

Experimental Program Analysis: A New Program Analysis Paradigm

Joseph R. Ruthruff, Sebastian Elbaum, and Gregg Rothermel
Department of Computer Science and Engineering
University of Nebraska–Lincoln
Lincoln, Nebraska 68588-0115, U.S.A.
{ruthruff, elbaum, grother}@cse.unl.edu

ABSTRACT

Program analysis techniques are used by software engineers to deduce and infer characteristics of software systems. Recent research has suggested that a new form of program analysis technique can be created by incorporating characteristics of experimentation into analyses. This paper 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 show how the use of an experimental program analysis paradigm can help researchers identify limitations of analysis techniques, improve existing experimental program analysis techniques, and create new experimental program analysis techniques.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—*debugging aids, testing tools*; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—*program analysis*

General Terms

Experimentation

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 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSTA'06, July 17–20, 2006, Portland, Maine, USA.
Copyright 2006 ACM 1-59593-263-1/06/0007 ...\$5.00.

Recently, researchers have begun to consider a new approach to program analysis that harnesses principles of experimentation [21, 31]. Such *experimental program analysis* techniques might be able to draw inferences about the properties of software systems in cases in which more traditional analyses have not succeeded.

As experimentalists, we do indeed recognize many characteristics of scientific experimentation that already are, or could potentially be, utilized by program analysis techniques. These characteristics include the formulation and testing of hypotheses, the iterative process of exploring and adjusting these hypotheses in response to findings, the use of sampling to cost-effectively explore effects relative to large populations in generalizable manners, the manipulation of independent variables to test effects on dependent variables while controlling other factors, the use of experiment designs to facilitate the cost-effective study of interactions among multiple factors, and the use of statistical analyses to assess results.

Anyone who has spent time debugging a program will recognize the relevance of several of the foregoing characteristics to that activity. Debuggers routinely form hypotheses about the causes of failures, conduct program runs (in 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 this “experiment”, 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 (e.g., [16, 22, 30, 32]).

In this paper 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 scientific experimentation. Building on this characterization, we present an operational definition of a new paradigm for program analysis, *experimental program analysis*, and we show how analysis techniques can be characterized in terms of this paradigm.

We show how our formalization of the experimental program analysis paradigm, together with our operational definition, could help researchers identify limitations of analysis techniques, improve existing program analysis techniques, and create new experimental program analysis techniques. We also discuss applications for experimental program analysis, and suggest how techniques following this paradigm can approach program analysis problems in manners that may not be possible by other types of techniques. As such, we believe that our formalization of the experimental program analysis paradigm will allow researchers to use it, to good effect, in various program analysis domains in which it has not previously been considered.

The remainder of this paper 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 an example, and also describes several potentially exploitable differences between experimental program analysis and experimentation in other fields. Section 4 illustrates the potential benefits of using experimental program analysis. Section 5 describes related work, and Section 6 concludes.

2. BACKGROUND

Empirical science is a mature field, and its methods are well-discussed in various monographs (e.g., [4, 14, 19, 20, 23, 26, 27, 28]). Because experimental program analysis techniques draw from empirical science, in this section we distill, from these monographs, an overview of the empirical method. In addition, we highlight material about experiments that is most relevant to the understanding of experimental program analysis. (Note that this overview is generalized from the sources just cited. In practice, empirical science takes on different forms in different domains. The generalization we present here, which we refer to as “traditional experimentation”, suffices for our subsequent discussion.)

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 the outcome of the questions, or limitations on those outcomes. As part of this step, the investigator also identifies the target *population* to be studied, and on which conclusions will be drawn.

Depending on the outcome of this first step, as well as the conditions under which the investigation will take place, different research 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 paper — 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 [19, 20]:

(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 isolated factors 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 typically 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 in the independent variables on the population. The observations elicited from the dependent variables are used in the experiment to conduct the tests that evaluate the predictions of the experiment.

(2) Choice of experiment design. Experiment design choice is concerned with structuring variables and data so that predictions 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 predictions about the potential effects of the treatments on the dependent variable through a *statement 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 units in the sample 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 should 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. In some experiments, 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 test hypotheses, then the process returns to experiment design 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 step 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, note that the actions taken during each step in empirical studies 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 statistical analysis used may not be of sufficient power to support valid conclusions, the nuisance variables may cause confounding effects that bias the results, and so on. These limitations are formally known as *threats*

to the experiment’s validity. The types of threats that we consider in this paper are those identified by Trochim [26]: (1) threats to internal validity (could other factors affecting the dependent variables of the experiment be responsible for the results), (2) threats to construct validity (are the dependent variables of the experiment appropriate), (3) threats to external validity (to what extent could the results of the experiment be generalized), and (4) threats to conclusion validity (what are the limitations of the experiment’s conclusions, and how could a stronger experiment be designed). The investigator 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

In this section we define and describe experimental program analysis. We then discuss several traits that distinguish this activity from traditional experimentation and “non-experimental” program analysis, as well as traits that they have in common. As a vehicle for this discussion, we illustrate the concepts that we present using an existing technique that (as we shall show) is aptly described as an “experimental program analysis” (EPA) technique — the technique implemented by HOWCOME [30].

HOWCOME 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 in order to “test” a hypothesis regarding whether the applied changes reproduce f .¹ If the applied subset “fails” this test (by not reproducing f), then a different subset is tested. If the subset “passes” the test (by reproducing f), then a subset of those incriminating variable values is considered. This process continues until no further subsets can be formed or are capable of reproducing the failure.

3.1 Definition and Illustration

We now descriptively define experimental program analysis:

Experimental program analysis is the evolving process of manipulating a program, or factors related to its execution, under controlled conditions in order to characterize or explain the effect of one or more independent variables on an aspect of the program.

There are a few aspects of this definition deserving elaboration. First, one important characteristic of experimental program analysis is the notion of “manipulation”. EPA techniques manipulate either concrete representations of a program such as source code, or factors related to its execution such as input or program states, in order to learn about the effect of those manipulations on an aspect of the program that is of interest. Viewed through the lens of traditional experimentation [19, 20], these manipulations correspond to the purposeful changes made to independent variables. 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.

¹HOWCOME uses the Delta Debugging [32] algorithm, which employs a more general form of experimental program analysis. To focus our discussion, in this paper we consider only the use of Delta Debugging in HOWCOME on variable values in program states.

Second, experimental program analysis is performed “to characterize or explain the effect of one or more independent variables on an aspect of the program”. The independent variables whose effect is being characterized or explained are the manipulations just described; yet it is important to understand the nature of this characterization or explanation. In rare cases, experiments might be able to “prove” a conjecture if an entire population is observed. In practice, however, EPA techniques, like experiments in other fields, will operate on a sample of a population, leading to answers that are not absolutely certain, but rather, highly probable. In general, the outcome of an EPA technique is a description or assessment of a population reflecting an aspect of the program of interest (e.g., what is the potential behavior of the program?), or the determination of likely relationships between the treatments and the dependent variable (e.g., what inputs are making the program fail?).

Third, experimental program analysis involves an “evolving process” of manipulating independent variables. In order for an EPA technique to build a body of knowledge regarding a program, it is often necessary to conduct multiple experiments in sequence, with the design of later experiments changed by leveraging findings gleaned from previous experiments; for example, by re-sampling the population and refining hypotheses accordingly. In terms of program analysis, these evolving experiments often have the effect of allowing the results of later experiments to efficiently 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 future experiments that will be conducted, and the conditions under which they will be conducted. Note that the selection of the treatments that are applied to the program in future experiments is tightly associated with how previous experiments have evolved during program analysis, thereby linking treatment selection to the aforementioned EPA process.

To further elucidate our descriptive definition, we augment it with an operational definition, presented in tabular form (Table 1). Because EPA techniques conduct experiments to analyze programs, this table is organized in terms of the experimentation guidelines presented in Section 2. To better characterize the sorts of experiments EPA techniques conduct, the table also summarizes the ways in which the HOWCOME technique [30] relates to the various steps.

The table also distinguishes between the automated tasks that can be performed by an EPA technique (gray rows), and the manual tasks performed by the creator of the technique (white rows). Note that the tasks corresponding to the technique can be controlled and performed repeatedly (utilizing the technique’s feedback loop) by the process that guides the EPA technique.

In the following subsections, we discuss each task in the definition in detail, in turn. We also discuss, where applicable, the ways in which tasks can contribute to validity threats in experiments conducted by EPA techniques.

3.1.1 Recognition and Statement of the Problem

This planning step consists of the formal recognition and statement of the problem, and is reflected in our operational definition by the “formulation of research questions” and “identification of population” tasks.

Formulation of research questions. This task guides and sets the scope for the experimental program analysis activity, focusing on particular aspects of a program.

Example: in HOWCOME, the research question is: “given a passing execution e_p and failure f in a failing execution e_f , what are the minimum variables $V \in e_f$ that cause f ?”

Guideline	Task Identifier	Role in Experimental Program Analysis	HOWCOME
RECOGNITION AND STATEMENT OF THE PROBLEM	<i>Research questions</i>	Questions about specific aspects of a program.	“Given e_p and f in e_f , what are the minimum variable values in e_f that cause f ?”
	<i>Population</i>	The aspect of the program that the experiment will draw conclusions about.	All program states $S_{e_p} \in e_p$.
SELECTION OF INDEPENDENT AND DEPENDENT VARIABLES	<i>Factors</i>	The internal or external aspects of the program that could impact the effect of the measured manipulations.	Variable value changes, failure-inducing circumstances, number of faults, outcome certainty, unchanging variable values, etc.
	<i>Independent variables</i>	The factors that are manipulated in order to impact the program aspect of interest.	Values of variables in e_f at each state $s \in S_{e_f}$.
	<i>Fixed variables</i>	The factors that are set or assumed to be constant.	Variable values that do not change.
	<i>Nuisance variables</i>	Uncontrolled factors that can affect the measured observations on the program.	Multiple failure-inducing circumstances, number of failures, outcome certainty, etc.
	<i>Dependent variables</i>	The constructs used to quantify the effect of treatments on the target program aspect.	Whether the execution reproduces f , succeeds as did e_p , or is an inconclusive result.
CHOICE OF EXPERIMENT DESIGN	<i>Treatments</i>	The specific instantiations of levels from the independent variables that are tested.	Difference in variable values between a state in e_p and the corresponding state in e_f .
	<i>Hypothesis statement</i>	Statements or conjectures about the effect of treatments on an aspect of the program.	A null hypothesis H_0 for each treatment is: “The value changes do not produce f in e_p .”
	<i>Sample</i>	A set of elements from the population.	Program states in e_p .
	<i>Treatment assignment</i>	An assignment of levels of independent variables to the sampled units.	A compound treatment of variable values are applied to states in e_p .
PERFORMING THE EXPERIMENT	<i>Experiment procedures</i>	An automated process using algorithms to assign treatments to units in the sample and capturing observations measuring the isolated effects of those changes.	An observation is collected for each test of variable value differences to a state.
ANALYSIS AND INTERPRETATION OF DATA	<i>Data analysis</i>	Analyzing the observations to discover or assess the effects of the treatments on the aspect of the program of interest.	The effects of applying the variable values is gauged by observing the effect on the execution’s output.
	<i>Hypothesis testing</i>	Using the analysis from the observations to confirm or reject the previously-stated hypotheses regarding treatment effects.	H_0 is rejected if a treatment reproduces f , not rejected if the execution passes, and not rejected if the outcome is inconclusive.
	<i>Interim analysis</i>	Making a decision regarding whether further tests are needed based on the results of the current and previous tests.	If H_0 was rejected and subsets of variable values can be formed, design new tests on those values. Otherwise choose different values from those remaining (if any remain).
CONCLUSIONS AND RECOMMENDATIONS	<i>Final conclusions</i>	The conclusions drawn from the application of experimental program analysis.	Using the reported cause-effect chain, or deciding to generate a new chain.

Table 1: Tasks in experimental program analysis. Each task is summarized in the third column. The tasks are grouped, in the first column, according to the experimentation guidelines in Section 2. The fourth column uses HOWCOME to illustrate each task. Tasks performed by EPA techniques are in gray rows, while tasks in white rows are performed by investigators. (Note that threats to validity are considered at all stages in experimental program analysis, and thus the consideration of such threats is not depicted in the table in any particular row.)

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 EPA techniques is some set of artifacts related to representations of the program, or factors related to the program’s execution.

Example: in HOWCOME, conclusions about variable values relevant to f are made by applying those values to program states in e_p . Each program state is a unit in the population of all program states S_{e_p} .

3.1.2 Selection of Independent/Dependent Variables

The outcomes of this step 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. This task identifies any internal or external aspect of the program and environment that could impact the effect of the manipulations that are being measured. An important byproduct of this step is the awareness of potentially confounding effects on the results.

Example: in HOWCOME, factors include: variable values (both those that are changed in a single experiment and those that are not) that can impact the final execution output; number of faults, as multiple faults may induce different failure circumstances; the failure-inducing circumstances themselves, including non-deterministic or synchronization issues on which failures may depend; and potential uncertainty about outcomes (oracle problems).

Selection of independent variables. This task identifies the factors that will be explicitly manipulated throughout the experiments performed by the EPA technique in order to impact and support the study and analysis of the program under investigation. As in traditional experiments, treatments are ultimately selected as levels from the ranges of the independent variables.

Example: in HOWCOME, the independent variable is the values of the variables in e_f at each program state. (The operative notion is that through modification of this variable, HOWCOME may find different variable values to be relevant to f). As an example of treatment design, 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 reduces the likelihood that they will affect the dependent variable in confounding ways. 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.

Example: in HOWCOME, the variable values that are not manipulated between e_p and e_f 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, which are threats to the internal validity of the technique's experiments. (As we shall see in Section 4.1, improvements to EPA techniques can come in the form of finding ways to reduce, or even eliminate, the potential impact of these nuisance variables.)

Example: in HOWCOME, 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.

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 should, or is intended to capture.

Example: in HOWCOME, the dependent variable involves whether f is reproduced, and the construct is a 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.

3.1.3 Choice of Experiment Design

The choice of experiment design is a crucial step for maximizing the control of the sources of variation in the program and the environment, and reducing the cost of an experimental program analysis technique. Tasks in the choice of experiment design are the

first of those that can be controlled by the process driving the EPA technique; this includes the opportunity to leverage the results from previous experiments to influence how the tasks are performed.

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.

Example: in HOWCOME, experiments are crafted through the selection of potential variable values — levels of the independent variable — 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.) Thus, potentially relevant variable value changes (to a program state in e_p) constitute a compound treatment in HOWCOME's experiments.

Formulate hypotheses. This task involves formalizing conjectures about the effects of treatments on the aspect of the program of interest. These hypotheses are later evaluated 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. As we discuss in Section 3.2, it can be cost-effective for EPA techniques to deal with multiple hypotheses concurrently in the same experiment.

Note that in this paper we use null hypotheses as a vehicle by which to analyze the impact of treatments. In general, however, hypotheses regarding treatments could be stated in various forms, as long as their evaluation leads to meaningful analyses regarding the effects of the treatments and the manipulations that should be performed next by the technique.

Example: in HOWCOME, when considering potential variable value changes to a program state in e_p , experiments assess whether the variable values reproduce f in the execution. 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. Sampling the population by collecting observations on a subset of the population is one way by which experimental program analysis achieves results while retaining acceptable costs. This step 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. If this task is not completed properly, the external validity of the experiment may be threatened, as conclusions may not generalize to the population.

Example: in HOWCOME, program states from e_p are sampled so that variable values from equivalent states in e_f can be applied. States at which relevant, minimal variables are found are used to form the reported cause-effect chain.

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 is a set of objects 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.

Example: in HOWCOME, the variable value differences selected as treatments are applied to a program state in e_p so that it can be observed whether the value changes reproduce f .

3.1.4 Performing the Experiment

This step is primarily mechanical and consists of just one task: obtaining a set of observations on the sampled units, according to experimental procedures, that measure the effect of the independent variable manipulations (treatments).

Execute experimental procedures. In experimental program analysis, this procedure 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 possible human factors that might confound results — and where an algorithmic representation is sometimes not as natural for representing the experimentation processes. The observations obtained during this task are those that relate to the dependent variable, and should capture only the isolated effects on that variable 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.

Example: in HOWCOME, 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.

3.1.5 Analysis and Interpretation of Data

EPA techniques must analyze the collected observations to evaluate hypotheses and determine the next course of action. 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. If they are not performed properly, the conclusion validity of the experiment can be threatened due to the use of analyses of insufficient power.

Performing data analysis. This task involves the analysis of collected observations for the purpose of evaluating hypotheses. In many cases, this includes statistical analyses to determine the appropriateness of the decision made regarding an hypothesis.

Example: in HOWCOME, the only information needed to evaluate hypotheses is whether or not the dependent variable indicated that the variable value treatment caused f to be reproduced, or whether it caused an inconclusive result.

Testing of hypotheses. Hypothesis testing is used to assess the effect of the manipulations made by the investigator (or, in the case of experimental program analysis, by the automated process that manages the experiment).

Example: in HOWCOME, 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. This tells the technique that it should concentrate on trying to find the minimally relevant variable value differences within this treatment. As such, the rejection of H_0 guides the manipulations in 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.

Example: in HOWCOME, 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 .

3.1.6 Conclusions and Recommendations

Investigators must draw conclusions based on experimental program analysis results and, if appropriate, recommend future courses of action, which can include the use of the EPA technique’s output or a repeated or refined run of the technique. These conclusions should be interpreted in light of the various validity threats that may have been introduced by the previous tasks during the experiments conducted by the EPA technique.

Draw final conclusions. These are the final conclusions drawn from experimental program analysis when the analysis is complete and no further experiments are needed or can be performed.

Example: in HOWCOME, 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.

3.2 Experimental Program Analysis Traits

Experimental program analysis has distinguishing traits that have various implications for program analysis; we now comment on several of these.

Replicability and sources of variation. Program analysis activities are not subject to certain 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 re-

sembles the concept of “testing effects” seen in some other fields. Experimental program analysis is susceptible to 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. Controlling for such behavior (e.g., controlling the scheduler) may threaten the generality of an EPA technique’s findings. Still, experimental program analysis has the advantage of dealing with software systems, which are abstractions and are more easily controlled than naturally occurring phenomena.

The cost of applying treatments. In most cases, the application 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. Two factors contribute to this trait. First, in experimental program analysis, application of treatments to experimental units is often automated and requires limited human intervention. Second, unlike in other fields there are no truly expendable units or treatments (e.g., seeds and soil in agricultural experiments, mice when experimenting with genetic disorders, energy when exploring atomic particles); 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). EPA techniques can take advantage of this trait by using increased sample sizes to increase the confidence in, or quality of, the findings, and adding additional treatments to their design in order to learn more about the research questions.

Sampling the input space. Experiments often sample from a population in order to draw inferences from the sample that reflect the population. 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.

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 statistical 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.

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, HOWCOME utilizes an algorithm to refine the stated hypothesis within the same experiment and guide successive treatment applications of variable values to a program state. We strongly suspect that other tasks in experimental program analysis, such as earlier tasks (e.g., choice of variables), can be represented in algorithmic form. This would allow such EPA techniques to be highly adaptable to particular instances of program analysis problems — without the intervention of investigators.

The role of feedback. Traditional experimentation guidelines advocate the separation of hypothesis setting, data collection, and data analysis activities. 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 [23]. For example, in clinical trials the 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 will enable 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 experiment designs in EPA techniques. Because they build a body of knowledge about programs during their analyses, EPA techniques can adjust their experiment parameters in order to improve the efficiency of their analyses, such as through the use of adaptive sampling procedures [24] to concentrate on areas of the program that may actually contain faults, based on the program information gathered thus far, during a debugging activity.

4. APPLICATIONS OF THE EXPERIMENTAL PROGRAM ANALYSIS PARADIGM

We now consider in detail the applications of the experimental program analysis paradigm described in Section 1.

4.1 Assessing and Improving EPA Techniques

Our operational definition of experimental program analysis provides a means for assessing the limitations of existing EPA techniques by specifically outlining each high-level task in techniques’ experiments so that incorrectly approached tasks, or tasks staged to rely on assumptions that may not be met, can be more easily identified. For example, experimental design errors, sampling bias, confusion of correlation and causality, and interactions between effects can all intrude on EPA techniques and bias or limit their results. (This is especially likely if the researchers creating techniques are unaware of the possibility of such limitations.) In doing so, our operational definition helps investigators explicitly consider validity threats that are inherent in controlled experimentation. Although considering the limitations of analysis techniques is not new to the program analysis community, explicitly considering the validity threats that typically accompany the design or assessment of experiments provides a new lens through which to view such limitations in EPA techniques.

Assessing techniques’ limitations is important, of course, so that conclusions can be appropriately interpreted; however, after assessing limitations, improvements that address such limitations can nat-

urally suggest themselves. Further, because our operational definition is closely tied to an outline of experiments that is generally accepted in the experiment design community, we expect that the definition will help researchers expose many opportunities to utilize established experiment design strategies to improve EPA techniques. As one example, there are many strategies for assigning treatments to the units of a sample (e.g., block designs, factorial designs, split-plot designs), and these designs could provide opportunities to increase the power of experiments and lower experiment costs (e.g., a block design can isolate nuisance variables into blocks, removing them as error effects, and a split-plot design can reduce the need for applying combined treatments). Through the experimental program analysis paradigm we explicitly expose the program analysis community to such opportunities.

Detailed example of assessing and improving techniques. In Section 3.1, we outlined some of the validity threats to the HOWCOME technique. For example, HOWCOME will suggest the minimally relevant variable values between the provided passing and failing executions. However, these may not be the *truly minimal* variables and values causing the observed failure: rather, they may just be the minimal variable values given the subsets isolated by the divide-and-conquer algorithm for the given executions.² The possibility of dependencies between variables influencing those suggested as minimally relevant in the cause-effect chain is another threat not explicitly outlined in [30]. Although such dependencies may be part of the causality chain between values that ultimately results in the failure, understanding the impact of these nuisance variables could help in isolating the fault, or in doing so more quickly. For example, HOWCOME already keeps the values of variables that are not modified constant to reduce their impact on results. One possible improvement to the technique could involve adding an option to manipulate only variables with no dependencies, to prevent dependencies between variable values from influencing results.

Another approach to improving HOWCOME involves its scalability limitations, not all of which have been specifically addressed in [30]. Although systematic, HOWCOME’s sampling procedures and strategies for selecting new variable values to test may become impractical (slow to converge) in the presence of programs with many variables and large traces. One example improvement to the technique would be to utilize random sampling over the program state space; the size of the sample could be based on the availability of resources (e.g., time). Although the use of random sampling could result in missing information in the cause-effect chain, it would provide an opportunity to employ statistical analysis to generate confidence as to the accuracy (and perhaps completeness) of the cause-effect chain thus created.

Because an implementation of HOWCOME was not available to us, we do not explore such improvements further in this paper; instead we focus on the creation of a new technique.

4.2 Creating New EPA Techniques

By formally exposing the use of experimentation in program analysis to the research community, we expect to enable researchers to better utilize EPA techniques to address program analysis problems. Our operational definition of experimental program analysis will assist researchers in this process by providing a “recipe” for creating new EPA techniques.

²In earlier work pertaining to input minimization [32], this threat is discussed in terms of “global minimums” versus “local minimums”. Local minimums are detected by HOWCOME, whereas a “global minimum” could be detected by exhaustively testing all combination of variable-value changes to program states.

Detailed example of creating new techniques. Refactorings [12] are semantic-preserving transformations to source code that aim to improve the targeted code’s maintainability. There has been an increasing amount of research regarding the use of refactoring support to improve software systems (e.g., [7, 17, 18]), and this support has been realized in both IDEs and stand-alone tools. To date, however, it remains unclear precisely when and where potential refactorings should be applied to the source code of any given system.

Although it may be possible to refactor particular segments of code, doing so may not always be advantageous. Further, many types of refactorings are possible (Fowler [12] introduces approximately 70), and refactored code can interact with other code in unexpected ways. In some cases, the effects of multiple beneficial refactoring instances may be compounded into a greater improvement than would be achieved by considering each instance individually. On the other hand, a refactoring instance may worsen the code when combined with certain other instances with which the original refactoring does not interact well. Thus, the benefit, or lack thereof, in using refactoring support is dependent on the manner in which potential refactorings of code are applied to a system.

One way to address this issue would be to combinatorially apply all possible combinations of code refactorings and keep the best combination; however, this is not feasible for larger software systems. To provide refactoring support at reduced cost, groups of potential code refactorings can be incrementally and *experimentally* evaluated according to some relevant standard in order to determine whether they should be kept or discarded.

As we expect other researchers to do, we used our operational definition to help build an experimental methodology for this refactoring approach, which we term RF_{epa} . We summarize how RF_{epa} fits into that operational definition. The independent variable being manipulated by RF_{epa} is the range of individual code refactorings that could be applied to a system. In RF_{epa} , compound treatments of one or more treatment code refactorings R are applied to applicable samples of source code S ; a null hypothesis H_0 can then be stated as, “The application of R to S does not worsen the system.” If R makes the system worse according to one or more dependent variables, then H_0 can be rejected, and the refactorings in R discarded from future consideration. Otherwise, R can be combined with other compound treatments of refactorings until no refactorings remain to be evaluated.

To select specific treatment refactorings, we first identify all possible refactorings and evaluate them individually. Refactorings with a negative, detrimental effect on the system are immediately discarded from consideration in order to save comparisons. We then incrementally build up combinations of code refactorings (as compound treatments) for assessment. First, we form the refactorings that were not discarded into groups of two and evaluate their effect on the system. If they produce a negative effect then both are discarded. We then combine groups of two beneficial code refactorings into groups of four refactorings. These groups of four are evaluated, and those groups that are beneficial are combined with others in the same manner until no further groups of beneficial refactorings remain to be evaluated.

To investigate the potential of this RF_{epa} approach, we conducted a small study using the `Siena` software system as a subject. `Siena` is written in Java, has 26 classes, and 6,035 lines of source code. (`Siena` is publicly available as a part of an infrastructure supporting experimentation [6].) In this study, we used RF_{epa} to attempt to improve 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 [7]. In object-oriented programming, cohesion among methods is impor-

tant because methods that are not similar may not belong in the same class. We selected cohesion because it is considered a good indicator of the maintainability of software systems, and has been the subject of previous refactoring work [7]. As a dependent variable to measure cohesion, we selected the “Lack of Cohesion of Methods” metric (LCOM) as defined by Henderson-Sellers [13]. In this paper, we consider the five *Siena* classes with the worst initial LCOM measurement. (Note that since LCOM measures a lack of cohesion, an LCOM of 0.0 is optimal.)

Given our decision to use refactoring support to improve cohesion, we decided to explore the use of the “Extract Method” and “Extract Class” refactoring operations. Extract Method identifies a set of statements that appear at more than one location in a class and creates a new method out of those statements. This new method is then invoked at each point where that set of statements previously appeared. Extract Class creates a new class out of selected methods and attributes residing in a separate, original class. We selected these two refactorings because they were identified in previous work [7] as having the potential to improve cohesion. Extract Method improves cohesion by making similarities among methods more explicit, while Extract Class creates a new class out of similar, connected methods and attributes [7].

The goal of this study was to illustrate the difficulty of refactoring in general, and to investigate whether RF_{epa} has the potential to effectively address this problem. Because we know of no well-accepted standards defining when particular code refactorings should (and should not) be applied, we considered the cohesion of the methods in the five *Siena* classes after applying both *all* possible Extract Method and Extract Class refactorings to the five classes, and after applying only those refactoring combinations indicated by RF_{epa} using LCOM as a dependent variable. As such, the application of *all* possible refactorings serves as a point of comparison against RF_{epa} .

Given these two refactoring approaches, we measured cohesion within the five classes, according to the LCOM metric, by considering three constructs: (1) the initial LCOM, denoted by `LCOM-Init`, measured without applying any refactorings; (2) the LCOM after applying all possible refactorings, which we denote by `LCOM-All`; and (3) the LCOM after RF_{epa} , which we denote by `LCOM-EPA`. The purpose of considering `LCOM-Init` is to provide a clear baseline with which to assess the improvement, or lack thereof, of each approach.

The LCOM of the *Siena* classes according to each construct is provided in Table 2. This table indicates that the code refactorings applied by RF_{epa} resulted in a cohesion that was either as good as or better than the cohesion after applying all possible refactorings. This result is encouraging because it indicates that an experimental approach for refactoring support such as RF_{epa} can provide guidance for deciding when and where refactorings should be applied. Such guidance is important because simply applying all possible code refactorings may actually worsen the system; Table 2 indicates that applying all possible code refactorings worsened the cohesion of the methods in the *SENP* and *HierarchicalDispatcher* (abbreviated *HDispatcher* in Table 2) classes, and impaired the improvement in cohesion in the *Monitor* class. In the *TCP-Handler* class (*TCPPacket* in Table 2), no Extract Method or Extract Class refactorings were possible.

We also noticed interesting interactions between combinations of individual code refactorings when we considered the use of both Extract Method and Extract Class refactorings. One interesting observation was that, at times, the benefits of different treatment refactorings stacked. For example, in the *Tokenizer* class, one possible Extract Method refactoring applied by itself would reduce the

Class	LCOM-Init	LCOM-All	LCOM-EPA
Monitor	1.179	1.125	1.095
TCPPacket	1.0	1.0 *	1.0 *
SENP	0.997	1.005	0.996
HDispatcher	0.899	0.92	0.899
Tokenizer	0.85	0.667	0.667

Table 2: LCOM of classes using Extract Method and Extract Class refactorings. The * symbol indicates that no refactorings of either type were identified for this class.

LCOM of *Tokenizer* from 0.85 to 0.848, while a second possible Extract Class refactoring applied by itself would reduce LCOM from 0.85 to 0.7. When these two refactorings are applied together, however, the LCOM of *Tokenizer* is reduced to 0.667. Thus, not only was applying these two refactorings together most beneficial, but the *interaction* between the refactorings appears to have caused an additional degree of benefit.

At other times, interactions between refactorings produced less desirable results. Among all classes, there were four instances of individual Extract Method refactorings that had no effect on the LCOM results summarized in Table 2. However, when combined with potential Extract Class refactorings for their respective classes, three of these four Extract Method refactorings made LCOM worse. In one other case, an interaction with Extract Method refactorings caused an Extract Class refactoring for *SENP* that was formerly beneficial to the class to instead make the LCOM worse.

These types of examples serve to illustrate two points. First, interactions between particular code refactorings make this problem complex, which may indicate why, to date, there exists no accepted methodology outlining the circumstances in which potential code refactorings should be applied to source code. We believe that an experimental program analysis approach may provide a means for deciding when and where refactorings should be applied to systems. The intuition behind this proposed approach ensures that refactorings will not be applied if they will not improve the system, according to specified criteria.

Second, the preliminary evidence gathered in this study indicates that an experimental program analysis approach can find and apply particular code refactorings that will improve the system. We hesitate to generalize regarding the effectiveness of the approach in performing refactoring due to the limited scope of this study. However, the goal of this study was to illustrate the difficulty of realizing automated refactoring support, and to investigate whether an experimental program analysis approach such as RF_{epa} has the potential to effectively address this problem. We believe that this study has succeeded on both fronts, and has indicated that such an approach warrants further investigation in future work.

In addition to more extensive study, we plan to harness some of the well-known methods in the experiment design literature to improve the proposed RF_{epa} refactoring approach. For example, by operating on the intuition that some classes may have more opportunities for applying code refactorings than others, the use of adaptive sampling would allow the approach to concentrate its refactoring effort on areas of the system where code refactorings have already proved successful, thereby increasing the efficiency of the technique. As another example, because there are many possible measures regarding the maintainability of source code that may be of interest to refactoring techniques, it may be desirable to consider many dependent variables, operating at many levels of granularity (e.g., method-level, class-level, package-level), which may require multivariate analyses.

In summary, as discussed in Section 3.2, we believe that the use of traditional experimentation strategies will provide valuable improvements to many EPA techniques, including the refactoring approach described here.

5. RELATED WORK

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. [28] introduce an experimental process tailored to the software engineering domain, Fenton and Pfleeger [11] 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. [15] provide guidelines for conducting empirical studies in software engineering.

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, 9, 16], 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., [5, 8]).

Within the program analysis domain, to the best of our knowledge, Zeller is the first to have used the term “experimental” in application to program analysis techniques [31]. Our work differs from Zeller’s, however, in several important 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 [31, page 1]. Thus, in discussing specific analysis approaches, Zeller provides only informal definitions. In this work, we accept Zeller’s challenge and apply our understanding of and experience with controlled experimentation to 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 that: “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 [31, 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 [14, 19, 20]. 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

quintessentially experimentation-related notions are present in our definition, and the utility of including them is supported.

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 interest in many research papers (see, e.g., [2, 10, 25, 29, 31]). Our goal in this paper is not to taxonomize; nevertheless, we would argue that experimental program analysis is not constrained to the traditional static or dynamic classification, but rather, is orthogonal to it. The experimental program analysis paradigm focuses on the *type* of analysis performed: namely, whether tests and purposeful changes are used to analyze software systems. As such, it can overlap with *both* static and dynamic analysis techniques.

Experimental program analysis therefore fills a gap that is not addressed by static or dynamic analysis techniques by offering (1) procedures for systematically controlling sources of variation, (2) experiment designs and sampling techniques to reduce the costs of experimentation, and (3) 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.

6. CONCLUSIONS AND FUTURE WORK

This paper has presented experimental program analysis as a new 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 EPA techniques, and create new EPA 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 paper we have considered only a few examples of how to adapt 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 the automation opportunities for EPA techniques. Thus far, we have focused on the automation of experimental program analysis tasks and the advantages therein, but little else. However, it seems likely 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 3.2 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 it offers distinct advantages over other forms of analysis — at least for particular classes of analysis tasks — including procedures for systematically controlling sources of variation in order to experimentally analyze software systems, and experimental designs and sampling techniques that reduce the cost of generalizing targeted aspects of a program. 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.

7. 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. We also thank the anonymous reviewers of earlier versions of this paper for comments that substantially improved the content of the work. This work has been supported by NSF under awards CNS-0454203, CCF-0440452, and CCR-0347518 to University of Nebraska–Lincoln.

8. REFERENCES

- [1] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proc. 2001 ACM SIGPLAN Conf. Prog. Lang. Des. Impl.*, pages 168–179, June 2001.
- [2] T. Ball. The concept of dynamic analysis. In *Proc. Seventh Euro. Softw. Eng. Conf. held jointly with the Seventh ACM SIGSOFT Int'l. Symp. Found. Softw. Eng.*, pages 216–234, Sept. 1999.
- [3] V. R. Basili, F. Shull, and F. Lanubile. Using experiments to build a body of knowledge. In *NASA Softw. Eng. Workshop*, pages 265–282, Dec. 1999.
- [4] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*. Wiley, New York, first edition, 1978.
- [5] 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 Trans. Softw. Eng.*, 23(7):437–444, July 1997.
- [6] H. Do, S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proc. 2004 Int'l. Symp. Empirical Softw. Eng.*, pages 60–70, Aug. 2004.
- [7] B. Du Bois, S. Demeyer, and J. Verelst. Refactoring — improving coupling and cohesion of existing code. In *Proc. 11th IEEE Working Conf. Reverse Eng.*, pages 144–151, Nov. 2004.
- [8] 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 *Proc. 19th Int'l. Conf. Softw. Eng.*, pages 205–215, May 1997.
- [9] S. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Trans. Softw. Eng.*, 31(4):312–327, Apr. 2005.
- [10] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *Proc. Workshop Dyn. Anal.*, pages 24–27, May 2003.
- [11] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Course Technology, second edition, 1998.
- [12] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [13] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.
- [14] R. E. Kirk. *Experimental Design: Procedures for the Behavioral Sciences*. Brooks/Cole, third edition, 1995.
- [15] 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 Trans. Softw. Eng.*, 28(8):721–734, Aug. 2002.
- [16] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proc. 2005 ACM SIGPLAN Conf. Prog. Lang. Des. Impl.*, pages 15–26, June 2005.
- [17] M. Mantyla, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proc. IEEE Int'l. Conf. Softw. Maint.*, pages 381–384, Sept. 2003.
- [18] T. Mens, T. Tourwe, and F. Munoz. Beyond the refactoring browser: Advanced tool support for software refactoring. In *Proc. Int'l. Workshop Principles Softw. Evolution*, pages 39–44, Sept. 2003.
- [19] D. C. Montgomery. *Design and Analysis of Experiments*. Wiley, New York, fourth edition, 1997.
- [20] C. Robson. *Real World Research*. Blackwell Publishers, Inc., Malden, Massachusetts, U.S.A., second edition, 2002.
- [21] J. R. Ruthruff. Experimental program analysis: A new paradigm for program analysis. In *Proc. 28th Int'l. Conf. Softw. Eng.: Doctoral Symp.*, May 2006 (to appear).
- [22] J. R. Ruthruff, M. Burnett, and G. Rothermel. An empirical study of fault localization for end-user programmers. In *Proc. 27th Int'l. Conf. Softw. Eng.*, pages 352–361, May 2005.
- [23] D. Siegmund. *Sequential Analysis: Tests and Confidence Intervals*. Springer-Verlag, New York, 1985.
- [24] S. K. Thompson. *Sampling*. Wiley, New York, second edition, 2002.
- [25] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3):121–189, 1995.
- [26] W. M. K. Trochim. *The Research Methods Knowledge Base*. Atomic Dog, Cincinnati, Ohio, U.S.A., second edition, 2001.
- [27] A. Wald. *Sequential Analysis*. Wiley, New York, 1947.
- [28] C. Wohlin, P. Runeson, M. Host, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering*. Kluwer Academic Publishers, Boston, 2000.
- [29] M. Young and R. N. Taylor. Rethinking the taxonomy of fault detection techniques. In *Proc. 11th Int'l. Conf. Softw. Eng.*, pages 53–62, May 1989.
- [30] A. Zeller. Isolating cause-effect chains from computer programs. In *Proc. 10th ACM SIGSOFT Symp. Found. Softw. Eng.*, pages 1–10, Nov. 2002.
- [31] A. Zeller. Program analysis: A hierarchy. In *Proc. Workshop Dyn. Anal.*, pages 6–9, May 2003.
- [32] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.