4

Techniques for Statistical Debugging

What is luck? Luck is probability taken personally. It is the excitement of bad math.

–Penn Jillette

Thus far we have focused on techniques for collecting sparsely sampled data from large numbers of users. However, this data is only as good as the sense we can make of it. This chapter presents several techniques for using sparsely sampled data to isolate the causes of bugs.

Sampled data is terribly incomplete. With 1/100 sampling, 99% of everything that happens is not even seen. Thus, we do not give strict causes and effects as one might look for using a symbolic debugger. Instead we use statistical models to identify those behaviors that tend to be strongly predictive of failure over many runs. We refer to this body of techniques as *statistical debugging*. Statistical debugging reaps the benefits of the Bernoulli sampling transformation developed in Sect. 2.1.1. While the data is incomplete, it is incomplete in a fair, statistically unbiased way. Thus the observed data is a noisy but representative sample of the complete behavior, and failure trends identified in the former are equally applicable to the later.

Section 4.1 defines some basic notation and terminology that we will use throughout the remainder of this chapter. In Sect. 4.2 we describe an algorithm for isolating single, deterministic bugs using a process of elimination. Section 4.3 extends our scope to non-deterministic bugs using a general-purpose statistical regression model. This approach has certain limitations, which we discuss in greater depth in Sect. 4.3.4 and Sect. 4.4. Better understanding of these limitations leads us to develop an improved algorithm in Sect. 4.5 that combines statistical ranking techniques with an iterative bug elimination process to manage multiple unknown deterministic and non-deterministic bugs. The ranking and iterative elimination is our best known algorithm to date. Section 4.6 offers several case studies demonstrating how the algorithm has been used to successfully isolate both known and previously unknown bugs in real applications.

4.1 Notation and Terminology

Let $\mathcal{P}$ represent the set of all fundamental and inferred predicates for a given program. A feedback report R consists of one bit indicating whether a run of the program

B. Liblit: Cooperative Bug Isolation, LNCS 4440, pp. 55–88, 2007.

succeeded or failed, as well as a vector with one counter for each predicate $P \in \mathcal{P}$. Let $R(P)$ represent the counter value for P in R. If P is observed to be true during at least once run R, then $R(P) > 0$; if P is never observed to be true during run R, then $R(P) = 0$.

Predicates arise from instrumentation sites. Let S be an instrumentation site. As a notational shorthand, define the count of a site $R(S)$ to be the sum of the counts of its constituent fundamental predicates. Note that for all of the instrumentation schemes described in Sect. 2.2, the set of fundamental predicates arising from a site form a partition of the set of all possible program states at that site. Thus, one observation at a site S entails one true observation of exactly one fundamental predicate $P \in S$. Conversely, if $R(S) = 0$, then site S must never have been observed.

Let B denote a bug (i.e., something that causes incorrect behavior in a program). We use $\mathcal{B}$ to denote a *bug profile*, i.e., a set of failing runs (feedback reports) that share a cause of failure. The meaning becomes clear in context. The union of all bug profiles is exactly the set of failing runs, but note that $\mathcal{B}_i \cap \mathcal{B}_j \neq \emptyset$ in general; more than one bug can occur in some runs.

A predicate P is a *bug predictor* (or simply a *predictor*) of bug B if whenever $R(P) > 0$ then it is statistically likely that $R \in \mathcal{B}$ (see Sect. 4.5.1). The goal of statistical debugging is to select a small subset $\mathcal{A} \subseteq \mathcal{P}$ such that $\mathcal{A}$ has predictors of all bugs. Ideally we would also like to rank the predictors in $\mathcal{A}$ from the most to least important according to some reasonable definition of "importance." The set $\mathcal{A}$ and associated metrics are then available to engineers to help speed the process of finding and fixing the most serious bugs.

It is occasionally useful to distinguish deterministic from non-deterministic bugs. A bug is *deterministic* with respect to a predicate P if whenever P is true, the program is guaranteed to crash at some future point. A bug is *non-deterministic* with respect to a set of program predicates if it is not deterministic for any predicate in the set (i.e., none of the considered predicates perfectly predicts program crashes).

4.2 Predicate Elimination

We begin with automatic isolation of deterministic bugs with the additional simplifying assumption that each program under analysis has only one bug. Deterministic bugs are quite common, though they are generally easier to find and fix using any method than non-deterministic bugs (see Sect. 4.3).

4.2.1 Instrumentation Strategy

As a case study in finding deterministic bugs we take release 1.2 of the CCRYPT encryption tool. This version is known to contain a bug that involves overwriting existing files. If the user responds to a confirmation prompt with `EOF` rather than `yes` or `no`, CCRYPT crashes.

The `EOF` sensitivity suggests that the problem has something to do with CCRYPT's interactions with standard file operations. In C, these functions commonly return