Experimental Program Analysis[*]

Joseph R. Ruthruff, Sebastian Elbaum, and Gregg Rothermel[†]

Abstract

Program analysis techniques are used by software engineers to deduce and infer characteristics of software systems. Recent research has suggested that certain program analysis techniques can be formulated as formal experiments. This article reports the results of research exploring this suggestion. Building on principles and methodologies underlying the use of experimentation in other fields, we provide descriptive and operational definitions of experimental program analysis, illustrate them by example, and describe several differences between experimental program analysis and experimentation in other fields. We also explore the applicability of experimental program analysis to three software engineering problems: program transformation, program debugging, and program understanding. Our findings indicate that experimental program analysis techniques can provide new and potentially improved solutions to these problems, and suggest that experimental program analysis offers a promising new direction for program analysis research.

Keywords: experimental program analysis, program analysis, experimentation

1 Introduction

Program analysis techniques analyze software systems to collect, deduce, or infer specific information about those systems. The resulting information typically involves system properties and attributes such as data dependencies, control dependencies, invariants, anomalous behavior, reliability, or conformance to specifications. This information supports various software engineering activities such as testing, fault localization, impact analysis, and program understanding. Researchers who are investigating these and other activities continue to seek new program analysis techniques that can address software engineering problems cost-effectively, and continue to seek ways to improve existing techniques.

Researchers for some time have harnessed principles of experimentation to aid program analysis techniques (e.g., [6, 15, 35, 49].) Zeller recognized that such program analysis techniques might be able to establish causality relationships between system variables of interest in cases in which standard analyses have not succeeded [50]. We further argued that such techniques might also be able to more cost-effectively draw inferences about properties of software systems to characterize them [35].

Anyone who has spent time debugging a program will recognize characteristics that are experimental in nature. Debuggers routinely form hypotheses about the causes of failures, conduct program runs (in

[*]This article is a revised and expanded version of a paper presented at the *ACM SIGSOFT International Symposium on Software Testing and Analysis*, July 2006 [37].

[†]Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, Nebraska 68588-0115, U.S.A., {ruthruff, elbaum, grother}@cse.unl.edu

which factors that might affect the run other than the effect being investigated are controlled) to confirm or reject these hypotheses, and based on the results of these "experiments," draw conclusions or create new hypotheses about the cause of the fault.

The "experimental" nature of this approach is reflected (in whole or in part) in existing program analysis techniques aimed at fault localization and debugging. For example, HOWCOME [49] is a tool intended to help engineers localize the cause of an observed program failure f in a failing execution e_f. HOWCOME attempts to isolate the minimal relevant variable value differences in program states in order to create "cause-effect chains" describing why f occurred. To do this, HOWCOME conducts an experiment where a subset of e_f's variable values are applied to the corresponding variables in a passing execution e_p to "test" an hypothesis regarding whether the applied changes reproduce f. If e_p with the applied value subset "fails" this test (by not reproducing f), then a different value subset treatment is tested. If the subset "passes" the test (by reproducing f), then a subset of those incriminating variable values is considered. This process continues exploring different hypotheses until no further subsets can be formed or can reproduce the failure.

While the use of principles of experimentation by program analysis techniques has increased in recent years, there remain many approaches by which techniques could draw on research in the statistical and experiment design literature to improve their efficiency or effectiveness. These approaches include (1) methodologies for manipulating independent variables of interest to test their effects on dependent variables, (2) procedures for conducting and adjusting hypothesis tests in program analysis contexts, (3) strategies for systematically controlling sources of variation during these tests, (4) experiment designs and sampling techniques to reduce the costs of experimentation, and (5) mechanisms to generate confidence measures in the reliability and validity of the results. To date, however, the opportunities offered by such approaches have not been pursued rigorously, with the support of a theory of experimental program analysis, by the program analysis research community. As a result, we believe that many program analysis techniques have not advanced to the degree that they could have.

This article takes a first step in formalizing an experimental program analysis paradigm, and demonstrating the potential utility of experimental program analysis, through three contributions. First, we argue that a class of program analysis approaches exists whose members are inherently experimental in nature. By this, we mean that these techniques can be characterized in terms of guidelines and methodologies defined and practiced within the long-established paradigm of experimentation. Building on this characterization, we present an operational definition of a paradigm for *experimental program analysis*, and we show how analysis techniques can be characterized in terms of this paradigm.

Second, we demonstrate how our formalization of the experimental program analysis paradigm can help researchers identify limitations of analysis techniques, improve existing program analysis techniques, and create new experimental program analysis techniques. We also show that techniques following this paradigm can approach program analysis problems in new ways. These results suggest that our formalization can help researchers use experimental program analysis effectively in various ways in which it has not previously been considered.

Third, we examine the applicability and potential cost-effectiveness of experimental program analysis by considering its use with respect to three well-researched software engineering problems: program transformation, program debugging, and program understanding. We consider each problem in detail, discuss the use of experimental program analysis techniques to address that problem, and explain how that use can help researchers tackle some of that problem's more difficult challenges. For each problem, we present one specific experimental program analysis technique in detail, and illustrate the use of that technique through an empirical study. The results of these studies indicate that experimental program analysis techniques can contribute to solving these problems in more cost-effective ways.

The remainder of this article proceeds as follows. Section 2 provides an overview of experimentation and relevant concepts. Section 3 presents our definitions of experimental program analysis, illustrates them by example, and discusses applications for them in practice. Section 4 discusses several exploitable differences between experimental program analysis and experimentation in other fields. Section 5 presents three experimental program analysis techniques, and reports an empirical study investigating each technique's effectiveness and applicability. Section 6 describes related work, and Section 7 concludes.

2 Background

The field of empirical research methods is mature, and has been well-discussed in various research monographs (e.g., [5, 24, 29, 34, 39, 44, 46, 47]). Because experimental program analysis techniques draw on empirical research methods to conduct their analyses, in this section we distill, from these monographs, an overview of the empirical method. In this presentation, we focus on *material about experiments that is most relevant to a general understanding of experimental program analysis*, and is discussed as being important to experimentation in the foregoing monographs.[1]

The initial step in any scientific endeavor in which a conjecture is meant to be tested using a set of collected observations is the **recognition and statement of the problem**. This activity involves formulating *research questions* that define the purpose and scope of the experiment, identifying the phenomena of interest, and possibly forming conjectures regarding likely answers to the questions, or limitations on those answers. Research questions can be exploratory, descriptive, or explanatory – attempting not just to establish causality but also to establish relationships and characterize a population [24, 29, 34, 44]. As part of this step, the investigator also identifies the target *population* to be studied, and on which conclusions will be drawn.

[1]In practice, experiments take on different forms in different domains. The overview presented is a generalization from the cited sources, which we later refer to as "traditional experimentation," and suffices for our subsequent discussion.

Depending on the outcome of this first step, as well as the conditions under which the investigation will take place, different strategies (e.g., case studies, surveys, experiments) may be employed to answer the formulated research questions. Conditions for selecting strategies may include the desired level of control over extraneous factors, the available resources, and the need for generalization. These strategies have different features and involve different activities.

In the case of experiments — the design strategy of interest in this article — scientists seek to *test hypothesized relationships between independent and dependent variables by manipulating the independent variables* through a set of purposeful changes, while carefully controlling extraneous conditions that might influence the dependent variable of interest. In general, when considering experiments, investigators must perform four distinct (and often interleaved) activities [29, 34]:

(1) Selection of independent and dependent variables. This activity involves the identification of the *factors* that might influence the outcome of the tests that will later be conducted on the identified population. The investigator must isolate the factors that will be manipulated (through purposeful changes) in investigating the population and testing the hypotheses; these are referred to as *independent variables*. Other factors that are not manipulated, but whose effects are controlled for by ensuring that they do not change in a manner that could confound the effects of the independent variable's variations, are referred to as *fixed variables*. Variables whose effects cannot be completely controlled for, or variables that are simply not considered in the experiment design, are *nuisance variables*. The response or *dependent variables* measure the effect of the variations on the independent variables on the population.

(2) Choice of experiment design. Experiment design choice is concerned with structuring variables and data so that conjectures can be appropriately evaluated with as much power and as little cost as possible. The process begins with the investigator choosing, from the scales and ranges of the independent variables, specific levels of interest as *treatments* for the experiment. Next, the investigator formalizes the conjectures about the potential effects of the treatments on the dependent variables through a *formulation of hypotheses*. To reduce costs, the investigator must determine how to *sample the population* while maintaining the generalization power of the experiment's findings. The investigator then decides how to *assign the selected treatments* to the sampled units to efficiently maximize the power of the experiment, while controlling the fixed variables and reducing the potential impact of the nuisance variables, so that meaningful observations can be obtained.

(3) Performing the experiment. This activity requires the codification and pursuit of specified *procedures* that properly gather observations to test the target hypotheses. These procedures are supposed to reduce the chance that the dependent variables will be affected by factors other than the independent variables. Thus, it is important that the investigator regularly monitor the implementation and execution of the experiment procedures to reduce the chances of generating effects by such extraneous factors.

(4) Analysis and interpretation of data. An initial *data analysis* often includes the computation of measures of central tendency and dispersion that characterize the data, and might help investigators identify anomalies worth revisiting. More formal analysis includes statistical significance assessments regarding the effect of the treatments on the dependent variables. Such assessments provide measures of confidence in the reliability and validity of the results and help interpretation proceed in an objective and non-biased manner. The data analysis allows the investigator to *test the hypotheses* to evaluate the effect of the treatments. The interpretation of the hypothesis testing activity during an *interim analysis* can lead to further hypothesis testing within the same experiment, either through the continued testing of current hypotheses or the formulation of new hypotheses. If more data is needed to evaluate the hypotheses, then the process can return to the experiment design stage so that such data can be obtained; this establishes a "feedback loop" in which continued testing may take place during the course of the experiment. If more data is not needed, then the investigation proceeds to the final stage in the process of experimentation.

The final step when performing any empirical study, regardless of the research strategy utilized, is the offering of **conclusions and recommendations**. An investigator summarizes the findings through *final conclusions* that are within the scope of the research questions and the limitations of the research strategy. However, studies are rarely performed in isolation, so these conclusions are often joined with recommendations that might include guidance toward the performance of replicated studies to validate or generalize the conclusions, or suggestions for the exploration of other conjectures.

As an overall issue, the actions taken during each step may result in limitations being imposed on the conclusions that can be drawn from those studies. For example, in experiments, the sample taken from the population may not be representative of the population for reasons pertaining to sample size or the area from which the sample was taken, the inferential analysis may be not be appropriate due to assumptions about the population not being met, the experiment design may not offer enough power to place sufficient confidence in the conclusions, or the nuisance variables may cause confounding effects that bias the results. These limitations are formally known as *threats to the experiment's validity.* The types of threats that we consider in this article are those identified by Trochim [44], who provides a general discussion of validity evaluation and a classification of four types of validity threats:

1. *Threats to external validity* involve limitations on the ability to generalize the results of the experiment. They do not concern the validity of the results from the experiment in question – only the applicability of those results to other settings, contexts, or the population the experiment is investigating.

2. *Threats to internal validity*, rather than dealing with how the experiment relates to contexts outside of the investigation as do external validity threats, concern limitations, or confounding and biasing factors, that can influence the reliability of the results obtained within the experiment.

3. *Threats to construct validity* deal with the adequacy and limitations of an experiment's dependent variables. They are directly tied to the sufficiency of the constructs chosen to evaluate the effect of the manipulations of the independent variables in the context of the sample.

4. *Threats to conclusion validity* concern limitations on the power or legitimacy of an experiment's conclusions. These validity threats suggest ways in which a stronger, and perhaps more accurate, experiment might be designed.

The designer of an experiment must consider these threats when designing each stage of an experiment, and must interpret the conclusions drawn at the end of the experiment in light of these validity threats.

3 Experimental Program Analysis

We now define and discuss experimental program analysis, drawing on the overview of principles of experiment design presented in Section 2. As a vehicle for our discussion, we illustrate the concepts that we present through an existing program analysis technique that (as we shall show) is aptly described as an "experimental program analysis" (EPA) technique – the technique implemented by HOWCOME [49].

We first descriptively define experimental program analysis:

> *Experimental program analysis* is the evolving process of manipulating program factors under controlled conditions in order to characterize or explain the effect of the manipulated factors on an aspect of the program.

In Section 1, we cited the HOWCOME tool as an example of a concrete technique that fits into our view of experimental program analysis. With our descriptive definition of the paradigm, this should become clearer, as HOWCOME can be related to this definition as follows:

- HOWCOME operates through an *evolving process* of systematically narrowing the consideration of variable values relevant to a program failure f, and eliminating those deemed irrelevant.
- The *program manipulations* that are made during this process are the variable value changes that are made to program states in e_f.
- These manipulations are made *in a controlled manner* such that only a selected subset of variable values of interest are manipulated and tested at a specific time.
- In the end, the goal of HOWCOME is to *explain the effect of these manipulations* on e_f in the form of a cause-effect chain that relates variable values that are relevant to f.

- Despite the controls used by HOWCOME, validity threats still limit the conclusions that can be drawn from the technique. For example, the presence of multiple faults in the source code may confound results by inducing different failure circumstances, while dependencies between variables may cause extra, irrelevant variable values to be reported in cause-effect chains.

There are several aspects of this descriptive definition that merit elaboration:

- First, one important characteristic of experimental program analysis is the notion of **manipulating program factors**. The program factors manipulated by EPA techniques are either concrete representations of the program such as source code and program input, or factors and byproducts related to the program's execution such as environment variables, program executions, and program states. These manipulations are made to learn about their effect on an aspect of the program that is of interest. Viewed through the lens of traditional experimentation [29, 34], these program factors correspond to independent variables that are manipulated during the experiments conducted by EPA techniques to perform program analysis. Learning about the effect of these manipulations is done (for the purposes of program analysis) by testing hypotheses regarding the effect of the manipulations on the aspect of the program of interest, with proper controls on the conditions of these tests. A specific manipulation being tested can be viewed as a treatment, whose effect on the program is the subject of the hypothesis and evaluated through hypothesis testing.

- Second, the manipulation of these independent variables occurs **under controlled conditions**. "Controlled conditions" refers to the idea that experiments require that the manipulated independent variables be the only factors that will affect the dependent variables, and that nuisance variables do not confound the effect of the treatments and limit the conclusions that can be drawn from the experiment.

- Third, experimental program analysis is performed **to characterize or explain the effect of the manipulated program factors on an aspect of the program**. The outcome of an EPA technique is a description or assessment of a population reflecting a program aspect of interest (e.g., "what is the program's potential behavior?"), or the determination of likely relationships between treatments and dependent variables (e.g., "what inputs are making the program fail?"). Most EPA techniques, like experiments in other fields, operate on samples of populations, leading to answers that are not absolutely certain, but rather, highly probable.

- Fourth, experimental program analysis involves an **evolving process** of manipulating program factors as independent variables. For EPA techniques to build a body of knowledge about a program, it is often necessary to conduct multiple experiments in sequence, changing the design of later experiments by leveraging findings of prior ones; for example, by re-sampling the population and refining hypotheses.

These evolving experiments allow the results of later experiments to converge to a desirable or useful outcome. An experimental program analysis process is then necessary to manage the experiments' evolution, including a feedback loop enabling the utilization of previous findings to guide the later experiments, and the conditions under which they will be conducted.

While our descriptive definition helps us clarify the ways in which techniques like HOWCOME are experimental program analysis techniques, we also wish to support *five key operations* related to experimental program analysis:

- *Providing a means for classifying techniques as EPA techniques* and facilitating an understanding of the formal experiments conducted by these techniques.
- Providing a "recipe" for *creating new EPA techniques* to cost-effectively solve problems that have not been adequately addressed by traditional program analysis techniques.
- Providing support to help researchers easily *assess the limitations of EPA techniques.*
- Suggesting opportunities for leveraging advanced procedures for designing and conducting effective and efficient experiments, thereby supporting the *improvement of EPA techniques.*
- Exposing features required by experimentation that can be encoded in algorithms, thereby *enhancing automatability within experimental program analysis.*

To provide this type of support and further elucidate our descriptive definition of experimental program analysis, we augment it with an operational definition, which we present in tabular form in Table 1. We term this definition "operational" because it facilitates the *use* of experimental program analysis.

Because EPA techniques conduct experiments to analyze programs, Table 1 is organized in terms of the experimentation guidelines presented in Section 2. The fourth column in Table 1 shows how the HOWCOME technique relates to the various "features" in the operational definition. Features whose role in experimental program analysis can be automated are in gray rows, while features in white rows are manually realized by the creator of the technique.

In the following subsections, we present each feature of this operational definition in detail. Because these experimentation features must be *implemented* in EPA techniques, our presentation is organized in terms of the *tasks* that are conducted to realize an implementation of each experimentation feature in a program analysis context. During this presentation, we also discuss how each feature contributes to the possible strengths of an EPA technique and, where applicable, a technique's weaknesses due to the introduction of validity threats. For each feature in the operational definition, we use the HOWCOME technique to provide an example of an EPA technique implementing the feature.

Activity	Features	Role in EPA	HOWCOME
Recognition and Statement of the Problem	*Research questions*	Questions about specific aspects of a program.	"Given execution e_p and failure f in e_f, what are the minimum variable values in e_f that cause f?"
	Population	Program aspects that the experiment draws conclusions about.	All program states $S_{e_p} \in e_p$.
Selection of Independent and Dependent Variables	*Factors*	Internal or external program aspects impacting the effect of the measured manipulations.	Variable value changes, circumstances inducing f, number of faults, outcome certainty, unchanged variable values.
	Independent variables	Factors manipulated to impact the program aspect of interest.	Values of variables in e_f at each state $s \in S_{e_f}$.
	Fixed variables	Factors that are set (or assumed to be) constant.	Variable values that do not change.
	Nuisance variables	Uncontrolled factors affecting measurements on the program.	Multiple circumstances inducing f, number of faults, outcome certainty.
	Dependent variables	Constructs characterizing the effect of the program manipulations.	Whether the execution reproduces f, succeeds, or has an inconclusive result.
Choice of Experiment Design	*Treatments*	Specific instantiations of independent variable levels that are tested.	Difference in variable values between an e_p state and corresponding e_f state.
	Hypothesis statement	Conjectures about treatment effects on an aspect of the program.	Null hypothesis H_0 for each treatment: "The state changes do not reproduce f."
	Sample	Selected unit set from population.	Selected program states in e_p.
	Treatment assignment	Assignment of independent variable levels to the sampled units.	Compound treatments of variable values are applied to states in e_p.
Performing the Experiment	*Experiment procedures*	Algorithms to measure and collect observations of treatment effects on program.	Observations are collected for each test of variable value changes within a program state.
Analysis and Interpretation of Data	*Data analysis*	Analysis of observations to discover or assess the effects of treatments on the targeted program aspect.	Effects of applying variable value changes is measured by observing the effect on the execution's output.
	Hypothesis testing	Tests to confirm or reject the previously-stated hypotheses based on the data analysis.	H_0 is rejected if f is reproduced, not rejected if e_p is reproduced, and not rejected for inconclusive outcomes.
	Interim analysis	A decision whether further tests are needed based on the results of the current and previous tests.	Form new subsets of variable values (if possible) if H_0 was rejected. Otherwise, choose different values from those remaining (if any).
Conclusions and Recommendations	*Validity threats*	Limitations or assumptions impacting confidence in results.	Dependencies causing extra variable reports, multiple faults confounding results.
	Final conclusions	Conclusions drawn from the experimental program analysis.	Using the reported cause-effect chain, or deciding to generate a new chain.

Table 1: Features in EPA techniques. Features are grouped in relation to the experimentation activities presented in Section 2. The rightmost column uses Howcome to illustrate each feature. Features that can be automated are in gray rows, while features in white rows are performed by investigators.

3.1 Recognition and Statement of the Problem

The outcomes of this planning activity are (1) the formulation of research questions, and (2) the identification of the experiment's population.

Formulation of research questions. This task guides and sets the scope for the experimental program analysis activity, focusing on particular aspects of a program such as source code or runtime behavior.

Howcome *Example:* Howcome aims to isolate the variable values relevant to a failure f observed in a failing execution e_f but not a passing execution e_p. The research question addressed by the technique is: "given e_p and f in e_f, what are the minimum relevant variable values $V \in e_f$ that cause f?"

Identification of population. This task identifies the aspect of the program about which inferences will be made. The population universe of the experiments conducted by an EPA technique is some set of artifacts related to representations of the program of interest, or factors related to the program's execution.

Howcome *Example:* Howcome draws conclusions about variable values that are relevant to f by applying those values to program states in e_p. Thus, each program state is a unit in the population of all program states S_{e_p}.

3.2 Selection of Independent/Dependent Variables

The outcomes of this activity are the identification of (1) the aspect of the program to be manipulated by the EPA technique, (2) a construct with which to measure these manipulations' effects, and (3) the factors for which the experiments performed by the technique do not account, that could influence or bias observations.

Identification of factors. The "factors" feature of EPA techniques is any internal or external aspect of the program that could impact the measurements regarding the effect of the manipulations that are being measured. An important byproduct of this task is the awareness of potentially confounding effects on the results.

Howcome *Example:* Many factors can influence the variable values identified by Howcome as being relevant to f, including the range of variable values in the program states that can impact the final execution output; the number of faults, as multiple faults may induce different failure circumstances; and the failure-inducing circumstances themselves, including non-deterministic or synchronization issues on which failures may depend.

Selection of independent variables. This task identifies the factors that will be explicitly manipulated by the EPA technique. The effect of these manipulations on the aspect of the program under study is measured in order to answer the research questions addressed by the technique. As in traditional experiments, treatments are ultimately selected as levels from the ranges of the independent variables.

Howcome *Example:* The independent variable manipulated by Howcome during its program analysis is the values of the variables in e_p at each program state. Changes to these variable values are made in order to test their relevance to f. The operative notion is that through modification of this variable, Howcome may find different variable values to be relevant to f. As an example, if the variable x is an 8-bit unsigned integer, then the range of the independent variable is 0–255, and a treatment from x is one of the 256 possible values (i.e., levels) of x.

Selection of fixed variables. This task chooses a subset of factors to hold at fixed levels. Fixing factors at certain levels ensures that these individual variables elicit the same effect on the observations obtained from the experiment, and that differences seen between observations are not due to these factors alone. As in traditional experiments, EPA techniques are most reliable if every factor that is not manipulated and tested during the program analysis is fixed to ensure systematic control in experiment conditions. If the proper factors are not controlled for, it is possible that they will introduce threats to the internal validity of the EPA technique. This is one way in which EPA techniques, like traditional experiments, reduce threats to validity – by ensuring that extraneous factors other than the independent variables do not impact results.

HOWCOME *Example:* The variable values that are not manipulated by HOWCOME are kept constant to control their effect on the program outcome. This attempts to prevent any variable values other than those in the treatment being evaluated from influencing the execution's output.

Identification of nuisance variables. This task identifies uncontrolled variables. These factors may intentionally be left uncontrolled because it may not be cost-effective to control them, or because it is not possible to do so. In any case, it is important to acknowledge their presence so that an EPA technique's conclusions can be considered in light of the possible role of these factors, as uncontrolled nuisance variables are inherently threats to the internal validity of the technique's experiments. (As we shall see, improvements to EPA techniques can come in the form of finding ways to reduce or eliminate the potential impact of these nuisance variables.)

HOWCOME *Example:* The presence of multiple faults, the existence of multiple failure-inducing scenarios or scenarios for which the outcome is not certain and cannot be classified as passing or failing, and dependencies between variable values are some nuisance variables that can confound, bias, or limit the results that are reported by HOWCOME.

Selection of dependent variable(s). This task determines how the effects of the manipulations to the independent variable (treatments) will be measured. A construct is then chosen that will capture these measurements in the form of observations that can be analyzed to evaluate the treatments. If this task is not performed properly, then the construct validity of the technique may be threatened, because the construct may not capture the effect that it is intended to capture.

HOWCOME *Example:* The dependent variable for HOWCOME measures whether f is reproduced when manipulations are made to the variable values in program states. The construct for this variable is a testing function that indicates if (1) the execution succeeded as did the original, unmodified execution e_p, in spite of the variable value changes; (2) f was reproduced by the treatments; or (3) the execution's output was inconclusive, as when inconsistent program states occur due to the modification of certain variables.

3.3 Choice of Experiment Design

The outcome of this activity, which formulates the design of the experiment(s) conducted by an EPA technique, are (1) the treatments that will be investigated in the experiment, (2) the hypotheses associated with those treatments, (3) the set of elements from the population that will be used in the experiment, and (4) the manner in which treatments will be assigned to the elements in the sample. The choice of experiment design is crucial for maximizing the control of the sources of variation in the program and the environment, and reducing the cost of an EPA technique. Features in the choice of experiment design activity can be automated by the EPA technique.

Design of treatments. This task determines specific levels from each independent variable's range at which to instantiate treatments. If there are multiple independent variables, or if multiple levels from the same variable are to be combined, then this task also determines how the instantiations will be grouped together to form compound treatments. It is these treatments — instances of specific manipulations made by a technique — that are evaluated in experimental program analysis using units from the population, and about which conclusions will be drawn.

HOWCOME *Example:*

The experiments conducted by HOWCOME are crafted through the selection of potential variable values to apply to a program state in e_p. Each variable change is an instantiation of the difference between the variable in the passing state and the corresponding failing state. These value changes are tested to see whether f is reproduced. If so, then either the variable value changes will be used to create a cause-effect chain or an attempt will be made to narrow those values further. The information that is gathered regarding the effect of these variable value changes on the program is later used to determine those variable value treatments that should be investigated further (by combining them with a different combination of variable values) and those that should be discarded.

Formulation of hypotheses. This task involves formalizing conjectures about the effects of treatments on the aspect of the program of interest. An hypothesis encodes a potential relationship between variables that is amenable to evaluation after observations are collected about the treatments' effects so that experimental program analysis can draw conclusions about the treatments' impact on the population of interest. We note that the constitution and later evaluation of the hypotheses may vary considerably depending on the number, type, and complexity of treatments, measurement construct, and unit of analysis involved. We start with a relatively simple hypothesis in this section but explore others in Section 5.

HOWCOME *Example:* When considering potential variable value changes that are made to a program state in e_p by HOWCOME, the technique's experiments assess whether the variable values reproduce f in e_p. A null hypothesis for a particular treatment of variable value differences therefore states that "the variable value changes do not reproduce f in e_p."

Sample the population. A sample is a set of elements from the population. Collecting observations on a subset of the population is one way by which experimental program analysis (like traditional experiments) achieves results while retaining acceptable costs. This task defines the sampling process (e.g., randomized, convenience, adaptive) and a stopping rule (e.g., execution time, number of inputs, confidence in inferences) to be used to halt the process of sampling. If this task is not completed properly, the external validity of the experiment may be threatened, as conclusions may not generalize to the population.

HOWCOME *Example:* HOWCOME samples program states from e_p so that the variable values from equivalent states in e_f can be applied and tested for relevance to f. States at which relevant variable values are found are used to form the reported cause-effect chain. Note that program states can be continuously re-sampled in order to draw conclusions about program states that are important to cause-effect chains.

Assign treatments to sample. This task involves assigning treatments to one or more experimental units in the sample. Some assignment possibilities include random assignment, blocking the units of the sample into groups to ensure that treatments are better distributed, and assigning more than one treatment to each experimental unit. The result of this task is a set of units of analysis from which observations will be obtained to evaluate the treatments during experimental program analysis. If this task is not performed correctly, the experiment could suffer from conclusion validity problems, as its conclusions may not be powerful enough due to issues such as having insufficient replications for each treatment. An attempt to avoid these types of problems can be made by choosing a methodology that is appropriate in terms of the conceptual relationship between treatments and the sample, and also ensures that quantitative information of sufficient power will be gathered to adequately address the research questions investigated by the technique.

HOWCOME *Example:* The variable value differences between e_f and e_p that are selected as treatments by HOWCOME are applied to the appropriate program states in e_p. This is performed so that HOWCOME can observe whether these treatment variable value changes reproduce f in e_p.

3.4 Performing the Experiment

This activity is primarily mechanical and has a single outcome: a set of observations on the sampled units that reflect the measured effect of the independent variable manipulations (treatments) according to experimental procedures. With EPA techniques, the procedures governing the collection of these observations are automated, which is an important characteristic of experimental program analysis because it grants techniques the scalability required to be applicable to large problems.

Execute experimental procedures. In experimental program analysis, experimental procedures can be represented algorithmically and can be automated; this differentiates experimental program analysis from experiments in some other fields, where the process of following experimental procedures and collecting observations is often performed by researchers — risking the intrusion of human factors that might confound results — and where an algorithmic representation is sometimes not as natural for representing the experimentation processes. During this task, the observations collected according to the dependent variable should capture only the isolated effects that follow from the manipulations of the independent variables. If this task is not performed correctly, the experiment's internal validity may be affected due to extraneous factors influencing the observations obtained.

Howcome *Example:* An observation is collected for each test of variable value changes to a state. The process of conducting such a test within an experiment involves running e_p, interrupting execution at the program state s_i under consideration, applying the treatment variable values from e_f to e_p at s_i, resuming the execution of e_p, and determining whether (1) e_p succeeded, (2) f was reproduced, or (3) the execution terminated with an inconclusive result. The outcome of this test is an observation collected.

3.5 Analysis and Interpretation of Data

The outcomes of this activity are (1) a data analysis used to test hypotheses, (2) the actual testing of those hypotheses, and (3) an interim analysis that determines whether further data needs to be collected in order to draw conclusions from the experiments. To ensure that conclusions are objective when a population sample has been used, statistical measures can assign confidence levels to results, helping control the effectiveness and efficiency of the experiment. These tasks can be automated by EPA techniques.

Performing data analysis. This task involves the analysis of collected observations for the purpose of evaluating hypotheses. This can include inferential analyses to determine the appropriateness of the decision made regarding an hypothesis. This also include checks on assumptions regarding the underlying unit distributions. If this task is not performed properly, or if the assumptions made by underlying statistical analyses for the data are not met, the conclusion validity of the experiment can be threatened.

Howcome *Example:* Hypotheses are evaluated based on whether the dependent variable indicated that the variable value treatment caused f to be reproduced, or whether it caused an inconclusive result.

Testing and evaluating hypotheses. Hypothesis testing assesses the effect of the manipulations made by the investigator, or, in the case of experimental program analysis, by the automated process managing the experiment.

Howcome *Example:* Using observations from the tests of applying variable values from e_f into e_p, the null hypothesis H_0 is rejected if the variable value treatment reproduces the original failure, indicating that the technique should try to find minimally relevant variable value differences within this treatment. As such, the rejection of H_0 guides the manipulations of the independent variable (i.e., guides the design of future treatments) by determining whether the particular treatment variables should be refined during the remainder of the analysis.

Performing interim analysis. After the hypotheses have been tested, a decision must be made about whether further treatments need to be evaluated or different experimental conditions need to be explored. EPA techniques determine automatically, based on the results of previous experiments, whether to continue "looping" (i.e., making further manipulations to the independent variables), or whether the experimental program analysis has reached a point where the technique can conclude and output results.

Howcome *Example:* If H_0 (for the treatment variable value differences) was rejected, indicating that those treatments reproduced f, then new experiments will be designed from those values to further minimize the variable values relevant to f (if further minimizations are possible). Otherwise, if different sets of variable values remain that have not yet been tested via experiments, they will be evaluated as treatments next. When no variable values remain to be tested, the cause-effect chain is reported by combining the isolated variable value differences into a sequential report explaining the causes of f.

3.6 Conclusions and Recommendations

The outcomes of this activity are (1) an assessment of the validity threats of the EPA technique's experiment(s), and thus the program analysis the technique conducts; and (2) the conclusions obtained from the experimental program analysis in the context of the limitations placed on them by validity threats.

Assessment of validity threats. As indicated throughout our discussion of the features in our operational definition, various threats to the validity of the experiments conducted by EPA techniques arise. Because some threats can be reported through automated mechanisms, this task is represented by a gray row. For example, power analyses could estimate the power of the experiment and quantify threats to conclusion validity, estimating the size of the sample could help to quantify threats to external validity, and estimating the number or severity of uncontrolled nuisance variables could help to quantify threats to internal validity.

Howcome *Example:* A cause-effect chain may not contain the true minimally relevant variables due to dependencies between tested variable values, or due to the initial variable value combinations tested, which influences the future combinations that are tested and the "minimal" set of variables thus found. Cause-effect chains can also be biased and confounded by multiple software defects interacting and influencing the report about the failure described by the chain.

Drawing final conclusions. This task results in the final conclusions that are drawn from experimental program analysis in the context of the EPA technique's validity threats. Final conclusions are made when the analysis is complete and no further experiments are needed or can be performed.

Howcome *Example:* a cause-effect chain can be used by engineers to track the root cause of the failure through its intermediate effects and ultimately to the failure itself, or to select different passing and failing executions to provide to Howcome.

4 Discussion of Experimental Program Analysis Traits

There are many opportunities for EPA techniques to utilize their distinguishing traits to address research questions in unique ways. Also, due to characteristics of the program analysis setting, EPA techniques have advantages available to them that are not as readily available to more traditional uses of experimentation. We now comment on several of these distinguishing traits, and specific benefits that they may bring to EPA techniques.

4.1 Replicability and Sources of Variation

Program analysis activities are not subject to many of the sources of spurious variations that are common in some other fields. For example, program analysis is conducted on artifacts and is usually automated, reducing sources of variation introduced by humans (as subjects or as experimenters), which are among the most difficult to control and measure reliably. We have also observed that some typical threats to replicability must be reinterpreted in the context of experimental program analysis. For example, the concept of learning effects (where the behavior of a unit of analysis is affected by the application of repeated treatments) should be reinterpreted in the program analysis context as residual effects caused by incomplete setup and cleanup procedures (e.g., a test outcome depends on the results of previous tests). Also, a software system being monitored may be affected by the instrumentation that enables monitoring, and this resembles the concept of "testing effects" seen in some other fields.

However, EPA techniques *are* susceptible to other sources of variation that may not be cost-effective to control. For example, non-deterministic system behavior may introduce inconsistencies that lead to inaccurate inferences, and should be cast as threats to internal validity. Controlling for such behavior (e.g., controlling the scheduler) may threaten the generality of an EPA technique's findings, which is an issue that investigators should think of as threats to external validity. Still, experimental program analysis has the advantage of dealing with software systems, which are not material entities and are more easily controlled than naturally occurring phenomena.

4.2 The Cost of Applying Treatments

In most cases, the applications of treatments to software systems have relatively low costs — especially in comparison, say, to the cost of inducing a genetic disorder in a population of mice and then applying a treatment to this population. Systems may thus be exercised many times during the software development and validation process. This is advantageous for experimental program analysis because it implies that multiple treatment applications, and multiple hypothesis tests, are affordable, which can increase the power of EPA techniques' conclusions and offers many directions for future work in this area.

Two factors contribute to this trait. First, in experimental program analysis, applying treatments to experimental units is often automated and requires limited human intervention. (This impacts the cost of treatment application, but EPA techniques are still likely to draw conclusions — i.e., produce results — that will likely need to be examined and comsumed by humans, and assessments of techniques will need ultimately to consider this.) Second, there are no expendable units or treatments; that is, the aspects of the system that are being manipulated, or the sampled inputs, can be reused without incurring additional costs (except for the default operational costs). Experiments in program analysis settings also avoid ethical issues that arise in other scientific domains such as those involving humans, livestock, wildlife, or the environment.

EPA techniques can leverage these traits by using increased sample sizes to increase the confidence in, or quality of, the findings, and adding additional treatments to their design to learn more about the research questions. We note, however, that the affordability of treatment application must be balanced with an analysis that takes into consideration the adjustments to confidence and test power required to reduce the possibilities of threats to conclusion validity.

4.3 Sampling the Input Space

Sampling allows researchers to control experimentation costs while maintaining the meaningfulness of the experiment by selecting a representative subset of the population. When experimenting with programs, we are often able to collect very large sample sets because of computing resources and the focus on a software program, which are not material entities. This can allow the experiments in EPA techniques to operate in ways that are difficult to achieve with traditional techniques. For example, in the case of EPA techniques that sample a program's execution space, it is not possible to sample the (infinite) number of executions for non-trivial programs. Sampling a limited, yet still sizable, subset of executions would provide these techniques with scalability as well as investigative power. Furthermore, we may find cases where the size of the population is small enough that the sample can constitute an important part of the population. We have even identified cases [51] where the population and sample size may be the same, resembling a census.

Sample quality is also an issue facing experiment designers, and this is no different in experimental program analysis; the power of EPA techniques to generalize and the correctness of their inferences is dependent on the quality of the samples that they use. Although this challenge is not exclusive to experimental program analysis (e.g., software testing attempts to select "worthwhile" inputs to drive a system's execution) and there will always be uncertainty when making inductive inferences, we expect the uncertainty of EPA techniques to be measurable by statistical methods if the sample has been properly drawn and the assumptions of the method have been met.

4.4 Assumptions about Populations

Software systems are not naturally occurring phenomena with distributions that follow commonly observed patterns. Experimental program analysis data reflecting program behavior is, for example, rarely normal, uniform, or made up of independent observations. This limits the opportunity for the application of commonly used inferential analysis techniques. One alternative is to apply data transformations to obtain "regular" distributions and enable traditional analyses. However, existing transformations may be unsuited to handling the heterogeneity and variability of data in this domain. Instead, it may be necessary to explore the use of "robust" analysis methods — that is, methods that are the least dependent on the failure of certain assumptions such as non-parametric statistical techniques [38].

4.5 Procedures versus Algorithms

EPA techniques are unique in at least two procedural aspects. First, EPA techniques' procedures are commonly represented as algorithms. The representation of these procedures using algorithms allows these techniques to at least partially avoid many of the human factors that can confound the results of experiments in some other fields. Furthermore, the algorithmic representation naturally lends itself to both analysis and automation, which reduces its application costs. Second, these algorithms can be extended to manage multiple experimental tasks besides the experiment's execution. For example, our experimental regression fault analysis technique in Section 5.2 utilizes an algorithm to refine the stated hypothesis within the same experiment and guide successive treatment applications of changes to a program.

We strongly suspect that other tasks in experimental program analysis, such as the choice of independent and dependent variables, design of the experiment, and sampling procedures, can be represented in algorithmic form and even fully automated as experimental program analysis evolves. This would allow such EPA techniques to be highly adaptable to particular instances of program analysis problems — without the intervention of investigators. EPA techniques, however, are still likely to draw conclusions that will be consumed by investigators, and studies of these techniques will ultimately need to take this into account.

4.6 The Role of Feedback

Traditional experimentation guidelines advocate the separation of hypothesis setting, data collection, and data analysis activities. This separation helps to make experiments more manageable, to reduce costs, and to avoid various types of potential bias. However, in the presence of costly units of analysis, these activities can be interleaved through sequential analysis to establish a feedback loop that can drive the experimental process [39]. For example, in clinical trials an experiment stops if the superiority of a treatment is clearly established, or if adverse side effects become obvious, resulting in less data collection, which reduces the overall costs of the experiment. EPA techniques can take sequential analysis with feedback even further, interleaving not only the sampling process and the stopping rules, but also determining what design to use and what treatments to apply based on the data collected so far. This enables EPA techniques to scale in the presence of programs with large populations or a large number of potential treatments, whereas the application of other program analysis approaches may not be affordable in such cases.

Another possible use of feedback involves the use of adaptive strategies in EPA techniques. Because they build a body of knowledge about programs during their analyses, EPA techniques can adjust their experiment parameters to improve the efficiency of their analyses, such as through the use of adaptive sampling procedures [42] to concentrate on areas of the program that may actually contain faults, based on the program information gathered thus far, during a debugging activity. Such analyses can allow techniques to provide more accuracy by narrowing in on the areas of the program showing promising results, and also to provide cost-effectiveness by not wasting effort on unimporant areas of the program.

5 Three Applications of Experimental Program Analysis

To investigate the capabilities and potential of experimental program analysis, we consider its use with respect to three software engineering problems: program transformation, program debugging, and program understanding. Where program transformation is concerned, we present a new EPA technique for performing program refactoring to improve the maintainability of programs. One unique aspect of this technique is its ability to handle the complex interactions that can take place between refactorings — the independent variable of interest — in the presence of a large refactoring space. Where program debugging is concerned, we present a new EPA technique for analyzing regression faults and reducing the set of source code changes between two sequential versions that might be responsible for the faults. One unique aspect of this technique is its ability to turn an optimization problem into a sampling problem. Where program understanding is concerned, we enhance an existing EPA technique for inferring likely program invariants — Daikon [15] — with a well-known design from the traditional experimentation literature to improve its cost-effectiveness, and with statistical tests to improve the reliability of its results.

Together, these three techniques illustrate how an understanding of the experimental program analysis paradigm can help researchers address various types of program analysis problems. The description of the experiments conducted by these techniques is provided in Table 2, and is elaborated on in the upcoming presentation of each technique.

5.1 Experimental Program Refactoring

Program refactorings [17] are semantic-preserving transformations to source code that improve a program's maintainability. There are many aspects to the identification, application, and evaluation of program refactorings that make this task difficult.

Feature	**Role in EPA**	**Refactoring**	**Regression Faults**	**Fractional Daikon**
Research questions	Questions about specific aspects of a program.	Given a program P with source code C, what are the refactorings R to apply to C such that the maintainability of P is maximized?	Given versions v_i and v_{i+1}, changes C between v_i and v_{i+1}, and execution e_f failing in v_{i+1} but not v_i, what is a subset $C' \subseteq C$ such that C' applied to v_i ($v_i(C')$) reproduces e_f?	Given a program P with execution traces T, what are the likely program invariants I at program points M in P?
Population	Program aspects to draw conclusions about.	All source code segments $S \subseteq C$ that can be individually refactored.	All functions in version v_i of the program.	A model of the program in terms of likely, candidate invariants.
Factors	Internal/external program aspects impacting effects of manipulations.	Target code and their dependencies, dependencies between refactorings.	Changed code, fixed code, code dependencies, change dependencies, execution e_f.	Invariant types, points M to infer I at, confidence level, dependencies in P, outcome certainty, T.
Independent variables	Factors manipulated to affect targeted aspects.	Refactored code at each segment $s \in S$ in P.	Differences C between modified functions in program.	Points $M' \in M$ at which to apply data from each trace $t \in T$.
Fixed variables	Factors that are set (or assumed) to be constant.	Code that does not change or has not been refactored.	Code that does not change, execution environment.	Level of confidence for reported invariants, invariant types.
Nuisance variables	Uncontrolled factors affecting measurements.	Dependencies between code, dependencies between transformations.	Dependencies between code, dependencies between changes.	Dependencies in P, outcome certainty of invariant holding for a trace t.
Dependent variables	Constructs quantifying effect of manipulations.	Cohesion of methods in P's classes, abstractness of P's packages.	Whether $v_i(C')$ reproduces e_f, does not, or is inconclusive.	Testing function deciding if data in t supports or falsifies an invariant.
Treatments	Specific instantiations of independent variable levels to be tested.	Instances of refactoring transformations R' to be applied to sampled source code.	Code differences $C_d \subseteq C$ between functions in v_i and v_{i+1} selected from C by algorithm.	Trace $t \in T$, and data therein, to be applied at sampled program points.
Hypothesis statement	Conjectures about treatment effects on an aspect of the program.	H_0 for a treatment is: "The refactorings R' do not worsen the code."	H_0 for a treatment is: "The changes C_d will not reproduce e_f."	H_0 for each invariant in treatment program points M' is: "The invariant i holds at M'."
Sample	Selected unit set from population.	Selected code $S' \subseteq S$ to apply R'.	Functions in v_i to apply C_d.	Candidate invariants at M' to apply t.
Treatment assignment	Assignment of independent variable levels to sampled units.	Apply R' as compound treatment of refactorings to applicable sampled S' code.	Apply C_d as compound treatment of changes to applicable sampled v_i functions.	Trace data is applied to invariants at "neediest" 50% of program points M' covered in t.
Experiment procedures	Using algorithms get observations measuring treatment effects.	Get observations by measuring lack of cohesion of methods and abstractness.	Get observations by running P with C_d, C_d if needed, and adjust granularity G if needed.	Get observations regarding whether data in t supports or falsifies the invariants at points M'.
Data analysis	Analyzing observations to assess treatment effects.	Test refactorings' effects on cohesion and abstractness.	Test effect of C_d to see if e_f was reproduced.	Test whether each invariant i at points M' were falsified by trace data.
Hypothesis testing	Evaluating hypotheses about treatment effects.	Reject H_0 if cohesion or abstractness lowers, and do not otherwise.	Reject H_0 if C_d reproduces e_f, and do not reject otherwise.	Reject H_0 if invariant i is falsified based on data in t.
Interim analysis	Deciding if more tests are needed based on hypothesis tests.	If reject H_0, do not consider refactorings in future treatments; else, do so until completed.	If reject H_0, test subsets of C_d; else, test complements, adjust G, or report best subset.	If reject H_0, discard i. Update coverage of M' so that other points may be selected for next trace.
Validity Threats	Validity threats and results' limitations.	See Section 5.1.1.	See Section 5.2.1.	See Section 5.3.1.
Final conclusions	Conclusions drawn from analysis.	Use good refactorings R to help maintainability.	Use reported changes C' to accelerate debugging.	Use non-rejected invariants I to better understand P.

Table 2: A summary of the experiments conducted by the three EPA techniques. Bold lines separate tasks according to the activities in Section 2. Gray rows denote features that can be performed by EPA techniques.

The first issue is related to the sheer scale of the problem: there are many types of refactorings (Fowler [17] introduces approximately 70), and in large programs there are potentially thousands of segments of source code that could be refactored.

Second, it may not always be advantageous to apply candidate code refactorings to particular segments of source code. While the effects of multiple beneficial code refactoring instances may be compounded into a much greater improvement than would be achieved by considering each instance individually, a code refactoring may also worsen the code, according to some metric of interest, when it is combined with other refactoring instances with which it does not interact well. Thus, when evaluating the utility of applied refactorings, designers must consider not only the effect of each individual refactoring but the potential *interaction* between refactorings.

Third, the application of one code refactoring may actually prevent other refactorings from being applicable. For example, it may be advantageous to extract a set of methods from a class to create a new class out of the extracted methods. If, however, a subset of the extracted methods are first moved into a second, preexisting class, it may no longer make sense to extract the remaining methods into their own class. Thus, the benefit, or lack thereof, in utilizing refactoring support is dependent on the manner in which particular code refactorings are applied to a program.

How should such challenges be addressed? In the case of program refactoring, various forms of experimental program analysis can be used to test potential refactorings (as treatments) applied to (sampled) areas of the program, and keep those that appear to be the most promising. In this setting, hypotheses can be made regarding the utility of one or more refactorings, and an experiment can be designed where the effect of those refactorings is tested for purposes of their evaluation. To explore possible interactions between refactorings and to avoid an expensive, exhaustive consideration of all combinations of refactorings, one possible strategy is to form groups of refactorings that can be considered together in an intelligent manner using experiment designs and procedures that focus on the most promising refactorings, as determined by feedback gleaned from previous experiments. We present and investigate an EPA technique following this strategy.

5.1.1 New Experimental Program Analysis Technique

Design

Because it is not feasible to examine all possible combinations of individual code refactorings for a program and keep the best combination — for non-trivial programs, there are likely to be hundreds of individual refactorings possible, and therefore millions of possible combinations — an alternative approach is needed. Experimental program analysis can provide such an approach by *experimentally* applying groups of potential, promising refactorings to the code, and measuring their effects in order to determine whether they should be kept or discarded. The feedback accumulated during previous tests of refactoring combinations can then be used to guide the refactorings that should (or should not) be considered further. The intuition behind this approach is that the set of individual code refactorings that could be applied to a program is manipulated as an *independent variable*. *Compound treatments* of one or more possible code refactorings R' are applied to the applicable *sampled* source code.[2]

For each compound treatment R', a *null hypothesis* H_0 is stated as, "The refactorings R' do not worsen the code." *Dependent variables* measuring the maintainability of the program are used to *evaluate this hypothesis* H_0 for each treatment R' after it is applied to the sampled source code S in the *treatment assignment* feature. If R' is detrimental to the program according to the observations collected during the *experiment procedures*, then H_0 can be rejected during the *hypothesis testing*, and the treatment refactorings in R' can be discarded from consideration during *interim analysis*. Otherwise, the refactorings in R' can be combined with other compound treatments of refactorings until none remain to be tested. In terms of our descriptive definition, the aim of this technique is to establish causality by measuring the effect of treatment refactorings on program maintainability.

Threats to Validity in the Experimental Program Analysis Technique

Because this refactoring approach conducts experiments to drive its analysis, there are threats to validity that must be considered when evaluating both the approach and the final set of refactorings indicated at the end of experimentation.

In terms of external validity, this experiment's population is the program's source code segments, and sometimes only a subset of the program will be selected for refactoring. In these cases, although individual refactorings (i.e., a refactoring operation and a location to apply that refactoring) may be beneficial when applied to some areas of the program, they may not be beneficial when applied as a whole to the program. If more of the program is sampled for refactoring, however, then this external validity threat is reduced, as the experiment would then resemble a census.

The refactoring technique also suffers from threats to internal validity due to the interactions and dependencies between individual refactorings. Some refactorings may be chosen, or discarded, based solely on how they interact with other refactorings, which in many cases may hinder the technique's ability to choose the truly optimal set of refactorings.

The dependent variables determine which refactorings will be retained and discarded. Thus, the construct validity of the results will be threatened if these variables are inadequate (e.g., not properly capturing the maintainability of the program).

Finally, in terms of the technique's conclusion validity, the refactorings identified may not be the *optimum*

[2]Our approach is to sample all relevant source code segments in the targeted source code. In Section 5.1.2, for example, we select five classes for program refactoring, and all source code within these classes is sampled for possible refactorings. An alternative approach not pursued here would be to consider individual samples of source code separately and investigate different refactorings for different areas of the program.

set because the approach does not exhaustively explore all possible combinations of refactorings for the sampled source code.

5.1.2 Pilot Study

To investigate our experimental refactoring approach we use `Siena`, a program artifact written in Java with 26 classes and 6,035 lines of source code. (`Siena`, as well as the artifacts used in the other studies in this chapter, is available as a part of the Software-artifact Infrastructure Repository (SIR), an infrastructure supporting experimentation [10].)

The primary goal of this study was to formatively investigate whether our experimental program analysis approach for refactoring has the potential to effectively address one of refactoring's inherent difficulties — namely, selecting a set of refactorings that improves program maintainability — and to gain insights into the applicability of experimental program analysis to program transformation problems.

Study Design

We utilize two metrics as *dependent variables* to measure the maintainability of `Siena`, and the utility of the refactorings tested against the program.[3] Our first dependent variable measures the cohesion of the methods within classes. Cohesion of methods is a measure of the similarity of the methods in a class in terms of their internal data usages [11], and is important in object-oriented programming because dissimilar methods may belong in separate classes. As a dependent variable to measure cohesion, we selected the "Lack of Cohesion of Methods" (LCOM) metric as defined by Henderson-Sellers [22]. LCOM is represented by a real number that can be as low as 0.0, which is optimal since the metric measures a *lack* of cohesion, and can (theoretically) grow arbitrarily large. Methods within a single class that have very similar internal data usage exhibit high cohesion, and therefore have an LCOM close to 0.0. As the internal data usage within class methods diverges, LCOM grows larger.

Our second dependent variable measures the "abstractness" (Abst) of Java packages. This metric is a ratio of the number of abstract classes and interfaces in the package to the number of concrete classes and interfaces in the same package. When the Abst value of a package is zero we have a completely concrete package while a value of one indicates a completely abstract package. Of course, both too little and too much abstractness can hurt the maintainability of a package, and it is not always the case that one is better than the other. To investigate the use of experimental program analysis for program refactoring,we assume a setting where a developer is attempting to decrease package abstractness.

The *independent variable* manipulated in this study is the approach used for performing refactoring. We consider three approaches:

[3]Many metrics can be used to estimate maintainability, our primary goal was not to evaluate metrics but rather to use a subset of reasonable maintainability metrics as a means for investigating our experimental program analysis methodology.

1. As a treatment variable, we consider LCOM and Abst after applying the experimental program analysis methodology; we label the LCOM and Abst measurements using this first approach LCOM-EPA and Abst-EPA, respectively.

2. As our first control variable, we consider the initial LCOM and Abst values, denoted by LCOM-Init and Abst-Init, which are measured without applying any refactorings. As a comparison with experimental program analysis, this control variable serves as a baseline for evaluating what would have occurred had no refactorings been applied.

3. As our second control variable, we consider the LCOM and Abst values after applying all possible refactorings, which we denote by LCOM-All and Abst-All. This control variable considers one possible refactoring strategy that might be employed, where refactorings are applied without regard to how they may (or may not) impact other refactorings – a key challenge that we attempt to address through the use of experimental program analysis. Although we are chiefly interested in evaluating experimental program analysis as a treatment variable, our presentation of this study's results will also discuss how this "blind" approach of applying all refactorings would compare with the baseline of applying no refactorings at all (i.e., our first control variable).

Our treatment and control variables consider the use of the "Extract Class," "Extract Method," and "Move Method" refactoring operations to identify treatment refactoring instances. We chose these refactoring operations for two reasons. First, all three operations were identified by Du Bois et al. [11] as having the potential to improve cohesion, while the Extract Class operation has the potential to improve (lower) abstractness by increasing the number of concrete classes in the package. Second, the application of instances of either the Extract Class or the Move Method operations can prevent instances of the other type from being reasonably applied. For example, creating a new class C_2 out of the methods belonging to a separate class C_1 would not be possible if every method from C_1 was moved into a third, preexisting class C_3. Thus, choosing these two operations lets us investigate how often situations like this may occur, and how our experimental program analysis approach handles such situations.

The goal of this study is to formatively consider the possible benefits of using an experimental program analysis approach for addressing program refactoring. Thus, as *objects of study* we selected the five `Siena` classes with the worst initial LCOM because these offer the *greatest possible opportunity* for the refactoring approaches to improve the program. (Of the 26 `Siena` classes, 16 began with an LCOM of 0.0, thereby providing no opportunities for improvement according to this dependent variable.) `Siena` has only one package, which we measured as an object of analysis using the Abst dependent variable. We used version 2.5 of the RefactorIt tool [32] within our experimental methodology as an Eclipse plug-in to apply refactorings to `Siena` and evaluate them using the LCOM and Abst metrics.

Threats to Validity in the Study Design

Where external validity is concerned, we selected `Siena` as a subject in part because it is a real program from a software infrastructure repository used by many researchers. However, because only five classes were considered in this study, there are external validity issues related to sample size. Also, Siena is just one program written in one particular language; the study of additional programs including programs written in different languages would be beneficial.

In terms of internal validity, the results observed in this study may be due to particular characteristics of `Siena`. Also, our experimental program refactoring methodology may have performed differently had we chosen refactoring operations other than "Extract Class," "Extract Method," and "Move Method," although we note that these are three common and well-known refactoring operations.

Where construct validity is concerned, although we considered two established metrics for measuring the maintainability and organization of programs, many other maintainability metrics are available, and these metrics might yield different results. Furthermore, even well-established metrics like LCOM have limitations, and other approaches may be more appropriate in some scenarios. As a possible alternative to LCOM, for example, Bowman et al. [4] propose object-oriented refactoring based on improving both cohesion and coupling at the same time. Future studies exploring these alternatives may be insightful. Finally, there are many methods for computing the LCOM metric in particular. We chose one of the more well-known methods defined by Henderson-Sellers [22], but others could also be utilized.

Results

To provide context for our results, we begin by providing data on the utility of all individual code refactorings (Table 3). As the table shows, refactorings were more often judged to be detrimental than beneficial, and there were many refactorings that, individually, had no effect on the program. Table 3 also shows that there were *nine* instances where beneficial or neutral refactorings, when combined, could interact in an undesirable manner by making the maintainability of the program worse.

To provide further context, we also consider the cost of exhaustively applying and testing all possible refactorings to find the optimum subset of refactorings, and to obtain insights into the possible savings of using our EPA refactoring technique instead. As Table 4 shows, our EPA technique provides a substantial savings in terms of the number of refactorings that must be applied to `Siena` and evaluated for utility.

We next consider the effects of our approach on our dependent variables. Table 5 summarizes the LCOM values, according to each construct, of the five selected `Siena` classes. Perhaps the most encouraging result from this table is that our experimental program analysis methodology was the best choice for improving LCOM, as seen by comparing the EPA values with those from INIT and ALL. This study serves as formative evidence of the potential of an experimental refactoring approach to improve the maintainability of source

Individual Utility:	
Beneficial	5
Neutral	8
Detrimental	10
Detrimental Interactions:	
Between Beneficial	2
Between Neutral	7

Table 3: Summary of the utility of all code refactorings, and interactions between beneficial or neutral refactorings causing detrimental effects.

Class	**All**	**EPA**
`Monitor`	4	2
`TCPPacketHandler`	0	0
`SENP`	1152	12
`HierarchicalDispatcher`	128	9
`Tokenizer`	16	6

Table 4: The number of refactorings that need to be applied and tested using an exhaustive search versus experimental program analysis (EPA).

Class	INIT	ALL	EPA
`Monitor`	1.179	1.125	1.095
`TCPPacketHandler`	1.0	1.0 *	1.0 *
`SENP`	0.997	1.006	0.996
`HierarchicalDispatcher`	0.899	0.918	0.896
`Tokenizer`	0.85	0.684 / 0.889	0.684

Table 5: The Lack of Cohesion of Methods of classes using baseline approaches and experimental refactoring. The * symbol indicates that no refactorings were identified for this class.

code. Moreover, Table 5 suggests that simply applying all refactorings may not be the best approach, as `All` actually hurt the overall cohesion, compared to INIT, for two of the classes that we investigated (`SENP` and `HierarchicalDispatcher`).

Table 5 shows that the benefits of applying refactorings experimentally, in terms of the LCOM metric, were not always dramatic. One class (`TCPPacketHandler`) experienced no benefit in terms of LCOM from the experimental methodology, and two classes (`HierarchicalDispatcher` and `SENP`) showed less than a 1% improvement. However, the `Monitor` and `Tokenizer` classes showed improvements in LCOM of 7% and 20%, respectively. Based on these results and our own intuitions regarding program refactoring, we conjecture that the benefits experienced through the use of experimental program analysis to address this problem will vary from setting to setting. Future work investigating this issue using a larger number of

Init	All	EPA
0.111	0.100 / 0.103	0.100

Table 6: The abstractness of the `Siena` package using baseline approaches and experimental refactoring.

metrics would be worthwhile.

In investigating these five classes further, we found that there were no possible Extract Method, Extract Class, or Move Method refactorings in the `TCPPacketHandler` class, a small class with four methods and 47 lines of code. However, many refactorings were possible in the other four classes. Of particular interest to us was the `Tokenizer` class. In this case, two conflicting refactorings (Extract Class and Move Method) could not be applied together because the application of the Extract Class refactoring would extract the method that the Move Method refactoring sought to relocate. The All result for `Tokenizer` in Table 5 reports the first case where only the Extract Class refactoring was applied, as well as the second case where only the Move Method refactoring was applied. (These two LCOM results are separated by the "/" character in Table 5.) Of the two possible cases, in the first case All was an improvement over Init (and equivalent to the cohesion achieved by the experimental approach) while in the other it was worse than the initial cohesion before any refactorings were applied. Our experimental approach selects this best case by separately evaluating both the Extract Class and Move Method refactorings, and keeping the Extract Class refactoring that dramatically improved the class's cohesion. This example, which is one of four that we observed in this study, shows how an experimental approach to refactoring can provide guidance as to the refactorings that should be selected in difficult cases involving interactions between potential refactorings.

Table 6 shows how the abstractness of the `Siena` package was affected due to different treatment code refactorings. The Abst of the package was improved for both EPA and All. However, in the case of naively applying all possible code refactorings, the same, previously-mentioned conflict between Extract Class and Move Method refactorings in the `Tokenizer` class faces the All methodology. The experimental program analysis methodology chose the Extract Class refactoring, which decreased the package's abstractness as we desired by adding an additional concrete class.

Discussion: The Benefit of Experimental Program Analysis

Recall that we observed cases in which the application of one refactoring prevented the application of other refactorings. These considerations illustrate not only why program refactoring can be difficult, but the difficulty of activities involving program transformation in general. There are many confounding factors, such as interactions between transformations, that require innovative techniques in order to decide which transformations to apply in cases where there are many possible combinations to choose from, and when it may not be feasible to exhaustively try all possibilities.

We believe, and the `Siena` example supports this belief, that experimental program analysis can be applied not just to refactoring problems, but to transformation problems in general. By incrementally designing and running experiments to evaluate individual transformations and their interactions, and decide which transformations to retain and discard, experimental program analysis may provide effective and automated program transformation methodologies.

Given the commonality among program transformation techniques, similar approaches could be applicable to other activities involving transformation. For example, source code instrumentation can be dynamically adapted based on the manner in which the program is being executed at any given point by hypothesizing an efficient instrumentation scheme and then adapting that scheme based on acquired profiling information [23].

5.2 Experimental Regression Fault Analysis

Program analysis is used in debugging for wide-ranging purposes, including to support both explicit fault localization (e.g., [27, 36, 49]) and other activities that simplify the process of debugging faults (e.g., [7, 51]). In most cases, debugging is a highly exploratory process where there are many attributes in, or artifacts from, the program that an investigator may wish to test the effect of. As in the case of the program transformation problem, we consider examples of the challenges and opportunities that this can present for EPA techniques.

In many debugging activities, the amount of code that debuggers must explore to locate a fault may be large. We consider one specific type of debugging where this can be the case: localizing regression faults. A *regression fault* in a program version v_{i+k} is a fault that was not present in a previous version v_i of the program, and was introduced by one or more modifications between v_i and v_{i+k}; hereafter, to simplify our discussion, we refer to regression faults in the context of two sequential program versions v_i and v_{i+1}. In the general case, a programmer attempting to locate a regression fault may have to consider an arbitrary number of source code changes between v_i and v_{i+1}, and any of these individual changes — or in a more complicated scenario, any combination of these changes — may be responsible for the fault. Programmers would clearly prefer to consider as small a set of changes as possible.

To utilize experimentation, EPA techniques systematically manipulate the objects of exploration related to debugging as independent variables, and conduct hypothesis tests to evaluate the effect of manipulating these objects. To explore this further, we have developed a technique that uses experimental program analysis to consider changes in a regression testing setting.

5.2.1 New Experimental Program Analysis Technique

Design

Regression faults present a unique opportunity for debugging because the regression faults responsible for a failing execution e_f are by definition caused, at least in part, by one or more specific changes between two versions v_i and v_{i+1} of a program. We have created an "experimental regression fault analysis technique" that conducts experiments on the changes C between v_i and v_{i+1} in an attempt to isolate the subset of changes $C' \subseteq C$ that are actually responsible for the failing execution e_f caused by the regression fault. The intent behind this technique is to reduce the effort required to locate the regression fault.

As depicted in Table 2, the experiments conducted by this technique sample source code at the function level. The source code changes between v_i and v_{i+1} in the *sampled functions* are identified and applied within these functions to test and evaluate their effect. Thus, the range of source code changes in each sampled function is manipulated as an *independent variable*, and the changes between v_i and v_{i+1} in each selected function are applied as *treatments* to the sampled source code in which those changes occur. *Hypothesis tests* are conducted to determine whether the treatment source code changes reproduce the failing execution e_f. If so, then the technique attempts to systematically isolate a subset of those implicated changes (in a manner determined during an *interim analysis*) until it has found a small-enough subset, or until no further subsets lead to a reproduction of e_f. In terms of our descriptive definition of experimental program analysis, this technique attempts to establish causality between the source code changes and their effect on regression faults in a program.

One difference between this technique and the refactoring technique presented in Section 5.1 is that refactoring attempts to incrementally build a list of ever-increasing refactorings that could be applied, while this technique does the opposite – incrementally reducing a list of changes that could be responsible for the failure. Another difference is that this technique uses the Delta Debugging algorithm [51] to dictate the manner in which treatments are tested, whereas the refactoring technique uses an experimental design.

There are many granularities at which changes could be considered between sequential versions v_i and v_{i+1} of a program. For example, changes could be considered by manipulating the entire source files in which changes occur, manipulating sets of functions forming a component in the program, manipulating individual functions containing changes, or manipulating the actual changes themselves at the finest source code granularity possible. We would expect smaller changes, such as those at the statement level, to generally produce a more precise result, thereby resulting in more useful feedback. However, as the granularity of change becomes more fine, the number of units of change will increase, thereby increasing the cost by causing more experiments to be required to isolate an appropriate subset of changes. Also, the use of smaller changes may increase the possibility of *nonsensical changes* whose application causes compilation of the program to fail due to dependencies between applied and non-applied changes. On the other hand, while using large, coarse-grained levels of change such as entire files of source code may reduce the cost of the technique, it may be less precise, and therefore of less use to debuggers. Thus, and because our goal is to demonstrate the capabilities of experimental program analysis in a debugging activity, we elected to test program changes at the function-level by sampling individual functions with changes between v_i and v_{i+1}.

Threats to Validity in the Experimental Program Analysis Technique

In terms of external validity, the set of changes tested by the technique may not be complete if areas of the program where changes have occurred are not sampled.

One threat to internal validity is the possibility of multiple regression faults interacting with each other, and even masking each others' effects; both of these cases could influence the changes that are reported by the technique. The possibility of dependencies between the changes, an uncontrollable nuisance variable that is another internal validity threat, must also be considered. The granularity level of the changes considered can be a factor influencing the quality of results. Finally, the possibility of nonsensical changes is also an internal validity threat, as some combinations of changes will result in compilation or linking errors. This can result in changes that are not relevant to the regression fault being included so that the program can be properly compiled and executed.

One obvious threat to conclusion validity involves the changes reported. Because our technique uses experimental procedures to drive selection of treatment changes for testing and their assignment to areas of the code rather than exhaustively considering all combinations of changes, it is possible that the reported changes are not the minimal set. (Because our procedures are patterned after Delta Debugging, we apply and test program changes within our local area of search, providing an approximation of a "local minimum" result [51]. Had we exhaustively applied and tested all possible combinations of changes, we could find a "global minimum" result [51].)

One tool that has already explored a similar (non-experimental) approach is the Chianti change analysis research tool [33, 41]. Chianti considers atomic changes between two versions of a program and processes the results of a test suite to estimate the changes that actually affect the program outcome in question. There are many differences between Chianti and the technique we present in this section. First, Chianti's notion of "atomic changes" implies a finer granularity of change than ours. Second, our approach requires only one (failing) test to provide results rather than a series of passing and failing tests from a test suite. Third, rather than using call graphs to determine affecting changes, our approach experimentally applies and tests changes to identify those relevant. Fourth, we report only a *subset* of changes that should contain the regression fault, without suggesting the changes that may be more or less likely to be relevant. Still, the fundamental difference between Chianti and our approach is that we repeatedly apply the changes to manipulate the program and observe their effect, while Chianti analyzes the changes and testing information as is – without manipulating the program itself.

5.2.2 Pilot Study

We investigate the potential of the experimental program analysis conducted by this technique using the `flex` program as an artifact. `flex` is a medium-sized program written in C, and contains 15,297 lines of code among 163 procedures. We selected `flex` because it is a non-trivial, publicly available program with known (seeded) faults and test suites. The seeded faults and test suites were created by SIR researchers [10] not involved in the work described in this article. The primary goal of this study was to formatively investigate whether our experimental program analysis approach for regression fault analysis has the potential to address some of the problem's inherent difficulties.

We selected two versions of `flex`, versions 2.4.3 and 2.4.7, and a single seeded fault within these versions. In total, 1,329 lines of code had changed within 27 of the 163 procedures in `flex` between these two versions. We selected these versions for two reasons. First, versions 2.4.3 and 2.4.7 are sequential versions as provided by SIR researchers. Second, we did not want to select a fault that was too easy or difficult to detect, as the former may be easier for debuggers to locate by themselves, while the latter may not be detected by any test cases, and our technique requires a failing execution to isolate changes. Therefore, we chose a criterion of a fault detected by at least one test case in a functional specification-based test suite, and by less than 20% of those test cases. Such a fault existed between versions 2.4.3 and 2.4.7.

Study Design

The *independent variable* evaluated in this study is the approach for performing regression fault analysis. We consider two approaches:

1. As a treatment approach, we use the experimental regression fault analysis technique just presented.

2. As a control approach, we consider a technique that applies and tests all possible (2^{27}) combinations of changes to find the optimal minimal subset responsible for the regression fault.

With regard to the control approach, the time taken to exhaustively consider each of the 2^{27} changes is prohibitively expensive – a reason why such a technique is not practical in real-world debugging settings. Thus, as a heuristic for estimating the size of this global minimum, we exhaustively applied and tested every combination of change *within the local minimum.* To estimate the time that would have been taken to apply and test each of these changes, we extrapolated the time taken by our experimental regression fault analysis technique to test its subset of changes. While it is possible that this extrapolation will not be a precise measure of the time that would be taken in the exhaustive case, we were interested in the *relative magnitude* in the differences in time between the two techniques in order to investigate the amount of time that might be saved by the experimental program analysis approach.

We utilize four metrics as *dependent variables* to measure the cost and the effectiveness of our regression fault analysis technique:

1. We measure the number of isolated functions containing relevant changes between the two versions of `flex` considered in this study.

2. We measure the number of isolated lines of code in the isolated functions.

3. We measure the number of "evaluations" made by the approach, where an evaluation is the application and testing of a set of changes C_d.

4. We measure the time taken to execute the technique and consider all of the evaluations reported by our third dependent variable.

As *objects of analysis*, we used the first five test cases in the version's test suite that reveal the fault and meet the above criterion. (We selected more than one test case because we did not want the particular test case selected to be an influencing factor on our results.) We individually ran our technique implemented through a series of Perl scripts on Linux machines containing Dual Core AMD Opteron 250 2.4 GHz 64-bit processors with four gigabytes of memory.

Threats to Validity in the Study Design

As with the refactoring study, we selected `flex` as a subject because it is a real program from a software infrastructure repository used by many researchers. While this was done with external validity in mind, internal validity is an issue facing this study because its observed results may be due to particular characteristics of `flex`, the two versions 2.4.3 and 2.4.7, and the five test cases that were selected. Also, this study considered only one fault, and different results may be observed using other faults.

In terms of construct validity, the extrapolated estimation of the time taken by an exhaustive technique may not be completely accurate. However, this measure primarily aims to show the magnitude of difference between the experimental regression fault analysis technique and an exhaustive technique – not to provide an exact difference between the cost of the two approaches. Also, in terms of the gathered timing information, it is possible that times would vary across different machines, or even on the same machine over multiple runs. To limit these concerns, we ran our techniques on machines with the same hardware specifications, and ensured that our techniques were the only active user processes on the machines during their execution. Finally, our measures do not account for costs and benefits related to actual use by engineers of the data produced by the approach.

Results

Table 7 summarizes the number of isolated changes for each of the five selected test cases t_1–t_5, as well as the number of evaluations — compiling changes into the version, running the test case, and analyzing the execution — required to isolate the changes for each test case. The estimated optimal minimal number of functions and source code lines that could have been isolated are shown in the "Exhaustive Combination Search" area, along with the 2^{27} evaluations that would be required to test all combinations of changes

	t_1	t_2	t_3	t_4	t_5	*Mean*
Experimental Regression Fault Analysis						
Isolated Functions	8	9	8	7	7	7.8
% Total Functions	*29.6%*	*33.3%*	*29.6%*	*25.9%*	*25.9%*	*28.9%*
Isolated LOC	221	417	404	394	394	366
% Total LOC	*16.6%*	*31.4%*	*30.4%*	*29.7%*	*29.7%*	*27.5%*
Evaluations	109	115	101	87	87	99.8
Time (min:sec)	10:09	10:09	10:31	8:54	8:37	9:40
Exhaustive Combination Search						
Isolated Functions	7	8	7	6	6	6.8
% Total Functions	*25.9%*	*29.6%*	*25.9%*	*22.2%*	*22.2%*	*25.2%*
Isolated LOC	219	415	402	392	392	364
% Total LOC	*16.5%*	*31.2%*	*30.3%*	*29.5%*	*29.5%*	*27.4%*
Evaluations	134217728					
Time (days:hrs:min:sec)	9075:11:27:27					

Table 7: Summary of the number of isolated changes, in terms of functions and lines of code, responsible for the seeded `flex` fault using five test cases. The cost of identifying these changes is also shown in terms of evaluations performed by the technique and time (Minutes:Seconds for the experimental regression fault analysis approach and Days:Hours:Minutes:Seconds for the exhaustive approach). For the exhaustive combination search, the same number of evaluations (2^{27}) were required for all five test cases.

to isolate this minimum and an estimate of the time required to test these changes. The result for the percentage of total functions and lines of code isolated are based on the 27 total functions and 1,329 lines of code that changed across versions 2.4.3 and 2.4.7 of `flex`. Time measurements are in Minutes:Seconds for the experimental program analysis approach, and Days:Hours:Minutes:Seconds for the exhaustive approach.

From the 27 initial changes, depending on the particular test case used to reproduce the failing execution, our technique isolated between seven and nine changes, or approximately 26%–33% of the original 27 changes, needed to expose the regression fault. On average, this process took less than 10 minutes for each test case. Furthermore, depending on the failing test case, between 17%–31% of the total lines of code that changed between these two `flex` versions were isolated by the experimental regression fault analysis technique.

The performance of the EPA technique, in terms of the precision and size of the isolated functions and source code, was comparable to that of the exhaustive approach. For each test case, as shown in Table 7, exhaustively considering all combinations of changes would have eliminated only one fewer change than the EPA technique. In analyzing these results further after the completion of this study, we found that the same additional change was eliminated by the exhaustive approach for each of the five test cases: a two-line change in the `version.h` header file of `flex`. Yet clearly the cost of exhaustively considering all combinations of changes is not justified by the elimination of this one two-line change, as exhaustively applying and testing all 2^{27} combinations of changes would require years (on the machines we considered in this study).

One possible change that could be made to our experimental regression fault analysis technique is an "a priori" analysis that would study the dataflow relationships between segments of source code changes. The goal of this approach would be to use this information to ensure that changes that depend on each other are made together to reduce the number of wasted evaluations that are applied and tested. The essential strategy of this approach — grouping changes based on their dependencies — is similar to the Hierarchical Delta Debugging approach [28] that aims to apply groups of related changes to reduce wasted tests.

Another possible improvement to our approach that could further reduce the changes identified by the technique would be to exhaustively consider all combinations of the changes *identified by the EPA technique*. For example, for test cases t_4 and t_5, seven changes were isolated by the technique. We could exhaustively consider all 2^7 combinations of these seven changes, which would eliminate the two-line change in `version.h` and leave us with six changes. However, this would require an additional 128 tests, which would more than double the total number of changes required by a technique conducting such "post-hoc" analysis, as only 87 changes were required by the EPA technique to isolate the original seven changes.

Discussion: The Benefit of Experimental Program Analysis

Our goal in conducting this study was to formatively investigate the potential of a purely experimental approach to debugging regression faults – without attempting to fine-tune the technique so that it performs a more sophisticated analysis. Our investigation revealed that experimental program analysis can contribute to an important debugging activity: isolating the source changes that may be responsible for regression faults. It also provides a means for effectively reducing the exploration space of changes that an engineer must consider when debugging a regression fault, which corroborates results from related work in a similar debugging activity [51].

Overall, experimental program analysis experienced success in this setting by experimentally testing and narrowing the set of relevant changes until a smaller result could not be obtained. However, two characteristics of this strategy deserve recognition. First, despite the potential number of program changes in a problem instance, as well as dependencies therein, the experimental program analysis approach proposed here was able to systematically control the changes that may be relevant to the regression fault, and experimentally test the relevance of these changes and dependencies, leading to a precise solution. Second, the sampling procedures of this EPA technique considered only functions in the program containing changes, which helped to reduce the cost of the program analysis.

In future work, it may be that considering a finer notion of "change," such as that proposed in related work [33, 41], would result in more precise results. However, due to the increased number of units of change that would result from this strategy, it could come at an unacceptably high cost by requiring that a greater number of combinations of changes be experimentally applied and tested to determine their relevance. Future investigation investigating these tradeoffs would be interesting.

5.3 Experimental Dynamic Invariant Detection

Aiding program understanding is one of the most well-known, traditional purposes of program analysis. As an example of an activity in this problem area, we consider dynamic invariant detection. Most invariant detection techniques investigate relationships between variables (i.e., invariants) in the program at specific program points. However, there are potentially millions of relationships between variables at the various program points that might be considered. A challenge, then, is to evaluate the potential invariants in a cost-effective manner, as *any* evaluation is likely to consume resources. This challenge must be met in a manner that ensures that the invariants that are reported are accurate and not spurious.

In an experimental setting, these potential relationships between variables can be viewed as a population under study, and the items that are manipulated to learn about this population are the independent variables. Such experiments can be designed to sample these large populations of potential invariants to control cost at the risk of obtaining results that may not be conclusive, or may be incomplete. Experiments can also leverage different designs in order to efficiently assign specific instances of the independent variable manipulations (treatments) to units in the sample. Somewhat similar to the drawbacks of sampling, the power of the conclusions drawn from such designs may diminish, but at the benefit of a less expensive (and perhaps more cost-effective) design.

This section introduces an improvement to an existing EPA technique that is designed to characterize program behavior. The improvement aims to increase the cost-effectiveness of this technique by altering its experiment design.

5.3.1 Improving Cost-effectiveness in an Experimental Program Analysis Technique Design

Daikon [15] is an implementation of a technique that can be characterized as experimental program analysis, and that infers likely program invariants from execution traces using a "library" of predefined invariant types. At each program point of interest, all possible invariants that might be true are evaluated by observing the values of variables during program executions. If an invariant is violated in an execution, it is discarded (falsified). If an invariant has not been falsified in any execution and has been evaluated enough that Daikon has sufficient confidence in its validity, it is reported as a likely invariant.

Daikon can be considered a technique in the program understanding area because the invariants reported by the technique help programmers learn about and understand the program, a well as provide support for other software-engineering-related activities. In Daikon, the relationships between a program's variables at specific points are the population that is learned about. A sample of that population, in terms of candidate invariants, is evaluated by applying the data from execution traces (the independent variable) and conducting an hypothesis test about whether the invariant is valid based on the applied data. Thus, Daikon is attempting to *characterize a population* (potential relationships between variables in a program) through the use of the data in execution traces. We showed how Daikon can be mapped to the experimental program analysis operational definition in Table 2. For details as to how Daikon operates algorithmically, we refer readers to the work presenting the technique [15].

One important limitation for Daikon is that it can take a great deal of time to process the provided execution traces in order to report invariants, especially if those execution traces are large, or great in number. Ideally, we would like to improve Daikon so that it does less processing of execution traces without sacrificing the quality of the reported invariants or reporting "false positive" invariants.

To do this, we have leveraged principles from a well-known practice in the traditional experiment design literature: *fractional factorial designs* [26]. Fractional factorial designs reduce costs by achieving a balance between the factors under consideration in experiments without testing all possibilities that could be considered. Unlike complete designs, where all possible treatments are applied to all available units in the sample, fractional factorial designs spread the treatments across units of analysis in the sample. Balance is achieved by considering combinations of k factors, and selecting the combinations that are used in the experiments based on the coverage of the factors' levels that is achieved.

We used these principles with respect to Daikon to create an experiment design that chooses a fraction of the treatment combinations (i.e., execution trace data applied to candidate invariants) in a manner designed to be adequate for the analysis. (Our construct for "adequacy" is described later.) The treatment combinations chosen for evaluation are selected in an attempt to achieve a balance among the coverage of the sampled program points through the careful selection of execution traces.

In our implementation of a fractional design within Daikon, we consider two factors: the execution traces applied to the invariants, and the program points at which those invariants are evaluated. If the execution traces versus the program points are viewed as a two-dimensional grid, a complete design fills as many of these boxes as possible, while a fractional design balances the boxes that are filled across both the execution trace and program point factors.

To achieve this type of behavior in Daikon, we modified the tool so that it considers only 50% of the possible comparisons of execution traces to program points. For each execution trace processed, the data regarding variable values in that trace, which are applied to prospective invariants, are considered for only 50% of the program points in that trace. Furthermore, the program points that are selected are those that have the *least* coverage thus far in terms of the number of traces whose data has been applied to invariants at those points; we term these the *neediest* program points because they are in the most need of further observations to validate their invariants. Thus, the subset of execution that is applied to the sampled program points is carefully selected with the goal of reducing cost while controlling coverage.

These decisions regarding the neediest points are made based on accumulated feedback about what program points have been covered so far, and how often they have been covered during the analysis, which is done using previously processed data from execution traces. By selecting the 50% of the program points that are least covered in each trace, our design seeks to lower costs in a reasonable way while balancing the distribution of observations so that certain program points and their prospective invariants are not “starved.”

Threats to Validity in the Experimental Program Analysis Technique

In terms of the limitations that are new to Daikon as a result of our modifications to the technique, the particular manner in which we select the treatment combinations of execution data to apply to prospective invariants is an internal validity threat, as the particular invariants that are reported and discarded may change if different execution data were distributed among different invariants.

Conclusion validity is the primary increased threat to the technique. Because we (purposefully) consider only a fraction of the available data, the power of the technique's results are reduced in an effort to help control the costs of the technique.

5.3.2 Pilot Study of Cost-effectiveness Improvement

To gauge the performance of Daikon when the tool utilizes a fractional experiment design, and to help demonstrate the use of experimental program analysis in application to program understanding problems, we implemented this design in Daikon version 3.1.7 [9]; we refer to this tool as Daikon$_{frac}$. We then investigated the capabilities of this EPA technique using the `Space` program as a subject.

`Space` functions as an interpreter for an array definition language (ADL). `Space` is written C, and contains 136 functions and 9,564 lines of source code. Previous studies [19, 45] have resulted in the creation of 13,525 test cases and 1,000 branch-coverage-adequate test suites for `Space`. (Like our previous subjects, `Space`, along with numerous test suites, is publicly available [10].)

Study Design

The *independent variable* evaluated in this study is the experiment design used within Daikon to select the treatment combinations that are evaluated during the technique's analysis. We consider three approaches in this study:

1. As a treatment approach, we consider the use of Daikon with the fractional design (Daikon$_{frac}$).

2. As our first control approach, we consider the use of Daikon where 50% of the treatment combinations are randomly chosen – not within the context of the “neediest” program points. This helps us assess how an alterative strategy for reducing the costs of Daikon compares with our approach. We label this approach Daikon$_{rand}$.

3. As our second control approach, we consider the use of Daikon as originally designed: with a complete design considering the maximum number of treatment combinations possible. This variable serves as a baseline for assessing the relative costs and benefits of Daikon$_{frac}$ and Daikon$_{rand}$ in comparison to the original implementation of Daikon.

This study utilizes two metrics as *dependent variables* to measure the cost and the effectiveness of our modifications to the experiment design within Daikon:

1. As our first dependent variable, we measure the number of additional false positive invariants reported by the technique that would not have been reported if the original version of Daikon had been used. This helps us assess the drawbacks in using a modified experiment design to reduce the cost of Daikon's analysis. We chose to measure false positives because we expect that inaccurate invariants will be the greatest barrier to adopting a modified experiment within Daikon that tests a fraction of the possible treatment combinations.

2. As our second dependent variable, we measure the amount of analysis saved by Daikon through the use of a modified experiment design. This is measured as the number of comparisons of execution trace data to candidate invariants at the sampled program points. This dependent variable helps us assess the benefits of using a modified experiment design by facilitating comparisons of the cost of the techniques. We chose this measure because it is normalized with respect to the machines on which Daikon could be run.

As *objects of analysis*, we randomly selected five test suites from the 1,000 test suites for `Space` that are available from the software infrastructure repository [10], and generated execution trace files for each execution of each test suite. We then compared the original Daikon implementation with Daikon$_{frac}$ to detect invariants using all five suites.

Threats to Validity of Study Design

The size of `Space` is an external validity issue facing this study, as `Space` may not be representative of real-world programs that are often much larger in size. We accepted this tradeoff in order to study a program that has already been used by other researchers working with Daikon, and that has an infrastructure of non-trivial test suites in place to facilitate our study's design. We also expect that, because each test suite was designed to be branch-coverage-adequate, the test suites are likely to resemble suites that might be created for a program such as `Space` in a real-world software development setting.

Daikon will report different likely invariants for programs based on the execution traces it is given to analyze. Thus, the particular test suites we chose may be responsible for the results seen in this study, as

Cost: False Positive Invariants	
Daikon: Number of Reported Invariants	33415
Daikon$_{frac}$: Additional False Positives	1738 *(5.2%)*
Daikon$_{rand}$: Additional False Positives	5790 *(17.3%)*

Benefit: Observations Saved	
Daikon: Number of Observations	176288
Daikon$_{frac}$: Observations Saved	59638 *(33.8%)*
Daikon$_{rand}$: Observations Saved	44910 *(25.5%)*

Table 8: Results of using a fractional design in Daikon.

other results might have been acquired using other execution traces from other test suites. We attempted to mitigate this threat by randomly selecting multiple test suites generated by other researchers.

In terms of construct validity, there are many measures we could have used to gauge the cost and benefit of using a fractional design. For example, we could have directly measured the savings in terms of time rather than the number of variable value comparisons to invariants. We chose the latter because it is normalized with respect to the machines on which Daikon could be run. Furthermore, comparing variable values to invariants for large sets of execution traces dominates the cost of Daikon, so we focus on the savings achieved during this expensive process. As in our prior study, however, our measures do not account for costs and benefits related to actual use by engineers of invariants reported by the approaches; our results thus bear on effectiveness and efficiently of the techniques in producing invariants, but not necessarily on usefulness of the reported invariants in practice.

Results

Table 8 summarizes the results of using the Daikon techniques. The top of the table summarizes the average number of invariants reported by Daikon for the five branch-coverage-adequate test suites, and the average number of additional false positives reported by Daikon$_{frac}$ and Daikon$_{rand}$. The bottom of the table summarizes the average number of observations required by Daikon to process the execution traces from each test suite and test the candidate invariants, as well as the savings achieved by Daikon$_{frac}$ and Daikon$_{rand}$.

We first consider, in detail, the results of Daikon$_{frac}$ as compared to the original implementation of Daikon. As can be seen, with a 50% fractional design, Daikon$_{frac}$ required, on average, two-thirds of the observations that would be required by the original Daikon implementation. (Two-thirds of the observations were required, rather than 50% of the observations, due to the number of candidate invariants that needed to be tested at each selection of 50% of program points.) Furthermore, these savings came at a cost of, on average, about 5% of precision, as the extra invariants that were not falsified using Daikon$_{frac}$.

We were also interested in how well Daikon$_{frac}$ improved the cost-effectiveness of the technique relative to other approaches that might have been considered. Comparing the number of additional false positive invariants reported by Daikon$_{frac}$ and Daikon$_{rand}$ indicates that, for the particular `Space` test suites selected for this study, Daikon$_{rand}$ reported over three times the number of false positive invariants than did Daikon$_{frac}$. Furthermore, Daikon$_{frac}$ experienced greater savings in terms of analysis cost than did Daikon$_{rand}$. This appears to be due to the fact that Daikon$_{frac}$ sought to give adequate coverage to program points that were not covered by many execution traces, and did not have as many invariants to test as more traces were used.

5.3.3 Discussion: The Benefit of Experimental Program Analysis

Our change to Daikon, and the study of that change, suggest that the cost-effectiveness of Daikon may be improved by incorporating advanced designs into the experiments conducted by the tool. The use of such experiment designs and statistical principles serves to provide a unique experimentation-based solution to an important program analysis problem. We suspect that other such opportunities exist in other program analysis techniques, both in the program understanding domain and elsewhere.

6 Related Work

6.1 Experimentation in Software Engineering Research

There is a growing body of knowledge on the employment of experimentation to assess the performance of, and evaluate hypotheses related to, software engineering methodologies, techniques, and tools. For example, Wohlin et al. [47] introduce an experimental process tailored to the software engineering domain, Fenton and Pfleeger [16] describe the application of measurement theory in software engineering experimentation, Basili et al. [3] illustrate how to build software engineering knowledge through a family of experiments, and Kitchenham et al. [25] provide guidelines for conducting empirical studies in software engineering. However, these approaches focus on experimentation to evaluate methodologies, or techniques and tools, whereas we focus on its use as the driving force behind program analysis techniques.

There are also instances in which software engineering techniques utilize principles of experimentation as part of their operation (not just for hypothesis testing). For example, the concept of sampling is broadly used in software profiling techniques to reduce their associated overhead [1, 13, 27], and experiment designs are utilized in interaction testing to drive an economic selection of combinations of components to achieve a target coverage level (e.g., [8, 12]). We believe that, in many cases, techniques that utilize certain principles of experimentation may in fact be appropriately characterized as EPA techniques, allowing investigators opportunities such as the use of advanced experiment designs to improve the cost-effectiveness of their techniques, and providing insights into technique limitations through the assessment of validity threats.

6.2 Experimentation in Program Analysis

Zeller is the first to have used the term "experimental" in application to program analysis techniques [50]. Our work differs from Zeller's, however, in several ways.

First, Zeller's goal was not to precisely define experimental program analysis, but rather to provide a "rough classification" of program analysis approaches and "to show their common benefits and limits", and in so doing, to challenge researchers to overcome those limits [50, page 1]. Thus, in discussing specific analysis approaches, Zeller provides only informal definitions. In this work, we provide a more precise notion of what experimental program analysis is and can be.

Second, our view of experimental program analysis differs from Zeller's in several ways. He writes: "Experimental program analysis generates findings from multiple executions of the program, where the executions are controlled by the tool", and he suggests that such approaches involve attempts to "prove actual causality", through an (automated) series of experiments that refine and reject hypotheses [50, page 3]. When considering the rich literature on traditional experimentation, there are several drawbacks in the foregoing suggestions. Experimentation in the scientific arena can be exploratory, descriptive, and explanatory, attempting not just to establish causality but, more broadly, to establish relationships and characterize a population [24, 29, 34]. For example, a non-causal question that can clearly be addressed by experimentation is, "is the effect of drug A applied to a subject afflicted by disease D more beneficial than the effect of drug B?" EPA techniques can act similarly. Further, experimentation (except in a few situations) does not provide "proofs"; rather, it provides probabilistic answers — e.g., in the form of statistical correlations.

Finally, Zeller's explication contains no discussion of several concepts that are integral to experimentation, including the roles of population and sample selection, identification of relevant factors, selection of dependent and independent variables and treatments, experiment design, and statistical analysis. He also does not discuss in detail the nature of "control", which requires careful consideration of nuisance variables and various forms of threats to external, internal, construct, and conclusion validity. All of these fundamental experimentation-related notions are present in our definition, and the utility of including them is supported.

6.3 Search-based Software Engineering

Harman and Jones [21] defined search-based software engineering as the reformulation of a software engineering task as a search problem. In contrast, through experimental program analysis, we attempt to formulate the targeted program analysis as an experimentation process.

Search-based software engineering involves three primary components [21]. The first component involves casting solutions to the problem domain in a representation that is amenable to symbolic manipulation. This is done so that a search can be conducted by making changes to this representation as a solution is sought. The second component involves defining a fitness function that measures the quality of prospective solutions identified by the search. These fitness functions should have a landscape that is amenable to being searched in order to find an optimal solution. Thus, fitness landscapes that are largely flat and unchanging, or that have sharp and sudden shifts that can be easily missed by a search technique, are generally not ideal. The third component involves operators for manipulating the representation of a possible solution as a search is conducted [21]. Search-based software engineering involves the use of metaheuristic techniques in software engineering settings [20, 21]. Metaheuristic techniques such as genetic algorithms [21, 40], tabu searches [18, 30, 40], and simulated annealing [31, 40] are used in place of optimization techniques such as linear programming and dynamic programming due to the size of large software engineering problems, such as those addressed by combinatorial testing.

Clearly, there are some similarities between the formulations of search-based and experimental program analysis. For example, the fitness function(s) of search-based software engineering techniques can be mapped to the dependent variables feature of experimental program analysis. The units that are measured in search-based software engineering techniques using fitness functions are similar to the treatments feature of experimental program analysis. The process guiding search-based software engineering techniques could be mapped to the interim analysis feature of experimental program analysis. And both approaches have their unique toolset to support cost-effective analysis.

Still, as two separate formulations of program analysis techniques, search-based software engineering and experimental program analysis techniques offer different perspectives with unique strengths. For example, search-based software engineering techniques may be desirable when the goal is to search for and identify an *optimal* solution. On the other hand, experimental program analysis techniques may be more appropriate when it is desirable to analyze causality or characterize a population with the support of measures of confidence to evaluate the accuracy or completeness of the identified solutions, providing a *different type of solution* than search-based software engineering techniques. We believe that researchers will find some research problems where search-based software engineering solutions are more attractive, and other problems where experimental program analysis solutions are preferable.

6.4 Static and Dynamic Analysis

One additional question of interest involves the relationship between experimental program analysis and other "types" of analyses, such as "static" and "dynamic" analysis. The characteristics of and relationships between techniques, and taxonomies of techniques, have been a topic of many research papers (see, e.g., [2, 14, 43, 48, 50]). In these papers, static techniques are generally described as those that operate on fixed representations of programs and have no knowledge of the types of execution behaviors those programs might exhibit in practice, while dynamic techniques utilize information gleaned from program executions.

While our goal is not to taxonomize, we argue that experimental program analysis is not constrained to the traditional static or dynamic classification, but rather, is orthogonal to it. The EPA paradigm focuses on the *type* of analysis performed: namely, whether tests and purposeful changes are used to analyze software. As such, it can overlap with *both* static and dynamic analysis techniques, as illustrated by program refactoring (a static technique) and Daikon (a dynamic technique).

Experimental program analysis thus presents a perspective that is not provided by static or dynamic analysis techniques by offering (1) methodologies for manipulating independent variables of interest to test their effects on dependent variables, (2) procedures for conducting and adjusting hypothesis tests in program analysis contexts, (3) strategies for systematically controlling sources of variation during these tests, (4) experiment designs and sampling techniques to reduce the costs of experimentation, and (5) mechanisms to generate confidence measures in the reliability and validity of the results. We believe that such advantages, which allow EPA techniques to address research questions in ways that other program analysis techniques cannot, will motivate the development of EPA techniques.

7 Conclusions and Future Work

While researchers have created various program analysis techniques incorporating features of experimentation, the notion of incorporating such features has not previously been investigated in its own right. This article has taken this step, and formalized experimental program analysis as a program analysis paradigm. We have shown that by following this paradigm, and using our operational definition of experimental program analysis, it is possible to identify limitations of EPA techniques, improve existing techniques, and create new techniques.

There are many intriguing avenues for future work on experimental program analysis. One direction involves the use of the paradigm to solve software engineering problems in more cost-effective ways by adapting existing non-experimental techniques or creating new EPA techniques. In this article we have considered only a few examples of how to adapt existing techniques or create new techniques, but we believe that there are many others. We conjecture that investigating software engineering problems from an experimental program analysis perspective can reveal new opportunities for addressing them.

A second direction for future work, as we have mentioned, involves automation opportunities for EPA techniques. Thus far, we have focused on the automation of experimental program analysis tasks and the advantages therein. It seems likely, however, that the selection of the *approach* for a task can be automated as well. For example, EPA techniques could be encoded to consider multiple experimental designs (e.g., blocking, factorial, split-plot, latin square), and select that which is best suited for a specific instance of a problem. Improvements such as these may allow techniques to perform more efficiently, thereby making them more affordable to solve different classes of problems.

A third direction for future work with somewhat broader potential impacts involves recognizing and exploiting differences between experimental program analysis and traditional experimentation in some other fields. As Section 4 points out, there are several such interesting differences including, for example, the potential for EPA techniques to cost-effectively consider enormous numbers of treatments. It is likely that further study of experimental program analysis will open up intriguing new problems in the fields of empirical science and statistical analysis.

In closing, we believe that experimental program analysis provides numerous opportunities for program analysis and software engineering research. We believe that framing program analysis problems as an experiment offers access to new approaches that are not available in existing forms of analysis — at least for particular classes of analysis tasks — including procedures for systematically controlling sources of variation while analyzing software systems, experimental designs and sampling techniques to reduce the cost of generalizing targeted aspects of a program, procedures for conducting and adjusting hypothesis tests in program analysis contexts, and mechanisms to generate confidence measures in the reliability and validity of the results. We believe that such advantages will lead to significant advances in program analysis research and in the associated software engineering technologies that this research intends to improve.

8 Acknowledgments

We thank Kent Eskridge and David Marx of the Statistics Department at the University of Nebraska–Lincoln for feedback on our definition of experimental program analysis. This work has been supported by NSF under awards CNS-0454203, CCF-0440452, and CCR-0347518 to University of Nebraska–Lincoln.

References

[1] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 168–179, Snowbird, Utah, U.S.A., June 2001.

[2] T. Ball. The concept of dynamic analysis. In *Proceedings Seventh European Software Engineering Conference held jointly with the Seventh ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 216–234, Toulouse, France, Sept. 1999.

[3] V. R. Basili, F. Shull, and F. Lanubile. Using experiments to build a body of knowledge. In *Proceedings of the NASA Software Engineering Workshop*, pages 265–282, Dec. 1999.

[4] M. Bowman, L. C. Briand, and Y. Labiche. Multi-objective genetic algorithm to support class responsibility assignment. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 124–133, Oct. 2007.

[5] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building.* Series in Probability and Mathematical Statistics. Wiley, New York, 1st edition, 1978.

[6] B. R. Childers, J. W. Davidson, and M. L. Soffa. Continuous compilation: A new approach to aggressive and adaptive code transformation. In *National Science Foundation Workshop Next Generation Software*, Nice, France, Apr. 2003.

[7] J. D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 210–220, Rome, Italy, July 2002.

[8] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, July 1997.

[9] Daikon invariant detector distribution. http://pag.csail.mit.edu/daikon/. Last accessed: April 23, 2008.

[10] H. Do, S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 60–70, Redondo Beach, California, U.S.A., Aug. 2004.

[11] B. Du Bois, S. Demeyer, and J. Verelst. Refactoring—improving coupling and cohesion of existing code. In *Proceedings of the* 11th *IEEE Working Conference on Reverse Engineering*, pages 144–151, Delft, The Netherlands, Nov. 2004.

[12] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing: Experience report. In *Proceedings of the* 19th *International Conference on Software Engineering*, pages 205–215, Boston, Massachusetts, U.S.A., May 1997.

[13] S. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions on Software Engineering*, 31(4):312–327, Apr. 2005.

[14] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *Proceedings of the ICSE'03 Workshop Dynamic Analysis*, pages 24–27, Portland, Oregon, U.S.A., May 2003.

[15] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001.

[16] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* Course Technology, 2nd edition, 1998.

[17] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, 1999.

[18] F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publishers, Boston, 1997.

[19] T. L. Graves, M. J. Harrold, J. M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, Apr. 2001.

[20] M. Harman. The current state and future of search based software engineering. In *Proceedings of the* 29th *International Conference on Software Engineering*, pages 20–26, Minneapolis, Minnesota, U.S.A., May 2007.

[21] M. Harman and B. F. Jones. Search-based software engineering. *Information & Software Technology*, 43(14):833–839, Dec. 2001.

[22] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity.* Prentice-Hall, 1996.

[23] A. J. Kinneer, M. B. Dwyer, and G. Rothermel. Sofya: Supporting rapid development of dynamic program analyses for Java. In *Proceedings of the* 29th *International Conference on Software Engineering*, pages 51–52, Minneapolis, Minnesota, U.S.A., May 2007.

[24] R. E. Kirk. *Experimental Design: Procedures for the Behavioral Sciences.* Brooks/Cole Publishing, 3rd edition, 1995.

[25] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, Aug. 2002.

[26] R. O. Kuehl. *Design of Experiments: Statistical Principles of Research Design and Analysis.* Brooks/Cole Publishing Company, Pacific Grove, California, U.S.A., 2nd edition, 2000.

[27] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 15–26, Chicago, Illinois, U.S.A., June 2005.

[28] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. In *Proceedings of the* 28th *International Conference on Software Engineering*, pages 142–151, Shanghai, China, May 2006.

[29] D. C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, New York, 4^{th} edition, 1997.

[30] K. Nurmela. Upper bounds for covering array by tabu search. *Discrete Applied Mathematics*, 138(1–2):143–152, 2004.

[31] K. Nurmela and P. R. J. Ostergard. Constructing covering designs by simulated annealing. Technical report, Digital Systems Laboratory, Helsinki University of Technology, 1993.

[32] RefactorIt—Aqris Software. http://www.aqris.com/display/ap/RefactorIt/. Last accessed: April 23, 2008.

[33] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proceedings of the* 19^{th} *Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Language, and Applications*, pages 432–448, Vancouver, British Columbia, Canada, Oct. 2004.

[34] C. Robson. *Real World Research*. Blackwell Publishers, Malden, Massachusetts, U.S.A., 2^{nd} edition, 2002.

[35] J. R. Ruthruff. Experimental program analysis: A new paradigm for program analysis. In *Proceedings of the* 28^{th} *International Conference on Software Engineering Doctoral Symposium*, pages 977–980, Shanghai, China, May 2006.

[36] J. R. Ruthruff, M. Burnett, and G. Rothermel. Interactive fault localization techniques in a spreadsheet environment. *IEEE Transactions on Software Engineering*, 32(4):213–239, Apr. 2006.

[37] J. R. Ruthruff, S. Elbaum, and G. Rothermel. Experimental program analysis: A new program analysis paradigm. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 49–59, Portland, Maine, U.S.A., July 2006.

[38] S. Siegel and N. Castellan Jr. *Non-parametric Statistics for the Behavioral Sciences*. McGraw Hill, Boston, 1998.

[39] D. Siegmund. *Sequential Analysis: Tests and Confidence Intervals*. Springer-Verlag, New York, 1985.

[40] J. Stardom. Metaheuristics and the search for covering and packing arrays. Master's thesis, Simon Fraser University, 2001.

[41] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in Java programs using change classification. In *Proceedings of the* 14^{th} *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 57–68, Portland, Oregon, U.S.A., Nov. 2006.

[42] S. K. Thompson. *Sampling.* Series in Probability and Mathematical Statistics. Wiley, New York, 2^{nd} edition, 2002.

[43] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[44] W. M. K. Trochim. *The Research Methods Knowledge Base.* Atomic Dog Publishing, Cincinnati, Ohio, 2^{nd} edition, 2001.

[45] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 44–53, Bethesda, Maryland, U.S.A., Nov. 1998.

[46] A. Wald. *Sequential Analysis.* Wiley, New York, 1947.

[47] C. Wohlin, P. Runeson, M. Host, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering.* Kluwer Academic Publishers, Boston, 2000.

[48] M. Young and R. N. Taylor. Rethinking the taxonomy of fault detection techniques. In *Proceedings of the 11^{th} International Conference on Software Engineering*, pages 53–62, Pittsburgh, Pennsylvania, U.S.A., May 1989.

[49] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10^{th} ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1–10, Nov. 2002.

[50] A. Zeller. Program analysis: A hierarchy. In *Proceedings of the ICSE'03 Workshop on Dynamic Analysis*, pages 6–9, Portland, Oregon, U.S.A., May 2003.

[51] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, Feb. 2002.