**PlanAlyzer: Assessing Threats to the Validity of Online Experiments**

EMMA TOSCH, CICS, University of Massachusetts Amherst, USA  
EYTAN BAKSHY, Facebook, Inc., USA  
EMERY D. BERGER, DAVID D. JENSEN, and J. ELIOT B. MOSS, CICS, University of Massachusetts Amherst, USA

Online experiments have become a ubiquitous aspect of design and engineering processes within Internet firms. As the scale of experiments has grown, so has the complexity of their design and implementation. In response, firms have developed software frameworks for designing and deploying online experiments. Ensuring that experiments in these frameworks are correctly designed and that their results are trustworthy—referred to as *internal validity*—can be difficult. Currently, verifying internal validity requires manual inspection by someone with substantial expertise in experimental design.

We present the first approach for statically checking the internal validity of online experiments. Our checks are based on well-known problems that arise in experimental design and causal inference. Our analyses target PlanOut, a widely deployed, open-source experimentation framework that uses a domain-specific language to specify and run complex experiments. We have built a tool called PlanAlyzer that checks PlanOut programs for a variety of threats to internal validity, including failures of randomization, treatment assignment, and causal sufficiency. PlanAlyzer uses its analyses to automatically generate *contrasts*, a key type of information required to perform valid statistical analyses over the results of these experiments. We demonstrate PlanAlyzer’s utility on a corpus of PlanOut scripts deployed in production at Facebook, and we evaluate its ability to identify threats to validity on a mutated subset of this corpus. PlanAlyzer has both precision and recall of 92% on the mutated corpus, and 82% of the contrasts it generates match hand-specified data.

CCS Concepts: • Software and its engineering → Scripting languages; Domain specific languages;  
Software testing and debugging: Specialized application languages; Empirical software validation.

Additional Key Words and Phrases: Experimental Design, Threats to Validity, Online Experiments

ACM Reference Format:
Emma Tosch, Eytan Bakshy, Emery D. Berger, David D. Jensen, and J. Eliot B. Moss. 2019. PlanAlyzer: Assessing Threats to the Validity of Online Experiments. Proc. ACM Program. Lang. 3, OOPSLA, Article 182 (October 2019), 30 pages. https://doi.org/10.1145/3360608

1 INTRODUCTION

Many organizations conduct online experiments to assist decision-making, and many of the largest organizations now employ some form of federated experimentation management [Bakshy et al. 2014; Crook et al. 2009; Kohavi et al. 2009; Tang et al. 2015, 2010]. These systems often include software

Authors' addresses: Emma Tosch, etosch@cs.umass.edu, CICS, University of Massachusetts Amherst, 140 Governors Drive, Amherst, MA, 01002, USA; Eytan Bakshy, ebakshy@fb.com, Facebook, Inc. 1 Hacker Way, Menlo Park, CA, 94025, USA; Emery D. Berger, emery@cs.umass.edu; David D. Jensen, jensen@cs.umass.edu; J. Eliot B. Moss, moss@cs.umass.edu, CICS, University of Massachusetts Amherst, USA.

Permission to make digital or hard copies of part or all 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. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2019 Copyright held by the owner/author(s).
2475-1421/2019/10-ART182
https://doi.org/10.1145/3360608

Proc. ACM Program. Lang., Vol. 3, No. OOPSLA, Article 182. Publication date: October 2019.
components that make designing experiments easier or that automatically monitor experimental results. One popular example component of such experimentation systems is Facebook’s PLANOUT: a domain-specific language for experimental design.\(^1\)

The state of the art for validating experimental designs (i.e., the procedure for conducting an experiment) is manual human review. The most common experimental design on the Web is the A/B test, where users see one of two variants (A and B). The state of the art for analyzing the results of experimental designs depends on the design: for A/B tests, the outcomes of interest (e.g., click rates) can be computed and compared automatically, but more sophisticated designs require specialized analysis. Many experiments written in a DSL such as PLANOUT can be cumbersome to validate, and they cannot be analyzed using existing automated methods.

**Validation of Experimental Designs.** Experiments expressed as programs can have errors that are unique to the intersection of experimentation and software. Shadish et al.\(^2\) enumerate a taxonomy of nine well-understood design errors in the experimental design literature, referred to as threats to internal validity—i.e., the degree to which valid causal conclusions can be drawn within the context of the study. Seven of these errors can be avoided when the researcher employs a randomized experiment that behaves as expected.\(^2\) Therefore, ensuring randomization is critical to internal validity. However, randomization failures in programs manifest differently from randomization failures in the physical world: for example, a program cannot disobey an experimental protocol, but data flow can break randomization if a probability is erroneously set to zero.

**Validation of Statistical Conclusions.** Control-flow operators, calls to external services, and in-language mechanisms for data recording prohibit simple automatic variable monitoring. For example, an experiment that sets variables differently on the basis of the current country of the user cannot naively aggregate results across all participants in the experiment. Such an experiment would require additional adjustment during post-experiment analysis to account for the potential confounding introduced by a user’s current country.

**Contributions.** The work presented in this paper: (1) statically identifies sources of statistical bias in programmatically defined experiments, and (2) presents methods for automatically generating the most common statistical analysis for the largest class of such experiments. We make the following contributions:

**C1 Software for the static analysis of experiments.** We introduce PLANALYZER, the first tool, to our knowledge, for analyzing online experiments statically (§4).\(^3\) PLANALYZER produces three key pieces of information: (1) a list of the variables in the environment that are actually being randomly assigned; (2) the variables that are recorded for analysis; and (3) the variables that may be legitimately compared when computing causal effects. These three pieces of information are required in order to determine whether there are any valid statistical analyses of the recorded results of an experiment, and, when possible, what those analyses are.

**C2 Characterizing errors and bad practices (code smells) unique to programmatically defined experiments.** Traditional errors in offline experimentation can take on unusual forms in programmatically defined experiments. Additionally, some coding practices can lead to faults

\(^1\)[http://facebook.github.io/planout](http://facebook.github.io/planout)

\(^2\)The two remaining threats to validity that are *not* obviated by randomization are *attrition*, described in §3.2, and *testing*. Testing in experimental design refers to taking an initial measurement and then using the test instrument to conduct an experiment. Analysis may not be able to differentiate between the effect that a test was designed to measure and the effect of subjects learning the test itself. Testing is a form of *within-subjects* analysis that is not typically employed in online field experiments and whose analyses are outside the scope of this work.

\(^3\)[http://www.github.com/KDL-UMass/PlanAlyzer](http://www.github.com/KDL-UMass/PlanAlyzer)
during downstream statistical analysis, highlighting the potential utility of defining code smells for experiments [Fowler 2018]. In §5, we introduce errors and code smells that arise from the intersection of experiments and software.

C3 Empirical analysis of real experiments. We report PlanAlyzer’s performance on a corpus of real-world PlanOut scripts provided by Facebook (§6–7). Due to the vetting process at Facebook, few errors exist naturally in the corpus. Therefore, we perform mutation analysis to approximate a real-world distribution of errors. Facebook also provided a corpus of human-generated contrasts (the set of variable values that are allowed to be compared, necessary for estimating causal effect).¹ We demonstrate PlanAlyzer’s effectiveness in finding major threats to validity and in automatically generating contrasts.

Organization. We begin with a motivating example experiment in Section 2. Section 3 gives background in experimental design and causal inference (§3.1), outlines common threats to validity in causal inference (§3.2), and presents the PlanOut framework (§3.3). Section 4–Section 6 describe our contributions and Section 7 presents our evaluation. We discuss related work in Section 8 and summarize our findings in Section 9. We also include a glossary of relevant terminology in Appendix A and a discussion of features that make analyses in PlanOut nontrivial in Appendix B.

2 MOTIVATING EXAMPLE

Consider an engineering team aiming improve the quality of video streaming for users of its mobile streaming service and thus increase its usage [Krishnan and Sitaraman 2013]. Users access the service over heterogeneous networks, and one way to improve quality is to change the video bit rate; the firm cannot control network bandwidth, but they can control how much data to transmit per second. Streaming video at higher bit rates will result in higher quality video, at the expense of increased data use. The team knows it wants to incorporate location information into how they determine bandwidth, but they disagree about what location information to use, and how to implement their design. They discuss two alternatives: (1) assign users to bit rates based on country, which has previously proven to be effective; or (2) assign users to bit rates based on real-time estimates of network latency.

Country-Level Design. The team considers two different bit rates: 400 kbit/s (low) and 750 kbit/s (high). Streaming videos at high bit rates may result in poor performance for individuals in emerging markets, so the team chooses to allocate fewer users to the high bit rate within these markets. Furthermore, the team chooses to constrain the population to markets the team understands well, and for which they have a large quantity of data (e.g., India, Brazil, the US, and Canada).

Dynamic Design. Users may change networks throughout the day; each of these networks may have different levels of latency. Therefore, the team considers a design that uses a personalized dynamic treatment regime, which maps users to different bit rates depending on network connection type [Murphy 2003]..

The country-level design is similar to a classic A/B test over the possible max bit rates, per country: it is straightforward and should return results quickly. However, it is much more constrained than the dynamic design, using country as a coarse-grained proxy for bandwidth availability. The dynamic design takes into account contextual information that may change over the course of the experiment, causing the devices to receive different treatments at different times. Given the tradeoffs between these two experiments, the team decides to randomly assign participants to either one experiment or the other.

¹Appendix A provides a glossary for the terminology of experimental design.
1 dynamic_policy = bernoulliTrial(p=0.3, unit=userid);
2 context = getContext(deviceid=deviceid, userid=userid);
3 country = getUserCountry(userid=userid);
4 emerging_market = (country == 'IN') || (country == 'BR');
5 established_market = (country == 'US') || (country == 'CA');
6 if(dynamic_policy) {
7 weights = getBanditWeights(context=context);
8 choices = getBanditChoices(context=context);
9 max_bitrate = weightedChoice(choices=choices, weights=weights, unit=userid);
10 } else {
11 if (emerging_market) {
12 max_bitrate = weightedChoice(choices=[400, 750], weights=[0.9, 0.1], unit=userid);
13 } else if (established_market) {
14 max_bitrate = weightedChoice(choices=[400, 750], weights=[0.5, 0.5], unit=userid);
15 } else {
16 max_bitrate = 400;
17 }
18 }

Fig. 1. The example experiment of §2, written in the PlanOut language. This experiment aims to improve video streaming experiences for users with potentially heterogeneous network conditions by testing different bit rates for video transmission in a mobile application software, according to a randomly chosen policy. One policy is dynamic; treatments depend on user features, potentially varying over the course of the experiment. As discussed in §3.3, PlanOut programs are executed within a containing environment. getContext, getUserCountry, getBanditWeights, and getBanditChoices are all functions that exist within that environment.

Conditioning set: {}

Avg(Y|dynamic_policy=true) - Avg(Y|dynamic_policy=false)

Conditioning set:
{emerging_market : true}

Avg(Y|max_bitrate=400, dynamic_policy=false) - Avg(Y|max_bitrate=750, dynamic_policy=false)

Conditioning set:
{emerging_market : false; established_market : true}

Avg(Y|max_bitrate=400, dynamic_policy=false) - Avg(Y|max_bitrate=750, dynamic_policy=false)

Fig. 2. Valid contrasts and their associated conditioning sets (i.e., constraints on the analyses), presented using one output option for Planalyzer (C1), for the program in Figure 1. The outcome of interest is the average percentage of videos watched. As we discuss in §3, we generally abstract over the outcome and simply refer to it as Y. Planalyzer finds three conditioning sets, each having one set of valid contrasts. A naive approach to comparing bit rates might average over all data points. If this were valid, Planalyzer would have produced Avg(Y|max_bitrate=400)-Avg(Y|max_bitrate=750) for the empty conditioning set.

The script in Figure 1 depicts one way of representing these experiments in PlanOut. There are four paths through the program, and three of them randomly assign values to max_bitrate directly. The fourth path can only be said to randomly assign max_bitrate indirectly, via random branching.

Figure 2 depicts Planalyzer’s output in a human-readable format. This output specifies the valid contrasts, or the variables that may be legitimately compared. While there is a relationship between paths through a program and contrasts, the mapping is imperfect: only the latter two of the three contrasts correspond to a path.
Each contrast has two components: a possibly empty conditioning set, which corresponds to a kind of constraint on analysis, and a list of valid pairwise comparisons: only cases for which the constraint holds should be used when analyzing the associated contrast(s).\(^5\) The first contrast corresponds to comparing between the two approaches. This type of comparison may not actually be of interest to the team, but it is valid. The second and third contrasts must be analyzed separately: whether a user is in an emerging market or an established market determines their probability of being assigned the high or low bit rate, but the market may also have an influence on the average percentage of videos watched (\(Y\)). Therefore, naively aggregating over the high and low bit rates for the country-based experiment would not be correct.

Now consider an alternative version of this program. Assume that we have the functions `inEmergingMarket` and `inEstablishedMarket` available and that these functions take no arguments, but use the current location of the device to determine whether the user is currently in the appropriate market. Then we might replace the call to `getUserCountry` and lines 11-17 with:

```java
if (inEmergingMarket()) {
    max_bitrate = weightedChoice(choices=[400, 750], weights=[0.9, 0.1], unit=userid);
} else if (inEstablishedMarket()) {
    max_bitrate = weightedChoice(choices=[400, 750], weights=[0.5, 0.5], unit=userid);
} else {
    max_bitrate = 400;
}
```

The results of the calls to `inEmergingMarket` and `inEstablishedMarket` would then be stored in intermediate values and therefore would not be recorded. The logic of experimentation remains the same, but now we have no way to recover which market users were in from the recorded data. In this case, the only valid contrast would be the first one listed in Figure 2. If this were a standalone experiment, random assignment would be completely broken.

### 3 BACKGROUND

Experiments are procedures designed to measure the effects of interventions. Two common elements of experimentation are randomization, in which experimental units (e.g., users or devices) are randomly assigned to treatments, and contrasts, in which some set of outcomes are compared between units in each treatment group. One part of the example experiment shown in Figure 1 randomly assigns users to different bit rates (treatments). The other part of the experiment assigns users to either a dynamic or a country-based policy. The outcome might be the amount of time spent watching videos.

#### 3.1 Experimental Design and Causal Inference

Experimental design refers to the different methods of deliberately assigning treatments to units, in service of estimating the effect of those treatments on an outcome of interest that is typically denoted as \(Y\). In the example of Fig 1, \(Y\) corresponds to the average amount of time people spend watching videos. One set of treatments \((T)\) is `dynamic_policy=false`, which we will encode as 0, and `dynamic_policy=true`, which we will encode as 1. Then \(Y^T=1\) corresponds to the outcome for users receiving dynamic treatment assignment.

The scope of this work applies to field experiments, named for their origins in agriculture, where weather and ecosystems provide myriad additional variables that can influence experimental outcomes [Fisher 1936]. Ascertaining effects from a blend of human behavior and computerized

---

\(^5\)Note that our default output format is a collection of comma-separated value files (CSVs) that partition the contrasts by those that may be compared. We expect end-users to load these CSVs into a database. Producing the full list of pairwise comparisons when there are a large number of variables to compare is inadvisable. We produce the format presented here because it is the standard way of expressing contrasts in the experimental design community.
systems requires researchers to treat these experiments as field experiments, rather than laboratory experiments, which are more common in computer science research [Blackburn et al. 2008]. Section 3.3 describes some of the technical difficulties associated with running experiments on complex, heterogeneous systems.

The collection of experimental units is the sample (e.g., users, cookies, or devices). For a sufficiently large sample, random assignment ensures that the effects of any other variables on outcome are equally distributed across treatment groups, allowing accurate estimation of causal effects.

In addition to recording outcome data, an online experimentation system must also record the treatment actually assigned, rather than just the intended treatment. Practitioners often refer to this data collection process as logging, but we will refer to it as recording, to avoid overloading existing terminology in the software engineering literature.

The function that estimates causal effect may take many forms. Nearly all such functions can be distilled into estimating the true difference between (1) an outcome under one treatment and (2) its potential outcome(s) under another treatment.

**Average Treatment Effect (ATE).** In the case of a randomized experiment, we wish to know how \( Y \) is affected by a treatment \( T \). If \( T \) is assigned completely at random, for example, according to:

\[
T = \text{uniformChoice}(\text{choices}=[400, 750], \text{unit}=\text{userid});
\]

then the causal effect of \( T \) (the average treatment effect) can be estimated by simply taking the difference of the average outcome for units assigned to \( T = 1 \) and \( T = 0 \): \( \text{Avg}(Y|T=1) - \text{Avg}(Y|T=0) \). Such an experiment could be useful for learning how video watch time differs for equivalent individuals experiencing videos at the 400 or 750kbps setting.

It is not uncommon to use different probabilities for different kinds of users, such that some observed subgroup \( S \) causes us to assign users to treatments with different probabilities. This is illustrated in Figure 1: a user’s country, and therefore \( \text{emerging}_\text{market} \), will depend on some property of the population of users. If we did not consider \( \text{emerging}_\text{markets} \) in constructing our treatment control contrast, shown in Figure 2, then our analysis would be incorrect. We can still estimate causal effects, but must instead compute the difference in means separately for different values of the variables in \( S \). This is often referred to as **subgroup analysis**.

Were we to compare average percentage of videos watched for users outside India, Brazil, the US, or Canada with the average percentage of videos watched for users within these markets, our estimates of causal effect of \( \text{max}_\text{bitrate} \) on abandonment would almost certainly be biased; the way lines 9-17 are written almost guarantees this. However, if we condition on the variables \( \text{emerging}_\text{market} \) and \( \text{established}_\text{market} \), it becomes valid to estimate \( \text{max}_\text{bitrate} \). This estimand is known as the **conditional average treatment effect** (CATE). In this paper, we will refer to ATE and CATE as both belonging to the same family of estimators, since ATE is a specific case of CATE (i.e., one with an empty conditioning set). Average effect estimators over finite sets of treatments can be expressed in terms of their valid contrasts.

### 3.2 Threats to Validity of Causal Inference

Three of the most serious threats to validity in causal inference are selection bias, confounding, and a lack of positivity [Hernán and Robins 2016]. Some instances of these threats can be readily identified in programmatically defined experiments.

**Selection Bias.** Selection bias occurs when the sample chosen for treatment and analysis in the experiment differs in a systematic way from the underlying population. This bias can arise due to a variety of reasons:
Non-proportional sampling causes selection bias by sampling from a subgroup that does not resemble the population. Wang et al. [2015] showed how to correct for this bias during post-hoc analysis of political polling data from XBox users; the correction for a non-proportional sample requires the researcher to have access to a model of how the sampling population differs from the target population. When online experiments seek to answer questions about a platform’s user base (e.g. hypotheses about user engagement with ads), non-proportional sampling is not likely to be an issue. When researchers seek to extend hypotheses to the general population or offline behaviors, they may find that their sample differs in unexpected ways. For example, researchers across many disciplines use Amazon’s Mechanical Turk (AMT) to collect data. Ahler et al. [2019] have shown that while both online and offline subjects engage in satisficing and trolling behavior, online respondents can spoof IP addresses, faking locations, while participating in studies more than once. These subjects pose a greater threat to the validity of AMT results than, e.g., random respondents or bots, due to their non-uniform (and therefore biased) behavior.

Although non-proportional sampling is a cause of selection bias, in an online context it is tied to hypotheses, reasoning over which is currently out of scope of this work. We see an opportunity for future work in joint reasoning about hypotheses (e.g. predicates over the distribution of units, treatments, and the outcome variable Y) in order to truly validate programmatically defined experiments. Doing so requires additional information about the population, any proxy variables used in the hypotheses (e.g., using “sad” words denotes a sad mood), and ground-truth hypotheses from researchers. We know of no experimentation system that includes formally specified hypotheses.

A failure causes complete loss of data between the assignment of units to treatment and the recording of results (known as dropout or attrition). Analogous behavior for online experiments may not be entirely relevant—e.g., were users to disable JavaScript or quit a social network, they may no longer be considered part of the population. On the other hand, if, for example, during a storm a data center were to lose power and an experiment not be deployed to users within in particular region, we would have a form of dropout.

Since dropout, by definition, only occurs after treatment assignment, recognizing it would require a form of dynamic analysis and thus it is outside the scope of this paper.

During the mapping from subjects to treatments, randomization may fail. In offline contexts, this could be due to a breakdown in the experimental protocol. For example, a post-publication analysis of a large-scale randomized field experiment regarding the effects of the Mediterranean diet revealed that one of the investigators improperly assigned an entire village to the same treatment, despite an experimental protocol indicating that individuals be randomly assigned treatment [Estruch et al. 2013; Martinez-Gonzalez and Bes-Rastrollo 2014; McCook 2018]. Although such deviations from the experimental protocol are unlikely to occur in programmatically defined experiments, it is always possible that there are flaws in the program itself that cause deterministic, and therefore biased, assignment of treatments to subjects.

This type of selection bias can be detected statically. In Section 5, we expand on biases of this variety, and describe how they may be expressed and detected in PlanOut programs.

Confounding. Confounding occurs when there exists a variable that causes both treatment and outcome. For example, a common type of observational study of epidemiological effects compares between medicines prescribed, but the underlying disease causes both the choice of medicine and
the subsequent symptoms [Signorello et al. 2002]. Confounding is a major threat to validity in observational studies, where researchers have no control over treatment assignment.

Randomized treatment assignment ensures that there are no confounding variables. However, a fully randomized experiment is not always possible or desirable. In some cases, random assignment can be conditioned on some feature of the unit. This can introduce confounding, especially when an experimentation script is sufficiently complex or where the feature that influenced assignment is not recorded. As long as the confounding variable can be measured, its effect may be factored out at analysis time, provided the analyst properly adjusts for it [Pearl 2009].

**Lack of Positivity.** When a treatment has some nonzero likelihood of being observed in a data set for each combination of relevant variable values (i.e., across covariates), it has positivity. A lack of positivity can lead to statistical bias, especially during subgroup analysis. Suppose you are conducting a drug trial for breast cancer. Due to suspected heterogeneity of the efficacy of the drug by sex, you analyze the data for men and women separately. However, your sample has no men, due to the significantly lower rate of breast cancer among men [Society 2019a,b]. In this case, your study lacks positivity among men, since they have zero probability of treatment.

While an experimenter can sometimes correct for a lack of positivity in downstream statistical analysis, they must first be able to detect it. Positivity is not typically considered an issue in experiments, since experimenters are thought to completely control the assignment process. However, in the context of online experiments, there are some nontraditional and surprising reasons why positivity may not be achieved. Some of these reasons can be detected statically (§5).

### 3.3 The PlanOut Framework

Many firms have pre-existing methods for computing metrics of interest, and record these data regardless of the presence of an experiment. Figure 3 depicts these existing systems in the “Data” and “Application” components of the diagram. The PlanOut framework can be added to record the assignment of units to treatments. Once the execution of a PlanOut script completes, there is a store that contains the mappings from variables defined in PlanOut to their values; these variables and their values are recorded in the “Experiment Store.”

![Diagram of PlanOut framework](image-url)
A PlanOut script specifies the assignment of treatments to units, although assignment does not happen until the PlanOut script is executed by the PlanOut interpreter. The interpreter takes as input: (1) the PlanOut program, (2) each unit $u$ of the sample of units specified in the selected sample(s), and (3) any unit-related data queried by external operators or application-specific data that might be required by the treatment procedure. The interpreter deterministically hashes units to treatments in an effectively random fashion.

Treatments and outcomes must be recorded in order to estimate causal effect. Outcomes typically correspond to variables or metrics already being recorded by the application’s data-recording infrastructure. The PlanOut framework includes additional data-recording capabilities that record experiment metadata.

Important Language Features. As a DSL built by domain experts, PlanOut implements functionality only relevant to experimentation. Consequently, PlanOut is not Turing complete: it lacks loops, recursion, and function definition. It has two control flow constructs (if/else and return) and a small core of built-in functions (e.g., weightedChoice, bernoulliTrial, and length).

On its surface, PlanOut may appear to share features with probabilistic programming languages (PPLs) [Gordon et al. 2014a,b; Minka et al. 2014; Pfeffer 2016; Wood et al. 2014]. PPLs completely describe the data generating process; in contrast, PlanOut programs specify only one part of the data generating process—how to randomly assign treatments—and this code is used to control aspects of a product or service that is the focus of experimentation.

There are two critical features of PlanOut that differentiate it from related DSLs, such as PPLs: (1) the requirement that all random functions have an explicit unit of randomization, and (2) built-in control of data recording via the truth value of PlanOut’s return. Only named variables on paths that terminate in return true are recorded. This is similar to the discarded executions in the implementation of conditional probabilities in PPLs. A major semantic difference between PlanOut and PPLs is that we expect PlanOut to have deterministic execution for an input. Variability in PlanOut arises from the population of inputs; variability in PPLs come from the execution of the program itself.

Framework System Assumptions. PlanOut abstracts over the sampling mechanism, providing an interface that randomly selects from pre-populated partitions of unit identifiers, corresponding to samples from the population of interest, as depicted on the far-left-hand side of Figure 3. The interface that selects samples is the Namespace Mapper. This component extracts the application parameters manipulated by a PlanOut script and hashes them, along with the current experiment name, to one or more samples. We have spoken with data scientists and software engineers at several firms that use PlanOut, and they have stated that the mapping from experiments to samples was what drew them to the PlanOut framework. The mapping avoids clashes between concurrently running experiments, which is one of the primary challenges of online experimentation [Kohavi et al. 2013, 2009]. Readers interested in the specifics of PlanOut’s hashing method for scaling concurrent experiments can refer to the paper [Bakshy et al. 2014]; it is not relevant to Planalyzer’s analyses.

4 PLANALYZER STATIC ANALYSIS TOOL

Planalyzer is a command-line tool written in OCaml that performs two main tasks. It: (1) checks whether the input script represents a randomized experiment by validating the presence of random assignment and the absence of any failures related to selection bias, unrecorded confounders, or positivity; and (2) generates all valid contrasts and their associated conditioning sets for the ATE estimator. Planalyzer translates PlanOut programs to an intermediate representation (§4.1) and
assigns special labels to variables in the program (§4.2). It then builds a data dependence graph (DDG), which it uses to generate contrasts for the ATE estimator (§4.3).

4.1 PlanOut Intermediate Representation

Upon parsing, PlanAlyzer performs several routine program transformations. It: (1) converts variables to an identification scheme similar to SSA, (2) performs constant propagation, and (3) rewrites functions and relations (such as equality) in A-normal form [Aho et al. 1986; Cytron et al. 1991; Muchnick 1997; Sabry and Felleisen 1993]. Expressions may contain external function calls as subexpressions. Since it may not be possible to reason about the final values of a variable defined in a PlanOut program, PlanAlyzer reasons about intermediate values instead and reports results over a partially evaluated program [Futamura 1999].

After these routine transformations, PlanAlyzer splits the program into straight line code via tail duplication, such that every path through the program may be evaluated in isolation of the others. Although this transformation is exponential in the number of conditional branches, in practice the branching factor of PlanOut programs is quite small (discussed in §7 and Figure 9).

PlanAlyzer then converts guards into assertions and uses the Z3 SMT solver to ensure that variables assigned along paths are consistent with these assertions [De Moura and Bjørner 2008]. For each assertion, PlanAlyzer queries Z3 twice—first to obtain a satisfying solution, and then to test whether this solution is unique. Evaluation of the intermediate representation may contain un-evaluated code, so if there is more than one solution, PlanAlyzer keeps the code chunk abstract. For example, the intermediate representation of code that branches on a threshold such as that depicted in Figure 4 would contain the lines \( fv1 = \max_{br} > 550; \) \( \text{assert }fv1; \). PlanAlyzer would instantiate \(fv1\), but would keep \(\max_{br}\) abstract. When pretty printing the output as in Figure 2, PlanAlyzer does not show intermediate values; the csv output, however, contains all intermediate variables and unevaluated expressions.

PlanAlyzer uses SSA and A-normal form because they aid in contrast generation: a single execution of a PlanOut program corresponds to the assignment of a unit to a treatment. However, additional intermediate variables can have somewhat ambiguous semantics when attempting to model a programmatically defined experiment causally; although they aid in e.g., the detection of causal sufficiency errors (§5), they make reasoning about causal inference using tools such as causal graphical models quite difficult (§4.3).

4.2 Variable Labels for Causal Inference

The PlanOut language contains only some of the necessary features for reasoning about the validity of experiments. Given only programs written in PlanOut, PlanAlyzer may not be able to reason about some common threats to internal validity. The interaction between random operators and control flow can cause variables to lose either their randomness or their variation. Furthermore, we need some way of guaranteeing that external operators do not introduce confounding.

To express this missing information, we introduce a 4-tuple of variable labels \((\text{rand, card, tv, corr})\) that PlanAlyzer attempts to infer and propagate for each PlanOut program it encounters [Denning 1976; Sabelfeld and Myers 2003]. Unsurprisingly, inference may be overly conservative for programs with many external functions or variables. To increase the scope of experiments PlanAlyzer can analyze, users may supply PlanAlyzer with global and local configuration files that specify labels.
Randomness (rand). PlanOut may be used with existing experimentation systems; this means that there may already be sources of randomness available and familiar to users. Furthermore, since PlanOut was designed to be extensible, users may freely add new random operators.

Cardinality (card). The size of variables’ domains (cardinality) impacts an experiment’s validity. Simple pseudo-random assignment requires high cardinality units of randomization to properly balance the assignment of units into conditions. In the example program of Figure 1, all variables have low cardinality, except for context.

Time Variance (tv). For the duration of a particular experiment, a given variable may be constant or time-varying. Clearly, some variables are always constant or always time varying. For example, date-of-birth is constant, while days-since-last-login is time varying. However, there are many variables that cannot be globally categorized as either constant or time-varying. The tv label allows experimenters to specify whether they expect a variable to be constant or time-varying over the duration of a given experiment.

Since ATE/CATE assumes subjects receive only one treatment value for the duration of the experiment, PLANALYZER cannot use them to estimate the causal effect of treatments or conditioning set variables having a tv label. A PLANOUT program may contain other valid contrasts assigned randomly, and independently from the time-varying contrasts; PLANALYZER will still identify these treatments and their conditioning sets as eligible for being analyzed via ATE/CATE.

Example. The first branch through the program in Figure 1 leads to assignments of max_bitrate that vary with time. This part of the script encodes a contextual bandits experiment, an approach to experimentation that shares many features with reinforcement learning. Because bandits experiments use information from the environment in a loop to determine treatment, multiple visits to a website containing this sort of experiment could result in different treatments. Aggregating across individuals for a particular treatment (as ATE would do) is not sound in this case.

Since assignments along this branch cannot be compared in a between-subjects analysis, PLANALYZER excludes them from the contrasts returned in Figure 2; note that none of the contrasts compare across max_bitrate when dynamic_policy is set to true.

Covariates and Confounders (corr). Many experiments use features of the unit to assign treatment (a type of covariate; see Appendix A), which may introduce confounding. PLANALYZER automatically marks external variables and the direct results of non-random external calls as correlated with outcome (i.e., Y). This signals that, if the variable is used for treatment assignment, either their values must be recorded, or sufficient downstream data must be recorded to recover their values.

Example. PLANALYZER marks the variable country in Figure 1 as correlated with Y. Although data from the country variable affects treatment assignment, this information is captured by variables emerging_market and established_market, so PLANALYZER raises no error.

4.3 Data Dependence Graph (DDG)
PLANALYZER builds a DDG to propagate variable label information [Ferrante et al. 1987]. Since PlanOut only has a single, global scope, its data dependence analysis is straightforward:

(1) Assignment induces a directed edge from the references on the right-hand side to the variable name.
(2) Sequential assignment of \( var_i \) and \( var_{i+1} \) induces no dependencies between \( var_i \) and \( var_{i+1} \), unless the r-value of \( var_{i+1} \) includes a reference to \( var_i \).
(3) For an if-statement, PLANALYZER adds an edge from each of the references in the guard to all assignments in the branches.
(4) In the case of an early return, PlanAlyzer adds edges from the variables in dependent guards to all variables defined after the return.

Random, independent assignment implies independence between potential causes, so long as the (possibly empty) conditioning set has been identified and recorded. PlanAlyzer computes the DDG for the full script and uses the full DDG to determine when it is possible to marginalize over some variables. Figure 5 shows an example DDG for the example experiment of Figure 1.

Propagating Variable Labels. PlanAlyzer marks variables directly assigned by built-in random functions or external random functions as random. The randomness label takes a tuple of identifiers as its argument. This tuple denotes the unit(s) of randomization, used for reasoning about causal estimators. Any node with a random ancestor is marked as random (with the exception of variables that do not vary), with units of randomization corresponding to the union of the ancestors’ units. max-bitrate in Figure 5 is random, even though it is set to a constant in Figure 1. This is because assignment is still randomly assigned on the basis of dynamic_policy.

If a random operator uses a low-cardinality unit of randomization, it will be marked as non-random. Note, however, that if the unit of randomization for a random function is a tuple with at least one high cardinality variable, then the resulting variable will remain random.

PlanAlyzer propagates time-varying labels in the same manner as random labels. Unlike randomness, there is no interaction between the time-varying label and any other label.

Converting DDGs to Causal Graphical Models (CGMs). Readers familiar with graphical models may wonder whether the DDG can be transformed into a directed graphical model. Programmatical definitions have two features that, depending on context, make such a transformation either totally inappropriate or difficult to extract: (1) deterministic dependence; and (2) conditional branching. These two features can induce what’s known as “context-sensitive independence,” which limits the effectiveness of existing algorithms that would otherwise make graphical models an appealing target semantics. Although some work has sought to remedy branching, treatment of context-sensitive independence in graphical models more broadly is an open research problem [Minka and Winn 2009]. Furthermore, from a practical perspective, it is unclear how the versioned variables in the DDG ought to be unified, and some variables simply don’t belong in a CGM (e.g., userid).

5 DETECTING THREATS TO INTERNAL VALIDITY

We characterize some static threats to internal validity based on the forms of bias in experimental design, described in §3. Note that because there is currently no underlying formalism for the correctness of online field experiments that maps cleanly to a programming language context, we cannot define a soundness theorem for programmatical defined experiments. Some of the
threats described below would be more properly considered code smells, rather than outright errors [Fowler 2018].

Randomization Failures. There are three ways a PLANOUT program may contain a failure of randomization: (1) when it records data along a path that is not randomized, (2) when the units of randomization have low cardinality, and (3) when it encounters path-induced determinism.

Recording data along non-randomized paths occurs when there exists at least one recorded path through the program that is randomized and at least one recorded path through the program that is not randomized. Imagine the engineering team from Section 1 ran only the country-level design (i.e., Lines 9–17 of Figure 1). Then, data for users from outside the four countries of interest would be recorded in the experiment store in Figure 3, alongside data for users from within those countries. PLANALYZER raises an error for this path. The fix is simple: add \texttt{return false} after Line 15.

Units of randomization must have significantly higher cardinality than experimental treatments to ensure that each treatment is assigned sufficient experimental units to make valid statistical inferences about the population. Users can correct this by either annotating the unit of randomization as having high cardinality, or re-assessing their choice of unit.

Data-flow failures of randomization occur when inappropriate computations flow into units. PLANOUT allows units to be the result of arbitrary computations: e.g., one example PLANOUT script in the corpus described in Section 6 sets the unit of randomization to be \texttt{userid \* 2}. A PLANOUT user might want to do this when re-running an experiment, to ensure that at least some users are assigned to a new treatment. However, this feature can lead to deterministic assignment when used improperly. The following is a syntactically valid PLANOUT program; PLANALYZER detects an error in it when it converts the program to its intermediate representation:

\[
\begin{align*}
\text{max_br} & = \text{uniformChoice}(\text{choices}=[400, 900], \text{unit}=$\text{userid}$); \\
\text{dynamic\_policy} & = \text{bernoulliTrial}(p=0.3, \text{unit}=$\text{max\_br}$);
\end{align*}
\]

When writing this code, the researcher may believe that there are four possible assignments for the pair of variables. However, because the assignment of input units to a particular value is the result of a deterministic hashing function, every user who is assigned \texttt{max\_br=400}, is assigned the same value of \texttt{dynamic\_policy} because the input to the hash function for \texttt{bernoulliTrial} is always 400. Therefore, they will never record both (400, true) and (400, false) in the data, which likely contradicts the programmer’s intent.

Treatment Assignment Failures. PLANALYZER requires that all assigned treatments along a path have the possibility of being assigned to at least one unit, and that at least some treatments may be compared. There are three ways a PLANOUT program may contain a failure of treatment assignment, when: (1) some treatment has a zero probability of being assigned (i.e., a positivity error); (2) there are fewer than two treatments that may be compared along a path; and (3) dead code blocks containing treatment assignment.

Syntactically correct PLANOUT code permits users to set probabilities or weights to zero, either directly or as the result of evaluation. A zero-valued weight may flow in from earlier computation or be due to type puns or conversions. Furthermore, to establish a causal relationship between variables, there must be at least two alternative treatments under comparison.

When PLANALYZER expands all possible worlds to generate all possible treatments, it checks that there is at least one assignment for the free variables in the guard that causes the guard to evaluate to true and that there is at least one assignment that causes it to evaluated to false. If only one final value (i.e., true or false) is possible, then PLANALYZER raises an error.
Causal Sufficiency Errors. One of the main assumptions underlying causal reasoning is causal sufficiency, or the assumption that there are no unmeasured confounders in the estimate of treatment effect. Barring run-time failures, we have a complete picture of the assignment mechanism in PlanOut programs. Unfortunately, a PlanOut program may allow an unrecorded variable to bias treatment assignment.

Consider a program that assigns treatment on the basis of user country, accessed via a getUserCountry function:

```java
if (getUserCountry(userid=userid) == 'CA') {
    mxbr = uniformChoice(choices=[75, 90], unit=userid);
} else {
    mxbr = uniformChoice(choices=[40, 75, 90], unit=userid);
}
```

Treatment assignment of mxbr depends on user country, so user country is a potential confounder. Planalyzer will convert the guard to A-normal form and assign the getUserCountry function call to a fresh variable. Because this variable does not appear in the input program text, it cannot be recorded by the PlanOut framework’s data recording system. Therefore, the program and resulting analyses will violate the causal sufficiency assumption.

If Planalyzer encounters a static error or threat, it reports that the script failed to pass validation and gives a reason to the user. Some of the fixes are easy to determine from the error and could be interpolated automatically. We leave this to future work. Other errors require a more sophisticated understanding of the experiment the script represents and can only be determined by the script’s author.

6 PLANOUT CORPORA

Facebook provided a corpus of PlanOut scripts that we used to evaluate Planalyzer via a single point of contact. This corpus contains every PlanOut script written between 3 August 2015 and 3 August 2017. The actual dates (i.e., the choice of 3 August) was arbitrary. The start year was at a late enough point after PlanOut’s introduction at Facebook that the language implementation was stable. The timeframe allowed us to be sure that the experiments in the corpus were completed at the time of analysis. Facebook also provided us with a corpus of manually specified contrasts that were used in the analysis of the experimentation scripts that were actually deployed.

Each experiment may have been updated in vivo or may have a temporary (but syntactically valid) representation captured by a snapshotting system, leading to multiple versions of a single experiment. Some experiments are programmatically generated, leading to verbose experiments that are much longer than what a human might write.

The tool used for analyzing scripts can only be used for ATE analysis (not CATE), and so it provides a meaningful point of comparison for Planalyzer. While we do not have access to the custom analyses of more complex experiments (e.g., database queries, R code, etc.) we can infer some characteristics of the intended analysis by partitioning the corpus into three sub-corpora:

**PlanOut-A** This corpus contains scripts that were analyzed using some form of ATE (i.e., $\text{Avg}(Y|T_1 = t_1, \ldots, T_n = t_n) - \text{Avg}(Y|T_1 = t_1', \ldots, T_n = t_n')$), where the variables $T_1, \ldots, T_n$ were manually specified and automatically recorded during the duration of the experiment. Users may manually specify that a subset of the recorded variables be continuously monitored for pairwise ATE. Neither the recording, nor the data analysis tools have any knowledge of PlanOut. This is the main corpus we will use for evaluating Planalyzer, since the goal of Planalyzer is to automate analyses that firms such as Facebook must now do manually.

**PlanOut-B** Some scripts have data recorded, but no automated analyses. This may be because the scripts are not suited to ATE. We analyze the scripts in this corpus to see whether there are
Table 1. Descriptive statistics for the corpus. Input variables correspond to the number of unique variables in the input program. IR paths correspond to the number of paths generated for the intermediate representation of the program. The unique authors listed refer to authors that only appear in that corpus. There was overlap in authorship between each of the corpora, with 15 authors appearing in all three. The data was collected over a two year period.

| Corpus | PLANOUT-A: Confirmed Deployed and Contrasts Analyzed |  |
|--------|-----------------------------------------------------|----|
|        | Unique PLANOUT Scripts | 566 |
|        | Unique Experiments      | 240 |
|        | Unique Authors (Total)  | 30 (70) |
| Min. Versions | 1 Min. LOC | 1 Min. Input Vars. | 1 Min. IR Paths | 1 |
| Med. Versions | 3 Med. LOC | 45 Med. Input Vars. | 9 Med. IR Paths | 4 |
| Avg. Versions | 4 Avg. LOC | 78 Avg. Input Vars. | 10 Avg. IR Paths | 29 |
| Max. Versions | 28 Max. LOC | 691 Max. Input Vars. | 60 Max. IR Paths | 6561 |

| Corpus | PLANOUT-B: Confirmed Deployed and Contrasts Recorded |  |
|--------|-----------------------------------------------------|----|
|        | Unique PLANOUT Scripts | 381 |
|        | Unique Experiments      | 130 |
|        | Unique Authors (Total)  | 25 (72) |
| Min. Versions | 1 Min. LOC | 1 Min. Input Vars. | 1 Min. IR Paths | 1 |
| Med. Versions | 3 Med. LOC | 26 Med. Input Vars. | 7 Med. IR Paths | 3 |
| Avg. Versions | 4 Avg. LOC | 41 Avg. Input Vars. | 7 Avg. IR Paths | 7 |
| Max. Versions | 32 Max. LOC | 495 Max. Input Vars. | 26 Max. IR Paths | 253 |

| Corpus | PLANOUT-C: Not Deployed |  |
|--------|-------------------------|----|
|        | Unique PLANOUT Scripts | 493 |
|        | Unique Experiments      | 74 |
|        | Unique Authors (Total)  | 23 (47) |
| Min. Versions | 1 Min. LOC | 1 Min. Input Vars. | 1 Min. IR Paths | 10 |
| Med. Versions | 3 Med. LOC | 56 Med. Input Vars. | 13 Med. IR Paths | 4 |
| Avg. Versions | 8 Avg. LOC | 137 Avg. Input Vars. | 16 Avg. IR Paths | 368 |
| Max. Versions | 124 Max. LOC | 883 Max. Input Vars. | 48 Max. IR Paths | 27675 |

any CATE analyses that PLANALYZER can identify. This corpus may also contain custom analyses for within-subjects experiments, contextual bandits experiments, experiments that must account for peer effects, etc.

**PLANOUT-C** These are scripts that have never been deployed and therefore may not have had the oversight of domain experts. This corpus provides the best approximation of the kinds of mistakes that PLANOUT users actually make.

Note that users at Facebook are typically either experts in the domain of the hypotheses being tested or they are analysts working directly with domain experts. Therefore, PLANOUT-C was our best chance of finding scripts by non-experts in experimental design (although they were likely still domain experts). In all cases, experiments undergo review before being deployed. Table 1 gives a more detailed description of the corpora.

### 6.1 Characterizing Representative PLANOUT Programs

We designed PLANALYZER’s analyses on the basis of the universe of syntactically valid PLANOUT programs and our domain knowledge of experimentation. We built PLANALYZER from the perspective that (1) PLANOUT is the primary means by which experimenters design and deploy experiments, but (2) they can use other systems, if they exist. Facebook uses many experimentation systems and has a variety of human and code-review methods for the functionality that PLANALYZER
show_feature = true;
in_exp = false;
if (userid == 123456789) {
in_exp = true;
if (post_has_photo == true) {
show_feature = false;
}   
if (post_has_video == true) {
show_feature = false;
}   
return true;
}
in_pop = extPred(ep="in_pop", userid=userid);
if (!in_pop || userid == 0) {
return false;
}
in_exp = bernoulliTrial(p=1/100, unit=post_id);
if (!in_exp) {
return false;
}
if (post_has_photo == true) {
show_feature = false;
}
else {
enabled = true;
}
else if (extPred(userid=userid, ep="feat_rollout")) {
enabled = true;
}
if (enabled) {
animation_type = "WINDOW_ANIMATION";
} else {
animation_type = "CONTROL";
}
if (skip_logging) {
return false;
}

skip_logging = true;
enabled = false;
if (extPred(userid=userid, ep="feat_exp")) {
test = bernoulliTrial(p=1/25, unit=userid);
if (!test) {
enabled = bernoulliTrial(p=1/2, unit=userid);
skip_logging = false;
} else {
enabled = true;
}
else if (extPred(userid=userid, ep="feat_rollout")) {
enabled = true;
}
if (enabled) {
animation_type = "WINDOW_ANIMATION";
} else {
animation_type = "CONTROL";
}
if (skip_logging) {
return false;
}

Fig. 6. Some representative, lightly edited and anonymized experiments written in PlanOut. Left: This script mixes testing code with experimentation code. Lines 5-12 set values for user 123456789 and record those values. The actual experiment is in lines 14-26. It is only conducted on the population defined by the external predicate and the user being recorded in (represented here when the userid is 0). PlanAlyzer raises an error for this script. Right: PlanAlyzer raises a Causal Sufficiency error for this lightly edited and anonymized experiment, assuming that the call to extPred in line 3 is non-random.

Ambiguous Semantics and Type Errors. Since PlanAlyzer must initially perform type inference, it found 87 scripts in PlanOut-A that had typing errors. By far the most common issue flagged was the treatment of 0 as falsey. Upon manually inspecting the scripts, we found that most scripts could be modified so that these variables were consistently Boolean or numeric, depending on usage.
Table 2. The counts of code smells, static script errors, and tool failures found when running PlanAlyzer on the corpora. A PlanAlyzer error does not necessarily indicate that the experiment was run in error. A single experiment may have many script versions, not all of which were deployed. The numbers for PlanOut-A reflect the state of the corpus after adjustments for easily fixed type inconsistencies (initially 87), since we know those scripts ran in production, and wanted to see if PlanAlyzer could find more interesting errors or smells. There were no adjustments to the other two corpora. The grey band represents manual analysis, only performed on PlanOut-A. *We analyzed PlanOut-A with a max number of random variable choices of 100 and the other two corpora with the default setting of 20.

| Output Category                        | PlanOut-A Scripts (566) | Exps. (240) | PlanOut-B Scripts (381) | Exps. (130) | PlanOut-C Scripts (493) | Exps. (74) |
|----------------------------------------|-------------------------|------------|-------------------------|------------|-------------------------|------------|
| Not an experiment                      | 10                      | 10         | 8                       | 5          | 22                      | 8          |
| Low cardinality unit                  | 7                       | 1          | 6                       | 2          | 1                       | 1          |
| Ambiguous semantics                   | 5                       | 2          | 0                       | 0          | 0                       | 0          |
| Type inconsistencies                  | 10                      | 4          | 36                      | 12         | 4                       | 2          |
| Causal sufficiency errors             | 111                     | 54         | 75                      | 22         | 77                      | 17         |
| False positive                        |                         |            |                         |            |                         |            |
| Testing code                          |                         |            |                         |            |                         |            |
| Possible random assignment            |                         |            |                         |            |                         |            |
| Recorded no randomization             | 25                      | 11         | 83                      | 23         | 214                     | 35         |
| Missed paths (tests)                  |                         |            |                         |            |                         |            |
| No randomization (config)             | 12                      | 7          |                         |            |                         |            |
| Possible random assignment            |                         |            |                         |            |                         |            |
| Random variable no variation          | 2                       | 2          | 22                      | 15         | 0                       | 0          |
| Exceeds max choices*                  | 0                       | 0          | 21                      | 14         | 0                       | 0          |
| No positivity                         | 7                       | 3          | 0                       | 0          | 9                       | 4          |
| Dead code                             | 5                       | 4          | 1                       | 1          | 4                       | 2          |
| Feature not implemented in tool       | 29                      | 8          | 29                      | 10         | 37                      | 7          |
| Overflow error                         | 0                       | 0          | 0                       | 0          | 80                      | 4          |

Other typing issues included string values such as "default" and "status_quo" for numeric variables and guards such as userid == 0 || userid == "0", which suggest there might be some utility in providing our type checking facility to users of PlanOut.

We also found three scripts from one experiment that applied the modulus operator to a fraction; since PlanOut uses the semantics of its enclosing environment for numeric computation, this script will return different values if it is run using languages with different semantics for modulus, such as PHP versus JavaScript.

Modifying Deployment Settings within Experimentation Logic. Some of the scripts marked as not experiments begin with return false and had an unreachable and fully specified experiment below the return statement. PlanAlyzer flags dead code in PlanOut programs, since it can be the result of a randomly assigned variable causing unintended downstream control flow behavior. However, every dead code example we found had the form condition = false; if (condition) ... These features occurred exclusively in experiments that had multiple scripts associated with them that did not raise these errors. After discussing our findings with Facebook, we believe that this might be a case of PlanOut authors modifying the experiment while it is running to control deployment, rather than leaving dead-code in by accident, as it appears from PlanAlyzer’s perspective.
Using PLANOut for Application Configuration. One of the most surprising characteristics we found in PLANOut-A was the prevalence of using PLANOut for application configuration, à la Akamai’s ACMS system or Facebook’s Gatekeeper [Sherman et al. 2005; Tang et al. 2015]. When these scripts set variables, but properly turned off data recording (i.e., returned false), PLANALYZER marked them as not being experiments. When they did not turn off logging, they were marked as recording paths without randomization. Some instances of application configuration involved setting the support of a randomly assigned variable to a constant or setting a weight to zero. Since experiments require variation for comparison, PLANALYZER raises an error if the user attempts to randomly select from a set of fewer than two choices. Three scripts contained expressions of the form `uniformChoice (choices=[v], unit=user_id)` for some constant value $v$.

As a result, users who aim to use PLANOut as a configuration system have no need for PLANALYZER, but anyone writing experiments would consider these scripts buggy.

Mixing External Calls to Other Experimentation Systems. Almost 20% of the scripts (106) include calls to external experimentation systems. In a small number of cases, PLANOut is used exclusively for managing these other systems, with no calls to its built-in random operators.

Non-read-only Units. One of the other firms we spoke to that uses PLANOut treats units of randomization as read-only, unlike other variables in PLANOut programs. Facebook does not do this. Therefore, programs that manipulate the unit of randomization may be legal: for instance, the aforementioned instance where the unit was set to `user_id * 2`. We also observed a case where the unit was set to be the result of an external call—without knowing the behavior of this external call it is assumed to be low cardinality. In this case, the experiment was performing cluster random assignment, which is not covered by ATE and out of scope for PLANALYZER.

Inspection for CATE. We investigated the conditioning sets PLANALYZER produces to see whether there were any contrasts that ought to be computing CATE, rather than ATE. Unfortunately, our gold standard annotation set PLANOut-A does not include any experiments that were analyzed with CATE, as the labels were collected with a system that only supports ATE. PLANALYZER produced conditioning sets for seven experiments; warning that ATE would not be valid for certain subsets of variables, but the gold truth data told us these experiments were aimed at learning variables for which ATE was valid (i.e., they were similar to the comparison between policies listed first in Fig 2). Thus, these seven experiments produced false alarms. However, it is reasonable for PLANALYZER to produce these conditioning sets because it does not have access to the variables of interest through the PLANOut language.

Tool Limitations. Twenty-two scripts required some more complex transformations to SMT logic that we have not yet implemented: all cases involved reasoning about map lookups or null values. The remaining seven scripts were all versions of a single experiment that used the `sample` function, which PLANALYZER does not currently support; this function generates $\binom{n}{k}$ subsets of size $k$ from a list of size $n$, but was left for future work since it is so rarely used.
We did not expect to see any real causal sufficiency errors, due to the expert nature of the authors of PlanOut-A. Rather, we expect to see some false positives, due to the fact that PlanAlyzer is aggressive about flagging potential causal sufficiency errors. We made this design choice because the cost of unrecorded confounders can be very high. Furthermore, the fix is quite easy and can be automated, were PlanAlyzer to be integrated in a PlanOut editor. The errors that were not false positives were either cases in which the author intermingled testing code with experiment code, or where branching depending on an external function call that is sometimes random.

As an example, the code example on the left side of Figure 6 contains testing code mixed with experiment code. This scripts raises an error due to the fact that there is a recorded path with no randomization on it. The right side of Figure 6 is an example of a causal sufficiency error; the result of the call to extPred is not recorded, but could be correlated with both treatment and outcome.

6.1.2 Characteristics of PlanOut-B. Our main interest in PlanOut-B is to identify whether there are experiments that could benefit from CATE analysis. Note that in §6.1.1, we found several experiments that were eligible to be analyzed with both ATE and CATE. In PlanOut-B we found 14 scripts spanning nine experiments that contain analyses eligible for CATE.

PlanOut-B, as a corpus, has very similar characteristics to PlanOut-A: authors still mix deployment logic with experimentation logic, use PlanOut for what appears to application configuration, and use idiomatic expressions that may not typecheck.

6.1.3 Characteristics of PlanOut-C. Recall that scripts in PlanOut-C were never deployed. Investigating a subset of these scripts, we believe that this corpus is largely filled with scripts trying out certain features. For example, one extremely large script appears to be automatically generated. PlanOut-C was the only corpus that caused the tool to crash. As depicted in Table 1, the maximum number of paths in this corpus is an order of magnitude more than PlanOut-A. Figure 9, which we discuss in depth in 7.3, depicts how this very large number of paths contributes to the runtime.

6.1.4 Findings. PlanOut scripts in deployment at Facebook represent a range of experimental designs. We observed factorial designs, conditional assignment, within-subjects experiments, cluster random assignment, and bandits experiments in the scripts we examined.

PlanOut has the look and feel of writing Python, R, or other scripting languages popular among data scientists. However, without a unified coding style, and no restrictions on program correctness other than parsing, there is considerable variability in the ways experiment authors use PlanOut. This variability includes implementing behavior that PlanOut is not suited to solve.

7 EVALUATION

Real-world PlanOut scripts unsurprisingly contained few errors, since they were primarily written and overseen by experts in experimental design. For example, of the 25 recorded paths with no randomization, nine contained a special gating function that may sometimes be random, depending on its arguments. Four of the scripts appeared to be using PlanOut for configuration, leaving twelve scripts that essentially implemented application configuration logic.

Therefore, to test how well PlanAlyzer finds errors, we selected a subset of fifty scripts from PlanOut-A and mutated them. Table 3 describes the mutations we performed and the type of effect on output we expected. We then validated a subset of the contrasts PlanAlyzer produced against a corpus of hand-selected contrasts monitored and compared by an automated tool used at Facebook. Finally, we report on PlanAlyzer’s performance, since its effectiveness requires accurately identifying meaningful contrasts within a reasonable amount of time.

### Table 3. Mutations injected into sample PlanOut programs.

| Mutation | Description                                                                 | Fault?                                                                 |
|----------|-----------------------------------------------------------------------------|----------------------------------------------------------------------|
| SAI      | Subpopulation Analysis Insert; wraps a node of the AST in an if-then-else state where the guard is a feature of population, such that if the guard is true, the wrapped node is executed and recorded, and if the guard is false, the program returns false. | Never: creates a subpopulation frame that should not be in the conditioning set. |
| CI       | Constant Insert; inserts a variable assignment from a constant.             | Never: constant assignment should have no bearing on the presence of errors. They may sometimes affect the treatments or conditioning sets. |
| EFCI     | External Function Call Insert; inserts a variable assignment from an external function call that could be correlated with outcome. | Never: the variable defined cannot be correlated with treatment assignment. |
| RSI      | Return Statement Insert; inserts a return statement at an arbitrary point in a statement tree. The only restriction is that a new return statement cannot be added after another return statement. | Sometimes: a return true statement may be inserted before a return false, causing an error. |
| CSE      | Causal Sufficiency Error; wraps a node of the AST in an if-then-else statement where the guard is a feature of the population. | Sometimes: if the mutation induces a dependency between the guard and treatment assignment, then the script will contain an error. |
| URE-1    | Unit of Randomization Error; replaces a unit of randomization with an expression containing the former unit and constants. | Sometimes: some operations can reduce the cardinality of the unit, for example, modulus. |
| URE-2    | Unit of Randomization Error; replaces the unit with another high cardinality unit. | Sometimes: replacing the unit may make some treatments within-subjects. |
| URE-3    | Unit of Randomization Error; replaces the unit with another variable defined previously in the program | Sometimes: variables defined in the program should almost always have low-cardinality, however sometimes authors include functions of e.g. userid. |

### 7.1 Mutation Methodology

We first identified scripts that were eligible for this analysis. We modified the PlanOut-A scripts that raised errors when it was appropriate to do so. For example, we updated a number of the scripts that erroneously raised causal sufficiency errors so that they would not raise those errors anymore. We excluded scripts that, for example, contained testing code or configuration code. This allowed us to be reasonably certain that most of the input scripts were correct.

All of our mutations operate over input PlanOut programs, rather than the intermediate representation. We believed this approach would better stress PlanAlyzer. We perform one mutation per script.

We considered two approaches when deciding how to perform the mutations:
Table 4. We apply each type of mutation at a rate proportional to the eligible nodes in the input program’s AST. We found the overall rates by applying our mutations over the PlanOut-A corpus. We assessed PlanAlyzer on over fifty randomly selected scripts from PlanOut-A. PlanAlyzer had a precision and recall of both 92%. We these outcomes in §7.1.

(a) Mutation proportions.

| Mutation | Probability |
|----------|-------------|
| CI       | 0.22        |
| CSE      | 0.20        |
| EFCI     | 0.18        |
| RTI      | 0.15        |
| SAI      | 0.22        |
| URE      | 0.02        |

(b) Mutation results.

|          | True Pos. | False Pos. | True Neg. | False Neg. |
|----------|-----------|------------|-----------|------------|
| CI       | 0         | 0          | 8         | 0          |
| CSE      | 9         | 1          | 3         | 0          |
| EFCI     | 0         | 0          | 9         | 0          |
| RTI      | 1         | 0          | 3         | 0          |
| SAI      | 0         | 0          | 13        | 1          |
| URE      | 2         | 0          | 0         | 0          |

1. Randomly select a mutation type, and then randomly select from the eligible AST points for that mutation.
2. Generate all of the eligible AST points for all of the mutations, and then randomly select from this set.

Method 1 leads to an even split between the classes of mutations in the test corpus; method 2 leads to frequencies that are proportional to the frequencies of the eligible AST nodes. We chose the latter because we believed it would lead to a more accurate representation of real programming errors. Table 4a gives the probability of a script containing a particular mutation type.

To select the subset of scripts to evaluate, we sampled fifty experiments and then selected a random script version from that experiment. We then manually inspected the mutated script and compared the output of the mutation with the original output.

**Findings:** Fault Identification over Mutated Scripts. When analyzing our sample of fifty mutated scripts, PlanAlyzer produced only one false positive and only one false negative. The precision and recall were both 92%. On the one hand, this is very surprising, given both the false positive rate in the PlanOut-A corpus for causal sufficiency errors (8%) and the proportion of CSE mutations in this sample (28%). However, we found that most of the CSE mutations caused the program to exit before random assignment, causing PlanAlyzer to raise legitimate errors about recorded paths with no randomization. The rest were true causal sufficiency errors (i.e., they would cause bias in treatment).

The one false negative we observed occurred in a script that re-defined the treatment variable for two userids, in what appears to be testing code. The mutation wrapped the redefined treatment, so this is a case where PlanAlyzer should have raised a “no randomization error” in both the input script as well as the mutated script.

7.2 Validation against Human-generated Contrasts

We decided whether an experiment should be in the subset according to the following three criteria: (1) all variables in the human-generated contrasts appeared in the original script; (2) PlanAlyzer was able to produce at least one contrast for the experiment; and (3) PlanAlyzer produced identical contrasts across all versions of the experiment. Criteria (1) and (2) ensure that analysis does not require knowledge unavailable to PlanAlyzer. Criteria (3) is necessary because the tool that monitors contrasts logs them per-experiment, not per-version. If the possible contrasts change between versions, we cannot be sure which version corresponded to the data. Ninety-five of the 240 unique experiments met these criteria.
foo = bernoulliTrial (p=0.4, unit=userid);  
if (get_subpop (userid=userid)) {  
  if (foo) {  
    mode = 2;  
    target_level = 3;  
    compression_gain = 3;  
    lim_enabled = 1;  
  } else {  
    return true;  
  } 
} else {  
  max_delay = uniformChoice (choices=[500, 1000, 2000, 4000, 10000], unit=userid);  
  if (max_delay == 500) {  
    horizon_seconds = 1;  
    max_packets = 9;  
  } else if (max_delay == 1000) {  
    horizon_seconds = 10;  
    max_packets = 166;  
  } else {  
    horizon_seconds = 100;  
    max_packets = 166;  
  }  
  limit_probability = uniformChoice (choices=[1, 5, 10, 20], unit=otherid);  
}  

Fig. 8. Example mutations of anonymized scripts from the evaluation set. Left: CSE; lines 3, 10-12 have been inserted. This script previously produced valid output, but now raises a causal sufficiency error at the mutation point. Right: URE; the unit of randomization in line 14 has been changed from userid to otherid. We supply an annotation for otherid to indicate that it has high cardinality. This mutation raises no errors when producing conditioning sets, but it does raise an error when producing human readable output, saying that the script is not suitable for ATE.

Findings: Contrast Generation. PLANALYZER found equivalent contrasts for 78 of the 95 experiments. For 14 experiments, it produced either partial contrasts or no contrasts. In each of these cases, the desired contrast required summing over some of the variables in the program (marginalization), or more sophisticated static analysis than the tool currently supports. Since it is computationally expensive to produce every possible subset of marginalized contrasts, we consider the former to be an acceptable shortcoming of the tool. Finally, 3 experiments had issues with their human-generated contrasts (no contrasts, or ambiguous or unparsable data).

7.3 PLANALYZER Performance

All analyses were run on a MacBook Air (OSX Version 10.11.6) with a 1.6 GHz Intel Core i5 processor having 4 logical cores. The longest runtime for any analysis was approximately 3 minutes; runtime scales linearly with the number of “paths” through the program, where a path is defined according to the transformed internal representation of the input PLANOUT program and is related to the number of conditioning sets. PLANALYZER uses the Z3 SMT solver [De Moura and Bjørner 2008] to ensure that conditioning sets are satisfied and to generate treatments [Fredrikson and Jha 2014; Valiant 1979], so both the number of variables in the program and the number of paths in the internal representation could cause a blowup in runtime. We found that runtime increases linearly with the number of internal paths, but possibly exponentially with the number of variables, as depicted in Figure 9.

Findings: Performance. PLANALYZER produces meaningful contrasts that are comparable with the human-specified gold standard, automatically generating 82% of our eligible gold-standard contrasts. PLANALYZER runs in a reasonably short amount of time; likely due to PLANOUT’s generally small program sizes.
7.4 Threats to the Validity of the Evaluation

Although we were able to interview data analysts and experimental designers from several firms that have used PLANOUT, we were only able to obtain a corpus from one firm. Although experimentation is ubiquitous and consequential, it can also be contentious, leading firms to be skittish about sharing experimentation scripts.

One person annotated and verified the scripts, serving as the point of contact with Facebook to answer domain-specific questions. The point of contact at Facebook is an expert in experimental design. Due to the availability constraints of our contact, we could only verify false positives in the...
corpus, not false negatives. For the latter, we had to rely on the one annotator reading through the corpus of scripts and using their best judgment. This is a labor-intensive process. As a result, we limited the analyses we did on PlanOut-B and PlanOut-C and resorted to analyzing a sample of the available scripts for mutation testing, which was the most labor-intensive.

Ideally, our mutations would be defined on the basis of mistakes that people actually make when writing PlanOut scripts. In the absence of that, we defined the mutation operators according to mistakes that we made when learning PlanOut and our experiences working with novices in both programming and experimental design.

8 RELATED WORK

For large firms, concurrent and overlapping experiments complicate the competing goals of enforcing unbiased treatment assignment and preserving high quality experiences for end-users. Google Layers [Tang et al. 2010] is an experimentation infrastructure that addresses the massive scale of having many different parameters, some of which are not independent and therefore cannot be running concurrently. Kohavi et al. have developed heuristics and best practices for sound experimentation [Crook et al. 2009; Kohavi et al. 2013; Kohavi and Longbotham 2015; Kohavi et al. 2009]. On a smaller scale, 3X [Shin et al. 2013] and TurkServer [Parkes et al. 2012] provide open source implementations of complementary resources for experimentation: 3X uses database concepts to store experimental conditions for reproducibility, while TurkServer manages running synchronous experiments on Amazon’s Mechanical Turk.

Systems such as AutoMan [Barowy et al. 2012], SurveyMan [Tosch and Berger 2014], VoxPL [Barowy et al. 2017] and InterPoll [Livshits and Kastrinis 2014] post tasks to crowd-powered backends and address validity questions, or automate statistical results for the user. All use randomization to marginalize some measurement bias, and in some cases estimate power for extremely limited null hypothesis significance tests where the null hypothesis pertains to adversarial respondent behavior. None of these systems are equipped to conduct experiments over a more general hypothesis space.

Two major schools of thought dominate causal inference and influence PlanAlyzer’s design: the causal graphical models of Pearl [Pearl 2009] and the potential outcomes framework of Neyman and Rubin [Sekhon 2008; Splawa-Neyman et al. 1990]. Pearl’s models have served as inspiration for some probabilistic programming languages [Goodman et al. 2008; Wood et al. 2014], while many empiricists in economics [Angrist et al. 1996], political science [Gelman and King 1990], and medicine [Little and Rubin 2000] have used the potential outcomes framework [Rubin 2005]. Recent work has aimed to unify these two approaches [Hernán and Robins 2016; Morgan and Winship 2014], and to generalize the work across experiments and datasets [Bareinboim and Pearl 2015].

Prior work on automated analyses of experiments can be found in the information retrieval and knowledge discovery literature. This includes work on the automated search for relationships between data in order to generate testable hypotheses or identify natural or quasi-experiments [Jensen et al. 2008; Spangler et al. 2014; Sybrandt et al. 2017]; tools for specifying causal graphical models and inferring valid queries over them [Schein 2003]; and R packages [Grömping 2014, 2016, 2017; Tikka and Karvanen 2017] and commercial software [Sall 1989] for identifying causal effects from graphical models, and for generating randomized assignment, replication, and repeated measurements with the appropriate statistical power. None of these approaches or tools treat experiments as software, capable of having errors, or being analyzed.

Spreadsheet analysis occupies a similar problem space as programmatically defined experiments. Both domains feature highly flexible tools (spreadsheet programs and the PlanOut runtime, respectively) where the intent of authors is typically unknown, and where a notion of soundness may not be appropriate. We see parallels between our approach and the development of code smells
for spreadsheet features [Cunha et al. 2012a,b; Hermans et al. 2012, 2015] and take inspiration from the qualitative and empirical analyses of spreadsheets [Hermans and Murphy-Hill 2015; Panko 1998].

9 CONCLUSIONS

The state of the art for auditing experiments, and for generating their associated statistical analyses, is almost entirely a manual process. This is the first work that analyzes field experiments statically. We propose a new class of errors and threats unique to the programmatic specification of experimentation. We have implemented a tool that, for the most common class of experiments, automatically identifies threats and generates statistical analyses. We compare the output of PLANALYZER against human-generated analyses of real PLANOUT scripts and find that PLANALYZER produces comparable results.

A GLOSSARY

between-subjects An experiment wherein an experimental subject receives only one treatment. Treatment effect is determined at a population level, by comparing measurements across subjects.

causal inference The process of inferring whether one variable can change the outcome of another.

code smell A programming pattern that is not an error, but correlates with errors or bad practices, originally defined in Martin Fowler’s text on refactoring [Fowler 2018]. Code smells are a popular alternative to fault localization in contexts where faults may be difficult to define, or are tied to programmer intent.

conditioning set The set of variables whose values must be fixed to the same set of values for all treatments being compared; can be thought of as a constraint on contrasts.

contrast A set of variables that may be compared in order to determine the presence of a causal effect: e.g., treatment and control, A/B/C/etc.

covariate A variable that is not a treatment but could be correlated (i.e., could vary) and thus be predictive of outcome. When a covariate is a cause of both outcome and treatment, it is a confounder.

estimator A function that estimates the value of a parameter from data. For example, ATE is an estimator for causal effect.

experimental design A field of study that focuses on the process of experimentation, from an operational and procedural point of view.

online field experiment An experiment conducted over a large, heterogeneous software system, typically involving human interaction, where the treatments are variables in a software system, and the outcome variable of interest is typically a function of human behavior.

potential outcome The value of the outcome variable, had treatment assignment been a particular value. Notational convention places the treatment assignment in the superscript of the outcome: $Y^{(\text{treatmentA}=a, \text{treatmentB}=b, \text{etc.})}$.

statistical bias The difference between between the true value of a parameter of interest and the expected value of an estimator of that parameter.

subgroup analysis The estimation of treatment effect, split out according to one or more covariates; usually performed when there is heterogeneity in treatment effect.

treatment A treatment can refer to any non-empty subset of variable or collection of variables being manipulated, or the value that that variable takes on. In Figure 1, this corresponds to non-empty subsets of $\text{dynamic\_policy}$, $\text{max\_bitrate}$, either variable individually, or the values that either of these may take on, e.g. \{ $\text{dynamic\_policy}$ = false, $\text{max\_bitrate}$ = 400 \} or \{ $\text{dynamic\_policy}$ = true \}.
treatment assignment  The process of how a variable (e.g., the label or left-hand side of a variable assignment) is assigned a value. In order to estimate causal effect, treatments must either be assigned in an unbiased manner (e.g., randomly), or correct for biased assignment after data has been collected, during analysis. In the context of programmatically defined experiments, treatment assignment refers to the function that maps treatment variables to their values.

within-subjects  An experiment wherein the experimental subject receives multiple treatments, and can function as their own control.

B DISCUSSION OF PLANOUT FEATURES THAT COMPLICATE STATIC ANALYSIS

We have tagged these with a kind of impact assessment. Each feature may have high (H), medium (M), or low (L) impact on our ability to statically analyze scripts. The projected impact is in terms of correctness, performance, and maintainability of the code. In each case, we explained where and why it causes problems, how it might have an impact on future versions of the tool and extension, and how we can fix it.

Tests (H).  It would be helpful to have some kind of directive or pragma at the top of PlanOut scripts to differentiate cases where people are essentially writing configuration files (where they should really be using a different tool) versus test cases, where experiment authors might want to try out versions of the experiment deterministically first.

Unit of Analysis (H).  We assume that units of randomization are units of analysis. We return the same ATE for a userid as we do for a clusterid. One thing that would help is to explicitly set the unit of analysis at the top of a script, especially in cases where it should be different from the unit of randomization.

Types (M/L).  There are two core issues with types: (1) there are type errors in some programs, and (2) we need type information to run the SMT solver.

Regarding (1), in practice most of the type inconsistencies appear in treating one of the external operators as a string versus a container. Types are critical features for end-users. Type errors have often obfuscated more interesting errors in a program. In practice, we have corrected obvious type errors in scripts in order to get to those more interesting errors. (2) is more critical for the tool, but occurs only in certain cases. Since all built-in random assignment operators must explicitly state their possible outcomes, we only have type issues when we are really badly missing information. In these cases, it is fine for the tool to come back and say it can’t generate estimators, but that the script is not invalid. However, when we use data from guards, those often come from external operators having unknown codomains. Types would be very useful for these.

Potential solution: add some lightweight type annotations to PlanOut and add them into the serialized PlanOut as metadata.

Default null (M).  There is no undefined type in PlanOut, but there is null. This interpretation has been informed by both the Python reference implementation and the execution model as an embedded interpreter in JavaScript. It is not stated anywhere whether the containing environment (e.g., the JavaScript engine in the browser) is expected to have some a particular semantics for when the variable attempts to bind to a variable not defined in the PlanOut script. If a variable is used before definition, we interpret this to mean that the containing environment could either have a concrete value bound to the variable, or the variable could remain null. However, it is also possible that we could call it undefined behavior and throw an error. One of the complexities currently in the static analyzer is that both cases are considered. Note that we always assume that external functions used in the program are defined.

Potential Solution: Formally define and release a semantics for PlanOut as an embedded language.
**Open-world assumption/Partial evaluation/Concolic evaluation (H).** This is related to the previous issue. Since we have an open world assumption, variables that are used but not defined are not errors. **PlanAlyzer** runs “concolic” (i.e., concrete and symbolic) execution – when it can enumerate finite values for a variable, it will [Sen et al. 2005]. When it can’t, it will leave the expression symbolic, reducing and normalizing as it goes.

Since not every variable can be evaluated down to a final value, we need to store these partially-computed values. When we run the SMT solver, we may have insufficient information to concretize the remaining symbolic values. When that happens, we keep the expression symbolic and return that as the “else” case alternative for the concrete values.

**Potential Solutions:** The open-world assumption is one of the key features in promoting **PlanOut**’s adoption. The main challenge in the open-world assumption is just that evaluation can be tricky, there are edge cases, and generating possible values is time-intensive. **PlanAlyzer** currently only concretizes a variable if there is a unique solution; otherwise, it leaves the expression symbolic. There may be further optimizations and heuristics to the tool that could help with the latter issue.

**No knowledge of hypotheses (M).** A valid **PlanOut** script implies a finite set of analyses. Right now we can only produce a subset of those analysis. However, even that subset often contains estimators that are not interesting to the **PlanOut** user. Systems at e.g., Facebook, already have a mechanism for specifying “public parameters,” for more fine-grained logging and monitoring. There is currently not way to couple tracking outcome (i.e., Y), so **PlanAlyzer** completely abstracts over effects.

**Potential Solution:** **PlanAlyzer** can produce out actual SQL and/or R code if we have some additional information. This feature would be very useful to **PlanOut** end users. PlanOut could support e.g., a pragma for specifying the hypothesis we wish to test. This could serve as documentation for the program. It could also connect with some existing formalisms for databases to formally specify experiments [Spivak 2014; Spivak and Kent 2012].

**No user-defined types (L).** User-defined types, especially disjoint unions, would be a nice feature to have. Some of the functionality currently in the **PlanAlyzer** tool could then be pushed into the SMT solver.

**ACKNOWLEDGEMENTS**

This material is partially based upon work supported by the United States Air Force under Contract No, FA8750-17-C-0120. We thank Facebook, Inc. for providing the **PlanOut** corpus and additional funding. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force, nor Facebook, Inc. Additionally, we thank KDL members Kaleigh Clary, Amanda Gentzel, and Reilly Grant, as well as Sam Baxter, Javier Burroni, John Foley, Cibele Freire, and Bobby Powers for comments, feedback, and suggestions throughout the various incarnations of this work. We thank Dean Eckles for early feedback on this work. Finally, we thank our OOPSLA reviewers, whose detailed comments substantially improved the clarity of this paper.

**REFERENCES**

Douglas J Ahler, Carolyn E Roush, and Gaurav Sood. 2019. The Micro-Task Market for Lemons: Data Quality on Amazon’s Mechanical Turk. Presented at the Meeting of the Midwest Political Science Association.

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. *Compilers, Principles, Techniques*. Addison Wesley, 75 Arlington Street, Suite 300 in Boston, Mass.

---

6 e.g., https://github.com/facebook/planout/issues/119
Joshua D Angrist, Guido W Imbens, and Donald B Rubin. 1996. Identification of causal effects using instrumental variables. J. Amer. Statist. Assoc. 91, 434 (1996), 444–455.

Eytan Bakshy, Dean Eckles, and Michael S. Bernstein. 2014. Designing and Deploying Online Field Experiments. In Proceedings of the 23rd International Conference on World Wide Web (WWW ’14). ACM, New York, NY, USA, 283–292. https://doi.org/10.1145/2566486.2567967

Elias Bareinboim and Judea Pearl. 2015. Causal inference from big data: Theoretical foundations and the data-fusion problem. Technical Report. DTIC Document.

Daniel W. Barowy, Emery D. Berger, Daniel G. Goldstein, and Siddharth Suri. 2017. VoxPL: Programming with the Wisdom of the Crowd. In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI ’17). ACM, New York, NY, USA, 2347–2358. https://doi.org/10.1145/3025453.3026025

Daniel W. Barowy, Charlie Curtisinger, Emery D. Berger, and Andrew McGregor. 2012. AutoMan: A Platform for Integrating Human-based and Digital Computation. In Proceedings of the ACM International Conference on Ob- ject Oriented Programming Systems Languages and Applications (OOPSLA ’12). ACM, New York, NY, USA, 639–654. https://doi.org/10.1145/2384616.2384663

Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2008. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. Commun. ACM 51, 8 (Aug. 2008), 83–89. https://doi.org/10.1145/1378704.1378723

Thomas Crook, Brian Frasca, Ron Kohavi, and Roger Longbotham. 2009. Seven pitfalls to avoid when running controlled experiments on the web. In Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, New York, NY, USA, 1105–1114.

Jácime Cunha, Joao Paulo Fernandes, Pedro Martins, Jorge Mendes, and Joao Saraiva. 2012a. Smellsheet detective: A tool for detecting bad smells in spreadsheets. In 2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE, Piscataway, NJ, USA, 243–244.

Jácime Cunha, João P Fernandes, Hugo Ribeiro, and João Saraiva. 2012b. Towards a catalog of spreadsheet smells. In International Conference on Computational Science and Its Applications. Springer, Berlin, Germany, 202–216.

Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems (TOPLAS) 13, 4 (1991), 451–490.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Germany, 337–340.

Dorothy E Denning. 1976. A lattice model of secure information flow. Commun. ACM 19, 5 (1976), 236–243.

Ramón Estruch, Emilio Ros, Jordi Salas-Salvadó, Maria-Isabel Covas, Dolores Corella, Fernando Arós, Enrique Gómez-Gracia, Valentina Ruiz-Gutiérrez, Miquel Fiol, Jose Lapetra, et al. 2013. Primary prevention of cardiovascular disease with a Mediterranean diet. New England Journal of Medicine 368, 14 (2013), 1279–1290.

Jeanne Ferrante. Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS) 9, 3 (1987), 319–349.

Ronald Aylmer Fisher. 1936. Design of experiments. Br Med J 1, 3923 (1936), 554–554.

Martin Fowler. 2018. Refactoring: improving the design of existing code. Addison-Wesley Professional, Boston, MA, USA.

Matthew Fredrikson and Somesh Jha. 2014. Satisfiability modulo counting: A new approach for analyzing privacy properties. In Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). ACM, New York, NY, USA, 42.

Yoshihiko Futamura. 1999. Partial evaluation of computation process–an approach to a compiler-compiler. Higher-Order and Symbolic Computation 12, 4 (1999), 381–391.

Andrew Gelman and Gary King. 1990. Estimating incumbency advantage without bias. American Journal of Political Science 34, 4 (1990), 1142–1164.

Alan S Gerber and Donald P Green. 2012. Field experiments: Design, analysis, and interpretation. WW Norton, New York, NY, USA.

Noah D Goodman, Vikash K Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2008. Church: A language for generative models. In Proc. 24th Conf. Uncertainty in Artificial Intelligence (UAI). JMLR: W&CP, Online, 220–229.

Andrew D. Gordon, Thore Graepel, Nicolas Rolland, Claudio V. Russo, Johannes Borgström, and John Guiver. 2014a. Tabular: a schema-driven probabilistic programming language. In The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL ’14, San Diego, CA, USA, January 20-21, 2014, Suresh Jagannathan and Peter Sewell (Eds.). ACM, New York, NY, USA, 321–334. https://doi.org/10.1145/2535838.2535850
Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. 2014b. Probabilistic programming. In Proceedings of the on Future of Software Engineering, ACM, New York, NY, USA, 167–181.

Ulrike Grönnping. 2014. R package FrF2 for creating and analyzing fractional factorial 2-level designs. Journal of Statistical Software 56, 1 (2014), 1–56.

Ulrike Grönnping. 2016. FrF2: Fractional Factorial Designs with 2-Level Factors. http://CRAN.R-project.org/package=FrF2

Ulrike Grönnping. 2017. CRAN Task View: Design of Experiments (DoE) & Analysis of Experimental Data. http://CRAN.R-project.org/view=ExperimentalDesign

Felienne Hermans and Emerson Murphy-Hill. 2015. Enron’s spreadsheets and related emails: A dataset and analysis. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, Piscataway, NJ, USA, 7–16.

Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2012. Detecting code smells in spreadsheet formulas. In 2012 28th IEEE International Conference on Software Maintenance (ICSM). IEEE, Piscataway, NJ, USA, 409–418.

Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2015. Detecting and refactoring code smells in spreadsheet formulas. Empirical Software Engineering 20, 2 (2015), 549–575.

Miguel A Hernán and Jamie M Robins. 2016. Causal Inference. Forthcoming.

David D Jensen, Andrew S Fast, Brian J Taylor, and Marc E Maier. 2008. Automatic identification of quasi-experimental designs for discovering causal knowledge. In Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, New York, NY, USA, 372–380.

Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. 2013. Online controlled experiments at large scale. In Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, New York, NY, USA, 1168–1176.

Ron Kohavi and Roger Longbotham. 2015. Online controlled experiments and A/B tests. Encyclopedia of Machine Learning and Data Mining, C. Sammut and G. Webb, Eds.

Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M Henne. 2009. Controlled experiments on the web: survey and practical guide. Data mining and knowledge discovery 18, 1 (2009), 140–181.

S Shunmuga Krishnan and Ramesh K Sitaraman. 2013. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. IEEE/ACM Transactions on Networking (TON) 21, 6 (2013), 2001–2014.

Roderick J Little and Donald B Rubin. 2005. Causal effects in clinical and epidemiological studies via potential outcomes: concepts and analytical approaches. Annual review of public health 21, 1 (2000), 121–145.

Benjamin Livshits and George Kastrinis. 2014. Optimizing human computation to save time and money. Technical Report. Technical Report MSR-TR-2014-145, Microsoft Research.

Miguel Angel Martinez-Gonzalez and Maira Bes-Rastrollo. 2014. Dietary patterns, Mediterranean diet, and cardiovascular disease. Current opinion in lipidology 25, 1 (2014), 20–26.

Alison McCook. 2018. Errors Trigger Retraction Of Study On Mediterranean Diet’s Heart Benefits. https://www.npr.org/sections/health-shots/2018/06/13/619619302/errors-trigger-retraction-of-study-on-mediterranean-diets-heart-benefits [Online; Last accessed 5 August 2019].

Tom Minka and John Winn. 2009. Gates. In Advances in Neural Information Processing Systems. JMLR: W&CP, Online, 1073–1080.

T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. 2014. Infer.NET 2.6. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

Stephen I Morgan and Christopher Winship. 2014. Counterfactuals and causal inference. Cambridge University Press, Cambridge, UK.

Steven S Muchnick. 1997. Advanced compiler design implementation. Morgan Kaufmann, Burlington, MA, USA.

Susan A Murphy. 2003. Optimal dynamic treatment regimes. Journal of the Royal Statistical Society: Series B (Statistical Methodology) 65, 2 (2003), 331–355.

Raymond R Panko. 1998. What we know about spreadsheet errors. Journal of Organizational and End User Computing (JOEUC) 10, 2 (1998), 15–21.

David C Parkes, Andrew Mao, Yiling Chen, Krzysztof Z Gajos, Ariel Procaccia, and Haoqi Zhang. 2012. Turkserver: Enabling synchronous and longitudinal online experiments. Fourth Workshop on Human Computation (HCOMP’12).

Judea Pearl. 2009. Causality: Models, Reasoning and Inference (2nd ed.). Cambridge University Press, New York, NY, USA.

Avi Pfeffer. 2016. Practical probabilistic programming. Manning Publications Co., Shelter Island, NY, USA.

Donald B Rubin. 2005. Causal inference using potential outcomes: Design, modeling, decisions. J. Amer. Statist. Assoc. 100, 469 (2005), 322–331.

A. Sabelfeld and A. C. Myers. 2003. Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21, 1 (Jan 2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

Amr Sabry and Matthias Felleisen. 1993. Reasoning about programs in continuation-passing style. Lisp and symbolic computation 6, 3-4 (1993), 289–360.

John Sall. 1989. JMP: Design of Experiments. https://www.jmp.com/en_us/about.html

Proc. ACM Program. Lang., Vol. 3, No. OOPSLA, Article 182. Publication date: October 2019.
Richard Scheines. 2003. Causal Reasoning: Disseminating New Curricula with Online Courseware. Presented at the American Education Research Association.

Jasjeet S Sekhon. 2008. The Neyman-Rubin model of causal inference and estimation via matching methods. In The Oxford Handbook of Political Methodology. Oxford University Press, Oxford, UK, 271–299.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In ACM SIGSOFT Software Engineering Notes, Vol. 30. ACM, New York, NY, USA, 263–272. Issue 5.

William R. Shadish, Thomas D. Cook, and Donald T. Campbell. 2002. Experimental and quasi-experimental designs for generalized causal inference. Houghton Mifflin Company, Boston, MA, USA.

Alex Sherman, Philip A Lisiecki, Andy Berkheimer, and Joel Wein. 2005. ACMS: The Akamai configuration management system. In Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2. USENIX Association, Berkley, CA, USA, 245–258.

Jaeho Shin, Andreas Paepecke, and Jennifer Widom. 2013. 3X: A Data Management System for Computational Experiments (Demonstration Proposal). Technical Report. Stanford University. http://ilpubs.stanford.edu:8090/1080/

Lisa B Signorello, Joseph K McLaughlin, Loren Lipworth, Søren Friis, Henrik Toft Sørensen, and William J Blot. 2002. Confounding by indication in epidemiologic studies of commonly used analgesics. American journal of therapeutics 9, 3 (2002), 199–205.

American Cancer Society. 2019a. How Common is Breast Cancer? https://www.cancer.org/cancer/breast-cancer/about/how-common-is-breast-cancer.html [Online; Last accessed 12 August 2019.].

American Cancer Society. 2019b. Key Statistics for Breast Cancer in Men. https://www.cancer.org/cancer/breast-cancer-in-men/about/key-statistics.html [Online; Last accessed 12 August 2019.].

Scott Spangler, Angela D Wilkins, Benjamin J Bachman, Meena Nagarajan, Tajhal Dayaram, Peter Haas, Sam Regenbogen, Curtis R Pickering, Austin Comer, Jeffrey N Myers, et al. 2014. Automated hypothesis generation based on mining scientific literature. In Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, New York, NY, USA, 1877–1886.

David Spivak. 2014. Category theory for the sciences. MIT Press, Cambridge, MA, USA.

David I Spivak and Robert E Kent. 2012. Ologs: a categorical framework for knowledge representation. PLos One 7, 1 (2012), e24274.

Jerzy Splawa-Neyman, DM Dabrowska, TP Speed, et al. 1990. On the application of probability theory to agricultural experiments. Essay on principles. Section 9. Statist. Sci. 5, 4 (1990), 465–472. [Updated 1900].

Justin Sybrandt, Michael Shutman, and Ilya Safro. 2017. MOLIERE: Automatic Biomedical Hypothesis Generation System. In Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD ’17). ACM, New York, NY, USA, 1633–1642. https://doi.org/10.1145/3097983.3098057

Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic configuration management at Facebook. In Proceedings of the 25th Symposium on Operating Systems Principles. ACM, New York, NY, USA, 328–343.

Diane Tang, Ashish Agarwal, Deirdre O’Brien, and Mike Meyer. 2010. Overlapping experiment infrastructure: More, better, faster experimentation. In Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, New York, NY, USA, 17–26.

Santtu Tikka and Juha Karvanen. 2017. Identifying Causal Effects with the R Package. Journal of Statistical Software 76 (February 2017), 1–30. Issue 12.

Emma Tosch and Emery D. Berger. 2014. SurveyMan: Programming and Automatically Debugging Surveys. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA ’14). ACM, New York, NY, USA, 197–211. https://doi.org/10.1145/2660193.2660206

Leslie G Valiant. 1979. The complexity of computing the permanent. Theoretical computer science 8, 2 (1979), 189–201.

Wei Wang, David Rothschild, Sharad Goel, and Andrew Gelman. 2015. Forecasting elections with non-representative polls. International Journal of Forecasting 31, 3 (2015), 980–991.

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In Proceedings of the 17th International Conference on Artificial Intelligence and Statistics, Vol. 33. JMLR: W&CP, Online, 1024–1032.