching definitions are visualized as dashed edges in the extended CFG, shown in Fig. 8.

Intuitively, this graph captures the basic elements of program behavior. The complex interrelationships between data and control are summarized in a single graph. The intuitive value is clear; it is straightforward to visually trace how different variables affect each other, and how different sets of variables feature in particular paths through the code. In practice such graphs

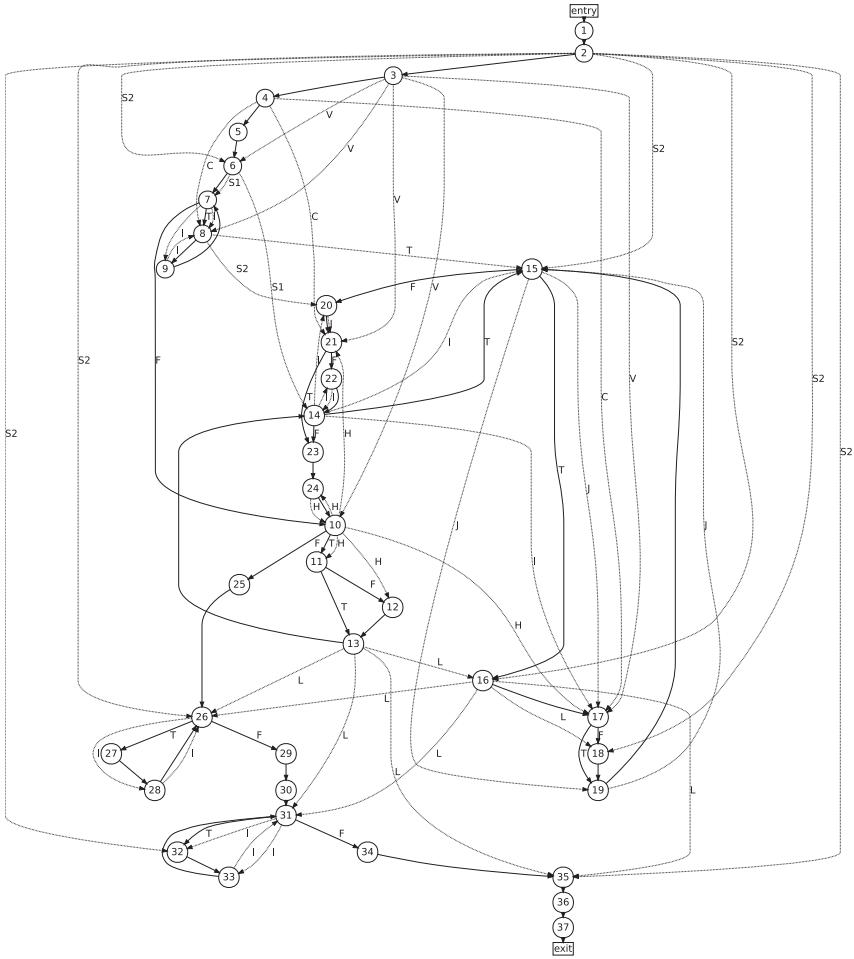


Fig. 8. CFG with reaching definitions shown as dashed lines.

are rarely used as visual aids, but form the basis for more advanced analyses, as will be shown below.

3.2.1.2 Call Graphs

The above representations are solely concerned with the representation of individual procedures or functions. In practice, for programs that exceed the complexity of the `bounce` program, behavior is usually a product of interactions between multiple functions. The possible calls from one method or function to another can be represented as a "call graph" [47]. Such graphs

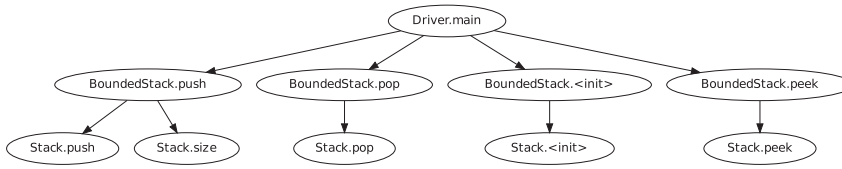


Fig. 9. Call graph for code in Fig. 1.

can, when used in conjunction with individual function CFGs, be used to compute data and control-flow relationships that span multiple procedures.

For this chapter an intuitive description of a call graph will suffice. A standard (context-insensitive) call graph consists of nodes that represent call statements within methods or functions, and edges represent possible calls between them. An example of a call graph with respect to the BoundedStack code is shown in Fig. 9. Due to the simplicity of the code, this call graph is straightforward; it is tree-shaped, there are no loops, and each call has only one possible destination.

For larger systems, call graph computation can be exceptionally challenging [18], and has been the subject of an extensive amount of research. Calls in the source code do not necessarily have a unique, obvious target. For example, the target of a call in object-oriented system (because of mechanisms such as polymorphism and runtime binding) is often only decided at execution-time. The *pointer-analysis* algorithms that underpin these computations [18] lie beyond the scope of this chapter, and the possible problems of inaccuracy will be discussed in the wider context of static analysis in Section 3.5.

3.3 Answering Questions about Behavior with Static Analysis

Having covered the basics of static analysis, we consider the essential questions that they can answer about software behavior. As discussed in Section 2, software behavior is multifaceted. Data models capture the values of data variables (and possibly any associated transformations) at different points in a program. Control models consider the order in which events or data states can arise. Combined models, such as the EFSM, capture the interplay between these two facets.

CFGs (with or without data annotations), coupled with call graphs, can convey information about program behavior, albeit at a low level. They make explicit the control and data flow between individual statements and functions. However, if we recall the three problems of feature location, abstraction,

and dynamism discussed at the beginning of this section, such a low-level representation is of limited use.

These can however be attenuated by a selection of well-established analysis techniques that build upon these basic representations. They can, at least to an extent, be used to answer specific questions about the possible behavior of a system. They can expose the dependence between statements in a program, can determine whether certain statements can or cannot be executed, can estimate the value ranges for variables at various points in the program, and can even provide the constraints on input parameters that are required to reach particular areas of the source code. The rest of this subsection will present an overview of these individual techniques. This will be followed by a subsection that provides an overview of existing approaches that have composed these techniques to build comprehensive behavioral models of software systems.

3.3.1 Dominance Analysis, Dependence Analysis, and Slicing

The possible order in which events can occur, or the chains of causality that lead from one state to another, form a fundamental part of the analysis of software behavior. Looking at the representations discussed in Section 2, this notion of sequence forms a fundamental aspect of state machines, Message Sequence Charts, and their variants. This information can be extracted from a CFG by two types of analysis: Dominance analysis [2] and Dependence analyses [10].

3.3.1.1 Dominance Analysis

Given a CFG, dominance analysis can be used to draw out the order in which statements are executed. A dominance analysis will, for each statement, identify other statements that *must* be executed beforehand, or that can *only* be executed subsequently. More formally, node *B post-dominates* a node *A* if all paths from *A* to the exit node must pass through *B*. Conversely, node *A dominates* node *B* if all paths from the entry node must pass through *A*.

These general notions of dominance and post-dominance are not particularly concise; lots of statements can dominate and post-dominate each other, which makes it difficult to impose a specific order on the execution. Instead of considering *all* possible dominating and post-dominating nodes, this can be narrowed down by considering only the *immediate* dominators and post-dominators. A node *A immediately* dominates another node *B* if there are no other nodes that dominate *B* on the path in the CFG from *A* to *B*. Conversely, a node *B immediately* post-dominates another node *A* if there

are no other nodes on the path from *A* to *B* that post-dominate *A*. Thus, each node can only have a unique dominator and post-dominator.

These immediate relationships can be visualized as a tree. The dominance and post-dominance trees for the `bounce` program are shown in Fig. 10. In terms of program behavior, these are interesting because, for each statement, they make explicit the set of other statements that *must* be executed before or after (depending on the tree). If one selects a statement in the dominator tree and takes the unique path to the root, all of the statements on that path dominate that statement, and will be executed beforehand (in the order in which they appear in the path). Similarly for the post-dominator tree; if one selects a statement and traces its paths to the root, all of the statements that appear on that path *must* be executed after the statement (this time in the reverse-order in which they appear in the path).

This dominance analysis provides a useful layer of information about the control-aspects of a program. It tells us which statements *must* be executed before or after each other. If we want to know, in coarse terms, the order in which statements are executed to calculate a particular output, then analyzing them in terms of their dominance is a good starting point.

3.3.1.2 Control Dependence Analysis

Dominance trees omit an important aspect of information: conditions. Source code is composed of logical constructs such as `if-then-else` blocks and `for` loops. The execution of a statement is generally contingent upon the outcome of decision points throughout the program. This conditional relationship between statements is not captured by dominance alone.

Control dependence analysis [10] builds upon dominance analysis and data-flow analysis to capture the conditional relationships between statements. A statement *B* is control-dependent upon a statement *A* if the execution of *B* is determined by the outcome of *A*, but *B* does not post-dominate *A*. In other words, the execution of *B* must solely depend on the evaluation of the predicate *A*; if *B* is executed regardless of the outcome of *A*, it is not control-dependent.

These control dependence relationships can again be viewed in a graphical format, referred to as the *control dependence graph* (CDG). The CDG for the `Bounce.bas` example is shown in Fig. 11. By default, any top-level statements that are not contained within a conditional block are control-dependent upon the entry node to the CFG. Nested branches indicate nested conditional blocks.

It is important to distinguish the nature of the information contained within the dominance trees and the dependence graph. Dominance trees

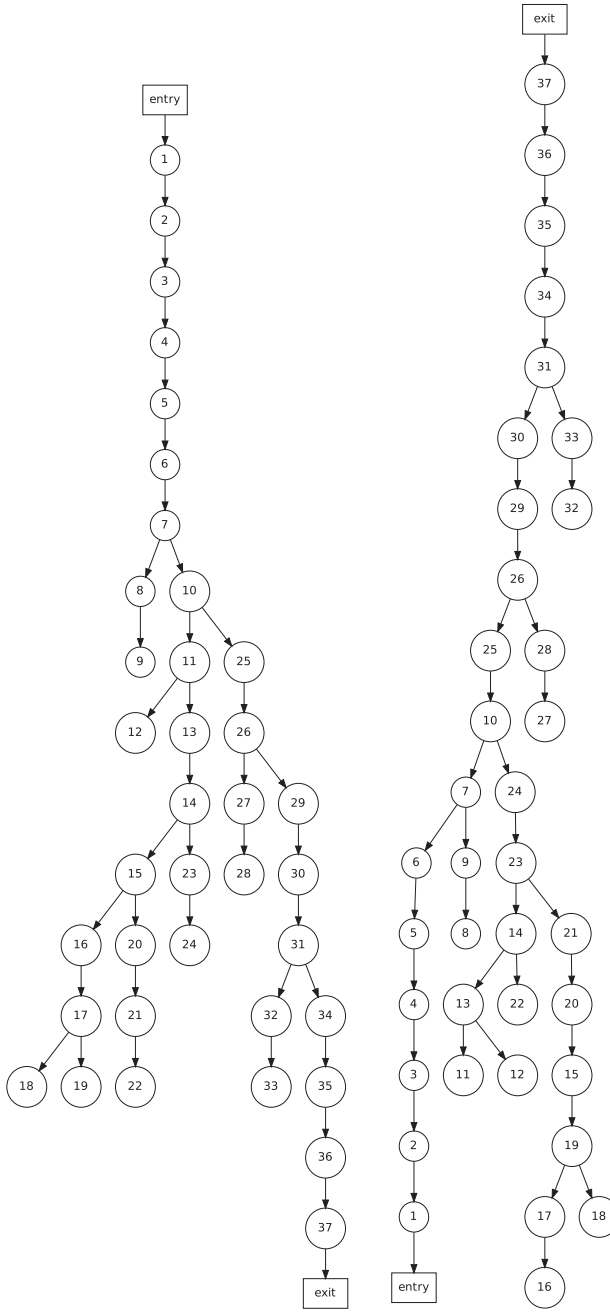


Fig. 10. Dominator and post-dominator trees for bounce (shown on the left and the right respectively).

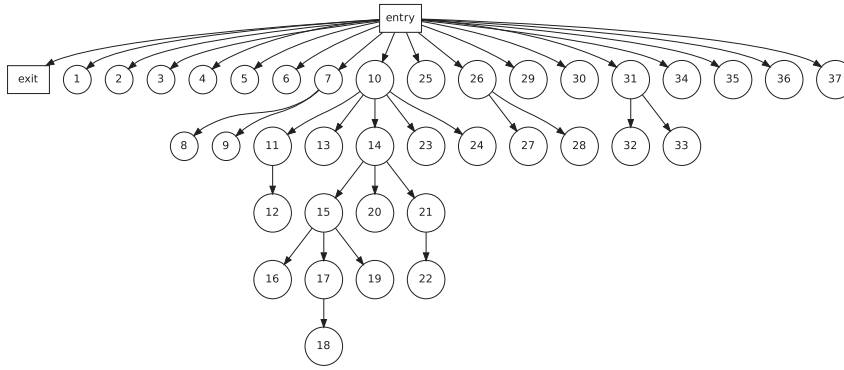


Fig. 11. Control dependence graph for bounce.bas.

contain sequential information, and can be used to answer questions of the nature *"Which statements must follow statement X ?"* or *"Is statement Y always preceded by statement X ?"*. On the other hand, the CDG ignores most of the sequential information. It identifies only the *essential* orders between statements that depend upon each other. It can identify relationships between statements where there is not necessarily a dominance relationship, but where one can still have a bearing on others' execution.

3.3.1.3 The Program Dependence Graph and Slicing

Whereas dominance and control dependence solely focus on control—whether or not (or the order in which) statements execute—the *program dependence graph* (PDG) [16] combines control dependence with the dataflow between statements. This (at least for a single-routine) includes all of the dependencies that tie statements to each other. For any pair of statements a and b that is connected by some path $a \rightarrow \cdots \rightarrow b$ through the program dependence graph, it is possible that a will either affect whether or not b is executed, or will have some effect on the variable values used at b. In other words, it is possible that a will affect the *behavior* of b. The PDG for the bounce.bas program is shown in Fig. 12.

This ability to automatically relate statements to each other in terms of their underlying dependencies is clearly valuable from the perspective of analyzing software behavior. Several automated tools have been developed that can use the program dependence graph to compute *program slices* [57]. A slice is defined with respect to a particular statement in the program, and a particular set of variables at that statement (together these are referred to as the *slicing criterion*). It is defined as the subset of statements in the program

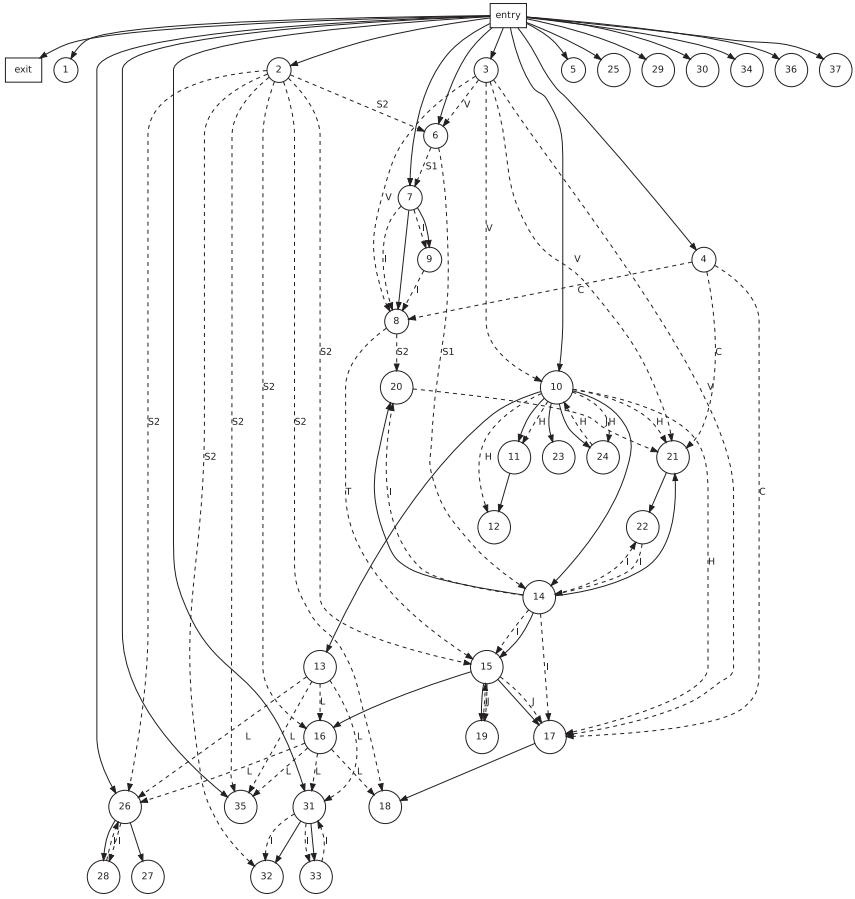


Fig. 12. Program dependence graph for bounce.bas.

that are responsible for computing the variables in the slicing criterion. When computed in terms of the PDG, a slice simply consists of all of those statements in the PDG that can reach the node in the graph representing the criterion.

Slices are clearly useful in their own right. As mentioned at the beginning of this chapter, one of the core problems of understanding software behavior is the information overload. There are lots of irrelevant source code, and the code that is relevant to the feature in question might be spread across the code base. Slices can (at least in theory) locate the relevant source code.

To get an intuition of how useful slicing can be, let us suppose that we are interested in finding out which statements affect the behavior of variables

`S1` and `l` at line 35. Extracting a slice from the dependence graph is simply a matter of tracing back along every incoming path to node 35, marking the statements that are traversed in the process. The resulting slice is shown in Fig. 13—both in terms of the reduced PDG, as well as the corresponding source code.

The "headline" value of slicing is clear. In reference to the challenges discussed in Section 3.1, it limits the amount of source code that has to be navigated with respect to a particular comprehension task, reducing the mental overhead on the developer. Importantly, these benefits also transfer to other non-manual tasks, including reverse-engineering. Even though these source code analysis techniques do not directly reverse-engineer a model, they can reduce the amount of source code that needs to be considered for a re-engineering task. Slicing can be used to reduce the code base to something more manageable. Slicing is used for the same purpose in other domains, such as source-code verification [13], and (its originally intended purpose) debugging [58].

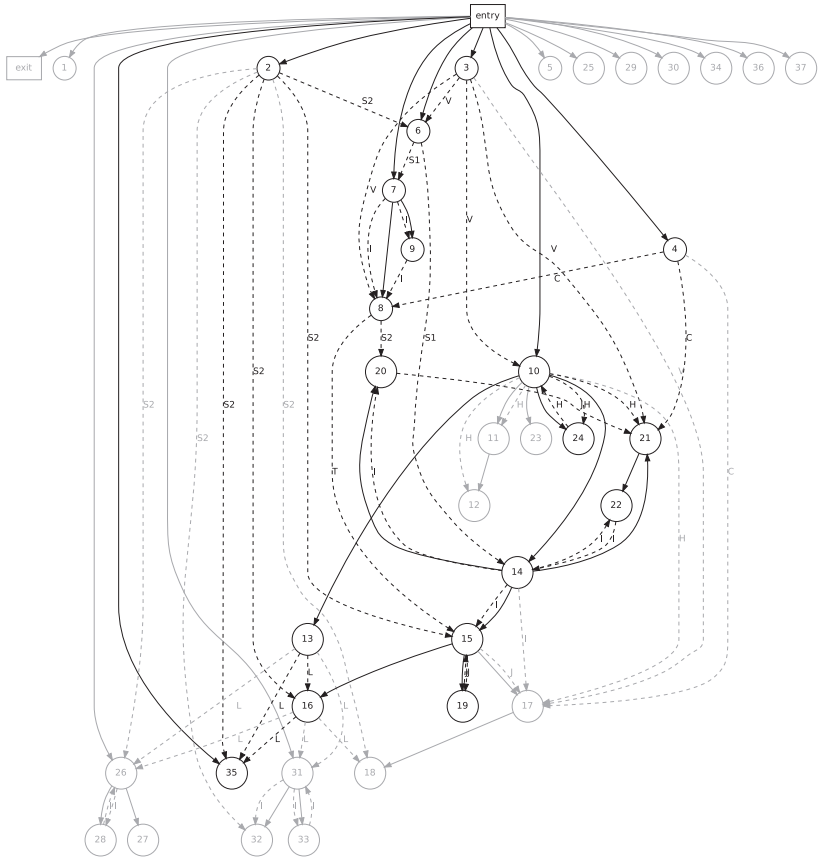
3.4 Building behavioral Models by Static Analysis

As illustrated above, source code (1) provides a (partial) ordering on the events that can occur during program execution, and (2) maps out the way in which data can flow through the program. It is the ultimate blue-print for program behavior. Although the low-level analyses discussed above can provide some basic information about execution order, it remains difficult to ascertain useful, abstract information about program behavior.

Numerous techniques have been developed that build upon the low-level analyses presented above to generate models of software behavior. These date back to work by Kung et al. [30], but the field has flourished only relatively recently, with work by numerous groups around the world [50, 54, 48]. All of these approaches share what is essentially the same underlying process. They first derive a set of "static traces," a tree-like representation of the possible paths through the source code. These static traces are then combined and abstracted, to produce an abstract model that captures the behavior of the system as a whole.

3.4.1 Collecting "Static Traces"—Identifying Potential Program Executions

Most static analysis approaches to deriving models rely on identifying a set of potential program execution paths through the code. These can subsequently be mined to generate the final model. The broad task of identifying




```
 2 INPUT "TIME INCREMENT (SEC)";S2
 3 INPUT "VELOCITY (FPS)";V
 4 INPUT "COEFFICIENT";C
 6 S1=INT(70/(V/(16*S2)))
 7 FOR I=1 TO S1
     8 T(I)=V*C^(I-1)/16
     9 NEXT I
10 FOR H=INT(-16*(V/32)^2+V^2/32+.5) TO 0 STEP -.5
     13 L=0
     14 FOR I=1 TO S1
         15 FOR J=0 TO T(I) STEP S2
             16 L=L+S2
             19 NEXT J
         20 J=T(I+1)/2
         21 IF -16*J^2+V*C^(I-1)*J<H THEN 23
         22 NEXT I
     24 NEXT H
35 PRINT TAB(INT(L+1)/(2*S2)-2);"SECONDS"
```

Fig. 13. Computing the slice in bounce.bas, where the slicing criterion is line 35, variables S1 and l. This shows that 18 of the 37 lines are relevant to the computation.

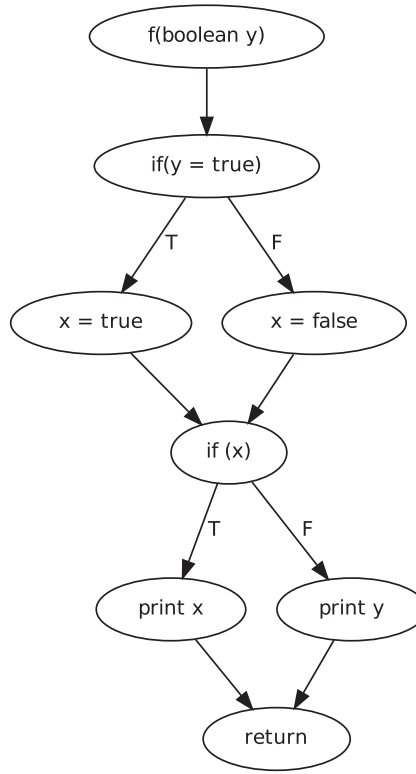


Fig. 14. An example of infeasible paths; the only feasible paths through this CFG are those that take *both* true branches, or *both* false branches. Paths that mix true and false branches are infeasible.

feasible executions from source code is (as will be elaborated in Section 3.5) generally undecidable. However, there are numerous techniques that have emerged from the field of software verification that can at least provide an approximation of feasible executions.

The challenge of identifying a feasible path through a program is best contemplated in terms of the CFG. In a CFG, a program execution corresponds to a path from the entry to the exit node. The problem is that not every walk through the CFG corresponds to a *feasible* execution. One branch in the graph might give rise to a configuration of data values that prohibit the execution of certain subsequent branches. This is illustrated in Fig. 14. Only two of the four paths through the graph are actually feasible.

The task of collecting "static traces" involves processing the CFG to elicit a collection of paths that are feasible. This is generally accomplished by

```
...
10 FOR H=INT(-16*(V/32)^2+V^2/32+.5) TO 0 STEP -.5
  11 IF INT(H)<>H THEN 13
  12 PRINT H;
  13 L=0
  14 FOR I=1 TO S1
    15 FOR J=0 TO T(I) STEP S2
      16 L=L+S2
      17 IF ABS(H-(.5*(-32)*J^2+V*C^(I-1)*J))>.25 THEN 19
      18 PRINT TAB(L/S2);"0";
      19 NEXT J
    20 J=T(I+1)/2
    21 IF -16*J^2+V*C^(I-1)*J<H THEN 23
    22 NEXT I
  23 PRINT
  24 NEXT H
25 PRINT TAB(1);
...
```

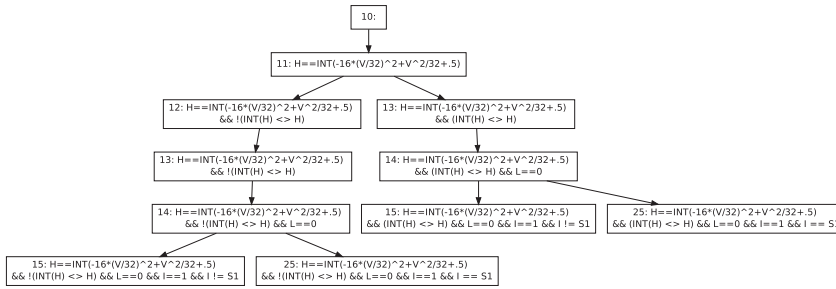


Fig. 15. Small portion of a symbolic execution tree, for a portion of code taken from the "bounce" code in Fig. 6. For the sake of readability the tree only reaches a depth of six execution steps, which suffices to give an intuition of how it is constructed.

adopting static-analysis techniques that are capable of systematically exploring the state-space of the program by traversing the CFG (without actually executing it). Commonly used techniques include abstract interpretation [9], and symbolic execution [27] (which is a specialized form of abstract interpretation).

To provide an intuition of how such techniques work, a brief example of symbolic execution is shown in Fig. 15. In general, abstract interpretation approaches can vary substantially from this, so the purpose here is merely to provide an intuition of what a possible approach looks like (this approach is similar to those taken by Kung et al. [30] and Walkinshaw et al. [54]). A "symbolic execution tree" is constructed by building a tree of all possible paths through the CFG, and associating each node in the tree with a *path condition*. The path condition represents the current constraints on the data variables that must hold at that given point, as deduced from the predicates

and variable assignments that have occurred so far. The feasibility of a path is reflected in the satisfiability of its path condition.

Even from this example, it becomes apparent that the tree expands rapidly. The tree is limited to a depth of 6, but does not even contain the full bodies of the nested for-loops. Each iteration of the outer for-loop would entail ten nodes (three of which are branch-points). As the depth of the tree increases, the number of nodes in the tree increases exponentially.

One way of limiting the expansion of the tree is to use SAT-solvers. These can prevent the expansion of any nodes with path conditions that are unsatisfiable (and so correspond to infeasible program executions). However, this is often of limited use. If the program in question couples loops with non-trivial predicate conditions (e.g., involving floating-point arithmetic) it can become impossible to determine whether particular paths are feasible or not. If loop-termination depends on the values of unknown input parameters, there remains no option but to expand the loop indefinitely (or, in practice, up to some arbitrary limit).

3.4.2 Deriving Models from Static Traces

Once the possible paths through a program have been extracted, it becomes possible to derive behavioral models. Most published approaches focus on FSMs or EFSMs [30, 50, 54, 48]. All techniques use some form of abstract interpretation/symbolic execution to produce static traces (as described above), but vary according to the nature of the state machine produced. Given the variety in approaches, there is no single generic description that can serve to capture how all of these techniques produce their state machines. However, to provide an intuition we focus on one approach that is shared by the work by Kung et al. and Walkinshaw et al. [30, 54]. Both start from an execution tree and produce a state machine using roughly the same approach (although they have different ways of producing the execution trees and seek to derive models for different purposes).

Given an execution tree, the techniques operate in two phases. The first phase is to identify points in the execution tree that correspond to "state transition points"—triggers that indicate the transfer from one state to another. For example, these might be calls to an API (if the target model is to represent the API usage), or method-entry points if the model is to represent the sequential behavior within a class.

Finally, the tree is used to construct the state machine. This can be done very simply; for every pair of annotated states A and B in the tree such that there is a path from A to B (without intermediate annotated states), it is

possible for execution to reach state B from state A. In other words, there is a state transition $A \rightarrow B$.

3.5 Limitations of Static Analysis

Deriving models of software behavior from source code is fundamentally limited by undecidability. Seemingly simple behavioral properties, such as whether a program will terminate, have been proven to be undecidable in the general case. More damning is the proof of Rice's theorem, which shows that the same limitations that give rise to the halting problem also mean that *all* non-trivial behavioral properties are undecidable [46].

Representations that form the basis for static analysis—CFGs, dataflow relations, symbolic execution trees, etc.—are intrinsically conservative. Wherever the feasibility of a branch or a path cannot be decided, static analysis is forced to return a result that allows for *all* possibilities. Thus, any analyses that build upon these foundations tend to be severely hobbled by inaccuracy.

The practical value of techniques that seek to predict dynamic program behavior from source code is severely limited. For languages such as Java, symbolic execution environments do exist, but cannot (at the time of writing) be used "out of the box." Although some systems can be used in certain circumstances at a unit-level, they cannot readily scale to larger systems. Complex data-types and constraints exacerbate this problem even further.

Finally, there is the problem of libraries. Even a trivially small Java program can make extensive use of built-in libraries (`System.out.println`, etc.). Calls to libraries can be instrumental to the behavior of the system under analysis. For example, the question of whether a call to a `Collections` object is a mutator (changes the state of the object) or an observer (merely returns the state) is crucial for determining whether that call should be treated as a *def* or a *use*. This can only be determined by analyzing the source code (or byte-code) of the class in question.

There are only two possible solutions to this. The first solution is to develop all-encompassing static analysis techniques that pull in the respective source code or byte-code for the libraries in question. This can severely affect scalability. The only other option is to produce manual "stubs" to the libraries that summarize the relevant information. This is, for example, carried out for the JavaPathFinder model-checker (and its symbolic execution framework) [51]. This however also has the clear downside that it is prohibitively time consuming.

4. DYNAMIC ANALYSIS

Whereas static analysis is concerned with deriving information from the source code syntax, dynamic analysis is broadly concerned with analyzing the program while it executes. This can involve keeping track of data and control events—either as externally observable input/output, or internal information such as the execution sequence of individual statements or values that are assigned to particular variables.

Whereas static analysis often consumes an overwhelming amount of effort to increase precision by eliminating infeasible predictions, this is not a problem with dynamic analysis. If anything, the challenge with dynamic analysis is the converse [14]. We begin with a set of observed executions, and from this have to make generalizations about program behavior, based on inferences from the given set of executions.

As with static analysis, dynamic analysis encompasses a very broad range of techniques with different purposes. For example, profilers can provide a high-level view of the performance of a programming, debuggers can be used to make explicit the internal state of a program at particular points, and testing can check that a program is returning the correct outputs. This section focusses on a selection of dynamic analysis techniques that are particularly useful from a reverse-engineering perspective.

The chapter is structured as follows. It begins by presenting the process of tracing—the fundamental step of recording the information about program executions, which forms the basis for the subsequent analysis. It then continues to discuss in broad terms the general relationship between the inference of software models, and the general Machine Learning problem of inferring models from given examples. This is followed by the presentation of the more specific challenges of inferring state machines and data models from traces. The section is, as was the case with the static analysis section, finished with a discussion of the intrinsic limitations of dynamic analysis.

4.1 Tracing and the Challenge of Inductive Inference

In broad terms, the goal of dynamic analysis (in the context of reverse engineering) is to produce a model that generalizes some facet of program behavior from a finite sample of program executions. Program executions are recorded by a process known as *tracing*. Tracing can be time and resource consuming, and requires the considered selection of program inputs. Then

there is the challenge of inferring a model that is accurate and complete. This subsection introduces these problems, to provide a sufficiently detailed background for the rest of this section.

4.1.1 Tracing

Software tracing is, in broad terms, the task of recording software executions and storing them for subsequent analysis. Intuitively, the task is simple. To monitor program execution, and to feed the name of every significant event (such as the name of a method call), along with the values of relevant data parameters and state variables, to a text file. Many modern programming languages are accompanied by relatively sophisticated trace generators (c.f., the Eclipse TPTP tracing plugin for Java[3], or the Erlang OTP built-in tracing functions[4]).

A trace is, put simply, a sequence of observations of the system. Each observation might contain an element of control (e.g., a method-entry point, or some other location in the code), and an associated data state (variable values, object-types, etc.). In simple terms, for the purpose of discussion here, we envisage a trace of n observations in the following simple format:

$$T = \langle (p_0, D_0), \ldots, (p_n, D_n) \rangle$$

Each observation takes the form $\langle p, D \rangle$, where p is some point in the code (this same point may feature in several observations within the same trace), and D represents the state of the various relevant data variables at that point.

To produce a program trace, it is first necessary to determine *what* needs to be traced. Tracing is an expensive process; recording information is time consuming, trace data can consume large volumes of storage space, and traces can be time consuming to analyze. The question of what to trace depends on the nature of the analysis to be carried out. The question is critical because recording too much information can lead to traces that become unmanageable, and can lead to unintended side-effects that affect the subsequent behavior of the program (both of these problems are elaborated below).

The default option of recording *everything* at *every* point becomes rapidly intractable for all but the simplest programs. To provide an intuition, we can draw upon an example of a trace in JHotDraw (a Java drawing problem that is popular for case-studies in the Software Engineering community—see Fig. 16). The system (version 5.2) is relatively simple, consisting of 142

[3] http://www.eclipse.org/tptp/.
[4] http://www.erlang.org/doc/.

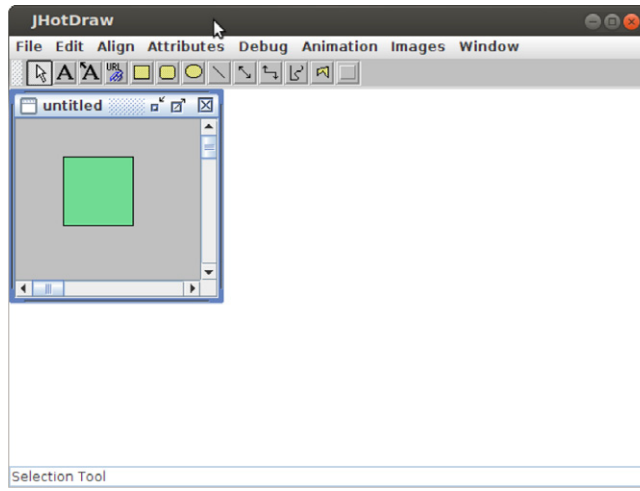


Fig. 16. Screenshot from a simple JHotDraw sample application execution.

classes. However, the trace that captures the sequence of method signatures involved in the (roughly four-second long) process of starting the program, drawing the rectangle shown in the figure, and closing the window, contains a total of 2426 method invocations. And this is still relatively lightweight; the trace only traces method-entry points (omitting individual statements, data values, and object-identities), which would lead to an order-of-magnitude increase in the size of the trace.

This illustrates an important trade-off between the utility of the traces and their scale. On the one hand it is important to collect all of the information that is necessary for the analysis in question. If we are constructing a behavioral model, this should include all of the information at each point during the execution that is relevant to the behavior that is of interest. On the other hand, it is important to avoid an information overload, which could reduce the accuracy of the model, and increase the expense of the analysis.

There is also the question of *which traces* to collect. Programs tend to accept inputs from an infinite domain. Given that dynamic analysis techniques are restricted to a finite sample of these, it is vital that they are in some sense representative of "general" or "routine" program usage. Otherwise the results of the dynamic analysis risk become biased.

This tension between scale and the need for a representative set of traces represents the core of the dynamic analysis challenge. This is exacerbated when the analyst lacks an in-depth familiarity with the program (which

is probable in a reverse-engineering scenario). In reality it is unrealistic to expect the developer to know exactly what information needs to be extracted from which program points. It is also unlikely that they will be aware of (and have the time to exhaustively execute) the inputs that are required to elicit a "typical" set of program executions. Indeed, these factors set out the parameters of the dynamic-analysis challenge; to make-do with the given information, and to nonetheless infer a model that is at least approximately accurate.

4.1.2 The Essential Challenge of Dynamic Analysis

The broad task of reverse-engineering an accurate general model from a (potentially small) selection of traces is challenging. The process of deriving an accurate model is invariably a product of guesswork, where the accuracy of the final result depends on several factors, such as the breadth and availability of traces. This subsection briefly covers the essential reasons that underpin the problem of model inference (and dynamic analysis in general).

Ultimately, a software system can usually produce an infinite number of different executions, taking inputs from an infinite domain, and producing an infinite range of outputs. This means that a given selection of traces is inevitably only a partial representation of the general program behavior. A model that is generated from these traces cannot merely parrot-back the trace-values that have been observed. It has to somehow use the traces to also guess any features of program behavior that might not been explicitly available in the actual traces. In other words, the inference process must include a process of *induction*—of making guesses, based on the given traces, of the more general system behavior that the model should reflect.

As such, many dynamic analysis techniques belong to a family of techniques from the field of Machine Learning, known as *inductive inference* techniques [38]. These tend to be based on some form of statistical reasoning; they might for example derive rules from data variables that occur frequently, or method sequences that are repeated often. Ultimately, any technique that is essentially founded on guesswork is at risk of making mistakes [14]. The given sample of traces might be a poor sample, with misleading patterns that point toward non-existent rules in the system, or that miss out crucial features.

An illustrative example of the induction problem is shown in Fig. 17. Consider the relationship between two numerical variables, where the observed values from a trace are plotted in a scatter plot. Based on this information, an inference technique might guess that the data points are produced by the straight-line function on the left when, in fact, they are produced by the polynomial function on the right.

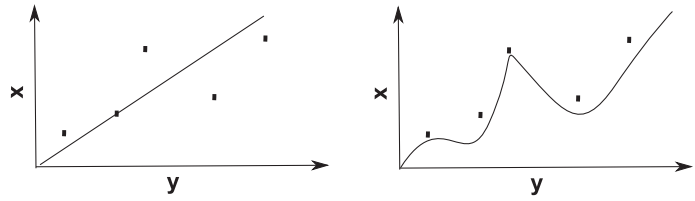


Fig. 17. Illustration of the problem of inferring behavior from limited data points.

The challenge of inferring the correct model places a great premium on the provision of the traces. To reduce the chances that an inference algorithm makes mistakes, it is necessary to make the set of traces as "rich" and "representative" of typical behavior as possible. Finding such a set of traces is of course a significant problem in its own right. The problem of not knowing which inputs have to be exercised to capture every significant facet of behavior is akin to the test-data generation problem [60].

In the broad field of inductive model inference, it is taken for granted that the sample of observations (read "traces" in our context) has been carefully collected to ensure that it is at least representative to some basic extent. This is a necessary precondition for dynamic analysis too [14]. Traces must be selected with care, otherwise it becomes difficult to make any reliable assertions about software behavior, because the model could be skewed by the underlying set of traces.

4.2 Practical Trace Collection Approaches

Depending on the system we seek to reverse-engineer, the challenge of collecting a sufficient set of traces is potentially achievable, despite the problems discussed previously. Two techniques that can be used are briefly presented here. The first assumes that the subject system is an isolated component, where we only have a vague idea of how the input parameters affect program behavior. The second assumes that the subject system is a component embedded within a wider system, where its usage is governed by its surrounding components.

4.2.1 Selecting Inputs by the Category Partition Method

The Category-Partition (CP) method [41] is a popular test set generation technique. The idea is that we have some software system, that accepts as input a set of parameters, each of which has a certain bearing on the output produced by the system. The CP Method provides a framework within

Table 1 Possible categories for push method.

Input	Categories
parameter o	Null, non-null object
lim	Negative, zero, positive
s.size()	Zero, positive

which to select "significant" inputs that ought to collectively cover every important facet of program behavior.

For each parameter, the first task is to identify its "categories." These are ranges of values that (are thought to) yield the same software behavior. So, for example, let us suppose that we want to test the `BoundedStack.push(Object)` method in isolation (shown in Fig. 1). The inputs that would affect the behavior of the method are the parameter object (the obvious input), as well as the current state of the stack—so the value of `lim`, and the size of the underlying stack `s`.

Table 1 shows some possible categories for the push method. A category represents a particular value or range of values that could (at least potentially) individually affect the output of the program. So one might wish to explore what would happen if a parameter is null, or some arbitrary object. The limit `lim` is merely an integer; we are unaware of the bounds, so it makes sense to see what happens if it is set to zero, negative, or positive value. With s.size(), given that s is an array that was instantiated elsewhere, one can surmise that its size cannot be negative. Nonetheless, it still makes sense to establish what happens when it is empty or populated.

Once the categories have been developed, they can be combined in a systematic way to yield "test-frames"—combinations of inputs that systematically exercise the input-space. For the categories in Table 1 this would yield $2 \times 3 \times 2 = 12$ different test frames, which capture every combination of categories. The test set that is obtained is shown in Table 2.

Depending on the number of input parameters, and the number of categories per parameter, systematically generating all possible combinations can result in too many combinations to execute exhaustively. In the context of testing, the problem generally comes down to time; too many test cases mean that testing will take too long. However, in the context of dynamic analysis there is an additional problem of space; each execution is being recorded as a data-trace which, depending on the contents of the trace, can rapidly become intractable.

Table 2 Final input combinations based on categories in Table 1.

lim	o	s.size()
Negative	Null	Zero
Negative	Not null	Zero
Negative	Null	Positive
Negative	Not null	Positive
Positive	Null	Zero
Positive	Not null	Zero
Positive	Null	Positive
Positive	Not null	Positive
Zero	Null	Zero
Zero	Not null	Zero
Zero	Null	Positive
Zero	Not null	Positive

One way to attenuate this problem is to prioritize the test cases. Certain category combinations will be more critical than others—some may elicit a greater variance of program behavior than others. Therefore, if it is going to be impossible to exhaustively execute the test cases, it makes sense to concentrate on the most important ones or, if possible, to order the combinations according to some order of importance, so that the key combinations are guaranteed to be executed.

4.2.2 Selecting Inputs by Exercising the Surrounding System

The category-partition approach makes the big assumption that there is a sufficient amount of a priori knowledge about the system to identify the key input categories. Although this is perhaps a reasonable presumption in the context of software testing, it is improbable that this is always the case in a reverse-engineering scenario. The system is being reverse-engineered precisely because of a *lack* of detailed knowledge about the system in question.

In this situation, the reverse-engineer is seemingly reduced to attempting arbitrary inputs. However, using such an input selection strategy is unlikely to amount to a representative set of traces. Any purely random test-input technique would require an enormous number of tests to produce a truly representative sample of behavior, which would far exceed the time and space limits that any practical reverse-engineer operates in (see point (2) on random input selection below).

There is however often an alternative. Software components rarely operate in isolation. They tend to operate within a broader software framework.

If the software we seek to reverse-engineer is perhaps an abstract data type or a utility class, the reverse-engineer is furnished with a valuable additional source of data: its typical runtime usage.

Obtaining runtime data in this way is straightforward. The component can simply be left within its original environment. Its behavior can then be traced while controlling the inputs to its host system. Whereas identifying the necessary inputs to an isolated, anonymous low-level component can be difficult, identifying inputs to the host system (e.g., via a GUI [20] or a set of user-level test cases) can be (1) much more straightforward and (2) can lead to a set of relatively representative traces.

It is important to note that gathering trace-data in this way places an important caveat on any interpretations that can be drawn from inferred models. In giving up control over the component-level inputs, any executions of the component are highly specific to the broader framework within which they operate. This naturally means that any models that are inferred from the traces are similarly biased.

4.2.3 Random Input Selection

As a last resort, in the absence of any prior knowledge about the categories of inputs or the context in which the system is being used, it is possible to resort to the random selection of inputs. The approach is to be used as a last resort because of two big problems:

1. **Generating random inputs:** For any system that takes non-trivial input parameter types (e.g., complex objects, sequences of button-clicks, etc.) there is no accepted basis for producing inputs that are truly random [22]. This means that there is a danger that the traces will be biased toward those aspects of behavior that happen to be the easiest to execute, potentially missing out important facets of behavior that are triggered by more subtle input combinations.
2. **Number of inputs required:** Even if it is possible to obtain a "truly" random input generation procedure, it usually takes a vast number of random test cases to even approximate the number of tests required to adequately exercise the full range of software behavior. Results from the field of Machine Learning indicate that the number of random tests required is exponential with respect to the complexity of the system (in terms of the state-space) [32].

Nonetheless, there are upsides to adopting this technique. There are numerous automated quasi-random input-generation techniques that will rapidly generate large volumes of random tests. At the time of writing, one of the most popular implementations is the Randoop tool [42] (which is

vulnerable to the first problem mentioned above). This will, for a given Java class, generate extensive numbers of inputs (in the form of JUnit tests), regardless of the complexity of the input parameter objects.

4.3 Data Function Inference

Data function inference aims to infer data-models from traces. As discussed in Section 1, these generally include pre-/post-conditions for functions, and invariants (for objects or loops, etc.). Given a trace, the task is generally to derive a separate specification for each unique point in the code p that is recorded in the trace (i.e., for each method-entry that appears in the trace). The trace is first processed to map each location in the code to the set of data configurations that have been observed at that location. This produces a one-to-many mapping. Subsequently, for each point, an inference algorithm is used to identify a data-model that fits the data.

There are numerous inference algorithms that can be used to identify useful data functions from multiple data observations. These involve numerous general-purpose data-mining algorithms, as well as techniques that are especially tailored to infer models from software traces. Examples of both are shown below.

4.3.1 Using General-Purpose Data Model Inference Algorithms

Perhaps the most straightforward way to infer a data model from traces is to employ general-purpose data mining algorithms for the task. There are hundreds of algorithms [38, 21]. These vary in several dimensions—from their scalability, to their ability to cope with noise, and the readability of the models they infer. Determining which algorithm is best for a particular type of software system or trace is beyond the scope of this chapter. This subsection provides an illustrative example of how a generic data mining algorithm can be applied, to provide an intuition of how such algorithms can be employed in general.

Let us suppose that we want to reverse-engineer the rule that governs the behavior of the `push` function in the `BoundedStack` class in Fig. 1. We begin by running a few executions to trace the relevant variable values in the `push` method throughout. This gives rise to the data in Fig. 18. At this point the dynamic-analysis reverse-engineering challenge is clear. How can we obtain from the data in Fig. 18 a model that describes the general behavior of the push method?

There are numerous general-purpose Data-Mining/Machine Learning algorithms that can be used to analyze such data [38]. For the sake of illustration we use the WEKA data-mining package [21] to analyze the trace. It

o	lim	s.size	return
obj1	5	0	TRUE
obj2	5	1	TRUE
obj3	5	2	TRUE
obj4	5	3	TRUE
obj5	5	4	TRUE
obj6	5	2	TRUE
obj7	5	3	TRUE
obj8	5	4	TRUE
obj9	5	5	FALSE
obj1	2	0	TRUE
obj2	2	1	TRUE
obj3	2	2	FALSE
obj4	2	2	FALSE
obj5	2	2	FALSE
obj6	2	0	TRUE
obj7	2	1	TRUE
obj8	2	2	FALSE
obj9	2	2	FALSE
obj1	100	0	TRUE
obj2	100	1	TRUE
obj3	100	2	TRUE
obj4	100	3	TRUE
obj5	100	4	TRUE
obj6	100	2	TRUE
obj7	100	3	TRUE
obj8	100	4	TRUE
obj9	100	5	TRUE

Fig. 18. Sample trace for the push method, containing 27 trace elements.

contains numerous data mining algorithm implementations, which can be used to derive models from data of the sort shown in Fig. 18. The sheer variety of algorithms is (to an extent) problematic; there are only few guidelines about which type of algorithm suits a given set of data characteristics. In our case, we choose to use the NNge (Non-Nested generalized exemplars) algorithm [37], which has been anecdotally shown to work well for examples with variables of different types.

The rules that are produced by the NNge algorithm are shown in Fig. 19. They hypothesize how the values of `o`, `lim`, and `s.size` affect the outcome

```
class TRUE IF : o in {obj9} ^ lim=100.0 ^ s.size=5.0  (1)
class TRUE IF : o in {obj3} ^ 5.0<=lim<=100.0 ^ s.size=2.0  (2)
class TRUE IF : o in {obj4} ^ 5.0<=lim<=100.0 ^ s.size=3.0  (2)
class TRUE IF : o in {obj1,obj2,obj6,obj7} ^ 2.0<=lim<=100.0 ^ 0.0<=s.size<=3.0  (12)
class TRUE IF : o in {obj5} ^ 5.0<=lim<=100.0 ^ s.size=4.0  (2)
class TRUE IF : o in {obj8} ^ 5.0<=lim<=100.0 ^ s.size=4.0  (2)
class FALSE IF : o in {obj9} ^ 2.0<=lim<=5.0 ^ 2.0<=s.size<=5.0  (2)
class FALSE IF : o in {obj3,obj4,obj5,obj8} ^ lim=2.0 ^ s.size=2.0  (4)
```

Fig. 19. Output from the NNge Algorithm—the numbers in parentheses after each rule indicate the number of trace observations that support a given rule.

(true or false—representing whether an object has been pushed onto the stack or not). For example, if `o` = "obj3," $5 \leq$ `lim` ≤ 100, and `s.size` = 2, the element will be pushed. However if `o`="obj3," `lim`=2, and `s.size`=2, it will not.

Looking at the model, it becomes clear that some of the rules seem inaccurate. They might pertain to a *particular* execution, but not be representative of the program behavior in general. For example, we can tell from the source code that the actual values of `o` do not affect the behavior of the push method. That is, unless `o`=null, but this is not captured by any of the traces. We can also tell from the source code that the rule we are after is much simpler than the one given here; an element is added if `lim` > `s.size`. However, these rules fail to capture this explicit relationship between the variables, instead focusing on the individual relative value ranges.

As discussed above, the problem of generality highlights a fundamental problem in dynamic analysis. If the given set of traces fails to highlight an element of program behavior (e.g., what happens when `o` = null), then this cannot be taken into account by any hypothesis model. If the set of traces is too small, there may simply not be enough evidence for a particular behavioral feature to factor in the model (e.g., that the value of `o` does not really matter).

It is also important to bear in mind that we are only able to tell that the model is inaccurate because we have access to the source code, and can derive this rule because the source code in question is very simple. Of course, in reality, the source code that produces the trace could be inaccessible (e.g., belonging to a closed-source library), or might simply be too complex to readily inspect and understand in this way. Thus, treating the system as a black-box, and making no presumptions about the roles of different variables, given these traces, this model would seem to be a reasonable guess at the rules.

The second criticism (the omission of the seemingly obvious relationship between `lim` and `s.size`) points toward a more general characteristic of data mining algorithms. They *do not assume* that the data was produced by

some computational process. The systems for which they were developed might produce noisy meteorological data, or arbitrary heights and genders of shoppers in a supermarket, or share-prices in a recession. They do not tend to look for explicit, discrete relationships between variables, but instead focus on statistical, probabilistically justified correlations and clusterings, which tend to produce models that seem counterintuitive when the system in question is a discrete logical software system.

4.3.2 Inferring Pre-Conditions, Post-Conditions, and Invariants

This leads us onto a slightly different class of inference techniques—purpose-built software reverse-engineering tools. Unlike generic data-mining and machine learning tools, these make the explicit assumption that groups of variables will either feed into or be the product of some software system.

The most popular tool in this field is Ernst et al.'s Daikon tool [15], which was published over a decade ago. This puts explicit functional relationships between variables at its heart. The tool is equipped with a set of standard rules that might be of interest to a software developer, such as $x > y$, or $a + b + c \leq d$, etc. (Daikon contains about 50 of these relationships).

Such rules are akin to the types of rules described in Section 2.1.1. As such, techniques such as Daikon can be highly effective at reverse-engineering pre-/post-conditions, or invariants that hold for objects or classes. In the original paper, Ernst et al. argued that these techniques could be embedded into the source code as assertions, to ensure that changes do not lead to unintended changes in behavior.

Given a set of traces, Daikon uses its internal list of rules as a checklist, and identifies all of the rules that can fit for a given observation point in the trace. If one rule is a more specific variant of another, it will choose the most specific variant, so that the user is given the most accurate possible set of rules, and is not overloaded with rules that are superfluous.

The tool is demonstrated with respect to our `BoundedStack` example in Fig. 1. The output produced by Daikon is shown in Fig. 20. The type of analysis offered by Daikon is more tailored to source code constructs than the more generic Machine Learning analysis discussed previously. It distinguishes between distinct exit points in the program, and links up the entry and exit conditions. Thus, the statement `this.lim == orig (this.lim)` implies that the value of `lim` remains the same at the entry and exit points. The rules are generally very simple in nature, picking out pertinent inter-variable relationships that remain constant for particular sets of executions.

```
===========================================================================
source.BoundedStack.push(java.lang.Object):::ENTER
o != null
o.getClass() == java.lang.String.class
===========================================================================
source.BoundedStack.push(java.lang.Object):::EXIT17
return == true
===========================================================================
source.BoundedStack.push(java.lang.Object):::EXIT17;condition="return == true"
===========================================================================
source.BoundedStack.push(java.lang.Object):::EXIT21
this.lim one of { 2, 5 }
return == false
===========================================================================
source.BoundedStack.push(java.lang.Object):::EXIT21;condition="not(return == true)"
===========================================================================
source.BoundedStack.push(java.lang.Object):::EXIT
this.lim == orig(this.lim)
this.s == orig(this.s)
(return == false) ==> (this.lim one of { 2, 5 })
(return == true) ==> (this.lim one of { 2, 5, 100 })
===========================================================================
source.BoundedStack.push(java.lang.Object):::EXIT;condition="return == true"
return == true
===========================================================================
source.BoundedStack.push(java.lang.Object):::EXIT;condition="not(return == true)"
this.lim one of { 2, 5 }
return == false
===========================================================================
```

Fig. 20. Daikon output for the push method, using the trace from Fig. 18. The points EXIT17 and EXIT21 represent the two individual return statements. The EXIT point covers both of these return points.

Daikon has proven itself to be a valuable tool for a range of programming activities (notably detecting conditions that can be hard-coded as assertions to prevent program deterioration). There are however downsides too. For one, a rule will only be suggested if it belongs to the list of provided rules. The need to contain every rule that might arise means that the rules have to be relatively simple and generic—staying restricted to elementary relationships between small numbers of variables. Though often useful, the downside is that this could miss out important, more complex rules.

4.4 State Machine Inference

State machines make the sequential behavior of a system explicit. A large number of algorithms have been developed to infer such specifications from traces. The first such approach (which was published 40 years ago [3]) is the *k*-tails approach, which forms the basis for most approaches that are popular today.

Without going into details, state machine inference algorithms tend to operate by a process known as "state merging." The set of traces is first arranged into a "prefix tree"[5], where paths from the root to a branch represent shared prefixes, and bifurcations represent points at which the traces differ. This in itself represents a state machine, albeit one that exactly mirrors the sequences of events embodied in the traces. The state-merging challenge is to identify those states in the tree that could, in fact, be equivalent to each other, and to merge them. Each merge results in a state machine that may contain loops, and may represent a broader range of sequence of events that better capture the possible behavior of the underlying software system.

Figure 21 illustrates the state-merging procedure with respect to a set of traces from the `BoundedStack` system. It starts from the prefix-tree (shown on the left), and then iteratively merges pairs of states until it arrives at the final state machine (shown on the right). The details of specific algorithms are beyond the scope of this chapter[6]; it is enough to know that all algorithms operate on the same premise, of merging pairs of states with similar subsequent behavior.

Recently, several approaches have been developed to infer more complex state machines than simple FSMs. A recent technique by Lorenzoli et al. [36] shows how this technique can be combined with Daikon to produce fully fledged EFSMs (see Section 2.1.3). Here, the Daikon specifications can act as guards to specify when a transition can be executed.

4.5 Limitations of Dynamic Analysis

The limitations of dynamic analysis are in many ways the converse of static analysis [14]. The accuracy of the results is entirely dependent on the provision of a sufficiently "representative" set of traces. If the set of traces is a poor sample, this will produce a behavioral model that only captures a specific facet of program behavior.

This risk has to be weighed up against the intrinsic expense of collecting and processing traces. Traces tend to use a large amount of storage, and become cumbersome to process. Thus, the essential challenge is to attempt to find a compromise, where the set of traces is sufficiently diverse without becoming too unwieldy.

[5] Also referred to as a "trie" in the Information Retrieval community.

[6] The algorithm used here was an implementation of the Blue-Fringe EDSM inference algorithm [31].

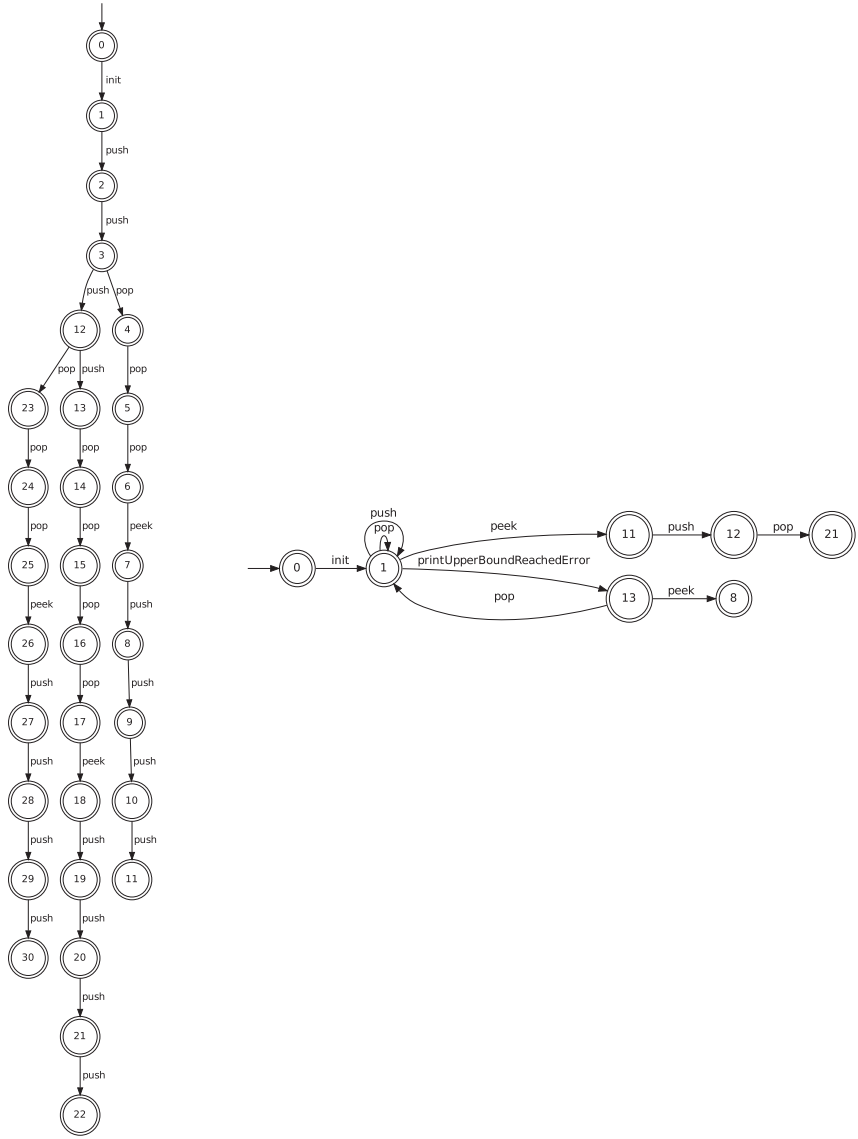


Fig. 21. Set of traces arranged into a prefix-tree at the top, and the final "merged" state machine on the bottom.

Depending on the type of program, this might simply be impossible. If its range of behavior is broad with lots of complex interactions between variables and system events, the accurate inference of a model might simply require more traces than can be obtained and processed in a tractable amount

of time. Alongside the space-constraints, it is also necessary to factor in the amount of human effort that can be required to collect such a set of traces; merely identifying the requisite set of inputs to trigger the necessary program executions can be prohibitively time consuming.

In practice, this limitation is well understood. It is generally accepted that the outputs from dynamic analysis techniques such as Daikon should (at least at first) be interpreted accordingly: as a reflection of the inputs that were used to derive them, not a reflection of the program itself. Depending on the developer's familiarity with the program and confidence in the representativeness of the traces, the results of dynamic analysis can at best be used to corroborate or challenge hypotheses program behavior. Without any strong convictions about the representativeness of the set of traces reverse-engineered models suffer the same problems as those produced by static analysis; they cannot be treated as exact or authoritative.

5. EVALUATING REVERSE-ENGINEERED MODELS

Having surveyed some of the key static and dynamic reverse-engineering techniques and their limitations, one could be excused for feeling somewhat despondent. Numerous techniques exist to generate models, but no guarantees can be made about their accuracy. What is the point in reverse-engineering a model without being able to attribute to it some measure of accuracy? This section surveys some of the approaches that can be used to assess reverse-engineered models.

Essentially there are two approaches that can be used. The first approach is to compare the internal syntax of the model against some reference. The alternative is to ignore the internal syntax, and to treat the model as a black-box, evaluating a model in terms of its behavior.

For the first situation, the choice of approach clearly depends on the type of model. Several approaches have been developed to establish syntactical differences between state machines and sequence diagrams [35, 53, 29, 43]. The basic challenges and techniques for doing so will be covered in Section 5.1.

The second approach of only considering models in terms of their external behavior is more established. Comparing models to actual behavior forms the core of Model-Based Testing [33] and Machine Learning [38]. One popular technique from the latter domain (*k-folds cross validation* [28]) is presented in Section 5.2.

5.1 Evaluating Syntactic Model Accuracy

The process of comparing the internal structures of the two models depends on the type of model. For example, if the model is a state machine, this approach would need to compare the states and transitions of the two models. If it were a sequence diagram, it would need to account for the sequencing of messages, etc.

Although several approaches have been developed that span different types of models [35, 53, 29, 43], this section will focus on comparing state machines, which have formed the basis of most of the research in this area. It is important to note that, even though syntactic notations might vary, the underlying mechanisms can remain essentially the same. For example, the MADMATCH approach proposed by Kpojedo et al. works on any graph-based specification, using the same matching principles [29] (which are similar to the state machine comparison approach discussed in this chapter).

Ultimately, comparing an inferred specification to some reference can be straightforward if the equivalent elements are easy to identify. For example, comparing state machines is easy if the same states can be identified by the same labels. However, the task becomes harder if the two specifications contain many elements where the equivalence relationship is unclear. In a state machine, this would mean that states have to be compared in terms of their surrounding incoming and outgoing states (which in turn have to be compared in terms of their surroundings). This can, depending on the size and complexity of the machines in question, rapidly become an expensive process.

Consequently, the key to efficiently comparing an inferred model to its target lies in identifying the equivalent points. Once these "landmarks" have been identified, the rest of a reverse-engineered model can be assessed in relation to them. This is the principle that underpins most of the automated tools that have been developed for this purpose [35, 53, 29, 43].

Once a selection of states has been matched (which is generally realistic for typical software models), it becomes possible to produce a "diff" of the two machines. An example of such a diff is shown in Fig. 22. The system in question represents the behavior of a CVS client. The reference model is shown at the top. The second model is reverse-engineered from a selection of traces (using a Markov-model inference technique [8]. Finally the diff is shown at the bottom, computed using the LTSDiff algorithm [53]. Here, the dashed edges show the edges and states that are added in the Markov model, the thin edges show the edges that are missing, and the thick edges show the edges and states that are present in both models.

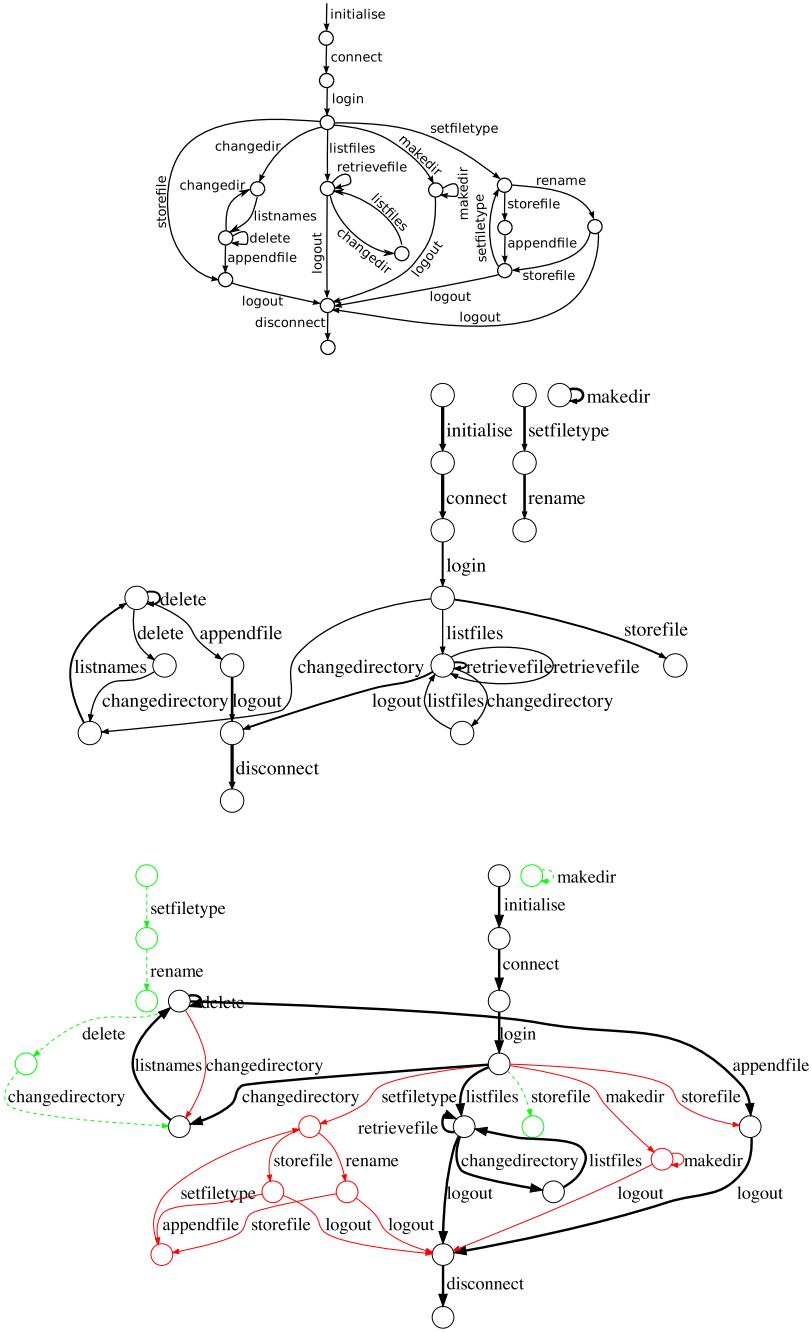


Fig. 22. Illustration of a state machine diff for a CVS client model—as computed by LTSDiff [53].

Table 3 Confusion matrix for classification of elements according to *Ret* and *Rel*.

	Rel	
Ret	**Element ∈ *Rel***	**Element ∉ *Rel***
Element ∈ *Ret*	True Positive (TP)	False Positive (FP)
Element ∉ *Ret*	False Negative (FN)	True Negative (TN)

5.1.1 Measuring the Difference

To quantify the accuracy of a reverse-engineered model, a further step is required to translate the structural difference into some form of "distance measurement." This can be achieved by resorting to standard assessment techniques from the field of Information Retrieval. The task of comparing a reverse-engineered model to some reference model is analogous to the task of comparing a retrieved set of documents to some ideal set.

The task of evaluating information retrieval techniques in this vein has been a subject of research since the 1970s, and has given rise to numerous useful measurements. These are all computed in terms of two sets: *Ret*—the set of elements that have been "retrieved," and *Rel*—the set of elements that are "relevant." These sets can in turn be broken down into four sets: True Positives, True Negatives, False Positives, and False Negatives, and can be represented in terms of a "confusion matrix" shown in Table 3.

Table 4 sets out some of the most popular measurements—Precision, Recall, and the *F*-Measure [44]. These measures can be applied directly to the "diff" information computed on model structures. With respect to the state machine diff example in Fig. 22, it is possible to extract $|TP|$, $|FP|$, and $|FN|$ directly from the bottom state machine. $|TP|$ is the number of bold edges (14), $|FP|$ is the number of thin dashed edges (6), and $|FN|$ is the number of thin undashed edges (13). Substituting these numbers gives rise to a Precision score of 0.7, a Recall score of 0.51, and an *F*-Measure (combined Precision and Recall score) of 0.59. The relatively high precision score shows that the set of transitions contained in the model is largely reflected in the reference model. However, the relatively low recall score indicates that the model is missing a large proportion of edges that should be present.

5.2 Comparing Model Behavior

Depending on the type of model, and on the context in which the model is being used, we might not care whether its internal structure is correct. We may simply wish to ensure that the behavior that is inferred by the model is broadly accurate. A benefit of ignoring the internal structure of the model

Table 4 Classical accuracy measurements.

Measure	Formula	Interpretation
Precision	$\frac{\lvert TP \rvert}{\lvert TP \cup FP \rvert}$	Proportion of elements in *Ret* that are in *Rel*
Recall (Sensitivity)	$\frac{\lvert TP \rvert}{\lvert TP \cup FN \rvert}$	Proportion of elements in *Ret* that are captured in *Rel*
F-Measure	$\frac{2 * Precision * Recall}{Precision + Recall}$	Harmonic Mean between Precision and Recall

is that this does away with the requirement for a "reference model." Given that we ultimately wish to assess the accuracy of the behavior of the reverse-engineered model, the "reference model" can be replaced with the software system itself.

In the absence of any internal structure to resort to, it becomes necessary to resort to observations of behavior—program traces. In this situation, where the model in question is inferred from a set of traces, there are several procedures that can be adopted to assess the accuracy of a model. These are routinely used in the field of Machine Learning.

The most popular approach is known as *k-folds cross validation* [28]. This approach works by dividing the set of traces into k partitions. All bar one of the partitions are used to infer a model, and the partition that was not used to train the model is used to evaluate it (the output of the model is compared to the corresponding output in the traces). This process is repeated k times (until all partitions have been used for evaluation) and the final accuracy is taken to be the average over all folds. This can then be used as an indicator for the accuracy of a model that is inferred from the trace as a whole. Usually a value of $k = 10$ is chosen, since this is the conventional value used for the evaluation of inferred models and has been shown to work well for this purpose [28].

The measurement used for accuracy depends on the type of behavior exhibited by the model. If the behavior consists of sequences of events, several authors have proposed Precision/Recall-based measures [35, 53]. However, if the output is numerical (e.g., the system under analysis computes some continuous numerical function), then the accuracy has to be measured by other means. For example, the outputs produced by the model could be compared to the actual output by computing the correlation between the two (this is what data-mining tools such as WEKA enable [21]).

Of course, this approach relies on the fundamental assumption that the set of traces is at least broadly representative of the full range of software behavior. If the set of traces is incomplete or biased toward a particular feature, then any score will be similarly biased. If the set of traces *is* biased or incomplete and there is no access to a reference model, there really is no suitable basis for establishing whether the inferred model is accurate or not.

6. CONCLUSIONS AND OUTSTANDING CHALLENGES

The task of reverse-engineering behavioral models is an essentially difficult one. The dynamics of software behavior—runtime variable values and their coupling with potential sequences of events—cannot be accurately predicted from the source code alone. The (usually) infinite ranges of inputs and outputs also make it is practically impossible to identify, execute, and collect the necessary runtime data by program execution.

Despite these fundamental barriers, reverse-engineering techniques have often proved to be reasonably successful at reverse-engineering models that are of value to software developers. This is testified by the widespread use in tools such as Daikon, both in academic as well as industrial contexts. Increasingly, reverse-engineering techniques for behavioral models are being integrated into a broad range of routine software-engineering techniques—a trend that is especially evident in the area of software testing [17].

Though promising, there still remain numerous substantial hurdles if such reverse-engineering techniques are to become truly applicable in a generic software development setting. The rest of this section provides a brief overview of some of these hurdles, along with some of the potential research avenues that could lead to progress.

6.1 Accommodating Scale

Reverse-engineering techniques (both static and dynamic) are beset by problems of scalability. Static techniques are currently limited to trivially small systems. Dynamic analysis techniques may not be restricted by source code complexity, but are still limited to systems that are sufficiently simple in terms of the number of traces required to expose their external behavior.

6.1.1 Static Analysis Technique

Static analysis techniques tend to be limited to unit-level analysis, as discussed in Section 3.5. As the volume of source code increases, the number

of interdependencies, control/dataflow relationships, static traces, non-trivial predicate conditions, and other elements that have to be considered rapidly increases to an intractable level. This explains why, for example, techniques that are based upon conservative static techniques such as symbolic execution tend to be limited to single methods or functions, or trivially small subsystems [54].

One solution that has proven to be popular is the integration of additional dynamic information. Even a very limited amount of dynamic information can be used to great effect, by providing simple, limited axioms about program behavior. These nuggets of information can be used to substantially curtail the range of behavior that has to be considered by a static analysis technique to a more manageable amount. At the same time, because only a limited amount of dynamic information is being used, this hybrid approach does not fall prey to the same problems of scale and inaccuracy that are faced by typical dynamic analysis techniques.

6.1.2 Dynamic Analysis Technique

Dynamic analysis techniques face their own challenges of scalability, as discussed in Section 4.5. Traces rapidly become too large to store and process. Merely identifying and executing the requisite inputs can be too time consuming, especially because this tends to be a manual task.

To an extent, this problem can potentially be addressed in an analogous way to the scalability challenges of static analysis. However, this relies on the availability of source code, which undermines one of the core benefits of dynamic analysis (that it does not rely on source code). Such an approach would also (at least to an extent) be subject to the other typical static analysis problems discussed above.

There is however an alternative solution, which does not rely on source code, and has proven to be highly effective in certain Machine Learning settings. Instead of developing model inference techniques that are effectively *passive*—which take arbitrary sets of traces and return some "best guess," the alternative is to put the inference approach in charge of selecting the traces as well as inferring models from them. Such *active* approaches tend to be powerful because they can be highly selective about the traces they use; they can select the traces with the *highest utility*, and thus attain a much higher accuracy from a given number of traces than would be the case if the same trace set were selected by some arbitrary basis.

Such *active learning* techniques are being increasingly applied in the context of software engineering. The QSM state machine inference approach

[11] is one such example, where queries are generated throughout the inference process, and can be answered by the developer. Adaptations of this algorithm have also been successfully applied in a fully automated context, where queries are automatically formulated into tests to the program [55].

6.2 Factoring in Domain Knowledge

Domain knowledge—axiomatic facts about the environment in which the system under analysis operates—is especially difficult to elicit by software analysis. This knowledge is often tacit, but can have a strong bearing on the behavior of a software system.

In a simple example, we might have a system that reacts to mouse-events. As users who are familiar with mice, might know several intuitive rules. For example, a "mouse-down" event cannot happen twice in a row (because a mouse-up event has to happen in between), or the x and y co-ordinates of the mouse should never be negative. Such nuggets of information may be obvious to a reverse-engineer, but might be impossible to derive from analysis of the source code alone.

It is often straightforward to extend conventional inference techniques to incorporate this additional information. For example, previous work by the authors [54] has extended a conventional state machine inference algorithms of the sort discussed in Section 4.4 to incorporate simple factoids about behavior expressed in LTL. Damas et al. [11] have incorporated facts in the form of data constraints.

Of course, the key premise for such techniques is that they rely on the ability of some form of a priori knowledge about the basic context in which a system operates. Even if the user possesses this knowledge, it is still necessary to select which facts are worth incorporating. Certain facts are invariably of greater utility to a reverse-engineering algorithm than others.

There are two apparent sources from which to obtain such facts, which are briefly listed below:

1. **By queries:** Active techniques (as discussed above) can be used to query the reverse-engineer, to enable them to provide their responses in the form of answers to targetted queries. This approach has been explored by the authors in the active variant of their LTL-driven state machine inference algorithm [52].
2. **Analyzing comments instead of code:** Over the past decade there has been a surge in the use of Natural Language Processing techniques to reverse-engineer the sort of domain knowledge that cannot be obtained from analyzing the raw source code syntax. Techniques such as Latent

Semantic Indexing and other forms of identifier analysis [4] have proven to be quite successful at pulling out nuggets of information about the semantics of a software system, and would seem to be an ideal fit for the challenge of augmenting conventional inference algorithms.

6.3 Concurrency and Time-Sensitivity

Concurrency and time-sensitivity represent perhaps the most urgent, but also the sternest challenge for reverse-engineering techniques. The matter is urgent because modern software systems are increasingly developed according to distributed, concurrent architectures. The popularity of languages such as Erlang and Scala that capitalize on lightweight concurrent processes, along with technologies such as multi-core processors and service-oriented architectures, is testament to this.

The task of inferring the underlying rules that determine *what* happens *when* in such systems is particularly challenging. Observable system behavior can be the product of complex interactions between multiple processes. Messages or signals can interleave in numerous unpredictable ways, some of which may have a significant bearing on program behavior. Faced with this huge increase in complexity, attempting to infer an accurate monolithic model of system-level behavior can become practically futile.

Even the "basic" processes of collecting traces that record this behavior become more challenging. Traces no longer merely correspond to a single sequence of events, but multiple streams of communication. Each trace-event corresponds to a signal or message has to be linked to the time it is sent, the time it is received, the IDs of the processes that send and receive the messages, as well as the contents of the message itself. If the system is time-sensitive, there is the added danger that the overhead of recording traces interferes with the system under analysis (especially if the volume of data is large). This hugely amplifies the challenges of storage and analysis that apply to conventional traces.

In practice, the types of reverse-engineering techniques that have been discussed in this chapter could only realistically be applied to individual parts or processes of a concurrent system. Concurrent systems are rarely (if at all) modeled as a single monolithic model. In practice they are broken down, and modeled in terms of their individual processes, following "agent-based" or "process-based" modeling principles. The manner in which the processes communicate with each other tends to be modeled separately, using formalisms such as CCS, or concurrent variants of UML interaction diagrams.

There has been very little work on inferring models of concurrent processes, perhaps because the comparatively straightforward task of inferring non-concurrent models is so challenging in its own right. Even if traces can be collected in an efficient way that does not interfere with the behavior of the system, and models of the individual processes can be accurately inferred from them, there still remains the task of inferring higher-level models of their interactions.

REFERENCES

[1] D.H. Ahl, BASIC computer games: microcomputer edition, Creative Computing, 1978.
[2] A.V. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, Tools, Addison-Wesley, 1986.
[3] A.W. Biermann, J.A. Feldman, On the synthesis of finite-state machines from samples of their behavior, IEEE Trans. Comput C. 21 (1972) 592–597.
[4] David Binkley, Dawn Lawrie, Steve Maex, Christopher Morrell, Identifier length and limited programmer memory, Sci. Comput. Program 74 (7) (2009) 430–445.
[5] S.N. Cant, Brian Henderson-Sellers, D. Ross Jeffery, Application of cognitive complexity metrics to object-oriented programs, JOOP 7 (4) (1994) 52–63.
[6] Kwang-Ting Cheng, A.S. Krishnakumar, Automatic generation of functional vectors using the extended finite state machine model, ACM Trans. Design Autom. Electr. Syst 1 (1) (1996) 57–79.
[7] J. Elliot, Chikofsky, James H. Cross II, Reverse engineering and design recovery: a taxonomy, IEEE Softw. 7 (1) (1990) 13–17.
[8] J. Cook, A. Wolf, Discovering models of software processes from event-based data, ACM Trans. Softw. Eng. Methodol. 7 (3) (1998) 215–249.
[9] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the Fourth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM, 1977, pp. 238–252.
[10] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Trans. Progr. Lang. Syst. (TOPLAS) 13 (4) (1991) 451–490.
[11] Christophe Damas, Bernard Lambeau, Pierre Dupont, Axel van Lamsweerde, Generating annotated behavior models from end-user scenarios, IEEE Trans. Softw. Eng. 31 (12) (2005) 1056–1073.
[12] W. Damm, D. Harel, LCSs: breathing life into message sequence charts, Formal Methods Syst. Design 19 (1) (2001) 45–80.
[13] M.B. Dwyer, J. Hatcliff, et al, Bogor: an extensible and highly-modular software model checking framework, in: ACM SIGSOFT Software Engineering Notes, vol. 28, ACM, 2003, pp. 267–276.
[14] Michael D. Ernst, Static and dynamic analysis: synergy and duality, in: Jonathan Cook (Ed.), WODA 2003: ICSE Workshop on Dynamic Analysis, Portland, OR, New Mexico State University, May 2003, pp. 24–27.
[15] Michael D. Ernst, Jake Cockrell, William G. Griswold, David Notkin, Dynamically discovering likely program invariants to support program evolution, IEEE Trans. Softw. Eng. 27 (2) (2001) 1–25.
[16] J. Ferrante, K.J. Ottenstein, J.D. Warren, The program dependence graph and its use in optimization, ACM Trans. Progr. Lang. Syst. (TOPLAS) 9 (3) (1987) 319–349.
[17] Gordon Fraser, Neil Walkinshaw, Behaviorally adequate software testing, in: Giuliano Antoniol, Antonia Bertolino, Yvan Labiche (Eds.), ICST, IEEE, 2012, pp. 300–309.

[18] David Grove, Greg DeFouw, Jeffrey Dean, Craig Chambers, Call graph construction in object-oriented languages, in: Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97), 1997, pp. 108–124.
[19] Gurevich, Sequential abstract-state machines capture sequential algorithms, ACM Trans. Comput. Logic 1 (1) (2000) 77–111.
[20] D.R. Hackner, A.M. Memon, Test case generator for guitar, in: Companion of the 30th International Conference on Software Engineering, ACM, 2008, pp. 959–960.
[21] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The weka data mining software: an update, ACM SIGKDD Explorations Newslett. 11 (1) (2009) 10–18.
[22] R. Hamlet, Random testing, in: Encyclopedia of Software Engineering, Wiley, 1994.
[23] C.A.R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580, 583.
[24] Mike Holcombe, An integrated methodology for the specification, verification and testing of systems, Softw. Test. Verif. Reliab. 3(3/4) (1993) 149–163.
[25] Message sequence chart (MSC), ITU-T Recommendation Z.120, International Telecommunications Union, November 1996.
[26] Daniel Jackson, Software Abstractions: Logic, Language, and Analysis, MIT Press, 2006.
[27] J.C. King, Symbolic execution and program testing, Commun. ACM 19 (7) (1976) 385–394.
[28] Ron Kohavi, A study of cross-validation and bootstrap for accuracy estimation and model selection, in: Chris S. Mellish, (Ed.), Proceedings of the 14th International Joint Conference on Artificial Intelligence, Morgan Kaufmann, San Mateo, August 1995, pp. 1137–1145.
[29] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Giuliano Antoniol, Yann-Gael Gueheneuc, Madmatch: many-to-many approximate diagram matching for design comparison, IEEE Trans. Softw. Eng. 99 (1) (2013) (PrePrints).
[30] D. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, On object state testing, in: Proceedings of the 18th Annual International Computer Software and Applications Conference, 1994. COMPSAC 94, IEEE, 1994, pp. 222–227.
[31] K.J. Lang, B.A. Pearlmutter, R.A. Price, Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm, in: V. Honavar, G. Slutzki (Eds.), Grammatical Inference Fourth International Colloquium, ICGI-98, vol. 1433 of LNCS/LNAI, Springer, 1998, pp. 1–12.
[32] K.J. Lang, Random DFA's can be approximately learned from sparse uniform examples, in: Proceedings of the Fifth Annual Workshop on Computational Learning Theory, ACM, 1992, pp. 45–52.
[33] D. Lee, M. Yannakakis, Principles and methods of testing finite state machines—a survey, Proc. IEEE 84 (1996) 1090–1126.
[34] Stanley Letovsky, Elliot Soloway, Delocalized plans and program comprehension, IEEE Softw. 3 (3) (1986) 41–49.
[35] David Lo and Siau-Cheng Khoo, QUARK: empirical assessment of automaton-based specification miners, in: WCRE, IEEE Computer Society, 2006, pp. 51–60.
[36] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè, Automatic generation of software behavioral models, in: Wilhelm Schäfer, Matthew B. Dwyer, Volker Gruhn (Eds.), ICSE, ACM, 2008, pp. 501–510.
[37] B. Martin, Instance-based learning: nearest neighbour with generalisation, PhD thesis, University of Waikato, 1995.
[38] Tom Mitchell, Machine Learning, McGraw-Hill, 1997.
[39] E.F. Moore, Gedanken-experiments on sequential machines, in: Automata Studies, number 34 in Annals of Mathematics Studies, Princeton University Press, Princeton, NJ, 1956, pp. 129–153.

[40] Object Management Group, Framingham, Massachusetts. OMG Unified Modeling Language Specification Version 1.3, June 1999.
[41] T.J. Ostrand, M.J. Balcer, The category-partition method for specifying and generating fuctional tests, Commun. ACM 31 (6) (1988) 676–686.
[42] C. Pacheco, M.D. Ernst, Randoop: feedback-directed random testing for java, in: Conference on Object Oriented Programming Systems Languages and Applications: Companion to the 22 nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications Companion, vol. 21, 2007, pp. 815–816.
[43] Michael Pradel, Philipp Bichsel, Thomas R. Gross, A framework for the evaluation of specification miners based on finite state machines, in: ICSM, IEEE Computer Society, 2010, pp. 1–10.
[44] Cornelis J. Van, Rijsbergen, Information Retrieval, Butterworths, 1979.
[45] Jr. Hartley Rogers, Theory of recursive functions and effective computability, in: McGraw-Hill Series in Higher Mathematics, McGraw-Hill Book Company, New York-St. Louis-San Francisco-Toronto-London-Sydney-Hamburg, 1967.
[46] A.L. Rosenberg, Introduction, The Pillars of Computation Theory, Springer, 2010, pp. 3–12.
[47] B.G. Ryder, Constructing the call graph of a program, IEEE Trans. Softw. Eng. 5 (3) (1979) 216–226.
[48] S. Shoham, E. Yahav, S.J. Fink, M. Pistoia, Static specification mining using automata-based abstractions, IEEE Trans. Softw. Eng. 34 (5) (2008) 651–666.
[49] Ian Sommerville, Software Engineering, 7th ed., Addison-Wesley, May 2004.
[50] P. Tonella, A. Potrich, Reverse Engineering of Object Oriented Code, Springer, 2004.
[51] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, Flavio Lerda, Model checking programs, Automat. Softw. Eng. 10 (2) (2003) 203–232.
[52] N. Walkinshaw, K. Bogdanov, Inferring finite-state models with temporal constraints, in: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, 2008, pp. 248–257.
[53] N. Walkinshaw, K. Bogdanov, Automated comparison of state-based software models in terms of their language and structure, ACM Trans. Softw. Eng. Methodol. (TOSEM) 2 (22) (2013).
[54] N. Walkinshaw, K. Bogdanov, S. Ali, M. Holcombe, Automated discovery of state transitions and their functions in source code, Softw. Test. Verif. Reliab. 18 (2) (2008) 99–121.
[55] Neil Walkinshaw, John Derrick, Qiang Guo, Iterative refinement of reverse-engineered models by model-based testing, in: Ana Cavalcanti, Dennis Dams (Eds.), FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2–9, 2009, Proceedings of the Lecture Notes in Computer Science, vol. 5850, Springer, 2009, pp. 305–320.
[56] Neil Walkinshaw, Marc Roper, Murray Wood, Feature location and extraction using landmarks and barriers, in: ICSM, IEEE, 2007, pp. 54–63.
[57] M. Weiser, Program slicing, in: Proceedings of the Fifth International Conference on Software Engineering, IEEE Press, 1981, pp. 439–449.
[58] M. Weiser, Programmers use slices when debugging, Commun. ACM 25 (7) (1982) 446–452.
[59] Jim Woodcock, Jim Davies, Using Z: specification, refinement, and proof, vol. 1, Prentice Hall, 1996.
[60] H. Zhu, P. Hall, J. May, Inductive inference and software testing, J. Softw. Test. Verif. Reliab. 2 (2) (1993) 69–81.

ABOUT THE AUTHORS

Neil Walkinshaw was awarded a BSc. Hons in Computer Science from The University of Sheffield in 2002, and a Ph.D. in Computer Science from the University of Strathclyde in 2006. He subsequently returned to Sheffield, where he worked for five years as a postdoctoral researcher with the Verification and Testing group. In 2010 he took up a Lectureship with the Department of Computer Science at The University of Leicester. His main interests lie in software analysis. These range from low-level static and dynamic source code analysis to model inference, testing, and software modularisation.