System Design for Digital Experimentation and Explanation Generation

Item Type	Dissertation (Open Access)
Authors	Tosch, Emma
DOI	10.7275/2ctg-ad35
Rights	Attribution 4.0 International
Download date	2026-03-14 14:33:11
Item License	http://creativecommons.org/licenses/by/4.0/
Link to Item	https://hdl.handle.net/20.500.14394/18376

SYSTEM DESIGN FOR DIGITAL EXPERIMENTATION AND EXPLANATION GENERATION

A Dissertation Presented

by

EMMA TOSCH

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2020

College of Information and Computer Sciences

© Copyright by Emma Tosch 2020

All Rights Reserved

SYSTEM DESIGN FOR DIGITAL EXPERIMENTATION AND EXPLANATION GENERATION

A Dissertation Presented

by

EMMA TOSCH

Approved as to style and content by:

David D. Jensen, Co-chair

J. Eliot B. Moss, Co-chair

Eytan Bakshy, Member

Lori A. Clarke, Member

Michael Ash, Member

James Allan, Chair of Faculty
College of Information and Computer Sciences

DEDICATION

In memory of Lorraine D'Zurilla, educator.

Nolite te bastardes carborundorum.

Margaret Atwood

ACKNOWLEDGMENTS

Thanks to my advisors over the years, who have not only shaped my view of research and materially affected my ability to attend graduate school, but have who have also effectively taught me the business of higher education. In my time at University of Massachusetts Amherst, I have had fortune of working with four very different, but very experienced advisors and have learned much from all of them. Thanks to Dr. David Jensen, for welcoming me into KDL and shepherding me through my latter years in the PhD, and then into an academic position. Thanks to Dr. Eliot Moss, who was there for me during a time of need, providing technical guidance as well as institutional support. Thanks to Dr. Emery Berger, who taught me a great deal and continues to be a role model to me years after we have both moved on to other pursuits. I owe much to Dr. Lee Spector, who welcomed me into the PhD program and supported me unconditionally in my early years, and continued to be a source of positivity and inspiration after I changed research areas. Finally, thanks to Dr. Eytan Bakshy, who mentored me at Facebook and has supported me ever since. While not an advisor, Eytan has periodically fulfilled that role in my graduate education, and I will be forever grateful for his time, knowledge, and kindness.

Thanks to my committee members not already listed: Drs. Michael Ash and Lori Clarke. I am additionally grateful to Lori for the kindness and support she has shown to me personally over the years, as well as her efforts to build a more supportive environment for women in computing.

Thanks to my professional mentors, who have helped me become the researcher and educator I am today. I am especially grateful to Dr. Kathryn McKinley, for her guidance navigating academia, and her invaluable work supporting young researchers

in the field of programming languages. I also owe much to Dr. Andrew McGregor, for being not only an excellent teacher, but an excellent teaching mentor. Finally, thanks to Dr. Susan Landau for giving me some of the best advice I didn't actually take. I owe much to the titans of computing who came before, and am fortunate to have access to so many who also happen to be women!

I have also had the great fortune of working with incomparable peers. While some can forge through a PhD alone, I cannot. My peers past and present, whom I met in KDL have enriched my view of research and broadened my interests: Amanda Gentzel, Sam Witty, Akanksha Atrey, Reilly Grant, Kenta Takatsu, Justin Clarke, Andy Zane, and Purva Pruthi. Thanks to Javier Burroni, who believed in the PlanAlyzer project even when I did not. Finally, my time in KDL was especially fruitful due to the many technical conversations and collaborations with Kaleigh Clary: you were the kind of person I hoped to meet in graduate school.

Prior to my time in KDL, I had the great fortune to be in PLASMA. Thanks to Rian Shambaugh for reading my operational semantics, even if it never made it into any papers! Thanks to Dan Barowy, Charlie Curtsinger, and John Vilk for the PLASMA movies nights and making conferences fun. Thanks to Bobby Powers for the Game of Thrones theories, good coffee, and introducing me to Beyond Meat burgers. Finally, thanks to Sam Baxter for the technical conversations, but mostly also for the climbing beta.

Thanks to the many friends and colleagues over the years, who have offered their support, friendship, and wisdom: Marco Carmosino, Meagan Day, Brittany Johnson, Sara Kingsley, Alex Passos, Amee Trivedi, and Emma Strubell. Thanks to Leeanne Leclerc: you are the unsung hero of many a lost graduate student; you are a friend and a mentor and have done more for our department than most will ever know.

Thanks to Deb Bergeron, Laurie Downey, Eileen Hamel, Malaika Ross and the many staff members who have helped me and other graduate students navigate bureaucracy.

Finally, I would never be where I am not if it weren't for my PhD family: John Foley and Cibele Freire. Thanks for dragging me through classes, making me coffee (even if you hate the stuff I love), and being there for me when times were hard. I can't believe we all made it and are professors now!

ABSTRACT

SYSTEM DESIGN FOR DIGITAL EXPERIMENTATION AND EXPLANATION GENERATION

SEPTEMBER 2020

EMMA TOSCH

B.A., WELLESLEY COLLEGE

M.A., BRANDEIS UNIVERSITY

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor David D. Jensen and Professor J. Eliot B. Moss

Experimentation increasingly drives everyday decisions in modern life, as it is considered by some to be the gold standard for determining cause and effect within any system. Digital experiments have expanded the scope and frequency of experiments, which can range in complexity from classic A/B tests to contextual bandits experiments, which share features with reinforcement learning.

Although there exists a large body of prior work on estimating treatment effects using experiments, this prior work did not anticipate the new challenges and opportunities introduced by digital experimentation. Novel errors and threats to validity arise at the intersection of software and experimentation, especially when experimentation is in service of understanding humans behavior or autonomous black-box agents.

We present several novel tools for automating aspects of the experimentation-analysis pipeline. We propose, describe, and evaluate new methods for evaluating online field experimentation, automatically generating corresponding analyses of treatment effects. We then draw the connection between software testing and experimental design and argue that applying software testing techniques to a kind of autonomous agent—a deep reinforcement learning agent—to demonstrate the need for novel testing paradigms when a software stack uses learned components that may have emergent behavior. We show how our system may be used to evaluate claims made about the behavior of autonomous agents and find that some claims do not hold up under test. We then show how to produce explanations of the behavior of black-box software-defined agents interacting with white-box environments via automated experimentation. Finally, we show how an automated system can be used for exploratory data analysis, with a human in the loop, to investigate a large space of possible counterfactual explanations.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS . . . . **vi**

ABSTRACT . . . . **ix**

LIST OF TABLES . . . . **xv**

LIST OF FIGURES . . . . **xvi**

CHAPTER

1. INTRODUCTION AND PROBLEM STATEMENT **1**

2. BACKGROUND **5**

2.1 Experimental Design 5
2.2 Threats to Validity in Experiments 8
2.3 Experimentation Management Systems 9
2.4 Causal Inference 11
2.5 Automation 11

2.5.1 Automated Intervention 11
2.5.2 Automated Experimental Analyses 14
2.5.3 Automated Validation of Design 14

2.6 Program Analysis 15

2.6.1 Software Faults 16
2.6.2 Code Smells 16

2.7 Software Testing 17

2.7.1 Fault Injection and Mutation Testing 18
2.7.2 Behavioral Acceptance Testing 18

2.8 Deep Reinforcement Learning 18
2.9 Causal Graphical Models 19

3. PLANALYZER: EXPERIMENTAL PROCEDURES AS SOFTWARE 21

3.1 Classifying Errors and Threats to Validity 26

3.1.1 Randomization Failures 27
3.1.2 Treatment Assignment Failures 28
3.1.3 Causal Sufficiency Errors 29

3.2 PlanAlyzer Tool 29
3.3 Core_{PO} Intermediate Representation 32
3.4 Evaluation 36

3.4.1 Accuracy via Mutation Testing 43
3.4.2 Contrast Validation 46
3.4.3 Performance/Efficiency 46

4. TOYBOX: DESIGNING ENVIRONMENTS FOR EXPERIMENTATION 49

4.1 Testing the Behavior of Autonomous Agents 51

4.1.1 Shortcomings of Score-Based Quantitative Measures 52
4.1.2 Behavioral Testing Framework 53
4.1.3 Behavioral Acceptance Testing vs. Safe RL 55

4.2 Designing Environments for Experimentation 56

4.2.1 Features for Experimentation 59
4.2.2 Toybox Architecture 60
4.2.3 Accuracy (vs. Atari) 61
4.2.4 Efficiency (vs. ALE) 63

4.3 Case Studies 63

4.3.1 Breakout Case Study 63
4.3.2 Amidar Case Study 70
4.3.3 Space Invaders Case Study 74
4.3.4 Case Study Findings 74

5. AUTOEXP: AN EXPERIMENT MANAGEMENT SYSTEM FOR EXPLANATION 78

5.1 Outcomes and Agents 79

5.1.1 StayAlive Agent 82
5.1.2 StayAliveJitter 83

5.1.2.1 SmarterStayAlive 85

5.1.3 Target 86
5.1.4 Agent and Outcome Rationale 89

5.2 Problem Formulation and Formalization 89

5.2.1 Basic Assumptions and Definitions: Agents and Environments 90
5.2.2 Formal Definitions of Counterfactual Explanations 95
5.2.3 Novel Explanation Hierarchy 97

5.3 Major Challenges Identified 99

5.3.1 Treatments and Outcomes over Time 99
5.3.2 Variable Definition 100
5.3.3 Challenges with framing Explanation as Optimization 101

5.4 AutoExp: Automating the Search for Explanations 102

5.4.1 Selecting the Appropriate Time to Intervene 104
5.4.2 Defining Treatments and Outcomes 106

5.4.2.1 Atomic variables 106
5.4.2.2 Composite variables 107
5.4.2.3 Behaviors 113

5.5 Why we need OFAT Experimentation 115

5.5.1 Challenge: Mixing Variable Types 115
5.5.2 Challenge: Environment Dynamics 116
5.5.3 Spanning the Design Space. 118

5.6 Evaluation of AutoExp via ToyBox 119

5.6.1 Simple, Interpretable, Rule-based Agents 120

5.6.1.1 StayAlive Family 120
5.6.1.2 Target 123
5.6.1.3 Performance 124

5.6.2 Deep Agents 124

5.7 Findings and Conclusions 126

6. CONCLUSIONS AND FUTURE WORK 131

6.1 Findings 131
6.2 Conclusions 133
6.3 Future Work 133

APPENDICES

A. GLOSSARY 137
B. PLANOUT SYNTAX 140
C. DETAILED CHARACTERIZATION OF THE PLANOUT-A CORPUS 143
D. BREAKOUT SCRIPTED AGENT GRAPHS 146
E. SAMPLING CONTINUOUS ATOMIC ATTRIBUTES 149
F. AUTOEXP RESULTS FOR TARGET AGENT 150

BIBLIOGRAPHY 153

LIST OF TABLES

Table **Page**

3.1 Descriptive statistics for the corpus. 37

3.2 PlanOut corpora analysis 41

3.3 Mutations injected into sample PlanOut programs. 44

3.4 PlanAlyzer mutation analysis. 45

4.1 ToyBox feature comparison with comparable environments. 57

4.2 ToyBox vs ALE performance 65

5.1 Counterfactual explanations for the StayAlive agent 121

5.2 Counterfactual explanations for the SmarterStayAlive agent 121

5.3 String counterfactual explanations for the StayAliveJitter agent 127

5.4 Weak counterfactual explanations for the StayAliveJitter agent 128

5.5 Outcome frequencies for control and intervention states. 129

5.6 Results from a constrained search over attributes for StayAliveJitter 129

F.1 Strong counterfactual explanations for the Target agent 151

F.2 Weak counterfactual explanations for the Target agent. 152

LIST OF FIGURES

Figure **Page**

2.1 The structure of the PlanOut framework 10

2.2 A traditional RL diagram 19

2.3 A classic causal graphical model 20

3.1 An example experiment written in PlanOut 23

3.2 An example set of contrasts and condition sets 24

3.3 The PlanAlyzer system architecture. 30

3.4 DDG for example PlanOut program 31

3.5 Syntax of Core$_{PO}$. 33

3.6 Example of SMT limitations 33

3.7 Experience matrix for PlanOut authors 40

3.8 Performance data for PlanAlyzer evaluated on the PlanOut corpus. 48

4.1 Sample Breakout States 51

4.2 ToyBox testing harness. 55

4.3 Agent Experimentation vs. Observer Experimentation 58

4.4 The ToyBox environment architecture. 61

4.5 Side-by-side comparisons of screen shots from ALE and ToyBox Atari games. 62

4.6 Fidelity evaluation between Atari ALE and ToyBox 64

4.7 Breakout test visualizations . . . 67

4.8 Breakout test code snippet . . . 69

4.9 Amidar test visualizations . . . 71

4.10 Amidar test code snippet . . . 72

4.11 Space Invaders test visualization . . . 73

4.12 Space Invaders test code snippet . . . 75

4.13 Preliminary results from mixing environments . . . 77

5.1 Agent movement behavior . . . 81

5.2 Frame trace for StayAlive agent . . . 84

5.3 Diagram of the AutoExp system. . . . 103

5.4 Example setter for composite variable (type 1) . . . 109

5.5 Example setter for composite variable (type 2) . . . 110

5.6 DDG for composite variable (type 1) . . . 111

5.7 DDG for composite variable (type 2) . . . 111

5.8 Getter and setter for complex composite variable (type 2) . . . 112

5.9 Example custom sampler for composite variable (type 1) . . . 113

5.10 Factorial experiments imply a shallow DAG. . . . 116

5.11 Washed out effects . . . 116

5.12 Example counterfactual explanations that AutoExp finds . . . 123

5.13 StayAliveJitter AutoExp performance . . . 125

5.14 Target AutoExp performance . . . 130

B.1 PlanOut's concrete syntax . . . 142

D.1 StayAlive Agent DDG . . . 146

D.2 StayAliveJitter Agent DDG 146

D.3 SmarterStayAlive Agent DDG 147

D.4 Target Agent Causal Model 148

CHAPTER 1

INTRODUCTION AND PROBLEM STATEMENT

Experimentation is everywhere in software, and can be formal or informal: debugging, performance analysis, A/B tests, and most system evaluations all involve experimentation. This is because when faced with decisions about the unknown, we must often *test* candidate hypotheses: e.g., whether a changed line of code causes a program to compile when it previously did not, or whether a change in webpage layout increases time spent by a user on a website. While there are methods for inferring causation from observational data [107, 119, 102], experimentation is still considered by many to be the gold standard for inferring cause and effect.

Because experimentation is a method for evaluation, and can be applied in a broad set of circumstances, different traditions have evolved to address *threats to the validity* of conclusions we might draw from experiments. One of the major threats to the validity of experiments and their associated conclusions is variability: in some circumstances there might be too much, leading to an inability to differentiate signal from noise; in other circumstances there might be too little, leading to questions about the generalizability of conclusions.

Experimentation is used for many possible objectives: e.g., *exploring* the relationship between a set of variables, *optimizing* a set of hyper-parameters for a machine learning algorithm, testing or *confirming* the predictive capacities of statistical models, or *explaining* the mechanistic relationships between environmental variables and an outcome of interest. Software requirements for sound experimentation and the

generation of sound statistical conclusions will vary on the basis of the experimenter's objectives.

This thesis explores the intersection of experimentation and software in contexts where a software engineer or experimenter has a great deal of control over the experimentation *environment*, but seeks to understand some aspect of the *behavior* of an autonomous agent interacting with that environment. The autonomous agent may be a person or it may be black-box software (e.g., an external program interacting with the environment over an API).

Chapter 3 presents techniques for validation of *online field experiments*. Online field experiments are a type of experiment that may be unfamiliar to computer scientists, but are critical to understanding population behavior on large software systems. We treat online field experiments as a program analysis problem. We discuss the complexity of experimentation on large, heterogeneous software systems and motivate the need for automated tools that connect experimentation with its associated statistical analyses. We identify novel threats to validity at the intersection of software and experimentation and provide PlanAlyzer, a tool to aid with validation. We then perform an empirical analysis of real experiments deployed at Facebook.

Experiments themselves are the object of study in Chapter 3; we are able to focus on experiment scripts because large Internet firms such as Facebook already have experimentation infrastructure in place. These firms design modular software systems so that the components of those systems are easily interchangeable [143]. These firms typically have a fixed set of outcomes or metrics they want to optimize and a fixed (though typically very large) set of interventions available. Because experiments in these contexts are deployed over massive numbers of participants, experimenters must be judicious about what data is recorded. Recording many variables for every participant at every time step would be prohibitively expensive.

After Chapter 3, we shift to applying that kind of principled experimental design to smaller scale environments where the experimental subjects are autonomous software agents. Like human subjects, these software agents are not inspectable and so must be treated as black boxes. Unfortunately, we found that the software infrastructure required to perform analogous experiments to be *ad hoc* at best and missing at worst.

Chapter 4 makes the case for designing smaller scale software environments for experimentation. Due to differences in usage, scale, and properties of the experimental subjects, such systems require support for aspects of field experiments, as well as aspects of *simulation experiments*.

Consequently, Chapter 4 presents a software architecture for experimentation with autonomous software agents. We implement this architecture via ToyBox, a suite of clones of Atari games from the Arcade Learning Environment (ALE), a common reinforcement learning benchmark. We clone these games in order to make them more suitable for intervention, and thus more suitable for answering causal questions about agent-environment interactions. We focus our evaluation on reinforcement learning (RL) agents; RL is an artificial intelligence learning *framework* wherein an agent interacts with an environment over time. The interaction between the agent and the environment may change over time. We provide a behavioral testing framework for ToyBox and show how it can be used to falsify hypotheses about the post-training performance of static RL agents (i.e., those that *do not* change their behavior when interacting with an environment, once already trained). ToyBox has since been used by others to test claims based on *saliency maps*, a deep learning technique that has been used to explain the output of deep agents.

Since *explanation* is a critical use-case for experimentation, we would like to be able to use ToyBox to this end. However, the ToyBox testing system requires the experimenter already have a well-formed hypothesis in mind. Given ToyBox's large parameter space (not uncommon for experimentation systems), we would like

a method for searching over that space to generate candidate causes for outcomes of interest. One of the shortcomings we have observed in the deep RL literature is the tendency of authors to anthropomorphize agent behavior, imposing their own biases on what they observe during game play. Unfortunately, it can be challenging to generate hypotheses about inscrutable software. Therefore, we need a system that can test conditions that an observer may overlook.

Chapter 5 introduces a system that automatically performs experiments to search for explanations of agent behavior. AutoExp is an exploratory data analysis tool. In the course of developing AutoExp, we identified key challenges to automating experimentation over large software systems for explanation. We find —contrary to the state-of-the-art of manual experimentation—that multi-factor experimentation is not suitable. In an open-domain we must experiment one factor at a time to ensure soundness of results. Also, we discover that a major challenge unique to automatic experimentation in this domain is a representational mismatch between interventions and variables used for explanation.

We conclude this dissertation by identifying future research problems in the area of automated experimentation over black-box autonomous agents interacting with complex software systems.

CHAPTER 2

BACKGROUND

The work described in this thesis spans many fields, including experimental design, reinforcement learning, explainability, software testing, and static program analysis. In this chapter we present some background material that is relevant to our contributions. We will refer the reader back to this section when appropriate.

2.1 Experimental Design

An experiment, in the general case, can be expressed as the relationship between two sets of variables: potential effects and potential causes. We treat experiments as having a one-to-one relationship with potential effects. In this context, we refer to the value of the potential effect variable as the *outcome*,[1] which is conventionally abstracted over and referred to as Y. Potential causes are variables such that changing any one of their values could cause the value of Y to change. The possible values that potential causes may take on are called *treatments*. "Treatment" can also refer to several variables in the abstract. Suppose we are conducting a medical study comparing two dosages of a medication (e.g., `dosage=high` and `dosage=low`). Each of these dosages would be a treatment. Now suppose that we are conducting a medical study that also wants to take into account exercise. We might in this case also refer to

[1] For the purpose of this thesis, a single experiment only has one outcome of interest. While it is certainly possible to have multiple outcomes associated with a single experiment and its associated data, doing so requires the researcher account for false discovery, due to the multiple comparisons problem. If an experiment is run to collect new data for multiple outcomes, we simply treat each outcome as the result of a separate experiment.

the variables `medication` and `exercise` as treatments. When the distinction matters, we will clarify.

When a potential cause has a default value that predates the experiment, we call it the *control*. In our prior example, that might be `dosage=none` if the medication is new and `exercise=exercise`$_0$ to indicated that the subject will continue their default exercise regime.

A common theme in each chapter of this thesis is the connection between experimental design, data recording, and statistical analysis in what we refer to as the *experimentation-analysis pipeline*. Two important elements of this pipeline are *randomization*, in which experimental *units*, e.g., users, devices, machines, etc., are randomly assigned to treatments, and *contrasts*, in which some set of outcomes are compared between units in each treatment group. The former is a common element of experimental design, while the latter constitutes part of the post-experiment analysis.

One of the main challenges when running experiments is that there may be variability in the data that cannot be controlled. This variability can make estimating causal effects very difficult. Many of the innovations in experimental design are techniques for reducing variability. However, in most cases, the researcher will not be able to eliminate all variability. One way to account for variability is to perform experiments over *replicates* that form a *sample* of experimental units (e.g., user IDs, cookie IDs, content IDs, device IDs).

When a single unit receives multiple treatments, we refer to this as a *within-subjects* experiment. Within-subjects designs decrease the variability of the estimate of the outcome by holding some aspects of the subject constant between treatments. However, it is not always possible to perform within-subjects experiments. This happens when it is not possible to "reset" the unit to the state it was in prior to the experiment. This is especially prevalent in human-subjects experiments: e.g., in some medical studies, upon delivering medicine to a patient, it is no longer possible to study the

effect on that patient having received no treatment, and different groups of subjects must be used.

When the treatment effect cannot be estimated by exposing the unit to multiple treatments, we must perform a *between-subjects* experiment. A between-subjects experiment aggregates outcomes over a sample. Random assignment ensures that the effects of the variables that would have been kept constant in a within-subjects design are equally distributed across treatment groups, allowing accurate estimation of causal effect.[2]

The function that estimates causal effect for between-subjects experiments may take on 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, called the *average treatment effect* (ATE)[3] [108]. If we wish to know how Y is affected by a treatment T assigned completely at random, then ATE can be estimated by simply taking the difference of the average outcome for units assigned to $T = 1$ (treatment) and $T = 0$ (control): $Avg(Y|T = 1) - Avg(Y|T = 0)$.

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. If we do not consider these subgroups in constructing our treatment control contrast, then our analysis will 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.* This is known as the *conditional average treatment effect* (CATE). We will refer to ATE and CATE as both belonging to the same family of estimators, since ATE is a specific case

[2]Random assignment is what allows experimenters to be free from having to worry about the distribution of covariates such as age, gender or income in each each treatment group.

[3]ATE is also defined for within-subjects experiments, but in this thesis, we will only discuss it in the context of between-subjects experiments.

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.

2.2 Threats to Validity in Experiments

Three of the most serious threats to validity in experiments are sampling bias, confounding, and a lack of positivity [60]. Some instances of these threats can be identified readily in programmatically defined experiments.

Selection bias occurs when the sample chosen for treatment and analysis in the experiment differs in a systematic way from the underlying population. This can happen in the initial selection of samples of the population due to non-proportional sampling [139]; during the mapping from subsets (samples) to treatments, due to a failure of randomization; or at some point between assignment of units to treatment and the recording of results (known as *dropout* or *attrition*).

Confounding occurs when there exists a variable that causes both treatment and outcome. Confounding is a major threat to validity in observational studies, where researchers have no control over treatment assignment. Under ideal conditions, random treatment assignment ensures that there are no confounding variables. However, a fully randomized experiment is not always possible or desirable. In some cases, attempts at random assignment can be conditioned on some feature of the unit, introducing confounding.

Unlike selection bias, confounding is not necessarily irreversible. As long as the confounding variable can be measured, its effect may be factored out at analysis time, provided the analyst properly conditions on it [102].

When a treatment has some nonzero likelihood of being observed over all joint values of covariates in a data set, it has positivity. A lack of positivity can lead to statistical bias, especially during subgroup analysis. 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 digital experiments, there are some nontraditional and surprising reasons why positivity may not be achieved. We will discuss these in detail in Chapter 3.

2.3 Experimentation Management Systems

Many organizations conduct online experiments to assist decision-making; the largest firms (Facebook, Amazon, and Google) now employ some form of federated experimentation management [7, 132, 75, 25, 131]. These systems often include software components to make designing experiments easier or to automatically monitor experimental results.

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 [132] 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 [25, 75, 73, 74]. On a smaller scale, 3X [121] and TurkServer [101] 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.

Chapter 3 focuses its analyses on PlanOut, a language and framework for defining and deploying online field experiments. Figure 2.1 depicts the experimentation management systems that back PlanOut, 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

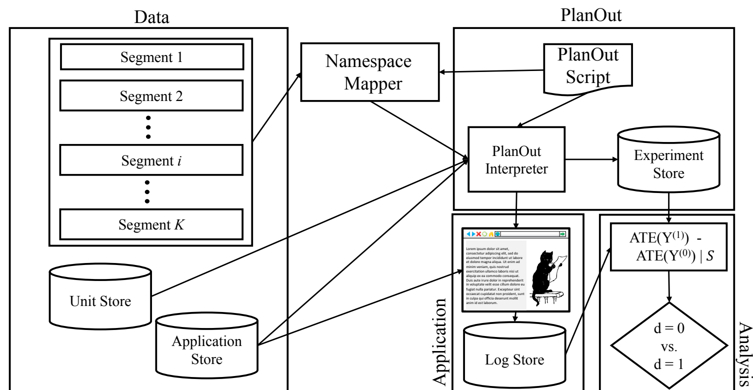


Figure 2.1: The structure of the PLANOUT framework. A *Namespace Mapper* randomly assigns a script to a randomly selected sample of the population of users. The PLANOUT *Interpreter* randomly assigns subjects to treatments, leveraging existing application infrastructure to record outcomes. The *Analysis* box uses standard notation (e.g., [46]): t represents a treatment instance from T, Y represents the outcome variable, and S represents the conditioning set.

store that contains the mappings from variables defined in PLANOUT to their values; these variables and their values are recorded in the "Experiment Store."

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. Appendix B gives PLANOUT's concrete syntax and highlights important language features.

2.4 Causal Inference

Two major schools of thought dominate causal inference and influence PlanAlyzer's design: the causal graphical models of Pearl [102] and the potential outcomes framework of Neyman and Rubin [127, 118]. Pearl's models have served as inspiration for some probabilistic programming languages [48, 141], while many empiricists in economics [3], political science [45], and medicine [82] have used the potential outcomes framework [109]. Recent work has aimed to unify these two approaches [60, 92], and to generalize the work across experiments and datasets [9].

2.5 Automation

Automation is not a new idea in the context of software-assisted experimentation. Since experimentation is a multi-stage process and is typically part of a larger pipeline, there are many opportunities for automation. We first discuss the state of the art on automating the search for viable interventions, and then discuss work on automating the downstream analyses and automating the validation of experimental designs themselves.

2.5.1 Automated Intervention

"Automated experimentation" has referred to several lines of work: most prominently, automated search for interventions that optimize a metric, and automated scientific discovery. Both lines of research of focus on automating the selection of where and how to intervene. The work in Chapter 5 is most closely related to this approach to automating experimentation.

Optimizing Experimental Designs

Experimentation can be costly: sometimes this is because, while a single run is inexpensive, the number of experiments needed to answer a query is very high.

Sometimes this is because a single run may be computationally expensive, time-consuming, or high risk.

When running a large volume of experiments is expensive, we will want to minimize the number of experiments to run, without compromising the accuracy of the estimates of effects. If we believe there exists only one cause for an outcome, or we are only interested in a single cause, then we can perform sequential one-factor-at-a-time experiments (OFAT), where a *factor* refers to the variable we intervene on and *factor level* refers to its value [37, 17]. The number of experiments we will need to perform scales linearly with the number of factors. However, if we want to find the effects of interactions between factors on outcome, we cannot run OFAT[4]

Fisher coined the term, and popularized the use, of "factorial" designs, where the experimenter performs multiple independent interventions concurrently [37]. Factorial experiments are an example of a "complex design," allowing researchers to test the effects of multiple hypotheses concurrently, as well as estimate any interaction effects.

If the objective of experimentation is to discover the settings of factors that optimize some outcome function, and if we expect there to be interactions between variables, then factorial designs are strictly more efficient than OFAT experiments: "full factorial" designs only require the number of experiments be the product of each factor level, while "fractional factorial" designs perform even fewer interventions, exploiting redundancies and symmetries in the full factorial design.

[4]We have not found a formal definition of OFAT experiments—the term has developed in contrast to the formally defined *factorial* experiment, which Fisher referred to as "complex experimental designs" [37] Therefore, either OFAT experiments are simply not defined for measuring the joint effects of two factors, or the experimenter must define new factors for each combination. OFAT experiments are typically discussed as sequential experiments; therefore, if we permit the definition of new factors and wish to identify the possibly joint effects of multiple factors, then the number of experiments we will need to run is exponential in the number of original factors: if there are n factors and each has d factor levels, this corresponds to $\sum_{s=1}^{n} \binom{n}{s} d^s \approx d^n$. Naive factorial designs are also exponential in the number of factors, but often employ algebraic structures and combinatorial properties of factors and their levels to reduce the number of runs to be polynomial in the number of factors. [38]

The family of factorial experiments work well when the factors are discrete. However, when factors are continuous, we may want to be selective about the regions they sample from. In such cases, we may want to introduce an additional constraint on the design, where we seek to e.g. minimize the variance of some property of the model [123].

When a single experiment is very expensive, researchers may want to run simulation experiments.[5] Simulation experiments are typically much cheaper to run, but they introduce epistemic uncertainty over the difference between the estimated effect and the true effect (because simulation is an approximation), while removing all aleatory uncertainty (because computer programs are deterministic).[6] Many traditional experimental designs have sought to control the variance of measurements; for simulation experiments, there is no variance, but the design space tends to be very large, since the simulations tend to have many parameters that may be continuous-valued. Thus, spanning the design space is the main challenge for simulation experiments, and is addressed via "space-filling designs" [112].

As software becomes increasingly complex—whether due to sheer lines of code, integration with (possibly black box) heterogeneous systems, or interaction with human users—researchers and software engineers are faced with an expanding number of configuration parameters to optimize. When the optimal configuration cannot be determined statically, researchers empirically search for the optimal configuration by generating and testing values, which amounts to sequential experimentation. Bayesian

---

[5]Early work in simulation experiments referred to them as "computer experiments." This was the term of art for the statistics community; due to its lack of precision in computer science, we will henceforth refer to them as "simulation experiments."

[6]Complex software such as modern machine learning stacks may have uncontrolled randomness. We will see examples of such software in Chapter 4. However, in the general case, software must generate its apparent randomness from *somewhere*; so long as that source can be intercepted and manually set, the software is deterministic. True randomness in computer behavior may arise from external events such as hardware failures or cosmic rays. However, modern hardware design has (with good reason) made such events extremely low probability, effectively rendering the output of a program with a set random seed and given input deterministic.

optimization is a popular strategy for when there are a large number of variables that may be continuous [90]. The deployment of Bayesian optimization as search via sequential experiments has come to be referred to as *adaptive experimentation* [6].

Automated Scientific Discovery

Classical AI has a long history of automated discovery, and automation of human-intensive systems. Discovery often proceeds by searching over a large space of candidate variables, programs, explanations, etc., and relies heavily on the representations used to encode domain knowledge [81]. These search systems often involve mutating candidates, which is analogous to intervention in classical experimental contexts, and in some cases has led to the automated discovery of natural laws [79, 115].

2.5.2 Automated Experimental Analyses

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 [125, 129, 63]; tools for specifying causal graphical models and inferring valid queries over them [114]; and R packages [133, 54, 53, 52] and commercial software [113] 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.

2.5.3 Automated Validation of Design

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 domain-specific language (DSL) such as PlanOut can be cumbersome to validate, and they cannot be analyzed using existing automated methods.

2.6 Program Analysis

Program analysis refers to any systematic study of a computer program. When the analyses are performed over the source code itself, we call this *static analysis*. When analyses are performed while the code is being run, we call this *dynamic analysis*.

Often program analysis is aided by the formalization of a language's syntax and semantics. This formalization serves as both a kind of documentation for others to re-implement a language, as well as the building blocks for proofs about the language, or building new analyses for the language. Language formalization can tell us how to identify undesirable program behavior, without having to run the program. Language designers often use their formalization to establish that a property of interest holds. When programs written in a language are sound (i.e., if a program passes the static analyzer, then the property of interest holds), then we can say that the program is *correct by construction*.

We will be using static program analysis to detect faults and generate statistical analyses for online experiments in Chapter 3. At present we do not have a soundness theorem (i.e., a formal definition of the property of interest we want every program to have), so our programs will not be correct by construction. Instead, we use static analysis for bug detection, and it is not guaranteed to be sound (i.e., if a bug exists in the program, the analyzer will catch it) or complete (i.e., the program catches all bugs). Some properties of programs are very challenging to define formally, and so

empirical analysis of precision and recall with human-annotated data provides an alternative route to analyzing the effectiveness of a particular program analyzer.

2.6.1 Software Faults

All software may have faults (commonly called "bugs"). Some faults may be automatically identified from the static program code. Other faults may arise due to anything from reasoning errors in the code to memory errors during runtime.

The first challenge for the programmer is to identify that a fault has occurred. Some software faults may cause a program to crash. Others simply produce incorrect output. Therefore, identifying faults must start with agreement about what correct program behavior is. Clearly a software crash is always a fault. Algorithms such as greatest common divisor (GCD) have precise definitions and notions of correctness. However, other types of program behavior may not be as clear. For example, does a function that adds two numbers execute successfully or does it raise an error when both integers and floating point numbers? Does the function % refer to remainder or modulus in a language, and what is its behavior on negative numbers? Are the results of a search engine query the most popular webpages, or ranked according to some other algorithm? In each of these cases, consensus on what constitutes a correct program is what dictates our approach to identifying whether a fault has occurred in the first place.

Once a programmer has identified the existence of a fault, they must then identify the source of the error. This process is referred to as *fault localization*.

2.6.2 Code Smells

The term "code smell" originally referred to a coding pattern that was likely to cause problems during *refactoring*, i.e., the reorganization of code. Code smells are not errors, but lead to unmaintainable code and, with high probability, future

errors. Language *linters*[7] are static analyzers that often highlight code smells, as well as statically-detectable errors and violations of stylistic conventions. Compilation warnings may include code smells.

Code smells have become a popular alternative method for fault localization in contexts such as spreadsheet analysis, where the intent of authors is typically unknown, and a notion of soundness may not be appropriate [59, 58, 26, 27].

2.7 Software Testing

In an ideal world, we would never need to programs for a property of interest because, if the program passes the static analyzer, it would always have that property by construction. However, there are many practical instances where encoding the property of interest in the static analyzer is not possible—for example, the property we want to be correct may not be verifiable statically, or the language encoding may not contain sufficient information to perform static analyses.

Software tests are a kind of *post-hoc* experiment on computer programs. Tests may be categorized along at least two axes: the size and structure of the code being tested, and the purpose of the test. For example, unit tests are over single functions or methods. End-to-end or integration tests are typically over several larger pieces of code, and are designed to ensure the integrity of a system.

Individual tests are typically run as part of a larger *test suite*. The high-level objective when constructing a test suite is to design individual tests to have sufficient coverage over the possible inputs that it will catch most important errors, based on the expected use of the program. The objective is similar to static analysis; however, the space of programs that can be tested is much larger than the space of programs that can be statically analyzed. Unfortunately, this comes as a trade-off with the

[7] Named after the C lint tool [65].

guarantees of static analysis—unlike static analysis, tests cannot prove that a program is correct; they can only show the absence of specific errors for specific inputs.

2.7.1 Fault Injection and Mutation Testing

Fault injection is a method used in some types of tests that involves knowingly modifying code to have some kind of defect. Fault injection can be used to *test* the correctness of test data when obtaining real-world input/state errors (i.e., ground-truth data) is expensive or impossible.

Mutation testing is a kind of fault injection over a program. The test program is altered at various points to test whether it is sufficiently sensitive to catch common programmer errors. *Mutation operators* are functions that alter the program in well-defined ways. Mutation operators are domain-specific, typically defined after performing an analysis of common errors that programmers actually make.

2.7.2 Behavioral Acceptance Testing

Acceptance testing typically occurs at the end of the software-development life cycle and focuses on assuring that the software conforms to end-users' expectations. Behavioral tests typically happen at the acceptance test phase and test higher-level and possibly emergent behavior of software.

2.8 Deep Reinforcement Learning

Reinforcement learning (RL) is a type of machine learning wherein an *agent* acts in an *environment* in pursuit of some goal (see Figure 2.2). The agent takes as input *state* at time t and chooses an *action* to perform at time $t+1$, receiving a real-valued reward in response to taking that action at $t+1$. The objective is to learn a mapping from state inputs to actions, called the *policy*, denoted π. Traditionally, this mapping was learned by dynamic programming and stored as a table [128].

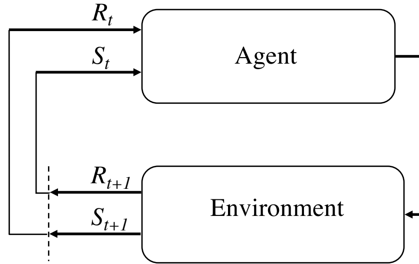


Figure 2.2: A traditional presentation of the RL paradigm. An agent interacts sequentially with an environments, potentially mutating it. The agent uses the current state in the environment and a reward to decide its next action.

Deep reinforcement learning represents the policy as a deep neural network [89, 15]. Unfortunately, these policies are complex and difficult to interpret [144]. For example, it may appear that agents learn some kind of generalized complex behavior like "avoids adversaries" in a game, but the "code" that represents adversary avoidance is encoded as numeric weights somewhere in the large matrix layers of a neural net.

Deep neural nets encode program behavior that cannot be analyzed statically, which is unfortunate because the guarantees provided by static program analysis are especially desirable for deep RL, since RL agents interact with environments as a long-running computer program.

Chapters 4 and 5 focus on testing the behavior of deep RL agents. Although RL is not featured directly in Chapter 3, there exists a powerful type of experimental design (contextual bandits) that is closely related to RL.

2.9 Causal Graphical Models

A directed graphical model is one of the primary formalisms for reasoning about causal relationships. A causal graphical model is represented by a set of nodes V and edges E: $G = \langle V, E \rangle$. The parents of $v \in V$, denoted $Pa(v)$ are the direct causes of v. Each node has associated with it a conditional probability distribution ρ_v that conditions on the values of v's parents: $\rho_v = P(v|Pa(v))$. Figure 2.3 depicts

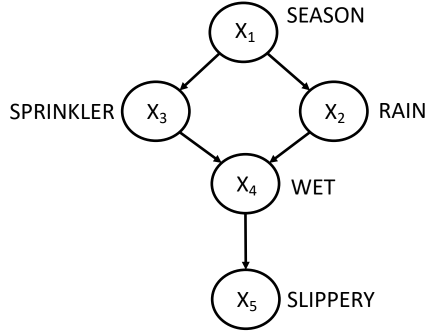


Figure 2.3: A classic causal graphical model from Pearl's classic text [102]. This graph models the causes of wet and slippery pavement. We let there be four seasons. All other variables are binary.

a classic example of a CGM, where the causal relationships being modeled are the effects of the season, a sprinkler being on or off, and whether it recently rained, on the pavement being wet and slippery. In this case, the vertices and edges are: $V = \{X_1, X_2, X_3, X_4, X_5\}$; $E = \{\langle X_1, X_2\rangle, \langle X_1, X_3\rangle, \langle X_3, X_4\rangle, \langle X_2, X_4\rangle, \langle X_4, X_5\rangle\}$. `SEASON` has no parents (i.e., $Pa(X_1) = \emptyset$), while `WET` has two parents (i.e., $Pa(X_4) = \{X_3, X_2\}$). A reasonable mapping for ρ_{X_1} would be a discrete uniform distribution (i.e., $\rho_{X_1} = 1/4$). A reasonable mapping for ρ_{X_4} might be: $\rho_{X_4} = \begin{cases} 0.01, & \neg X_2 \wedge \neg X_3 \\ 0.90, & X_2 \wedge \neg X_3 \\ 0.33, & \neg X_2 \wedge X_3 \\ 0.99, & X_2 \wedge X_3 \end{cases}$

Causal inference seeks to learn causal relationships through observational (rather than interventional, or experimental) data. Experimentation on a causal graph can be simulated with the `do` operator, which simulates intervention by setting a variable to a particular value and removing dependence on the parents of that variable.

CHAPTER 3

PLANALYZER: EXPERIMENTAL PROCEDURES AS SOFTWARE

In this chapter, we introduce the *task* of online field experimentation. We discuss the state of the art for online field experiments and makes the case for applying static program analysis for such experiments when they are written as computer programs.[1]

Example

Consider an engineering team aiming to improve the quality of video streaming for users of its mobile streaming service and thus increase its usage [77]. 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, which has never been tested. Because the company has already decided to vary bit rate, it is the bit rate *policy* that is being tested here.

[1] Work from this chapter was accepted to, and presented at OOPSLA 2019 [136]. Work from this chapter was also selected as a SIGPLAN Research Highlight and as a Research Highlight for the Communications of the ACM (CACM).

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 [95].

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. This allows them to conduct the two experiments concurrently, under similar conditions that they would not otherwise be able to control.

Experiments as Programs: Expected Behavior and Analyses

The script in Figure 3.1 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 3.2 depicts the *valid contrasts*, or the variables that may be legitimately compared. While there is a relationship between paths through a program and con-

```
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 }
```

Figure 3.1: The running example experiment, 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. Note: PlanOut programs are executed within a containing environment; `getContext`, `getUserCountry`, `getBanditWeights`, and `getBanditChoices` are all functions that exist within that environment.

trasts, 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). 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, naïvely aggregating over the high and low bit rates for the country-based experiment would not be correct.

```
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)
```

Figure 3.2: Valid contrasts and their associated conditioning sets (i.e., constraints on the analyses). The outcome of interest is the average percentage of videos watched. We generally abstract over the outcome and simply refer to it as Y.

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:

```
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 3.2. If this were a standalone experiment, random assignment would be completely broken.

As experiments evolve over time, it is possible for a variety of errors to appear, just like how bugs appear in programs. PlanAlyzer could detect these experimentation errors.

Contributions

The work presented in this chapter: (1) identifies static sources of statistical bias in programmatically defined experiments, and (2) presents methods for automatically generating the most common statistical analyses (average treatment effect) for the largest class of such experiments.

We introduce PlanAlyzer, the first tool, to our knowledge, for analyzing online experiments statically. PlanAlyzer takes PlanOut programs as input and produces three key pieces of information: (1) the entities in the environment that are actually being randomly assigned; (2) the data that are recorded for analysis; and (3) the data that may be compared when computing causal effects.

Experiments expressed as programs contain some of the same errors and threats to validity as traditional offline experiments. Some of these traditional errors take on unusual forms when defined in software. Additionally, some coding practices can lead to faults during downstream statistical analysis, highlighting the potential utility of defining code smells[2] for experiments. We introduce errors and code smells unique to programmatically defined experiments, which arise from the intersection of experiments and software.

We report PlanAlyzer's performance on a corpus of real-world PlanOut scripts provided by Facebook. Facebook also provided us with a corpus of human-generated contrasts (a key type of output necessary for estimating treatment effect). We demonstrate PlanAlyzer's effectiveness in finding major threats to validity and automatically generating key information necessary to correctly quantify causal effects.

We identify errors and code smells for programmatically defined experiments in Section 3.1. In Section 3.2 we describe the tool itself, and the intermediate

[2]Code Smells are patterns of code that suggest there might be a problem [40]. See Section 2.6.2 for more background.

representation (Section 3.3) that allows us to reason over these programs. Finally we discuss the corpus of real-world scripts and the efficacy of the tool in Section 3.4.

3.1 Classifying Errors and Threats to Validity

Experiments expressed as programs can have errors that are unique to the intersection of experimentation and software. Shadish et al. 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 [119]. Seven of these errors can be avoided when the researcher employs a *randomized experiment* that behaves as expected.[3] 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.

We characterize some static threats to internal validity based on the forms of bias in experimental design. 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 programmatically defined experiments. Some of the threats described below would be more properly considered code smells, rather than outright errors [40].

---

[3]The two remaining threats to validity that are *not* obviated by randomization are *attrition*, described in §2.2, 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.

3.1.1 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 our running example ran only the country-level design (i.e., Lines 9–17 of Figure 3.1). Then, data for users from outside the four countries of interest would be recorded in the experiment store in Figure 2.1, alongside data for users from within those countries. PlanAlyzer raises an error for this path. The fix is simple: add `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 3.4 sets the unit of randomization to be `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:

```
max_br = uniformChoice(choices=[400, 900], unit=userid);
dynamic_policy = bernoulliTrial(p=0.3, unit=max_br);
```

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 `max_br=400`, is assigned the same value of `dynamic_policy` because the input to the hash function for `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.

3.1.2 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) there are 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.

3.1.3 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:

```
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.

3.2 PlanAlyzer 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

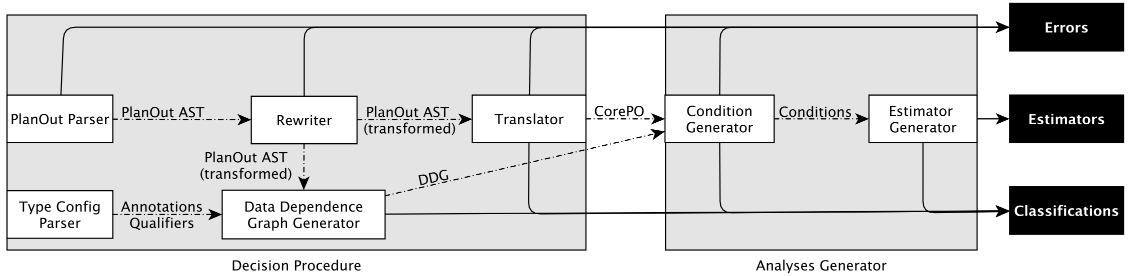


Figure 3.3: The PlanAlyzer system architecture.

to selection bias, unrecorded confounders, or positivity; and (2) generates all valid contrasts and their associated conditioning sets for the ATE estimator. Figure 3.3 depicts the system design.

Upon parsing, PlanAlyzer performs several routine program transformations. It: (1) converts variables to an identification scheme similar to static single assignment (SSA), (2) performs constant propagation, and (3) rewrites functions and relations (such as equality) in A-normal form[4] [2, 28, 111, 93]. 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 [43].

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.

PlanAlyzer also constructs a data dependence graph (DDG) [36], which it uses to propagate labels in Core$_{PO}$ (described below) and to generate contrasts for the

---

45

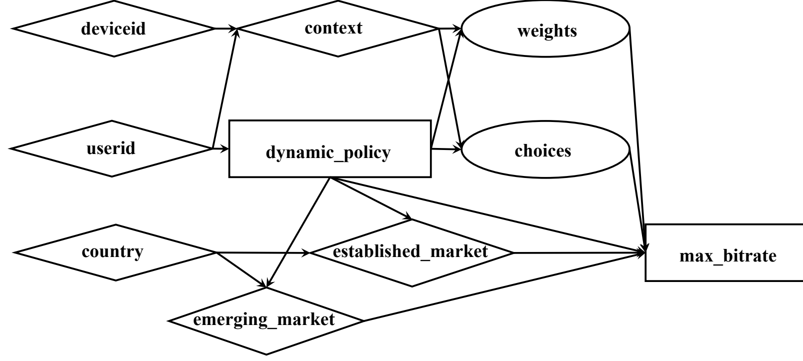


Figure 3.4: The DDG that PlanAlyzer produces for the PlanOut program in Figure 3.1. Diamonds denote non-random external variables and their dependents. Rectangles denote random variables. PlanAlyzer does not consider variables whose *existence* depends on a random variable, but have no variation in their values, to be random.

ATE estimator. 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 3.4 shows an example DDG for the example experiment of Figure 3.1.

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. Programmatically defined experiments 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 [86]. 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`).

3.3 Core$_{PO}$ Intermediate Representation

After transforming the input PlanOut program, PlanAlyzer then translates it to the Core$_{PO}$ intermediate representation and assigns special labels to variables in the Core$_{PO}$ program. Figure 3.5 depicts the syntax of Core$_{PO}$ programs.

PlanAlyzer then converts guards into assertions and uses the Z3 SMT solver to ensure that variables assigned along paths are consistent with these assertions [30]. 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 3.6 would contain the lines `fv1 = max_br > 550; assert fv1;`. PlanAlyzer would instantiate `fv1`, but would keep `max_br` abstract. When pretty printing the output as in Figure 3.2, PlanAlyzer does not show inter-

$$
\begin{aligned}
program &:= \{\pi_1, \ldots, \pi_n\} && (3.1)\\
\tau &:= \textbf{num} \mid \textbf{bool} \mid \textbf{cont} \mid \textbf{str} \mid \textbf{unknown} && (3.2)\\
d &:= \textbf{declare}\ \tau\ \ell\ \textbf{id} && (3.3)\\
n &:= \ell\ \textbf{id} = v \mid \ell\ \textbf{id} = expr \mid \textbf{assert}\ b && (3.4)\\
\pi_i &:= d\ (n+)\ \textbf{return}\ b && (3.5)\\
b &:= \textbf{id} \mid \neg b \mid b \wedge b \mid b \vee b \mid \textbf{true} \mid \textbf{false} && (3.6)\\
\ell &:= card\ rand\ tv\ corry && (3.7)\\
card &:= \textbf{high} \mid \textbf{low} && (3.8)\\
rand &:= \textbf{rand}\,(\textbf{id}, \ldots) \mid \textbf{det} && (3.9)\\
tv &:= \textbf{tv} \mid \textbf{const} && (3.10)\\
corry &:= \textbf{end} \mid \textbf{exo} && (3.11)
\end{aligned}
$$

Figure 3.5: Syntax of Core_{PO}.

```
if (max_br > 550) {
  // do something
} else {
  // do something else
}
```

Figure 3.6: PlanAlyzer leaves `max_br` abstract.

mediate values; PlanAlyzer's complete output,[6] however, contains all intermediate variables and unevaluated expressions.

PlanAlyzer uses SSA and A-normal[7] form to transform the input PlanOut program because these techniques 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 these

[6]PlanAlyzer can return output in a variety of forms; the general form is as comma-separated values that can be loaded into a database or spreadsheet program.

[7]A-normal form is a compiler optimization technique that simplifies complex expressions down to simple composable parts. In the example, it would lift the expression `getUserCountry(user=userid) == 'CA'` out of the guard and into its own variable. If the guard were a more complex expression, each subexpression would be assigned to a fresh, intermediate variable.

intermediate variables aid in, e.g., the detection of causal sufficiency errors, they make reasoning about causal inference using tools such as causal graphical models quite difficult.

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 expresses this missing information, we introduce a 4-tuple of variable labels (*rand*, *card*, *tv*, *corry*) that PlanAlyzer attempts to infer and propagate for each PlanOut program it encounters [110, 31]. 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 for externals.

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 3.1, 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.

Our target estimand is Average Treatment Effect (ATE, see Section 2.1 for background) and the related Conditional Average Treatment Effect (CATE).[8] 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 3.1 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 3.2; note that none of the contrasts compare across `max_bitrate` when `dynamic_policy` is set to true.

Covariates and Confounders (*corry*). Many experiments use features of the unit to assign treatment (a type of *covariate*; see Appendix A), which may introduce

[8]See Section 2.1 for background on ATE/CATE.

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 3.1 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.

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 3.4 is random, even though it is set to a constant in Figure 3.1. 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.

3.4 Evaluation

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

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	1o
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

Table 3.1: 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.

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 while deployed 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., $Avg(Y|T_1 = t_i^0, \ldots T_n = t_n^0) - Avg(Y|T_1 = t_i^1, \ldots T_n = t_n^1)$), 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 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 3.1 gives a more detailed description of the corpora.

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 provides. Therefore, we wanted to know: what are some characteristics of PlanOut programs that people actually write and deploy?

We found that analysts at Facebook used PlanOut in a variety of surprising ways and had coding habits that were perhaps indicative of heterogeneity in the programming experience of authors. Through conversations with Facebook, we have come to understand that most PlanOut users can be described along the two axes depicted in Figure 3.7.

Table 3.2 enumerates the errors raised by PlanAlyzer over the three corpora. Each warning does not necessarily indicate an error during deployment or analysis,

Experimental Design Experience	Programming Experience: High	Programming Experience: Low
High	I	II
Low	III	IV

Figure 3.7: Experience matrix for PlanOut authors. The horizontal axis represents programming experience or ability; the vertical axis represents experience in experimental design. We believe most authors represented in the PlanOut corpora are in quadrants I and II. PlanAlyzer's novel analyses target experiment authors in quadrants I-III and may be especially useful for authors in III, whom we believe are under-represented in the corpora. We conjecture, but cannot verify, that most of the errors PlanAlyzer flags in the corpora belong to authors in II.

due to the fact that there are pre-existing mechanisms and idiosyncratic usages of PlanOut.

PlanOut-A contains our ground truth data: all scripts were vetted by experts before deployment, with some component analyzed using ATE.

Most of the errors and threats to validity we identified in PlanOut-A were related to ambiguous semantics and type errors, modifying deployment settings within experimentation logic, using PlanOut for application configuration, mixing external calls to other experimentation systems, and using non-read-only units.[9]

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 ground truth annotation set PlanOut-A does not include any experiments that were analyzed with CATE, as the labels were collected with a system that supports only 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

[9]See Appendix C for a detailed analysis.

Output Category	PlanOut-A Scripts (566)	PlanOut-A Exps. (240)	PlanOut-B Scripts (381)	PlanOut-B Exps. (130)	PlanOut-C Scripts (493)	PlanOut-C 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	47	23				
Testing code	23	8				
Possible random assignment	41	23				
Recorded no randomization	25	11	83	23	214	35
Missed paths (tests)	4	1				
No randomization (config)	12	7				
Possible random assignment	9	3				
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

Table 3.2: 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, performed only on PlanOut-A. *We analyzed PlanOut-A with a maximum number of random variable choices of 100 and the other two corpora with the default setting of 20.

policies listed first in Fig 3.2). 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.

Our main interest in PlanOut-B is to identify whether there are experiments that could benefit from CATE analysis. Note that PlanOut found several experiments that were eligible to be analyzed with both ATE and CATE. In PlanOut-B PlanAlyzer 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 type-check.

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 3.1, the maximum number of paths in this corpus is an order of magnitude more than PlanOut-A. Figure 3.8, which we discuss in depth in Section 3.4.3, depicts how this very large number of paths contributes to the running time.

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.

3.4.1 Accuracy via Mutation Testing

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 3.3 describes the mutations we performed and the type of effect on output we expected.

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

Mutation	Description	Fault?
SAI	Sub-population 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 sub-population 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.

Table 3.3: Mutations injected into sample PlanOut programs.

(a) Mutation proportions.

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

Table 3.4: 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%.

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 selected the mutation by first generating all of the eligible AST points for all of the mutations, and then randomly select from this set. We believed this approach would lead to a more accurate representation of real programming errors. Table 3.4a 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%.

3.4.2 Contrast Validation

We 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.

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 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.

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).

3.4.3 Performance/Efficiency

We report on PlanAlyzer's performance, since its effectiveness requires accurately identifying meaningful contrasts within a reasonable amount of time.

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 running time for any

analysis was approximately 3 minutes; running time 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 [30] to ensure that conditioning sets are satisfied and to generate treatments [137, 41], so both the number of variables in the program and the number of paths in the internal representation could cause increases in running time. We found that running time increases linearly with the number of internal paths, but possibly exponentially with the number of variables, as depicted in Figure 3.8.

Findings: Performance. PlanAlyzer produces meaningful contrasts that are comparable with the human-specified ground truth, automatically generating 82% of our eligible ground truth contrasts. PlanAlyzer runs in a reasonably short amount of time, likely due to PlanOut's generally small program sizes.

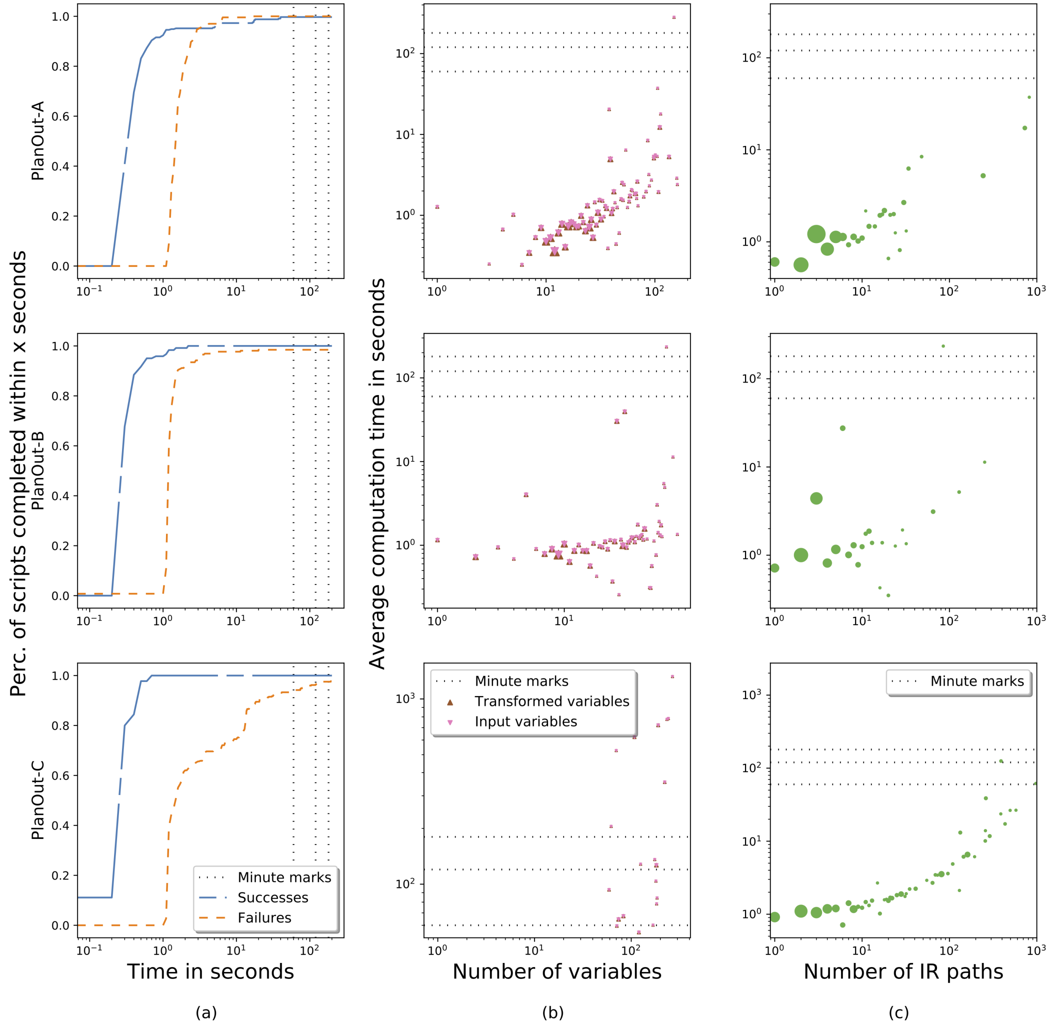


Figure 3.8: Wall-clock timing data for the PlanOut corpus. Plots in column (a) depict the empirical CDF of all scripts on a log-scale. Plots in columns (b) and (c) show the relationship between the running time and features of the PlanOut script we might expect to affect running time, on log-scale on both axes. Plots in column (b) show both the number of variables in the input PlanOut script, and the number of variables in the transformed, intermediate representation of the PlanOut program. Plots in column (c) depict the relationship between the number of paths through PlanOut programs and their running time. The times depicted in both (b) and (c) are averages over scripts satisfying the x-axis value, and the size of the points are proportional to the number of scripts used to compute that average. We chose this representation, rather than reporting error bars, because the data are not iid.

CHAPTER 4

TOYBOX: DESIGNING ENVIRONMENTS FOR EXPERIMENTATION

Execution environments such as Facebook have developed in tandem with experimentation because they must adapt to evolving user bases, regulations, accountability requirements, and other considerations. In this chapter, we argue in favor of designing a wider range of software environments with experimentation in mind. We focus on the case where uncertainty in experimentation arises not from a population of variable individuals interacting with the environment, but from repeated interaction of an autonomous software agent with the environment.[1]

Understanding how changes in the environment cause changes in the behavior of autonomous AI agents presents experimentation challenges that are similar to *within-subjects* experiments over human behavior on digital platforms. In some ways, software agents are easier to reason about, due to the lack of *carry-over effects*. On the other hand, an analyst will likely have many more causal queries for software agents. While social scientists performing experiments to understand human behavior typically have just one outcome variable in mind, analysts seeking to understand deep RL agents will have a variety of questions that require answers quickly.

Our objective in experimenting is to understanding the behavior of a specific type of software agent: an agent that is *learned* from data. We focus our analyses on learned autonomous agents backed by via deep neural networks: deep reinforcement

[1]Work from this chapter was accepted to, and presented at the 2018 NeurIPS Systems for Machine Learning Workshop [39].

learning (deep RL, see Section 2.8 for background). Deep RL agents are black-box autonomous AI programs that exhibit complex behaviors. If such agents are to be used in potentially high-stakes scenarios, stakeholders will require explanations for their behavior in certain circumstances, e.g., for compliance with legislation such as the General Data Protection Regulation (GDPR).

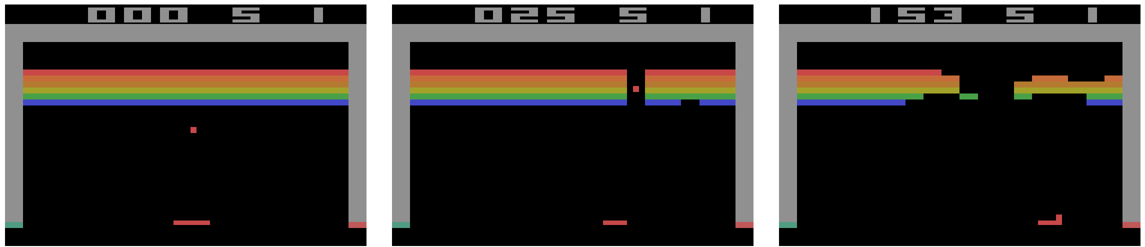
In this chapter, we first argue for the need for a behavioral testing protocol for deep RL agents. We present a behavioral testing framework that merges architectural insight from three sources: software support for reinforcement learning agents, between-subjects experimentation, and behavioral software testing. We then argue for transparency in RL environments used for research, and present an efficient and transparent re-implementation (ToyBox) of some of the popular Atari ALE environments.[2] Finally, we show how ToyBox can be used to significantly improve ease of testing for deep RL agents, and highlight recent work by others that has used ToyBox to evaluate the use of saliency maps (a common analysis tool for deep neural nets) for explaining deep agents.

Example: Breakout

The majority of our examples will use the Atari game Breakout. Breakout is the most commonly studied Atari ALE game. Its environmental dynamics are generally well-known and easily understood for humans, and skilled human players employ higher-level strategies when playing the game. Figure 4.1 gives an overview of game play and important Breakout concepts.

A deep agent that learns from images alone would not have prior knowledge of e.g., physics, and would therefore presumably need to learn to respond to such environmental dynamics in order to become a proficient player. Furthermore, published

[2] ALE is collection of games used for evaluating RL training algorithms, and consists of a wide variety of environments, including a collection of Atari 2600 games, executed from a hardware emulator [14].



(a) One of four possible starting states. (b) A "channel" or "tunnel." (c) Top row-bricks were removed via "breakout" behavior.

Figure 4.1: The game Breakout will be our primary example game. The goal in Breakout is to clear all of the bricks from the board. Figure 4.1a illustrates one of four starting states for Breakout. The ball may start from the center, left or right. At the center, it may head to the left or right. At each time step, the player may choose to move the paddle or not; the paddle may move only to the left or right. Figure 4.1b depicts the effect of the "tunneling" behavior, where a player targets a single column of the game board, clearing it so they may be able to initiate the titular "breakout" behavior, where the ball bounces against the top wall of the game space, and clears top-level bricks. Figure 4.1c illustrates the results of engaging in the breakout behavior: the top-row bricks could not have been cleared otherwise.

research has claimed that deep agents learn higher level behaviors during training [89]. However, such claims are not falsifiable due to the lack of a testing environment. Therefore, Breakout is an excellent example of the value of ToyBox.

4.1 Testing the Behavior of Autonomous Agents

The lack of testing capability for RL is concerning in its own right; after all, RL agents are software, and thus need to be tested. Testing also addresses reproducibility in deep RL, which has received increased attention due to the number of deep RL algorithms and uncertainty about their relative merits [57, 83, 64]. Nearly all of this work focuses on *how*[3] to replicate deep RL results, rather than focusing on

[3]For example, the mechanics of replication, such as seed selection, number of replicates, hyperparameter selection, etc. [16, 1].

what[4] is being replicated. The distinction between the replication mechanics and the reproduced result is especially important for environments such as Atari ALE, where anthropomorphized descriptions of agent behavior are qualitative, but also essentially subjective. Therefore, we argue for the use of precise *qualitative* descriptions of behavior in code.

4.1.1 Shortcomings of Score-Based Quantitative Measures

Quantitative evaluation of agent behavior typically consists of statistics over score and purportedly focus on assessments of the quality of the agent's training algorithm.[5] Note that achieving equivalent scores on ALE games is no easy feat; there are myriad sources of variation, and that lack of stability can manifest in dramatically different scores.[6] We argue that score is not enough to reproduce deep RL results over such environments.

Equivalent score does not entail equivalent behavior. Suppose two agents learn to play the game Pong. Both agents learn the same perfect score, but learn dramatically different methods for doing so: one agent exhibits human-like performance, while the other agent learns the no-movement exploit [83, 97].

Equivalent behavior does not entail equivalent score. Suppose two agents learn to play Breakout. Both exhibit the titular "breakout" behavior, creating tunnels in the playing field and rapidly earning reward as a result. However, one of the agents

---

[4] For example, metrics such as score or behaviors such as high-level strategies

[5] Examples include total accumulated reward (a single point estimate), and learning curves (point estimates over time). Recent work on distributional comparisons has argued that point estimates are inappropriate and lack external validity [67]. The development of novel evaluation strategies for quantitative metrics is complementary to the largely qualitative behavioral testing approach we present.

[6] Environment stochasticity, the effects of random seeds, algorithm implementation, unreported hyperparameters such as frame skip, frame blending, sticky action settings, grayscale conversion algorithm, episode resetting conditions, etc., are all examples of sources of instability and variation in deep RL problems that have been shown to have a significant effect on score [18, 20, 22, 57, 83].

is sensitive to the starting angle of the ball, frequently losing lives at one particular start orientation despite exhibiting expert behavior.

4.1.2 Behavioral Testing Framework

There is a precedence for behavioral acceptance testing in software engineering, via *behavior-driven development*. Behavior-driven development is related to the Agile[7] technique of test-driven development [12]. Behavioral tests express how a system is supposed to behave in *scenarios* or *stories*, where specific conditions trigger known *behaviors* and *outcomes* [124]. Behavioral tests also function as a kind of documentation and should be easy for an end-user of the system to read and understand. For example, in a fully realized behavioral testing system such as `behave` [35], a test of the Pong example from the previous section might include human-consumable output such as:

```
FEATURE: hit ball, robot!
  SCENARIO: arbitrary angle
     GIVEN random opponent-side ball position
      WHEN the ball is close to the agents' paddle
      THEN the agent hits the ball
```

The capitalized keywords correspond to the test case interface. Behavioral testing interfaces vary, but they share an emphasis on narrative.

Because RL agents are fundamentally tied to a complex environment, leveraging a testing paradigm that includes a sense of narrative helps us incorporate the setup of the environment into the discussion, design, and implementation of test cases.

The ToyBox testing framework is a lightweight layer on top of Python's built-in `unittest` framework. Each test case describes a *scenario*. The test harness (depicted in Figure 4.2) runs each test for some number of trials until the stopping condition

[7] "Agile" is a family of socio-technical techniques for managing software development [13].

is met. Tests are parameterized by models so that researchers may run the same test over many models, in order to compare performance. Five key methods specify behavior:

`isTrialDone` A Boolean-valued function that describes the stopping condition for the entire trial. This function is similar to the the `WHEN` keyword of `behave`. All of our Breakout tests stop when the agent misses the ball, completes the level, or times out (implemented in the shared infrastructure of the superclass).

`shouldIntervene` A Boolean-valued function that specifies when the system should apply the intervention. The intervention condition of Figure 4.10 is simple: it corresponds to the start of the trial. However, the ToyBox intervention API can be used to monitor state and create sophisticated triggers for intervention. For example, if we had an agent backed by a deep network with memory (e.g., a long short-term memory (LSTM) network [61]), we might want to test the effect of an early game intervention on a later game intervention.

`onTrialEnd` A function containing a test condition that is executed at the termination of each trial. The test condition may be confirmatory, containing an assertion, or exploratory, containing data collection. This function can be used to ensure minimal performance for each trial, e.g., shortcutting early if performance is drastically worse than random.

`onTestEnd` A function similar to `onTrialEnd`, but executed at the termination of the test harness. This function receives a list of the logged data, returned from `onTestEnd` across all trials. This function can be used for aggregated assertions across trials, e.g., to clear at least half of the bricks in 90% of trials.

`intervene` The function that performs the intervention. There may only be one intervention per test. If this function intervenes on multiple components of the state at once, they are considered a unit.

```python
def execute_test(test, model):
  trials_data = []
  for trial in range(test.trials):
      test.tick = 0
      while not test.isTrialDone():
          if test.shouldIntervene():
              test.intervene()
          action = test.selectAction(model)
          obs = test.stepEnv(action)
          test.obs = obs
          test.tick += 1
      test.obs = test.resetEnv()
      trials_data.append(test.onTrialEnd())
  test.onTestEnd(trials_data)
```

Figure 4.2: Behavioral test harness. The key methods a test may override are highlighted in blue. Methods that invoke ToyBox environment behavior are in purple, and `selectAction` is the only method involving the RL agent, where it must select an action from the current ToyBox state. Our Test harness generalizes to a variety of RL libraries, as it uses the test object itself to delegate to various back ends.

Readers familiar with statistical hypothesis testing may recognize the conditions of `onTrialEnd` as the *sharp null hypothesis* (i.e., a hypothesis that must be true for every experimental unit) and the conditions of `onTestEnd` as facilitating aggregated tests, such as *average treatment effect*.

The harness of Figure 4.2 also includes logic for the interaction of the agent with an environment; the API for this aspect is equivalent to the novel API of OpenAI Gym [19], which is an extremely popular and influential framework for encoding not only RL agents, but general time-varying agent-environment interactions (e.g., `ml-fairness-gym` [29]).

4.1.3 Behavioral Acceptance Testing vs. Safe RL

One major argument we make in favor of behavioral tests is that they are necessary to ensure that the agent does not engage in degenerate behavior. *Safe RL* is a line of work that seeks to encode constraints on agent behavior *during training*, or to *measure* constraints on agent behavior during deployment [44].

While it is certainly desirable to have provably safe agents, the researcher must know the safety constraint they want to enforce *a priori*, or else develop their safety constraint over time. Therefore, Safe RL is complementary to a behavioral testing framework when a safety constraint has been identified after training.

4.2 Designing Environments for Experimentation

Behavioral tests require an *intervenable environment* in which the researcher may change features of the potentially latent state, and not just the pixel input or agent's actions. Untestable claims about deep reinforcement learning agent behavior have made their way into journals, conference publications, and blog posts [89, 51], illustrating the need for a testing environment. Ideally, this testing environment would be ALE itself, given ALE's prominence in the rise of deep RL. ALE has several appealing qualities: it is non-trivial in that humans learn to play Atari and become more skilled with experience; it is a "real-world" environment that was not originally constructed to evaluate RL methods; and it is more complex than prior environments (e.g., GridWorld, Mountain Car).

Unfortunately, Atari ALE has a major drawback: it is effectively a black-box. Very little about individual games can be systematically altered, so ALE is poorly suited to testing how changes in the environment affect training and performance. This is a shortcoming in any environment that is effectively an emulator (e.g., OpenAI's *Sonic the Hedgehog* [96]). Even emerging efforts that re-purpose existing game engines [11, 72, 138, 68, 56] or involve procedural generation to inject variability [4, 23] stop short of convenient, programmatic intervention at the behavioral level. While many RL environments have been proposed and evaluated, few have attempted to solve our core issue; we summarize our understanding of similar environments in Table 4.1.

	Name	Year	2D	3D	OpenAI Gym	Configurable	Programmable	Public	Intervenable
Classical	Gridworlds [128]	1980-	✓		✗	Levels	Various	✓	✗
	Mountain Car [91, 128]	1990	✓		✓	✗	✗	✓	✗
	Mujoco [134] (Physics)	2012	✓	✓	✓	C API	✗	▲	▲
Compete with Humans	Infinite Mario [135]	2009	✓		✗	ProcGen	Java	▲	✗
	Atari (ALE) [14, 83]	2013	✓		✓	✗	✗	▲	✗
	Vizdoom [72]	2016		✓	✓	Levels	C++	✓	✗
	DeepMind Lab [11]	2016		✓	✓	Levels	Lua	✓	▲
	Go “AlphaGo” [122]	2016	✓		?	✗	✗	✗	✗
	StarCraft II [138]	2017		✓	✓	Levels	Protobuf	✓	▲
	Unity [68]	2018		✓	✓	Editor	C#	✓	▲
	MineCraft [56]	2019		✓	✓	XML	Java	✓	✗
	DOTA 2 [99]	2019		✓	?	?	?	✗	?
Generalization	Safety Gridworlds [80]	2017	✓		✓	Levels	Python	✓	✗
	Breakout-Variants [70]	2017	✓		✓	?	?	✗	?
	CoinRun [23]	2018	✓		✓	ProcGen	C++/Qt	✓	✗
	Sonic [96]	2018	✓		✓	Levels	✗	▲	✗
	Rogue-Gym [69]	2019	✓		✓	ProcGen	Rust	✓	✗
	Toybox	2019	✓		✓	JSON	Rust	✓	✓

(?): When environments are not available we can only speculate what support they have.

Public (▲): Some environments are under proprietary licenses or require ROM images that are not legally distributable.

Intervenable (▲): Environments with modern physics engines can modify variables and constraints on the fly; but modifying semantics will be difficult.

Table 4.1: A utility-based analysis of recent and influential environments available for reinforcement learning. Environments are increasingly selected to enable competition with humans in 3D worlds. While some environments were designed to study generalization, they often stop short of allowing intervention. Since no environments (to our knowledge) have been designed for intervention, we identify the languages in which they can be scripted and whether or not they have configurability built in, which would allow for some limited forms of testing.

Therefore, assertions about intelligent agent behavior remain untestable in Atari ALE: e.g., we cannot falsify the conjecture that agents trained on Breakout learn to build tunnels [89] or that they enter a tunneling mode [51], where the agent appears to target already-completed columns of the board, in order to score points very quickly

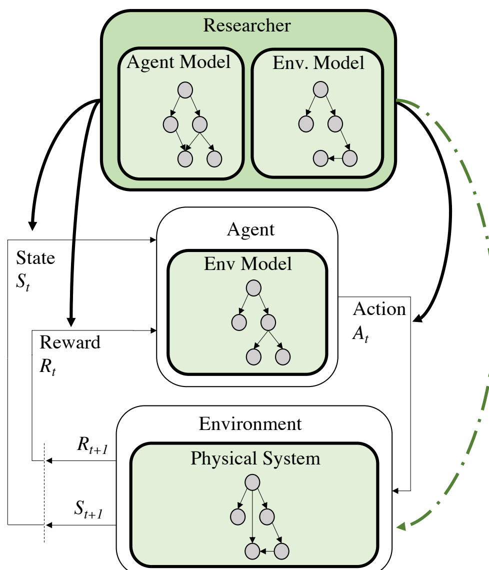


Figure 4.3: A modified RL diagram illustrating the need for intervenable environments for experimentation. All green shaded components are our contributions to the traditional RL diagram. In order to ensure that the agent is building a proper model of the environment, the researcher must have the correct model of the environment and the agent—this necessitates experimentation.

(See Figures 4.1b–4.1c). We know of no system currently that permits rapid iteration of experiments[8] to answer counterfactual questions about agent behavior.

To facilitate experimentation via intervention, we developed a suite of Atari clones within ToyBox, implementing clones for three Atari ALE games: Breakout, Amidar, and Space Invaders. Each of these games is in a different style (paddle-based, maze, and shooter), necessitating different hypotheses about behavior.

[8] Note that while RL agents are often framed as "experimenting" in their environments in service of learning their *policies* (i.e., the function that maps from the state to an action), when we seek to understand what features of an environment cause an agent to behave in a certain way, we require additional interventional capabilities. The "experimenting" that an agent performs may be related to underlying environment dynamics; the experimenting we wish to perform involves queries about *agent behavior*. Figure 4.3 illustrates the difference in objectives.

4.2.1 Features for Experimentation

As can be seen from Table 4.1, few existing environments permit arbitrary intervention on semantically-meaningful state objects. For example, while one can program "mini-games" in StarCraft II, arbitrary intervention on objects at arbitrary time points is not permitted [138]. Map files, which define the terrain for StarCraft II, have a proprietary encoding, making it a burdensome engineering task to, e.g., programmatically generate novel terrain from a distribution of terrain features.

The first key design feature we require is fast and human-understandable state serialization. Serde, a common Rust serialization library can efficiently export ToyBox state as JSON. Users can interact either with JSON directly (enabling experimentation in any programming language), or via the Python intervention library we wrote.

Our Python intervention library maps to the Rust internal objects. The intervention library also allows users to define custom interventions over collections of state variables, and automatically manages writing state to the ToyBox game state when appropriate. This allows users to programmatically manipulate arbitrary objects in ToyBox games at any time point.

The Python library also provides the ability to generate random states and to easily customize state equality comparisons. ToyBox ships with three built-in equality measures, and users can further customize state equality by simply sub-classing the `Eq` object.

This access from Python allows us to explore many aspects of the environments from within the same toolchain as the training and testing of RL agents. We took our raw access to environment internals and expanded it to be a fully-fledged testing system.

4.2.2 Toybox Architecture

Atari 2600 games were designed for human players. For ToyBox, the primary user is a reinforcement learning algorithm, and we expect machine learning researchers to be able to customize game play. To that end, we developed ToyBox to meet the following set of software requirements:

R1 ToyBox should be at least as efficient as the emulated version of the game that backs ALE. Since reinforcement learning algorithms require millions of frames of training data, we must be able to simulate and render millions of frames in reasonable time in order to enable efficient use of computation resources for learning.[9]

R2 ToyBox should have useful environments for core RL research to allow writing of tests that will advance knowledge of modern algorithms. For our Atari games, this means we need some level of fidelity to the original games to explore behavioral hypotheses.

R3 ToyBox should provide for data-driven user customization. Changing the bricks in Breakout, the board in Amidar, or the alien configuration in Space Invaders should not require re-compilation of the core game code, nor should it require the ability to write Rust code.

R4 ToyBox should be accessible through the ubiquitous (see Table 4.1) OpenAI Gym API, which is written in Python. Furthermore, ToyBox should be usable as a drop-in replacement for the analogous ALE environment.

Figure 4.4 depicts the ToyBox architecture. The game logic is written in Rust. Every game implements two core Rust structs ("classess"): `Config` and `State`. The

---

[9]Our original prototype of Amidar was written in Python and was not as efficient as Stella—the Atari hardware emulator inside Atari ALE—until we spent a few hours profiling and hard-coding more details into the code; conflicting with **R3**.

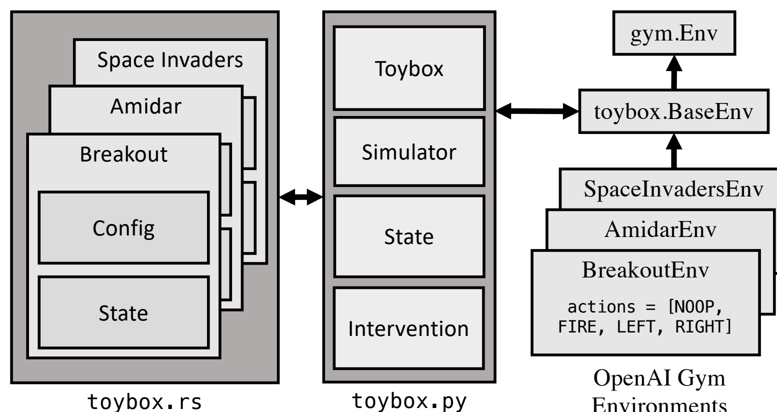


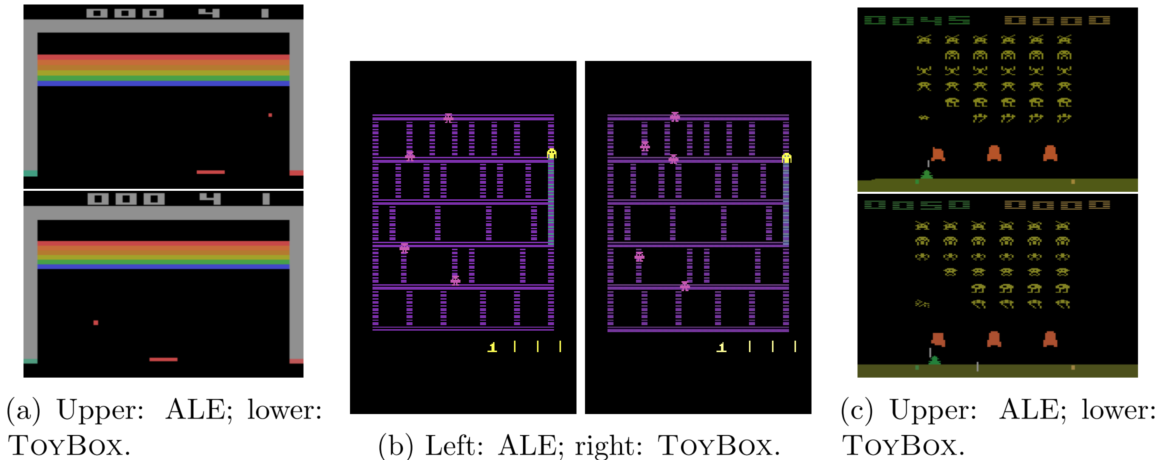
Figure 4.4: The ToyBox environment architecture. Solid arrows indicate the direction of data sharing. On the right, these data-sharing arrows also indicate inheritance (i.e., point to super classes).

`Config` struct contains data that we would generally expect to be initialized only at the start of an episode (i.e., a game, which may include multiple lives). The `State` struct contains data that may change between frames. At any point during execution, a ToyBox game can be paused, its state exported and modified, and resumed with the new state. A change to `Config` on the other hand, typically requires starting a new game to take effect.

The Python library interacts with the Rust code via a `Simulator` (which exports `Config`) and a `State` object. Both `Config` and `State` can be exported and modified as JSON, but we provide a higher-level `Intervention` module to aid in writing tests.

4.2.3 Accuracy (vs. Atari)

Three factors prevent exact replication of games: (1) Atari 2600 game source code is not available, (2) there are no formal specifications and few informal specifications of games, and (3) inferring arbitrarily complex programs from data is extremely challenging [106].



(a) Upper: ALE; lower: ToyBox.

(b) Left: ALE; right: ToyBox.

(c) Upper: ALE; lower: ToyBox.

Figure 4.5: Side-by-side comparisons of screen shots from ALE and ToyBox Atari games. Each game represents a different deterministic action trace, but traces are the same between ALE and ToyBox. ToyBox implementations of Breakout and Space Invaders have nondeterministic elements. Amidar is deterministic in both ALE and ToyBox. Due to idiosyncrasies at the start of ALE Amidar game play, the frames are not from identical points in the action trace; frames for Breakout and Space Invaders are. Note that for Amidar, ALE appears to be missing an enemy, which is due to Atari's rendering only a subset of sprites in each frame due to computational constraints.

Human players rely on unique problem-solving capabilities that deep RL agents have not yet achieved, while deep networks are undeterred by the kind of noise that can confuse humans [130, 34].

To measure whether ToyBox was sufficiently similar to Atari, from the perspective of deep agents, we trained a large collection of agents on each environment. We used three off-the-shelf implementations of training algorithms with default parameter settings for 5x10^7 steps from OpenAI Baselines [32]: `a2c` [88], `acktr` [142], and `ppo2` [117]. Due to issues with variability across agents and environments [57, 22, 66], we trained ten replicates for each of these training algorithms, differentiated by their random seed. Since there are various other uncontrolled sources of randomness, we evaluated each of these thirty agents per game using thirty unique random seeds. Figure 4.6 depicts our results: we find that agents achieve sufficiently similar performance in

each analogous environment, and have roughly equivalent rankings (idiosyncrasies are discussed in the Figure 4.6 caption). Therefore, we expect ToyBox games to be of comparable difficulty to their analogous Atari games.

Threats to Validity: Score

Although, as discussed earlier in this chapter, score is sometimes a poor window into these complex environments, it is our primary metric for ascertaining whether our environments have reasonable fidelity. More appropriate distributional comparisons of agent performance are still constrained by the limited information that more complex RL environments are capable of exporting [66, 67]. Furthermore, since RL performance metrics are an active area of research, experimenters may need to iterate on metrics or outcomes of interest, in contrast with the relatively stable metrics of the Internet firms of Chapter 3.

4.2.4 Efficiency (vs. ALE)

We report ToyBox efficiency in Table 4.2. Note that ToyBox permits researchers to process games entirely in grayscale and thus achieve substantial additional performance gains. However, since this is not a feature offered in ALE, we compared only against the ToyBox RGB(A) rendering.

4.3 Case Studies

We provide case studies for experimentation on each of the three games in the ToyBox system.

4.3.1 Breakout Case Study

For Breakout, we test three different expectations of agent behavior: whether the agent can clear the board (brick elimination), whether the agent can handle any starting angle (start angle invariance), and whether the agent has learned a correlation

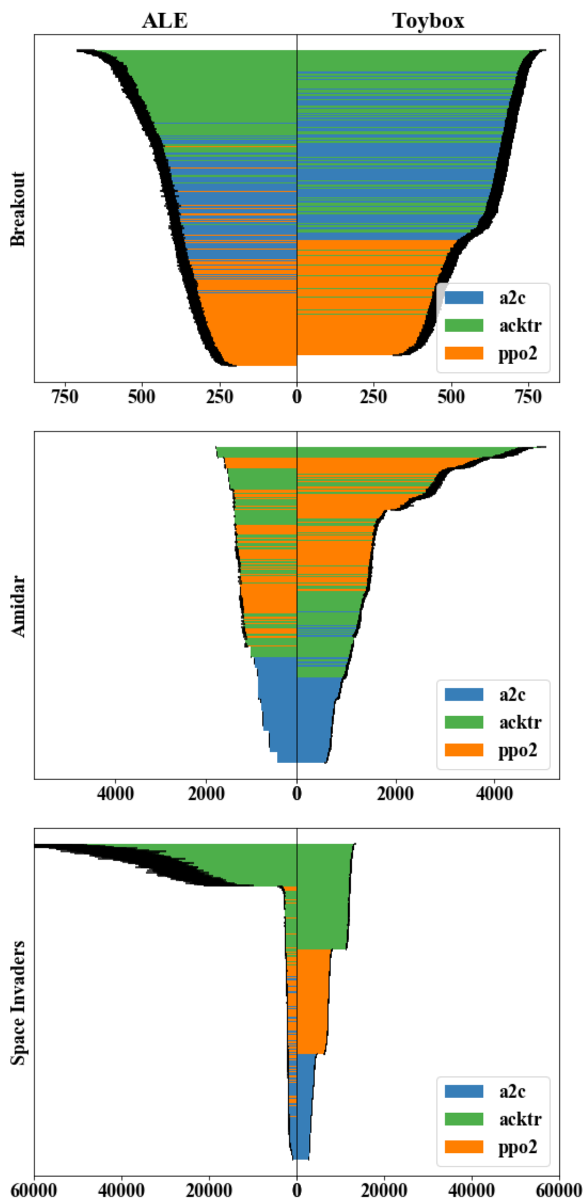


Figure 4.6: Fidelity evaluation between Atari ALE and ToyBox. Bar plots, ranked by score, across 300 model replicates per model, per game, with standard error. Each horizontal bar is the average performance for a trained model, evaluated over 30 games. Striations should be similar in both environments. **Breakout**: One training seed for `acktr` had poor performance in every trial (i.e., all average scores below 2). **Amidar**: The lack of within-game variation is reflected in the short error bars and similar rankings between back ends. **Space Invaders**: Due to the range of scores achieved, we evaluated Space Invaders for only one life.

		Raw kFPS	Gym kFPS
Breakout	ALE	52 (1.3)	3.4 (0.065)
	ToyBox	230 (5.4)	7.2 (0.23)
Amidar	ALE	61 (2.9)	3.0 (0.083)
	ToyBox	250 (2.3)	6.0 (0.112)
Space Invaders	ALE	55 (1.3)	3.9 (0.072)
	ToyBox	120 (3.4)	5.2 (0.082)

Table 4.2: ToyBox vs ALE performance: measured in thousands of frames per second (kFPS) on a MacBook Air (OSX Version 10.13.6) with a 1.6 GHz Intel Core i5 processor having 4 logical cores. Rates are averaged over 30 trials of 1x10^4 steps and reported to two significant digits, with standard error. We consistently observed an approximately 95% slowdown when interacting with both ALE (C++) and ToyBox (Rust) via OpenAI Gym. All benchmarks are run from CPython 3.5 and include FFI overhead (via `atari-py` for ALE).

between channels and the breakout behavior. We provide a sample code snippet using the ToyBox testing framework for the third test in Figure 4.8.

Brick Elimination. To win Breakout, an agent must eliminate every brick. A plausible state for such an agent might be after a “death” or lost ball, where there is one brick remaining. We might expect an agent that understands the physics of Breakout to remove the last brick quickly. A reasonable test for any successful agent might be one that measures whether an agent can remove the last brick in some acceptable percentage of games, within a pre-defined number of paddle hits, etc.

Figures 4.7a and 4.7d depict the reciprocal of the median number of steps taken to eliminate a given brick as a heat-map, with yellow indicating very few steps (median of 20) and red indicating very many steps (median of 400). For example, the agent typically targeted the paddle so that the ball would hit straight up when it was launched from a start state.

We suspect that the large difference we see between brick elimination scenarios is that improvements in paddle aim are not evenly distributed spatially. The same

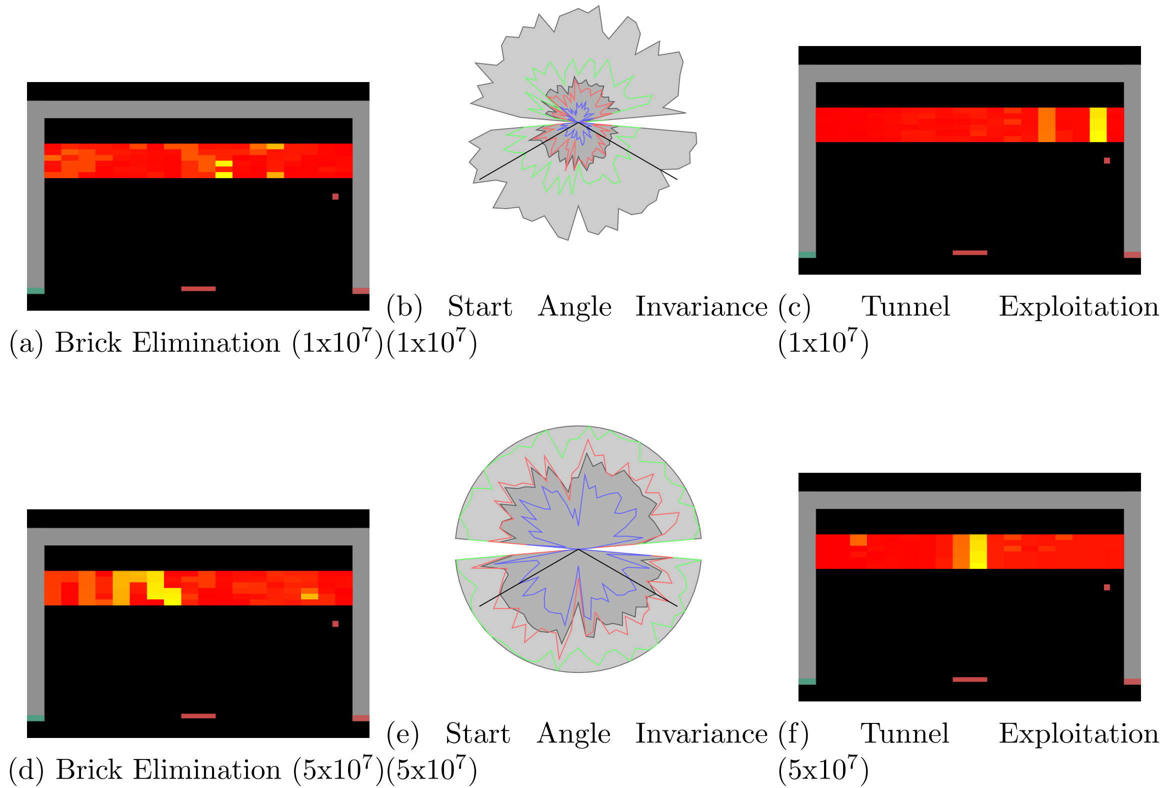
temporal range is visible on both charts, (maximums and minimums have not shifted) but there are many more bricks that are eliminated quickly for the agent with more training steps.

We had expected to see, for example, symmetry across the bricks. However, there is no obvious pattern here, suggesting that the agents are not particularly good at aiming at final, lingering bricks. In the future, we hope to explore whether these latencies are related to the frequency of their occurrence as the last brick in training data. Some alternative hypotheses for explaining the agent's behavior are that it has memorized a sequence of actions, or that it has learned only to survive and is not targeting the ball at all.

Start Angle Invariance. Our next behavioral test comes from observation of the game. Over time, the ball bounces through many different angles, and it is plausible (though maybe not strictly possible in the original version based on fixed-point math) to see all possible angles of movement of the ball. We therefore expect an agent to be able to hit the ball "thrown" at any start angle and to resume play.

Since Breakout operates over a finite set of angles, we did not expect as much success as this test indicated. Modifying a start state to change the initial launch angle of the ball led to a single life of game play, and a total score. Over 30 trials per agent per angle, we are able to visualize the mean (dark gray area), max (light gray area), median (red), 25th (blue) and 75th (green) percentiles on a polar plot in Figures 4.7b and 4.7e.

Both plots display failure to achieve any score with horizontal ball angles: since Breakout has no gravity, balls simply bounce horizontally forever, never hitting any bricks or threatening the paddle.



(a) Brick Elimination (1x10^7)

(b) Start Angle Invariance (1x10^7)

(c) Tunnel Exploitation (1x10^7)

(d) Brick Elimination (5x10^7)

(e) Start Angle Invariance (5x10^7)

(f) Tunnel Exploitation (5x10^7)

Figure 4.7: Test Data Visualizations: Color of bricks in Figures 4.7a, 4.7c, 4.7d, and 4.7f indicate the median number of steps required to clear the particular brick in that test: Bright yellow represents fewer steps and dark red indicate many more steps. In Figures 4.7b and 4.7e, the black lines indicate the starting angles seen during training, the light gray area the maximum score achieved from this starting angle, and the dark gray area represents the mean score achieved across trials. All tests were run with a 4 minute timeout.

The agents under test were remarkably resilient to starting angles. While displaying a large amount of variance,[10] the maximum score achieved from each angle was much higher, suggesting that an agent can be successful even with balls traveling at angles it may never have observed in training. The agent trained for 50 million steps was much more robust – out of the 30 trials, there was at least one trial that completed the full level for each starting angle.

Looking closer at the mean, we can see that the agent trained for 5×10^7 steps had more difficulty with vertical angles. When we observed this behavior, the agent would sometimes keep the ball aligned perfectly in the center of the board, hitting it precisely in the center of the paddle, over and over, and therefore failing to make progress. This is a fascinating behavior that is entirely unlike the kind of behavior we would expect from human players.

Tunnel Exploitation. The highest reward in Breakout comes from "digging" a tunnel through the wall of bricks and then bouncing the ball through the hole and onto the ceiling. When the ball is trapped above the bricks, it will result in many bricks being removed very quickly, and a higher near-term reward than usual. Mnih et al., speculated that a high-performing agent was learning this behavior [89].

One way to test whether an agent "knows" to exploit a tunnel is to give it a board with a nearly built tunnel, save for a single brick, and test whether the agent can aim at that single brick within a given amount of time. Just as in the test presented for **Brick Elimination**, we can generate a test for every brick.

In Fig. 4.7c and 4.7f, the value for each brick is the reciprocal of the median number of time steps before that brick was removed. The values range from 17 timesteps (bright-yellow) to 400 timesteps (dark red).

---

[10] Although outside the scope of this work, there is a recent growing interest in the role of variability during evaluation in machine learning, as well as RL specifically [24, 22, 66]. We suspect the higher variance observed during evaluation of the polar start invariance test is an instance of this phenomenon.

```python
class EZChannel(BreakoutToyboxTestBase):

  def shouldIntervene(self, obj=None):
    return self.tick == 0

  def intervene(self, obj):
    # obj refers to the particular brick on which we are intervening
    with BreakoutIntervention(self.getToybox()) as intervention:
      game = intervention.game
      game.lives = 1
      col = obj.col
      row = obj.row
      intervention.add_channel(col)
      # Now get the brick associated with the current target,
      # and make it live again
      _, brick = intervention.find_brick(lambda b: b.col == col and b.row == row)
      brick.alive = True

  def test_ezchannel_ppo2(self):
    seed = 8675309
    path = 'models/BreakoutToyboxNoFrameskip-v4.regress.model'
    bricks = BreakoutIntervention(self.getToybox()).game.bricks
    with tf.Session(graph=tf.Graph()):
      model = oai.getModel(self.env, 'ppo2', seed, path)
      self.runTest(model, collection=bricks)
```

Figure 4.8: Code snippet from our Breakout tunnelling test. Not shown: `onTrialEnd` logging data and `onTestEnd`.

In order to satisfy our requirement, we would expect that an agent could exploit a tunnel quickly anywhere on the board, however we find that agents hit the ball to predictable locations regardless of the board configuration. We feel comfortable rejecting the hypothesis behind our requirement. We can see as well that specific agents learn to hit the ball to specific locations – the `PPO2` model trained for 10 million steps prefers to send the ball to the right side of the screen, and the agent trained for 50 million steps prefers to hit the ball to the center column.

4.3.2 Amidar Case Study

One of the other games supported by ToyBox is Amidar[11], a maze game reminiscent of the more popular "Pacman" where enemies must be avoided and exploring the board leads to earned points.

Suppose we would like to test whether an agent has learned to avoid adversarial elements of a game: e.g., the enemies in Amidar. To test this, we might drop the agent around the corner from an enemy, or position the enemies to "gang up" on the player, forcing the agent to move in a particular direction.

This kind of intervention is meaningful only if enemy position is a function of current location. Observation led us to conclude that enemies move in fixed loops, likely implemented as lookup tables. This contrasts with "Amidar movement," which is believed to dictate enemy behavior.[12] The protocol matters for intervention because, for a lookup table, moving enemies will have no effect: enemies will simply "teleport" to the next location in the lookup table.

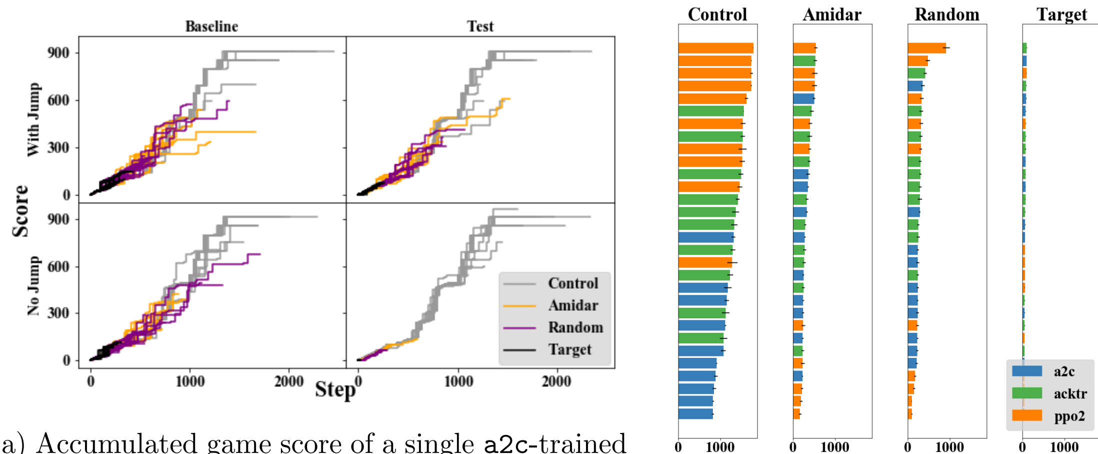
The upper left plot in Fig. 4.9a shows a baseline test for how an individual trained agent performs under each of four different enemy movement protocols: (1) a lookup table, on which the model was trained, (2) the "Amidar movement" protocol, (3) a random protocol, where at each junction the enemy chooses a random direction, and (4) an adversarial protocol, where enemies explore via random turns until the player is within line of sight, at which time they move toward the player's location. Note that, since enemies start far away from the player, the agent can (and does) easily make progress at the start of the game, regardless of enemy position. However, as the game progresses, the enemies close in and there are fewer opportunities for rewards.

---

[11] `https://en.wikipedia.org/wiki/Amidar`

[12] An enemy moves with a diagonal velocity, flipping the vertical direction when encountering the top or bottom of the board and horizontal direction when encountering the left or right edge (`https://en.wikipedia.org/wiki/Amidar`)

The upper right plot in Fig. 4.9a shows a test in which enemies "gang up" on the player: the enemies' start position is modified to be close to the player. We were at first surprised to see how well the agent did; however, upon examination, we found that the agent was using up the jump button, which allows the player to bypass enemies, at the beginning of the game. The lower half of Fig. 4.9a depicts the results of running the baseline and test for no jumps: while the baseline performs similarly, the player dies quickly for all non-lookup table enemy protocols.



(a) Accumulated game score of a single `a2c`-trained agent.

(b) Ranking of 30 model replicates.

Figure 4.9: **Amidar case study.** for four enemy movement protocols ("Control" is a lookup table, "Amidar" is the "Amidar movement," "Random" enemies move in a random direction at every junction, and "Target" causes enemies to pursue the player when it is in line of sight). **(a)**: *Upper left*: baseline performance of the agent on each of the four protocols. *Lower left*: baseline performance of the agent on each protocol without the ability to jump over enemies. *Upper right*: "Ganging up" test, where all agents start close to the player. *Lower right*: "Ganging up" test with no jump. **(b)**: Score ranking of 30 model replicates for the baseline condition with jumps (i.e., the upper left corner of the left graph) for each movement protocol.

```
1  class GangUpNoJumpTest(AmidarToyboxTest):
2
3    def shouldIntervene(self):
4        return self.tick == 0
5
6    def onTrialEnd(self):
7        with ami.AmidarIntervention(self.getToybox()) as ai:
8            unpt = ai.num_tiles_unpainted()
9            pt = ai.num_tiles_painted()
10           score = ai.get_score()
11       # Gang up is harder; try to paint half the board?
12       self.assertGreaterEqual(painted, unpainted)
13       return {'painted': pt, 'unpainted': unpt, 'score': score}
14
15   def onTestEnd(self, trials_data):
16       record(trials_data, '/path/to/log.csv')
17
18   def intervene(self):
19       with ami.AmidarIntervention(self.getToybox()) as ai:
20           # don't let it become invincible to escape!
21           ai.set_n_jumps(0)
22           for eid in ai.get_enemy_ids():
23               ai.set_enemy_protocol(eid, 'EnemyTargetPlayer')
24
25   def test_scenario_ppo2(self):
26       seed = 42
27       model = oai_test.getPPO2(self.env, seed, 'path/to/model')
28       tb_test.runTest(self, model)
```

Figure 4.10: Code snippet for a version of one of our 16 test scenarios from our Amidar Protocol test. This test frequently fails at the end of the first trial.

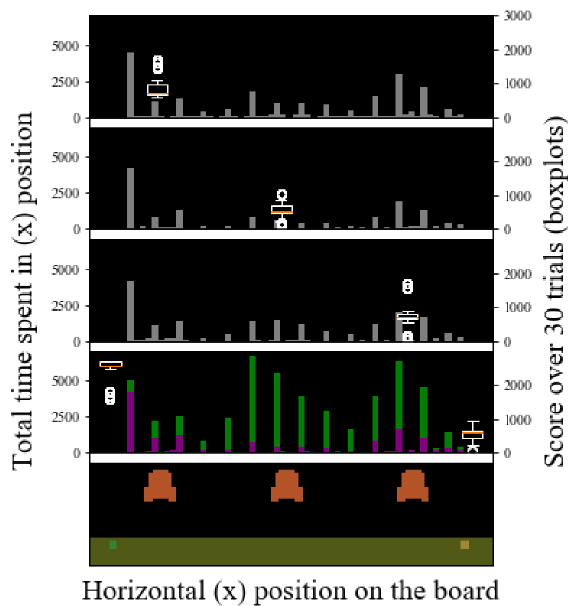


Figure 4.11: **Space Invaders case study.** The top three bands depict the tests of the solitary first, second, and third shields respectively. The bars depict the total number of steps spent in the corresponding horizontal location, while the box-plots depict the scores. The fourth band shows the behavior for all shields present (green; score box-plots on the left) and no shields present (purple; score box-plots on the right). The bottom band shows the Space Invaders terrain and default positions of the shields.

4.3.3 Space Invaders Case Study

Space Invaders[13] is a famous "shooter" game that was implemented in ToyBox to round out the genres supported by our intervenable games.

In Space Invaders, the player can seek refuge under three shields from the frontier of alien ships shooting down. We ran a test to see whether removing two of the three shields would cause an agent to use the remaining one more often. We also ran two baseline comparisons for a fixed amount of time: one where all shields are present (the default setting) and one where no shields were present.

Figure 4.11 shows the results under test. Since score provides an incomplete picture of agent behavior, we also tracked the agent's location (a simple query in ToyBox). We observe that the player does not appear to change its preferred locations under any of the tests.

A Note on Negative Results: Space Invaders, as we have implemented it, has turned out to be a fairly uninteresting game. Randomly selecting from the trimmed action set that OpenAI Gym allows can lead to fairly good performance. Furthermore, our implementation, which included both random and adversarial enemy behavior, led the agent's behavior to be invariant to randomness in enemy behavior.

4.3.4 Case Study Findings

We have shown a range of interventions and queries possible with ToyBox, all of which would be impossible to conduct using ALE. The interventions we demonstrated were designed to demonstrate the power of ToyBox's design and implementation, rather than to satisfy any particular RL research agenda. We were able to rapidly iterate on all of our experiments due to ToyBox's fast performance and its simple API for editing state. In addition to highlighting ToyBox's capacity for evaluating

[13] `https://en.wikipedia.org/wiki/Space_Invaders`

```
class ShieldXs(SpaceInvadersToyboxTestBase):

    def shouldIntervene(self, obj=None):
      if self.tick == 0:
        return True
      else:
        self.xs_observed[self.getToybox().query_state_json('ship_x')] += 1
        return False

    def intervene(self, obj=None):
      s1, s2, s3 = obj
      with si.SpaceInvadersIntervention(self.getToybox()) as intervention:
        game = intervention.game
        to_keep = []
        if s1:
          to_keep.append(game.shields[0])
        if s2:
          to_keep.append(game.shields[1])
        if s3:
          to_keep.append(game.shields[2])
        game.shields.clear()
        game.shields.extend(to_keep)

    def test_shieldxs_ppo2(self):
      model = oai.getModel(self.env, 'ppo2', seed, path)
      shield_configs = [(1,0,0), (0,1,0), (0,0,1), (0,0,0), (1,1,1)]
      self.xs_observed = Counter()
      self.runTest(model, collection=shield_configs)
```

Figure 4.12: Code snippet from our Space Invaders shield usage test. Not shown: `onTrialEnd` and `onTestEnd`

a single agent, we have shown how ToyBox may be used to evaluate models, by comparing the post-training performance ranking under test.

ToyBox has been used by others to evaluate previously untestable claims about saliency maps [5]. ToyBox enabled the authors to perform complex state manipulations, e.g., mirror the brick configuration, which could only have been done via error-prone and tedious pixel manipulation without ToyBox. Furthermore, any downstream effects due to the state manipulation coupled with environment dynamics are completely undetectable without a system such as ToyBox.

Finally, there are myriad ways ToyBox may be used that are currently under-explored. Figure 4.13 depicts the polar starts test for an agent that was trained on a mixture of ToyBox and ALE environments. Of note is that, while agents trained on ALE Amidar and Space Invaders *can* score points in ToyBox and vice versa, the same environment swap does not work for Breakout. We ran a preliminary test to see whether an agent trained on a mixture of environments could perform well. While we did not train the agent long enough (only 10 million steps), we saw potential for using ToyBox to explore training for a mixture of “real” (ALE) and simulation environments.

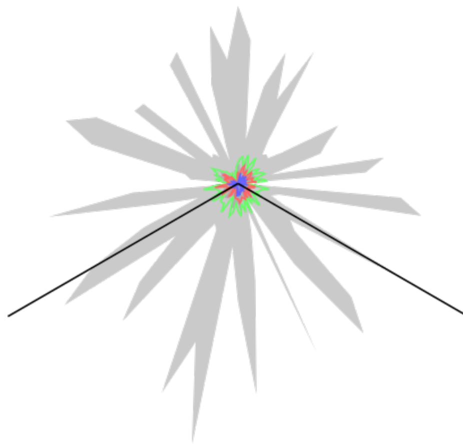


Figure 4.13: Preliminary results from training an agent on a mixture of ToyBox and ALE environments. Agents trained solely on one environment have not been able to score on the other environment, let alone generalize. At a minimum, this test shows that agents *can* learn *something* when trained on a mixture of environments, even if, for example, the agent is learning only conditional policies, based on image features used to classify whether the environment is ALE or ToyBox. This is an example of one of the many applications of ToyBox beyond what we have detailed here.

CHAPTER 5

AUTOEXP: AN EXPERIMENT MANAGEMENT SYSTEM FOR EXPLANATION

In contexts where experimentation is part of a software system or software pipeline, experimenters may use an experimentation management *system*. Such systems use a carefully designed software architecture to ensure that the underlying assumptions of how experiments should work are not violated. For example, two experiments that intervene on the same variable should not be run concurrently for the same set of users. For background on experimentation management systems, see Section 2.3.

The behavioral tests of Chapter 4 are sufficiently simple that they do not require an additional management system: each test evaluates only a single hypothesis for a specific agent, over a set of states that the experimenter believes should elicit equivalent behaviors. The tests rely on the researcher's already having a hypothesized model of behavior and are thus primarily a confirmatory data analysis tool—this form of testing is suitable for hypotheses that have already been identified.

In this chapter, we describe the requirements and challenges for designing a system that can provide counterfactual explanations to users given a black-box agent and a white-box environment. We present AutoExp, a prototype system for performing exploratory data analysis by automatically generating counterfactual explanations[1]

[1]Producing explanations for AI (i.e., *Explainable AI*, or XAI) is an emerging area of study that seeks to explain the behavior of learned, black-box agents or classifiers powered by deep learning. XAI has been the focus of several recent technical workshops, has become a standard subject area for technical sessions in major conferences, and has been the focus of a large Defense Advanced Research Projects Agency (DARPA) research program [55]. The research challenges of (and corresponding research progress on) this topic has also been regularly reported in the popular press [78].

of the behavior of black-box agents interacting with an intervenable environment. In order to provide useful explanations, AutoExp must make reasonable progress when searching through a large space of candidate interventions. It must be *possible* for AutoExp to find an explanation if it exists, and AutoExp must not bias the search except through top-level user-defined pruning of the search space that can be reproduced on another researcher's machine.

We will focus our evaluation on agents that play a single game: Breakout.[2] We evaluate AutoExp first on a set of scripted agents, whose policies are known. This allows us to: (1) stress test the system for known edge cases, and (2) establish that AutoExp works as intended. Then we apply AutoExp to a sample of the deep agents also used in Chapter 4.

Before we describe the AutoExp system and how it works, we begin by precisely defining the outcomes we care about in the game of Breakout and the scripted agents that were designed to exercise the AutoExp system in Section 5.1. We resume our more general discussion of automatic experimentation in Section 5.2, followed by major challenges (Section 5.3), the system itself (Section 5.4), why the search proceeds one factor at a time (Section 5.5), and we conclude with an evaluation of the system (Section 5.6).

5.1 Outcomes and Agents

Throughout this chapter, we will refer to scripted agents and a fixed set of outcomes for Breakout agents. Because we know how the agents are encoded, we can make reasonable ground-truth hypotheses about what explanations an automated system might generate for various outcomes. We will consider the following outcomes:

[2] As we did for Chapter 4, we will use the ToyBox system, which provides access to more games, but in this chapter we focus the evaluation on a breadth of agents rather than games.

Aim An agent can aim the ball left, right, or up. We determine aim by segmenting the paddle into thirds; if the ball's x position remains within the one of these segments for a specified window of time, then we say that the agent is aiming the ball in that direction. The ball's y position is not used when computing this outcome. We will be detecting only left and right in our later experiments.

HitBall We detect whether an agent has hit the ball if the ball begins the window heading down and then heads up. This outcome requires a minimum window size of three frames, and uses only the ball's y position.

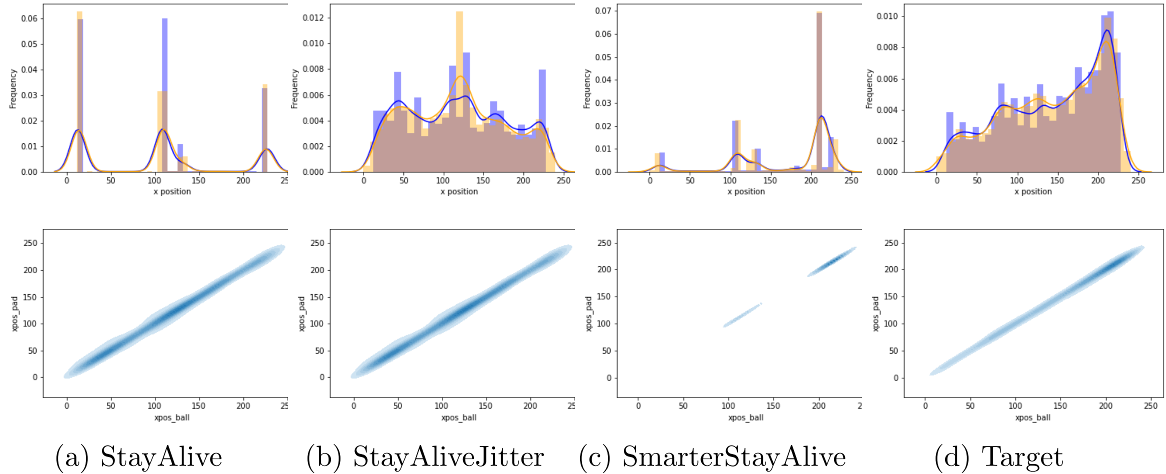
MissedBall This outcome is self-explanatory, but can sometimes be difficult to detect; when an agent misses, the ball is no longer in play. However, sometimes the window is too short and we can clearly see that the agent has missed the ball, even if the ToyBox system has not yet registered this. Therefore, we must compare the ball's y position with the paddle's y and x positions. Note that we do not forward simulate; this extra computation is enough to capture the edge cases where we need only one or two more frames to detect failure.

MoveSame This outcome denotes the case where the ball and paddle are moving in the same direction for over 50% of the window used to compute the outcome.

MoveOpposite This outcome denotes the case where the ball and paddle are moving in opposite directions on the x axis for over 50% of the window used to compute the outcome.

MoveToward This outcome denotes the case where the x distance between the ball and paddle is getting smaller for over 50% of the window used to compute the outcome.

MoveAway This outcome denotes the case where the agent and the paddle are *both* moving in opposite horizontal directions *and* the distance between the ball



(a) StayAlive (b) StayAliveJitter (c) SmarterStayAlive (d) Target

Figure 5.1: Observations of agent movement can inform the choice of outcomes we wish to explain. **Top**: Histogram and density estimation for the x positions of the ball (blue) and paddle (orange) for 30 trials each, for four agents playing Breakout for a maximum of 2000 steps. **Bottom**: Joint densities of the ball x position and paddle x position. A wider range of agents that have more variable performance might allow us to identify the critical time period for intervention dictated by environment dynamics discussed in Section 5.3.1.

and the paddle is increasing for over 50% of the window used to compute the outcome.

Below we describe the known behavior of four scripted Breakout agents that we use in the examples. Appendix D gives data dependence graphs (and in some cases, causal graphs) of the effects of variables from the ToyBox environment on agent action choice. We designed these agents to test known edge cases in the experimentation system.

We have manually verified that a sample of detected outcomes comports with what a human observer would expect the outcome name to denote. There may be cases where the outcome detectors fail; we want only to ensure that the cases we do find make sense to a human observer (i.e., we care about soundness, not completeness).

5.1.1 StayAlive Agent

The StayAlive agent follows a simple policy:

$$action = \begin{cases} \texttt{left}, & \texttt{ball.x} < \texttt{paddle.x} \\ \texttt{right}, & \texttt{ball.x} > \texttt{paddle.x} \\ \texttt{noop}, & otherwise \end{cases} \tag{5.1}$$

`action` at time t is deterministically computed from ball x position and paddle x position. Ergo, all uncertainty associated with this agent's model will be epistemic. There is variability in the state information at time t (i.e., variability in the data) because the ball start position and angle are together sampled from four possible configurations.

`ball.x` and `paddle.x` are both low-level autonomous variables that can be manipulated directly. `action` at time t is causal for both `paddle.x` and `ball.x` at time $t+1$. These variables will not only be drawn from very similar distributions for any agent that successfully plays Breakout, but they will also be correlated in the observed data (see Figure 5.1).

For a low-level outcome such as `action`, `paddle.x` and `ball.x` should suffice to explain the *causal mechanism* of the agent's choice of action at time t. However, higher level outcomes – including but not restricted to those that require reasoning over time – may require state information that captures *environment dynamics*.

We can now hypothesize how we expect StayAlive to perform for each of the outcomes:

Aim (left or right). As previously noted, the agent aligns the center of the paddle with ball's x position. The only point at which we *might* detect the agent aiming left or right is at the start of the game, when moving the paddle to the ball.

However, the window of state-action pairs used to compute the outcome would need to be fairly small to trigger this outcome.

HitBall. This agent, by design, always hits the ball and never misses. The only way we could induce the agent to miss the ball would be to increase the distance between the ball x position and paddle x position and decrease the ball y position so that there would be no way to move the paddle to the ball in time to hit it. If we allow changing only one factor at a time, it is exceedingly unlikely that we will generate an explanation within a window of reasonable size.

MoveSame. We would expect to detect this outcome early in game play, before the paddle is already aligned under the ball (once aligned under the ball, the paddle stops moving, and so it cannot trigger the MoveSame outcome).

MoveOpposite, MoveToward. The StayAlive agent moves only in the opposite direction of the ball it is moving toward it. Again, the only way to change this with a single variable is to change the ball's velocity.

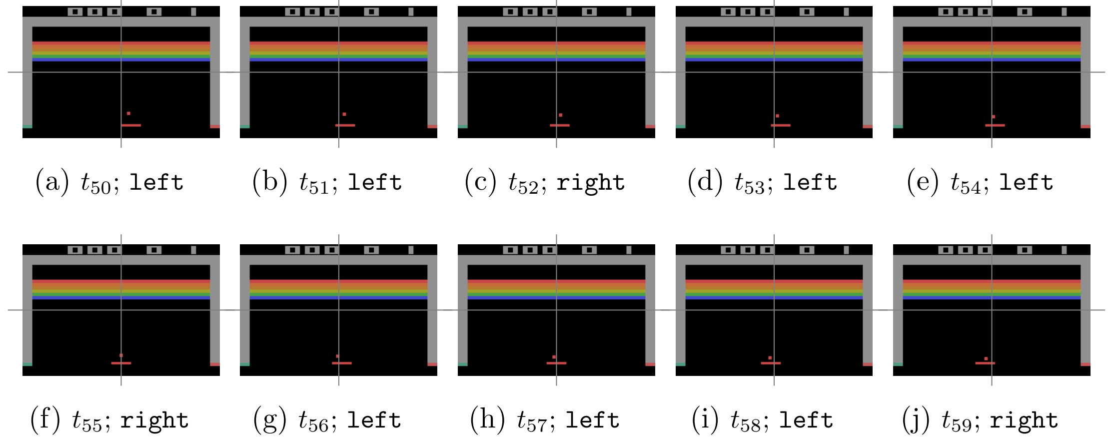
MoveAway. The agent, by definition, never moves away from the ball. If this outcome is detected, it is due to jerkiness of paddle movement and the particular window size.[3]

5.1.2 StayAliveJitter

The StayAliveJitter agent follows the same policy as the StayAlive agent, with two exceptions: it uses the previous time step's ball and paddle x positions to choose its action (thus eliminating the jerking behavior) and it sometimes moves randomly. As a result, we observe the agent exploring more of the state space (compare ball and paddle x positions in Figure 5.1). Figure D.2 depicts the data dependency graph.

---

[3]We are not actually sure that this outcome *can* be detected for the StayAlive agent, but leave the possibility open.



(a) t_{50}; `left` (b) t_{51}; `left` (c) t_{52}; `right` (d) t_{53}; `left` (e) t_{54}; `left`

(f) t_{55}; `right` (g) t_{56}; `left` (h) t_{57}; `left` (i) t_{58}; `left` (j) t_{59}; `right`

Figure 5.2: A trace of frames and actions for the StayAlive agent, where the "jerking" behavior can be observed. The action at time t_i can be observed at time t_{i+1}. We have bisected the frames to make the ball and paddle direction clearer upon visual inspection.

Using the same reasoning as for the StayAlive agent, we expect the outcomes and explanations for StayAliveJitter to be the same. That is, for most outcomes, we do not expect a system to generate a single-variable outcome often. We do expect occasionally to see explanations that are due to the random seed, that will not replicate over many seeds.

N.b.: The distribution of x positions for StayAliveJitter in Figure 5.1 is more uniformly distributed than we would have guessed *a priori*. This may have an impact on the outcomes and/or explanations produced. Furthermore, the likelihood that an agent chooses a random action is not uniformly distributed; let $X \sim Bern(\theta)$ and $Y \sim Bern(0.5)$, where θ is a parameter of the StayAlive agent, a real-valued number between 0 and 1:

$$
action_t = \begin{cases} \texttt{left}, & \texttt{ball.x}_t < \texttt{paddle.x}_t \land \texttt{ball.x}_t < \texttt{ball.x}_{t-1} \land \neg Bern(\theta) \\ \texttt{right}, & \texttt{ball.x}_t > \texttt{paddle.x}_t \land \texttt{ball.x}_t > \texttt{ball.x}_{t-1} \land \neg Bern(\theta) \\ \texttt{left} & X \land Y \\ \texttt{right} & X \land \neg Y \\ \texttt{noop}, & otherwise \end{cases} \tag{5.2}
$$

The function should be read sequentially, à la Scheme or Lisp cases. Note the first two cases: we sample from $Bern(\theta)$ more than once per state.

5.1.2.1 SmarterStayAlive

SmarterStayAlive is also based on StayAlive. Like StayAliveJitter, it uses the previous paddle and ball x positions to smooth out agent actions. Unlike StayAliveJitter, it does not take any random actions. When this agent finds itself aligned under the far left or the far right column, it attempts to move the paddle farther to the left or right, to break out of the state-action loop. It makes no such attempt when aligned under one of the two central columns. The data dependency graph is depicted in Figure D.3

We expect nearly all of the outcomes and explanations to be the same as for StayAlive, except for the aiming outcomes: when the agent is in the far left corner of the board, it should move the paddle farther to the left, so the ball hits the paddle on the right side, which should trigger the aim right outcome (and vice versa for the right side of the board). If we use aiming up as the counterfactual to either aiming right or left, we should be able to find an explanation: move ball x position to be under one of the non-extreme columns. It is not clear whether there is a single variable intervention that can change the outcome when the outcome is aiming left or right, and the counterfactual cannot be aiming up. This is because, in order for the agent to

aim left when it had been aiming right, we would likely need to intervene on both the paddle x position and the ball x position. Otherwise, we may simply miss the ball.

5.1.3 Target

The Target agent differs significantly from the StayAlive family of agents; it actually attempts to implement a strategy beyond simply surviving. Figure D.4 depicts a ground truth causal graph for the Target agent.

At a high level, the Target agent follows the StayAliveJitter agent's policy when (1) it has not yet scored and (2) it has already scored, and the ball is outside a bounding box that is proportional to the paddle size, relative to the paddle's current location. We omit the "causes" (i.e., inputs to the function that computes the bounding box) of the boundaries of the bounding box in this diagram for legibility purposes. A human observer might visually detect that the agent behaves differently when the ball "is close" to the paddle, but would have a great deal of uncertainty over how this variable should be measured, computed, or represented. We happen to know how this variable is computed deterministically from the ball x and y positions, the paddle x and y positions, and paddle width.

When the ball is within the bounding box, the Target agent estimates where it thinks the ball will cross the x axis. Then it selects the first non-empty column that has the fewest number of bricks, and attempts to align the paddle under the ball such that the ball will hit that column. If the ball is within some ϵ of the center of the target region of the paddle, then the agent may take a random move.

The Target agent also differs from the StayAlive agents in that it uses several internal variables, whose internal form we cannot access (although we could log them in order to measure how far off our estimates of their values are).

We can now hypothesize how we expect Target to perform for each of the outcomes:

Aim. By definition, the Target agent attempts to aim the ball at a particular column. It should be possible to change where the agent is aiming by changing the board configuration. Note, however, that the agent chooses the *first* column (reading left to right) that has the minimum number of bricks. This means that the agent will be biased toward moving to the right/aiming left. We can see evidence of this behavior in Figure 5.1. If we are permitted to change only a single low-level variable at the time of intervention, we may not find a change in state that changes outcome. However, if we allow arbitrary interventions on the board, there should be a set of changes that cause the agent to aim in a different direction.

HitBall. There are several interventions that might cause the agent to miss the ball. The agent computes the bounding box for deciding when to enter its "targeting mode" on the basis of the ball's y position and the paddle width. It is possible that by making the paddle *wider*, we could cause the agent to miss. This intervention may seem counter-intuitive, since a human player should find it easier to hit the ball when the paddle is wider. However, the Target agent's calculation of where the ball will cross the x axis is not a perfect model of the ball's path. For example, it does not include logic about what happens when the ball bounces off one of the side walls. This means that the agent's calculation of where the ball will be is actually an approximation. If we increase the paddle width, we increase the window of time where the agent is guessing where the ball will be, which increases the likelihood that the agent will miss the ball.

Furthermore, the agent's calculation of *how* to target the ball is imprecise, and the agent may sometimes take a random move. If the agent is targeting a column on the far end of the board, it will need to hit the ball off the far end of the paddle. It is possible that the agent misses here. Therefore, in contexts/states where the agent should try to hit the ball on the edge of the paddle, we could move the paddle slightly farther away, potentially causing the agent to miss.

Finally, it is also possible that a change in the board configuration could cause the agent to miss the ball, by changing where the agent needs to target.

MissedBall. We expect the relationships that informed our hypotheses about why an agent hits the ball to be symmetric for why an agent misses the ball: a smaller paddle width, small changes in the paddle x location toward the ball, and a change in the board configuration to create targets closer to the paddle's current position should all cause an agent that has previously missed the ball to hit it.

MoveSame. If we have detected that the agent is moving in the same direction as the ball and the ball is currently outside the bounding box for targeting, then we should be able to find x positions that cause the agent to move in the opposite direction of the ball. Although the agent uses the SmarterStayAlive strategy here, because it toggles to the Target strategy, it could end up in positions that allow us to detect this outcome and induce the counterfactual via intervention.

We should detect that the agent is moving in the same direction as the ball only when the agent is targeting a column to the far left or far right. We should be able to detect a counterfactual by changing the brick configuration so that the agent targets a column directly above the current paddle position. Note that if we move the paddle under the targeted column, the agent will probably miss the ball and we will not detect the counterfactual.

MoveOpposite, MoveToward, MoveAway. The agent moves in the opposite direction during the same time periods as the SmarterStayAlive agent, and when targeting. We do not expect to detect this outcome as frequently as MoveSame. This is because, after the initial downward path of the ball, the agent will following the ball, except when targeting. Targeting makes micro-adjustments, which we expect to occur over too small a window to detect most of the time. Again, increasing the paddle width could cause an increase in the frequency we observe this outcome. If

we do detect this outcome, we suspect any counterfactuals we discover will be due to random actions and therefore will not hold up under replication.

Since MoveToward and MoveAway are subsets of the MoveOpposite outcome, we do not expect many Target agents to trigger them, for the same reasons.

5.1.4 Agent and Outcome Rationale

The purpose of scripted agents is to validate our techniques under known and interpretable policies, so we are justified in applying them to deep agents, whose policies are not interpretable. We started with the simplest possible policy (StayAlive) and added complexity (randomness, temporal information). The Target agent code is significantly more complex and contains latent variables, approximations, and occasional random behavior. We have implemented an agent API that allows users to construct arbitrarily complex agents.

The outcomes were developed separate from agents. We strove to design outcomes that captured higher-level behavior that a user might want explained, and to design them irrespective of agent policy. We cannot say that they were developed independently, because they were developed by the same person (the author). Ideally, outcomes and agents would have been developed separately, by different subsets of researchers. Future evaluation should have some separation of these functions.

5.2 Problem Formulation and Formalization

The task of explaining an agent's behavior is often formulated based on a human observer who watches an agent interacting with an environment, witnesses some particular outcome, and then wishes to know why that outcome occurred. For example, an observer could watch an agent playing Breakout and ask "Why did the agent miss the ball?" Queries of this style can have many interpretations, so we

rephrase them as counterfactuals, e.g., "What would need to have been different in the environment to make the agent hit the ball?"

In order to search for explanations, we need to define what we mean by an explanation. In this section, we begin with formal definitions of the explanation state space and allowable intervention. Then we provide a sequence of three definitions of counterfactual explanation.

5.2.1 Basic Assumptions and Definitions: Agents and Environments

We assume that the environment is constructed similarly to ToyBox, where an agent interacting with an environment can be re-run from an arbitrary time point.

Experimentation in ToyBox resembles simulation experiments more closely than it resembles field experiments. This is because trials in ToyBox are deterministic, on the basis of a random seed. Therefore, we will need to incorporate reasoning about deterministic behavior under random seeds in our formal specification of the AutoExp system.

Recall from Section 2.8 that in reinforcement learning (RL), an agent can be defined by its policy π, a (possibly probabilistic) function that maps states to actions. Environments produce states, and obey a transition function T, a probabilistic mapping from a state-action pair to another state.

The learning procedure for an agent typically incorporates T into the Markov decision process (MDP) that formalizes the problem space [128]. An MDP is a 4-tuple $\langle S, A, P_a, R_a \rangle$ of states $\mathcal{S}$, actions $\mathcal{A}$, conditional probability functions P_a, and reward functions R_a. However, in order to reason about explanation, we need to reason about intervention, which requires that we consider states that may be outside of the distribution encountered during training. T typically refers to a probabilistic, denotational view of environment dynamics; when building an experimentation system for explanation, we require an operational view of environment dynamics. Therefore,

it is more appropriate to reason instead about a deterministic machine, M, that maps from state-action pairs to the next state.

When implemented as software, a probabilistic agent or machine will need to be parameterized by a random seed γ.[4] Let $s \in \mathcal{S}$ and $a \in \mathcal{A}$. Then we have:

$$M_\gamma(s_1, a) = s_2 \tag{5.3}$$

$$\pi_\gamma(s) = a \tag{5.4}$$

When we condition on the random seed and restrict s to the space of *on-policy states*, the agent policy and model are *total functions*, functions defined for all possible inputs of the correct type. However, we would like to reason about interventions that may be drawn from the *concept space*, as defined by Clary [21].[5] Let $\mathcal{C}_M$ denote the set of states from concept space and let $\mathcal{O}_{(\pi,M)}$ denote the set of on-policy states. Since $\mathcal{O}_{(\pi,M)} \subset \mathcal{C}_M$, $\forall s \left(s \in \mathcal{O}_{(\pi,M)} \rightarrow s \in \mathcal{C}_M\right)$. Thus, in order to proceed, π_γ and M_γ must be defined over $\mathcal{C}_M$:

Assumption 1 (π and M are total) *The policy and transition function are total for $\mathcal{C}_\mathcal{M}$:*

$$\forall \gamma, s \exists a \left(a \in \mathcal{A} \wedge s \in \mathcal{C}_M \rightarrow \pi_\gamma(s) = a\right) \tag{5.5}$$

$$\forall \gamma, s_1, a \exists s_2 \left(a \in \mathcal{A} \wedge s_1 \in \mathcal{C}_M \wedge s_2 \in \mathcal{C}_M \rightarrow M_\gamma(s_1, a) = s_2\right) \tag{5.6}$$

---

[4] The environment and agent need not have the same random seed. γ will, in the general case, refer to an existentially quantified seed; if we need to differentiate values of γ, we will note it.

[5] This work is unpublished and ongoing. Clary defines a state-space hierarchy: on-policy states are those that can be produced by running *this* agent; off-policy states are those that can be produced by running *any* agent, and concept space states include states that are neither off-policy, nor on-policy, but are "meaningful." For example, some experiments of Witty et al. partially complete a game board with configurations that *could never have been encountered during training, for any agent* [140]. However, a reasonable observer would still conclude that a well-performing agent could perform well on these states.

We can justify this assumption by noting that, for any software implementation of π_γ and M_γ, we can run each function and observe whether the program crashes.[6] It follows by induction from the assumption and the mapping that π and M are closed over $\mathcal{C}_M$.

Our problem formulation specifies that we expect a user to request an explanation after having observed some behavior. This behavior will be a function of state. Therefore, since we want to explain only *behaviors* that a particular agent could actually exhibit during normal execution, we will be interested in analyzing only *observations* of the environment that could happen during normal execution of this agent in this environment. We are not interested in explaining the behavior of something that *couldn't* happen during normal execution of π.[7] It follows from Definition 1 that any on-policy state is a valid observation:

Definition 1 (Valid Observation) *A state s is a valid observation for M and π iff there exist some γ, γ' such that $M_\gamma(s, \pi_{\gamma'}(s)) \in \mathcal{O}_{(\pi,M)}$.*

An observation is just a state; we would like to reason about outcomes. Let us define an outcome as a predicate indicating whether or not it happened:

Definition 2 (Valid Outcomes: Factual and Counterfactual Pairs) *Let (P, Q) be a pair of predicates, each over state-action traces (i.e., sequences of state-action pairs), where $P : (\mathcal{O}_{(\pi,M)} \times \mathcal{A})$ list $\rightarrow$ bool and $Q : (\mathcal{O}_{(\pi,M)} \times \mathcal{A})$ list $\rightarrow$ bool. If*

$$\exists t, t' \Big(P(t) \wedge Q(t') \Big) \textit{(Existential evidence of behavior)} \quad (5.7)$$

$$\wedge \forall t \Big(\big(P(t) \rightarrow \neg Q(t) \big) \wedge \big(Q(t) \rightarrow \neg P(t) \big) \Big) \textit{(Separability of evidence)} \quad (5.8)$$

[6] This assumes that both programs terminate.

[7] This would likely be the purview of researchers studying the effects of novelty.

then (P, Q) *is a valid outcome for* π*;* P *selects the factual outcome, and* Q *selects the valid counterfactual outcome.*

Outcomes, like observations, are tied to the behavior of a particular agent. Outcomes have a factual component (the outcome value actually observed) and a counterfactual component (an outcome value that was not observed during the same window). Factuals and counterfactuals are thus both defined as functions of sequential valid observations (and the associated actions for policy π). We require that both the factual and the counterfactual be observed at some point (Sentence 5.7), as existential evidence that the counterfactual is *possible* for the agent.

Example. The StayAlive agent (Section 5.1.1) follows the the x position of the ball. If our outcome is "hits the ball," and the counterfactual is "misses the ball," we will observe the outcome for every trial (i.e., for every seed γ), but we will never observe the counterfactual by design. Therefore, we do not consider this a valid factual and counterfactual pair for the specific StayAlive agent.

Note that in the general case, testing whether a given pair (P, Q) is valid for agent π on machine M is undecidable. In our analyses of Sections 5.6.1 and 5.6.2, when selecting suitable pairs (P, Q), we run an agent for a single episode and determine whether a pair is valid for that agent for the observed data from that episode.

Note also that a single outcome could have many counterfactual outcomes, but for a pair to be valid, they cannot both be true for the same input. They can, however, both be false for a given trace t. Sentence 5.8 describes this condition.

Example. Breakout allows three actions during gameplay: `left`, `right`, and `noop`. Suppose the outcome of interest is that the agent moves the paddle left. Then the counterfactual could be moving the paddle right, not moving the paddle, or the union of these two actions. It is up to the researcher to decide which counterfactual is appropriate for the given context; we must choose a counterfactual that is mu-

tually exclusive with the outcome (e.g., we should not choose "misses ball" as the counterfactual to "moves left").

Definition 3 (Valid Intervention) *Any total function* $f : \mathcal{C}_M \to \mathcal{C}_M$*, such that* $\forall s, s' \, (f(s) = s' \to s \neq s')$*, is a valid intervention for* M*.*

Any programmatic state manipulation that a machine accepts or is in the domain of the environment (i.e., is in $\mathcal{C}_M$) will be a total function with the correct signature; we should never be producing states that cause M to crash. We additionally require that, for a function over states in concept space, the intervention *actually changes* some aspect of the state.

As discussed in Chapter 4, ToyBox exports its state as JSON.[8] Consequently, the JSON can function as an ontology of objects, homomorphic to the internal object representation. These objects have attributes that can be other objects or base types. The core set of base types in ToyBox consists of integers, floating point numbers, Booleans, enums,[9] and strings. For Breakout, the set of base types is integers, floating point numbers, and Booleans.

Definition 4 (Atomic Attribute) *An atomic attribute is any attribute whose type is in the set of base types.*

The leaves of a JSON object will all be atomic attributes. An atomic attribute is the finest-grained unit of logging. Most atomic attributes in ToyBox are also manipulable, but in the general case we should not assume this to be true.

Conjecture 1 (Identifiability via $\mathcal{C}_M$**)** *Some causal effects can be identified only when we allow intervention in concept space.*

[8] `https://www.json.org/json-en.html`

[9] `enum` or enumeration types are often used to represent categorical variables.

We leave the proof of this theorem to future work, but it will likely rely on an argument about the tautological lack of positivity and inherent unobservability of some states, when we only consider those states that can be generated either on-policy or off-policy. In the general case, when the size of the Markov equivalence is greater than one, then causal effects cannot be identified by observational data alone. What we need for the proof to be complete is an argument that interventions in concept space are drawn from a distribution that we would expect an agent to be able to learn.

Definition 5 (State difference) *Let $s[X] = x$ denote the value x of an atomic attribute X for state s. Then we define the state difference*

$$s - s' = \{\langle X_1, x_1, x_1' \rangle, \ldots \langle X_n, x_n, x_n' \rangle\},$$

the set of keys that differ and their differing values.[10]

5.2.2 Formal Definitions of Counterfactual Explanations

We can now define what we mean by a counterfactual explanation in the context of an RL agent.

Definition 6 (Strong Counterfactual Explanation) *A strong counterfactual explanation for an outcome variable $Y = (P, Q)$ that takes on a factual predicate P with respect to a variable set $\mathcal{V}$ is any subset $\{V_1 = v_1', \ldots, V_n = v_n'\} \subseteq \mathcal{V}$ such that, for*

[10] For the moment, we care only about the *probability* that two values differ. However, our definition of set difference does open up the possibility of defining a metric for "size" or "magnitude" of intervention. For example, we may be interested the Hamming distance between the two states (i.e., $|\{\langle X_1, x_1, x_1' \rangle, \ldots \langle X_n, x_n, x_n' \rangle\}| = n$), an L_p distance between two states (i.e., $||\{\langle X_1, x_1, x_1' \rangle, \ldots, \langle X_n, x_n, x_n' \rangle\}||^p = (\sum_{i=1}^{n} (X_1' - x_1)^p)^{1/p}$), or a weighted ($w$) distance for some distance function d proportional to, e.g., the attribute depth or variability ($\sum_{i=1}^{n} w(X_i) d(x_i, x_i')$, where w is a function of attribute (e.g., $w(X) = 1/depth(X)$, so $s($`paddle.position.y`$) = 1/3$), standard deviation of the observed attribute, etc.).

some trace $t = [t_0, \ldots, t_n]$*, state* $s \in \mathcal{O}_{(\pi,M)}$*, and intervention* f *that maps* s *to a new state where each attribute* V_i *has been set to its corresponding* v_i*,*

$$P(M^*_\gamma(s_{t_i}, \pi_\gamma(s_{t_i}))) \wedge Q(M^*_\gamma(f(s_{t_i}), \pi_\gamma(f(s_{t_i}))))$$

where M^*_γ *refers to the repeated application of* $M\gamma$ *for each subsequent state, and each action as produced for each subsequent application of the policy.*

Definition 7 gives an operational definition of a counterfactual explanation: we perform a valid intervention on some intervention state s_{t_i} and forward simulate until t_n. If the counterfactual predicate Q holds under intervention, we have found an explanation.

When we control for the random seed in agents or environment, there is no variability. Therefore, any explanation we produce is *definitely* due to the change we induced. Unfortunately, in most cases, we will not be able to completely control the variability of the agent. Therefore, we need a weaker definition of counterfactual explanation:

Definition 7 (Weak Counterfactual Explanation) *A weak counterfactual explanation for an outcome variable* $Y = (R, Q)$ *that takes on factual value predicate* R *with respect to a variable set* $\mathcal{V}$ *is any subset* $\{V_1 = v'_1, \ldots, V_n = v'_n\} \subseteq \mathcal{V}$ *such that*

$$P(Q(t) \mid do(V_1 = v'_1, \ldots, V_n = v'_n), V - \{V_1, \ldots, V_n\}) >$$
$$P(R(t) \mid V_1 = v_1, \ldots, V_n = v_n, V - \{V_1, \ldots, V_n\})$$

where $v_1 \neq v'_1, \ldots, v_n \neq v'_n$*.*

Weak counterfactual explanation is a statistical definition, which we need to use when we cannot control the agent's variability. Since we expect agents potentially to be black boxes with opportunities for intervention, our approach must handle weak

counterfactual explanation. Note that `do` here refers to intervention (see Section 2.1); thus, this definition could be suitable in contexts where we cannot manipulate the environment directly, but must intervene in, e.g., a causal graphical model.

Note for both Definitions 6 and 7, any subset of $\mathcal{V}$ could be a counterfactual explanation.

Finally, we consider what it might mean for a counterfactual explanation to be "optimal." Note that what constitutes a *good* explanation is an open question. Some of the best-known framings of explanation in machine learning focus on building *interpretable models* [42, 33, 94]. Model-agnostic counterfactual explanation has had fewer attempts as formalization [55, 103, 71].

An eventual optimal counterfactual explanation might take the form:

Definition 8 (Optimal Counterfactual Explanation) *An optimal counterfactual explanation for an outcome $Y = (R, Q)$ with respect to a variable set $\mathcal{V}$ is the minimum subset $\{V_1 = v'_1, \ldots, V_n = v'_n\} \subseteq \mathcal{V}$ such that: $argmax_{v'_1,\ldots,v'_n} P(Q(t) \mid do(V_1 = v'_1, \ldots, V_n = v'_n), V - \{V_1, \ldots, V_n\})$, subject to $min_{v'_1,\ldots,v'_n} d(\langle v_1, \ldots, v_n \rangle, \langle v'_1, \ldots, v'_n \rangle)$ for some distance function d, given the observation $R(t)$.*

There are currently sufficiently many open questions about what makes an explanation "good" that we would consider a search for an "optimal" explanation premature. Sections 5.3 and 5.5 discuss the challenges inherent in defining d, and the role that an optimal definition would play in making the search for explanations more efficient.

5.2.3 Novel Explanation Hierarchy

In order to test AutoExp, we need to define agents that have varying types of explanations. To guide our understanding of explanation, we defined a novel explanation hierarchy:

Level 4: Random The least desired explanation in our hierarchy is that the agent is behaving randomly. However, per the discussion of Definitions 6 and 7, we

may not be able to control randomness in the agent's policy. Therefore, we use Definition 7 to rule out spurious explanations. We use the odds ratio of the factual and counterfactual under intervention and control to determine whether the intervention generally has an effect.

Level 3: Environment dynamics The next least desired explanation is that the outcome is due to environmental dynamics. For example, AutoExp could return an explanation for MissedBall that places the ball directly above the center of the paddle, thus inducing the counterfactual HitBall.

Level 2: Outcome dynamics Outcomes are defined for states and actions over time. Many of the explanations that AutoExp finds for the current batch of agents are functions of outcome dynamics. Agents may use the same set of variables to determine their next actions as the outcomes use to determine whether the factual or counterfactual occurred. However, discovering such explanations does not showcase the utility of an automated explanation system, since a user could simply use the encoded definition of the outcome to select variables to test. In this case, they could simply use the ToyBox testing system of Chapter 4.

Level 1: Agent policy The ultimate goal of our explanation is system is to be able to find explanation variables that are used only by the agent's policy, and not by the outcome. Such explanations would be tedious to find manually. We believe that such explanations will be useful for understanding deep agents.

Recall Figure 4.3 from Chapter 4: an explanation system seeks to differentiate the agent's internal model (i.e., variables used within the policy) from the environmental dynamics, and from the researcher's mental model (used to define outcomes).

5.3 Major Challenges Identified

Several features of our problem formulation and task necessitate a novel solution; we enumerate the major challenges below.

5.3.1 Treatments and Outcomes over Time

Computer simulation experiments, field experiments, and simulated experimentation on CGMs via the `do`-calculus all treat time similarly: an experimenter performs an intervention, some time passes, and then the experimenter measures the outcome. For simulation experiments, the intervention is the set of input or parameter settings at the start of simulation. For field experiments, the researcher may decide to run experiments at certain times (e.g., advertising campaigns that run only on the weekends, during elections, etc.), but this information is generally implicit, not encoded anywhere, and there will generally be some flexibility on timing. Most causal models are propositional, where the only encoding of time is the happens-before relationship of edges between variables. Although there is work on CGMs for time series models, defining treatment and outcome over time is a known challenge. Since time series models typically deal with real-world data, there is less fine-grained control over when intervention can happen, and re-running experiments at arbitrary time points is impossible.

For our problem formulation, we will need to decide *when* to intervene and *how long* to intervene. Games such as Breakout are episodic, where each episode ends when an agent misses the ball or times out. Suppose we observe the agent hit the ball in the center of the paddle at time t and want to know what would need to have been different at time $t-1$ to make the agent miss. Since the paddle is centered below the ball at time t, the ball would need to have been close to the center at time $t-1$ as well. No matter what action the agent chooses at $t-1$, it will still hit the ball. Therefore, even if we could change some variable in the environment, to cause the agent to miss,

the explanation produced would be addressing environment dynamics, rather than agent decision-making. We describe how AutoExp decides when to intervene in Section 5.4.1.

5.3.2 Variable Definition

Computer simulation experiments have well-defined treatments and outcomes: typically the treatment variables are the parameters or inputs to the system, and outcomes are established metrics in the domain of interest. Online field experiments in the style of PlanOut have access to configuration parameters in a software system, which correspond to treatments, and outcomes are also established metrics. Sometimes outcome metrics are a proxy for more qualitative, or difficult to define outcomes, but the mapping between quantities actually measured and their qualitative interpretation is clear. For CGMs, model variables (i.e., nodes in the graph) correspond to possible treatments or outcomes.

In our domain, there may be a representational mismatch between the low-level parameters we can manipulate and the higher-level concepts we wish to use for explanation. The correspondence between a data dependency graph of these low-level parameters and a causal graph of higher-level concepts may not be clear. For example, we may include aggregated variables such as the count of some collection of objects in, e.g., a CGM that describes a software system. The software system may have no reified internal representation of the count, and if that system permits intervention, it may allow intervention to occur only on a per-object level. This leaves a great deal of variability and choice to the researcher. In the example of "count", which objects should be deleted to intervene may be important or unimportant. Our hope is that the set of possible object-level interventions form an equivalence class with respect to the aggregated variable's intervention. If not, it is clear that the aggregated variable is an inappropriate choice for the CGM. Unfortunately, returning a collection

of low-level software parameters as a possible explanation may not be useful to the researcher. Thus, we believe our domain requires fast iteration for variable definition. We describe our solution to this issue in Section 5.4.2.

5.3.3 Challenges with framing Explanation as Optimization

Much existing work on automated or adaptive experimentation focuses on optimization: finding parameters that are the "best" for a particular task. The optimization task allows a system to use gradients as a guide for where to search next. However, there is no consensus on what makes one explanation better than another. While Definition 7 formally specifies what we mean by *an* explanation, there are many possible choices for how to compare explanations:

1. One possibility is to find the **minimal the distance between states** that would, with high probability, cause a difference in outcome. Thus, a small difference would be measured as $d(x, x') < \epsilon$ for some distance metric $d : X \times X \to \mathcal{R}$. When X has a numeric type *and* $x - x'$ has meaning within the context of M, then d takes on whatever L_p distance is appropriate. However, there are cases where X is numeric, but L_p distances are not meaningful. When X has a non-numeric type, there may not be a canonical distance metric, and so the user must craft one. Validating the utility of this distance metric may require, e.g., human studies or vast domain knowledge. This could be a significant undertaking and would not lend itself to automation.

2. Another approach recently discussed by Jensen [62] is to **minimize the ratio of the conditional probabilities of the treatment versus the control**. Here no distance metric is required. The challenges that come with this approach are typical causal modeling and inference issues: a lack of positivity, variable definition, variability in the data, etc. This work focuses exclusively on single-variable explanations. Unlike our system, this approach could produce alternative

explanations, by offering the user a list of single-variable explanations, ordered by the likelihood that they change the outcome.

3. A less discussed approach has been to find the **minimum set of variables** that, on average cause a change in outcome. The intuition is that we could probably cause a change in outcome if we randomly sampled from the entire state space, but such a large *number* of interventions could render the explanation useless to the observer. Furthermore, we suspect that, in some cases, a large number of interventions that are actually meaningful will correspond to a composite attribute or variable.

We make the argument in this chapter that experimentation for optimization is generally not suitable for explanation at this time. There are design choices we will need to make in order to find solutions for Definition 8 effectively (e.g., choice of d in a relational and dynamic environment) that are open problems. Therefore, we leave finding a solution to Definition 8 to future work.

5.4 AutoExp: Automating the Search for Explanations

We have implemented a tool—AutoExp, depicted in Figure 5.3 — that performs experiments for a single agent and environment for fixed random seeds (in line with Definition 6) until it finds an intervention that change the outcome of interest. We start by describing the architecture and design for performing many experiments in a single run (defined by Algorithm 1). This part of the system is fully automated, but customizable.

The upper left block in Figure 5.3 depicts the first step of the AutoExp system: selecting the attribute(s) on which to intervene. We have chosen to sample uniformly from the flat list of atomic attributes. One effect of this choice is that, with 108 bricks in Breakout, and each brick's having nine attributes, our search is dominated by

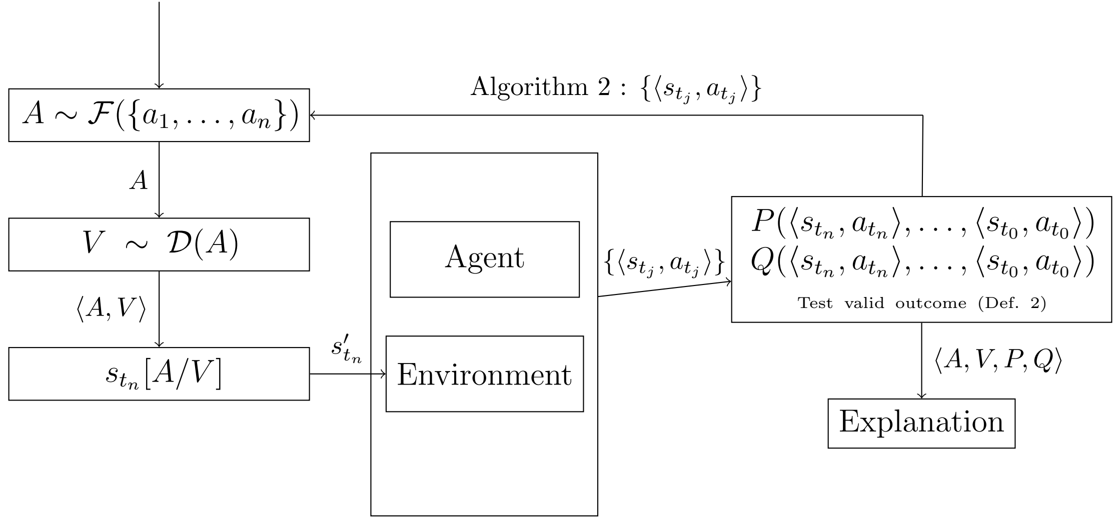


Figure 5.3: Diagram of a single run of the AutoExp system. Details of the iteration can be found in Algorithm 1.

Data: trace, a list of state-action pairs where the outcome was observed
Result: The mutated state that causes the counterfactual predicate to be true.

```
1  lag ← minimum window size to detect counterfactual
2  interventions_tried ← initialized empty set of interventions tried so far
3  possible_interventions ← generated list of possible mutation points for
     intervention
4  while lag < |trace| do
5  |   s ← state at position lag from the end of trace; while
   |     |interventions_tried| < |possible_interventions| do
6  |   |   intervened_state ← Algorithm 2 over possible_interventions,
   |   |     interventions_tried, and s
7  |   |   state_action_pairs ← play(agent, intervened_state, lag)
8  |   |   if counterfactual(state_action_pairs) then
9  |   |   |   return intervened_state
10 |   |   end
11 |   end
12 |   lag ← min(|trace|, lag * 2)
13 |   Clear interventions for interventions_tried
14 |   Reset possible_interventions
15 end
```

Algorithm 1: The top-level driver loop for automated experimentation.

features of bricks, most of which will likely have little impact on most outcomes. We have considered other options, e.g., sampling proportionally to the depth of the JSON, but this would be a premature optimization, given that there are other methods for changing the search space.

AutoExp provides two user customizations of the search space: the ability to *exclude* atomic attributes, and the ability to *include* user-defined composite variables. While, in the general case, A can be any subset of atomic attributes, we currently restrict A to be either a single atomic attribute, or the set of atomic attributes associated with the selected user-defined composite variable.

After the first iteration of Algorithm 1, AutoExp chooses the next attribute according to both $\mathcal{F}$ and Algorithm 2. It moves linearly through the list of attributes already selected, attempting to sample a new value for each. If sampling does not produce a new value, it moves on to the next attribute. If it exhausts the list of attributes, it samples a fresh one uniformly. Thus, AutoExp samples without replacement for its interventions, for each intervention state s_{t_n}.

5.4.1 Selecting the Appropriate Time to Intervene

Prior to running Algorithm 1, the user will have observed an outcome that they would like explained. Outcomes have a minimum window of time during which they can be measured. Outcomes are user-defined; we define eight for Breakout.

Figure 5.3 depicts AutoExp running in a loop. Each call of Algorithm 2 resets the state and tries another intervention. Since we expect a human to be involved in the loop, and since selecting the appropriate time to intervene is one of the known difficulties of our problem domain, we iteratively increase the length of time before the observed outcome, when selecting the intervention state.

Temporal precedence in repeated experimentation will need to be considered on two axes: time from the perspective of the observer or experimentation system, and

time from the perspective of environment dynamics. AutoExp searches for the appropriate t_i by first intervening at time $t_0 - 1$ and then doubling the number of time steps backwards if no suitable explanation can be generated, until the user-supplied window has been exhausted. The optimality of this search is well-known in, networking, AI, and complexity (cf. [76], [47], [105]). In the future we may also consider choosing a random number of timesteps, sampled from intervals of doubled size for each unsatisfactory explanation (as in randomized exponential backoff [85]).

Threats to Validity. There are two major threats to the validity of the above technique: (1) we may miss the critical window during backoff, and (2) we may still select an intervention that is not appropriate (e.g., transporting the paddle across the board). The former can be addressed via the aforementioned randomness of exponential backoff, and the latter can be addressed by the experimenter via user-defined interventions for user-defined variables. We discuss such variables in the next section.

Data: possible_interventions, a list of mutation points; s, the state on which we wish to intervene; interventions_tried, a set of the interventions already tried

```
1  for X in interventions_tried do
2  |   x' ← sample(X, s[X])
3  |   if x' is not ⊥ then
4  |   |   Create a new entry in interventions_tried for X and add x'
5  |   |   return A new state s′ such that such that s − s′ = {⟨X, s[X], x'⟩}
6  end
7  while |possible_interventions − interventions_tried| > 0 do
8  |   X ← random attribute from |possible_interventions − interventions_tried|
9  |   x' ← sample(X, s[x])
10 |   if x' is not ⊥ then
11 |   |   Create a new entry in interventions_tried for X and add x'
12 |   |   return A new state s′ such that such that s − s′ = {⟨X, s[X], x'⟩}
13 end
```

Algorithm 2: Searching for the next attribute and its value.

5.4.2 Defining Treatments and Outcomes

The block marked $V \sim \mathcal{D}(A)$ in Figure 5.3 depicts the second step of the AutoExp system: selecting new values (by Definition 3) for the appropriate attributes. Each variable shares a common interface called `Var` that requires a `sample` method. $\mathcal{D}$ refers to the distribution from which `sample` draws.

How we select the values of variables depends on the category of the variable; AutoExp provides users with the ability to define custom variables, and these custom variables do not behave in the same way as atomic attributes. As discussed in Section 5.3.2, there may be a representational mismatch between the atomic attributes that we can record and the variables on which we may actually want to intervene. Thus, we provide a taxonomy of the categories of variables for intervention.

5.4.2.1 Atomic variables

An atomic variable is just an atomic attribute (Definition 4). We will use the terminology "attribute" in the context of the experimentation environment as a machine. We will refer to it as a "variable" when we discuss it in terms of building a model for explanation. We will generally refer to all features, attributes, and variables simply as "variables" when we are discussing explanatory models or systems.

For any atomic variable X, we can sample from the marginal distribution of the values of X over sample data using the built-in capabilities of ToyBox. An atomic *intervention* simply sets the value of an atomic *attribute*:

Definition 9 (Valid atomic intervention) *Let $s[a] = v$ denote the value v of an atomic attribute a and let $s[a/v'] = s'$ denote the intervention of setting attribute a to have value v'. Then $s[a/v']$ is a valid atomic intervention iff $v' \neq v$ and $s[a/v'] \in \mathcal{C}_M$.*

We use the notations and conventions for substitution in logical formulae of Schöning [116].

5.4.2.2 Composite variables

We will sometimes be interested in variables that emerge from collections of atomic attributes; we refer to these as "composite variables." In Breakout, some examples include "the third column of the board" or "the top row color." Although any collection of atomic attributes can form a composite variable, we tend to think of them as more likely to be human-understandable concepts than atomic attributes. Furthermore, composite variables tend to be associated with "intelligent" behaviors in agents and could be useful for measuring generalization.

Composite variables may be further partitioned into two categories, depending upon whether the order matters, for the atomic attributes that define them.

Definition 10 (Valid composite variable intervention (Type 1)) *Let $A = [a_1, \ldots, a_n]$ be a composite variable and $s[A] = s[a_1, \ldots, a_n] = (v_1, \ldots, v_n) = V$. Then $S[A/V]$ is a valid composite variable intervention (type 1) iff*

$$\exists f_1, \cdots, f_n \left(s[A/V] = s[a_1/f_1(A), \ldots, a_n/f_n(A)] \right)$$

such that, for any $\forall i \in [1, n], s[a_i/f_i(A)]$ is a valid atomic intervention, and

$$\forall i, j \left(f_i \neq f_j \rightarrow s[a_i/f_i(A)][a_j/f_j(A)] = s[a_j/f_j(A)][a_i/f_i(A)] \right)$$

That is, for type 1 composite variables, we may apply the atomic interventions concurrently, or in any sequence.

There may be higher level concepts useful for explanation that a casual observer would be find unambiguous, but are in fact found to be under-specified when we are faced with encoding them. For example, distance between ball and paddle may seem unambiguous, but we need to specify the *type* of distance. Furthermore, there are

higher level concepts that are more qualitatively defined, where the mapping between the lower-level attributes may be tenuous, at best [126]. For example, in Breakout, we might categorize board configurations, and paddle and ball positions into "easy" and "hard," where the easy configurations are ones where scoring (i.e., clearing another brick) is easy (e.g., can be accomplished within a small window of time), and the hard configurations may result in an extended state sequence with no increase in score.

Definition 11 (Valid composite variable intervention (Type 2)) *Define A, V, and f_i as for Definition 10. Let π_k denote the function that maps the ith index of $[1, n]$ to the ith index of the kth permutation of the index set $\{1, \ldots, n\}$. Then $S[A/V]$ is a valid composite variable (type 2) iff $\exists f_1, \cdots, f_n \, (s[A/V] = s[a_1/f_1(A)] \cdots [a_n/f_n(A)])$ such that, for any $i \in [1, n]$, $s[a_i/f_i(A)]$ is a valid atomic intervention, and*

$$\exists k, k' \big(s[a_{\pi_k(1)}/f_{\pi_k(1)}(A)] \cdots [a_{\pi_k(n)}/f_{\pi_k(n)}(A)] \\ \neq s[a_{\pi_{k'}(1)}/f_{\pi_{k'}(1)}(A)] \cdots [a_{\pi_{k'}(n)}/f_{\pi_{k'}(n)}(A)] \big)$$

That is, for composite variables, some attributes values require that other variables' values *already be computed.* Therefore, composite attribute intervention cannot be performed concurrently.

Defining Composite Variables with the `Var` interface

Composite variables of both types are set the same way in AutoExp and are not currently differentiated from each other in terms of their API. Program analysis would reveal the difference: a data dependence graph for composite attributes would show each atomic attribute pointing into the state, but no flow between the attributes. The data dependence graph for composite variables would show at least one dependency (i.e., directed edge) between two atomic attributes.

While atomic variables rely on the underlying sampling method to select appropriate values for interventions, composite variables require user-defined sampling functions.

```python
def set(self, v: Tuple[int, int, int], g: Game):
  red, green, blue = v
  for avar in self.atomicvars:
    if avar.name.endswith('r'):
      avar.set(red, g)
    elif avar.name.endswith('g'):
      avar.set(green, g)
    elif avar.name.endswith('b'):
      avar.set(blue, g)
    else: assert False
```

Figure 5.4: Setter for top row color, a type 1 composite variable. All `Vars` have a fields `atomicvars` and `compositevars`, lists of the atomic or composite variables on which the variable depends. This variable's set of `atomicvars` is the list of RGB attributes for all of the bricks in the top row of the Breakout board.

Additionally, all `Vars` require setters, getters, and access to any atomic and composite variables on which they depend. These methods are trivial for atomic variables, but can be a bit more challenging for composite variables.

For example, in Figure 5.4, we have the code to set the top row of the board to be a particular color (RGB value). Each of the RGB values for each of the bricks in the row can be set independently: the set of atomic attributes that comprise this variable are an unordered set.

Contrast the setter in Figure 5.4 with how we set the x distance between the ball and paddle in Figure 5.5. For the x distance variable, not only do we have have an ordering over the atomic variables, but that ordering is actually a set of orderings because we randomly select which of the two atomic variables will be our independent random variable. We can see how RGB attributes are order independent in Figure 5.4.

Consider the dependence structure of these two setters in Figures 5.6 and 5.7.[11]

[11]In Figure 5.7, BT denotes a Bernoulli trial where the undirected edge on the left-hand side of the BT node is an atomic attribute that maps to 0, and the undirected edge on the right-hand side of the BT node is an atomic attribute that maps to 1. In this graph, another Bernoulli trial determines the *value* that the leaves of the graph take on, because we must decide between adding and subtracting the parent node from the distance are Bernoulli trials over the set $\{+, -\}$; once we know the operator,

```python
def set(self, v: int, g: Game):
  # randomly select which corevar will be set first
  random.shuffle(self.atomicvars)
  ivar, dvar = self.atomicvars
  # sample a value for the independent variable
  ivar_value = ivar.sample()
  # compute the two possible values of the dependent variable
  dvar_value_plus  = ivar_value + v
  dvar_value_minus = ivar_value - v
  # randomly select which of the two values we want
  dvar_value = random.choice([dvar_value_minus, dvar_value_plus])

  # set the two variables in the state object
  ivar.set(ivar_value)
  dvar.set(dvar_value)
```

Figure 5.5: Setter for horizontal distance between ball and paddle, a type 2 composite variable.

Both composite variables express possible constraints on the atomic variables. However, type 1 composite variables may change the joint probability distribution, but they will also preserve the structure of the dependencies of the underlying atomic attributes. Type 2 composite variables may not preserve that structure.

Composites variables may be composed of other composites variables. For example, suppose we have another variable that captures the y distance between ball and paddle, defined in an analogous way to our x distance variable. Then we can define a composite variable for the Euclidean distance between ball and paddle, as in Figure 5.8.

Thus, variables may be defined compositionally. By decomposing setting, getting, and sampling, we can define arbitrarily complex probability distributions for variables on an *ad hoc* basis, allowing users to focus on the domain, rather than the implementation details.

---

the value can be computed deterministically. We do not include this Bernoulli trial in Figure 5.7 because the Bernoulli trial affects only the value of the graph node, not the existence or orientation of its edges.

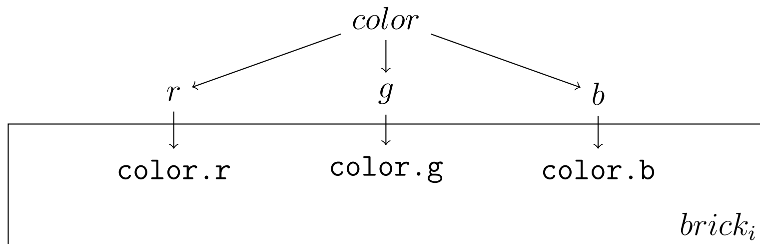


Figure 5.6: The data dependence graph for the top row color composite attribute. We have some input *color*, which is decomposed into its RGB values. We then deterministically set the RGB attributes of each brick instance. Note that all of the bricks on which we are intervening are set to have the same color and the order in which we set the bricks, as well as the order of the RGB values themselves, does not matter.

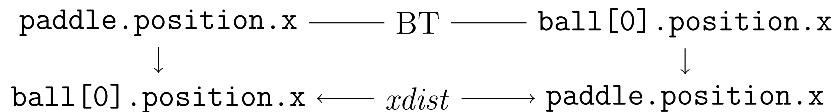


Figure 5.7: The data dependence graph for the x distance between ball and paddle. The input is a value for the x distance (*xdist*). We randomly select which of the ball or paddle x positions to set first. The other x position is then computed from the input distance and the position of the variable that was set first.

```
def get(self, g:Game):
  xdist = self._get_composite(XDistanceBallPaddle)
  ydist = self._get_composite(YDistanceBallPaddle)
  return (xdist.get(g)**2 + ydist.get(g)**2)**(1/2)

def set(self, v:int, g:Game):
  random.shuffle(self.atomicvars)
  ivar, dvar = self.atomicvars
  ivar_value = ivar.sample(ivar_name)
  # paddle must always be lower than the ball
  if 'ball' in ivar.name:
    dvar_value = v + ivar_value
  else:
    dvar_value = ivar_value - v

  ivar.set(ivar_value)
  dvar.set(dvar_value)
```

Figure 5.8: Setter and getter for Euclidean distance, a composite variable of other composite variables.

The separation of sampling from setting encodes an additional separation of concerns: the difference between encoding the distribution of the values of a variable vs. the causal structure that sets that value. The `set` method interacts with the distributions of the variables it manipulates only through those variables' `sample` method. Any additional randomness is over the topography of the causal graph.

Finally, note that for composite variables, we may need or want to define custom sampling methods. The default setting in AutoExp is to assume that the composite variable is a real-valued number, and to estimate density with a Gaussian kernel. Clearly this will not be an appropriate method for all numeric variables, especially important for variables where a small change in the variable's value does *not* imply a small change in the state space. For example, color is encoded via RGB values. A

```
def sample(self, g:Game):
  red   = Atomic(self.modelmod, 'bricks[0].color.r').sample(g)
  green = Atomic(self.modelmod, 'bricks[0].color.g').sample(g)
  blue  = Atomic(self.modelmod, 'bricks[0].color.b').sample(g)
  return (red, green, blue)
```

Figure 5.9: Example custom sampler for top row color, a composite variable (type 1). We can sample the red, green, and blue components of our composite "color" in any order, since they are independent.

small change in the integer representation of RGB will *always*[12] correspond to a small change in the R value because of the segmented nature of RGB encoding. Similarly, we can define the Breakout brick board configuration using a bit representation (i.e., an integer ranging from 0 to 2^{108} -1), where each bit corresponds to whether the brick is alive or dead. However, this encoding does not capture the spatial locality of brick liveness: it is not true that the board that corresponds to some value $x \in [0, 2^{108})$ has more bits in common with $x + 1$ than it does with $x + n; n \gg 1$; bits must be sampled independently.

That said, it may be desirable to set and get these encoded representations. However, we will want to write bespoke sampling methods for such variables. For example, Figure 5.9 shows how our top row color variable simply samples independently from each of its constituent atomic variables.

5.4.2.3 Behaviors

Behaviors are functions of states and actions over time. Behaviors may be evidence of higher-level strategies. Outcomes are, in the most general case, also functions of states and actions over time and so some outcomes are also behaviors (many outcomes ignore the action input, so they are not always behaviors). Outcomes are the predicate

[12]On little-endian systems, the lowest 8 bits of 24-bit RGB color will correspond to the 'R' component

form of behaviors (i.e., they detect whether or not a behavior has occurred). Therefore, behaviors can be either outcomes or explanatory variables. However, intervening to produce a behavior can be very challenging and is an instance of continuous intervention in time-series experiments.

AutoExp's Samplers Are an Approximation of the Desired Distributions

We learn the sampling distribution to sample from user-provided input data, which can be generated by having one or more agents play the game and recording the observed states. The details of the sampling procedure are built into ToyBox. There is a default function for learning these models, but it can be overridden as needed. If an experimenter wishes to use a different back end, they will need to implement the connection between that back end and the `Var` interface.

Note that, for any attribute X, AutoExp will use the default sampler. The default sampler assumes that X can be coerced to a real-valued number, and it learns a Gaussian kernel density function over the marginal distribution of input data (i.e., $P(X)$). To be consistent with our formal definition of counterfactual explanation, we actually want to be sampling from $P(X \mid \mathcal{V} - X)$. It is up to the experimenter to select the appropriate data or re-learn the density estimator for different values of $\mathcal{V} - X$.[13]

The primary threat to the soundness of using the default marginal distributions is the case where not only does the marginal have a higher probability than the conditional one (i.e., $P(X = x) > P(X = x \mid \mathcal{V} - X)$), but there also exists another x' that we should be sampling with higher probability, and *both x and x' change the*

[13] We expect the system to be fairly robust to misspecification. Because we have no expectation of completeness, it is acceptable to miss any value of X such that $P(X = x) < P(X = x \mid \mathcal{V} - X)$. Note that we mean only to say that it is acceptable to miss these cases in terms of our desire for sound or reasonable counterfactual explanations. It is absolutely worthwhile to improve the system so that it will sample the cases where $P(X = x) < P(X = x|\mathcal{V} - x)$. Indeed, these are the values of X that expect to be the most relevant explanations precisely because their co-occurrence (i.e., $P(X = x, \mathcal{V} - X)$) is so high.

outcome variable Y. That is, $P(X = x) > P(X = x')$ (we are more likely to sample the value x for X than x'), but $P(X = x \mid \mathcal{V} - X) < P(X = x' \mid \mathcal{V} - X)$ (x' is contextually more relevant).

5.5 Why we need OFAT Experimentation

AutoExp currently performs only OFAT interventions. However, we would like to be able to perform factorial experiments over both atomic and composite variables. At this time, we have identified three major challenges when attempting to run factorial experiments: mixing variables types, accounting for washed out interventions or downstream effects, and determining an efficient sampling procedure and/or stopping condition that spans the design space effectively.

5.5.1 Challenge: Mixing Variable Types

A single atomic intervention constitutes an OFAT experiment. When we mutate multiple atomic attributes concurrently, we may be trying to run a *factorial* experiment, which is more efficient than an OFAT experiment [17].

Ideally, atomic variables would be linearly independent: not only would we be able to vary them independently, but they would also minimally span the design space. Linear independence is the property that gives factorial experiments their power, allowing researchers to estimate an interaction effects between variables. However, clearly we cannot simultaneously set an atomic attribute X and a composite variable Y that depends on X; one will override the other *at intervention time*.[14]

---

[14] We highlight that this override occurs at intervention time to differentiate it from the washed out interventions of the second challenge, which are due to environment dynamics.

5.5.2 Challenge: Environment Dynamics

There may be unknown dependencies between atomic attributes. When we perform a factorial experiment, we are implicitly assuming the structure:

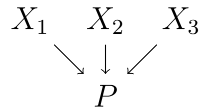


Figure 5.10: Factorial experiments imply a shallow DAG.

Here X_1, X_2, and X_3 are the attributes and P is the outcome. Because we are setting these attributes (i.e., applying the `do` operator to them), we sever any links from their causes (hence X_1, X_2, and X_3 have no parents). An implication of X_1, X_2, and X_3 being atomic attributes is that there are no observable mediators between X_1, X_2, and X_3, and P. Furthermore, by virtue of our intervention, we enforce *no collinearity between factors*, i.e., linear independence of the input values.

Unfortunately, inducing the above structure can actually be quite challenging. Because atomic attributes in software systems may be tied to low-level, mechanistic environment dynamics applied over time, we should not expect the above structure to hold after more than one time step. Consider the following structure:

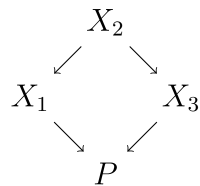


Figure 5.11: Dependencies between variables in system dynamics can cause inconsistencies between the counterfactual explanations produced by experiments and observational models.

Suppose X_2 is the x-velocity of the ball, X_1 is the direction, and X_3 is the speed. Suppose that Toybox sets X_1 and X_3 from X_2, and that the outcome is directly caused by X_1 and X_3. In an observational causal model, we can simulate setting X_1 via `do`.

However, in an experimentation system, if we try to set X_1, it will be immediately overridden by X_2.

One possible remedy for the challenge posed by variables that are consistent with Figure 5.11 is to have the experimentation system throw an error whenever a variable change is "washed out" after intervention (i.e., $|s - s'| > |M_\gamma(s, a) - M_\gamma(s', a)|$ for the valid intervention $s' = s[X/x']$) and issue a warning whenever a variable change causes downstream effects (i.e., $|s - s'| < |M_\gamma(s, a) - M_\gamma(s', a)|$).

If, after one time step, an intervention is washed out, we should remove it from the list of atomic attributes. The experimenter may have sufficient domain expertise to recognize a functional relationship between the washed out variable and a variable that causes downstream effects. In such a case, they can define a composite attribute that groups the two atomic attributes together, or remove the washed out atomic attribute from the list of eligible interventions. It is possible that, in the general case, there is some symmetry between washed out interventions and downstream effects that we could exploit.

One preliminary candidate definition for concurrent interventions is a set of interventions that have no effect on environmental dynamics, but could have an effect on the output of the policy:

Definition 12 (Environment Dynamics Invariant Intervention) *Given M_γ, $\pi_{\gamma'}$, valid observation s_0, and valid intervention f that modifies the nonempty set of atomic attributes $B \subset A$, we say that f is a* stable intervention *with respect to some trace $s_n, \ldots s_0$ iff, for any state s_i in the trace, intervened state $s'_i = s_i[B/f(A)]$, and action $a = \pi_{\gamma'}(s_i)$,*

$$|s_i - s'_i| = |M_\gamma(s_i, a) - M_\gamma(s'_i, a)| \rightarrow |s_i - s'_i| = |M^*_\gamma(s_i, a^*) - M^*_\gamma(s'_i, a^*)|,$$

where M^*_γ *denotes the repeated application of* M_γ *from the first argument, using the action sequence* a^* *produced by repeated application of the policy starting with the observed (control) state.*

The set of interventions that satisfy Definition 12 could form a factorial experiment.

5.5.3 Spanning the Design Space.

Factorial experiments with a small number of factors can perform simple random assignment for each factor independently and obtain coverage of the design space. In the presence of a large number of factors, we may want to perform a *fractional* factorial design, where we enumerate the combinations of concurrent treatments and sample from a *schedule* of interventions that exclude redundancies.

There are several features of our problem that make applying the factorial approach problematic:

1. Some of our factors are continuous-valued. We discretize these values online, so we would need to simulate sequential sampling. Because we do not use a fixed number of buckets, we would need to introduce a new hyper-parameter. Furthermore, discretization simply may not be appropriate here.

2. With over 1500 attributes, enumerating even a fractional factorial design leads to an intractable design space.

3. Factorial experiments are most appealing when the effect size of a combination of factors is greater than their individual effects. For a single run of our system, the only effect we are measuring is whether the outcome is false and the counterfactual is true. We will discuss the implications of this objective below.

We can address the first two concerns by instead framing counterfactual explanation as a Bayesian optimization problem [90]. From a distance, the approaches look similar: they are both black-box methods that perform a large number of experiments in a high

dimensional space, and have an adaptive component: for example, we use an adaptive discretization procedure (Appendix E) for selecting treatments, which resembles a space-filling experimental design. However, we do not have anything analogous to an acquisition function,[15] and are not explicitly computing posteriors.

Some of the difference between Bayesian optimization and this automated experimentation system can be explained by the fact that we have thus far discussed only how the system works for a single run; in the next section, we will delve into the implications of running many searches for explanations in parallel and explicitly connect that usage of our system with the counterfactual explanations and optimal counterfactual explanations described at the start of this document (Definitions 7 and 8).

5.6 Evaluation of AutoExp via ToyBox

All of the definitions and formalisms we have discussed thus far explicitly fix the random seed for both the agent and the environment. This facilitates our ability to "rewind and replay" and thus conduct repeated within-subjects experiments. The main threat we face is when the system finds a counterfactual that is a consequence of the random seed.

We run thirty concurrent instances of the automated experimentation system for every combination of agent and outcome below, over the atomic attributes. Some agents never observe an outcome or a counterfactual, and some searches exceed the time limit of twelve hours. All searches operate over 64 frame windows.[16]

Our objectives in this evaluation are to:

---

[15] For a black-box function f whose value we want to optimize, an *acquisition function* is another function that is cheaper to evaluate and used to selection the next point in the design/search space.

[16] Just under 3 seconds of normal game play, for a human player.

- Establish that AutoExp makes progress, and can find at least *some* changes in the environment that cause the factual to become false, and the counterfactual to become true.

- Test a variety of outcomes and qualitatively assess whether they will identify meaningful behaviors.

- Run a set of baseline evaluations on: two deterministic scripted agents that have very similar policies (i.e., the set of variables that control the decision making of one is a subset of the variables that control the decision making of the other), a simple scripted agent with a small amount of randomness, and a more complex agent that occasionally takes random actions, and has an internal (inaccurate) model of environment dynamics.

- Show how the AutoExp system can easily incorporate domain knowledge via constraints on atomic variables and the addition of composite variables.

- Run exploratory analyses on a selection of deep agents.

5.6.1 Simple, Interpretable, Rule-based Agents

We wrote four agents that play Breakout; three are variants of each other, designed to test edge cases in the system. The fourth agent follows a much more complex policy.

5.6.1.1 StayAlive Family

StayAlive and SmarterStayAlive functioned as baseline tests for our outcome definitions. As can be seen in Tables 5.1 and 5.2, they did not detect the test outcomes the vast majority of the time. We expected and desired this behavior, since we would prefer to detect an outcome when appropriate. Therefore, our deterministic agents did not run automated experimentation in the most cases.

Aim left	Aim right	Missed ball	Hit ball
*	*	*	**
Move same	Move opposite	Move toward	Move away
* 11	*	*	*
*** 19			

Table 5.1: Counterfactual explanations for the StayAlive agent: atomic attributes only, run over 30 random seeds for each of the eight outcomes. * This outcome was not observed during normal game play (i.e, no evidence found for behavior, violation of $P(t)$ in sentence 5.7). ** The counterfactual for this outcome was not observed during normal game play (i.e, no evidence found for behavior, violation of $Q(t)$ in sentence 5.7) *** The search timed out after 12h.

Aim left	Aim right	Missed ball	Hit ball
*	**	*	**
Move same	Move opposite	Move toward	Move away
**	*	**	*

Table 5.2: Counterfactual explanations for the SmarterStayAlive agent. Experiments run over atomic attributes only, for 30 random seeds for each of the eight outcomes. * This outcome was not observed during normal game play (i.e, no evidence found for behavior, violation of $P(t)$ in sentence 5.7). ** The counterfactual for this outcome was not observed during normal game play (i.e, no evidence found for behavior, violation of $Q(t)$ in sentence 5.7).

Tables 5.3 and 5.4 shows the strong and weak counterfactual explanations found for the StayAliveJitter agent, respectively. For the strong counterfactual explanations, *many* runs timed out before finding any explanations. For the weak explanations, many trials could not find the critical time period where a single change in the action an agent chooses would cause a change in outcome.

The only outcome with a promising counterfactual explanation is `paddle.position.x` for AimRight. We will look into validating this explanation momentarily; for now, let us consider: (1) the number of long-running trials that had to be terminated, and (2) the low-frequency explanations that the system produced for some outcomes.

Because StayAliveJitter's policy has a stochastic element to it, it will occasionally choose a different action due to that stochasticity, rather than the change made to the environment. MoveOpposite has the most egregious incidence of this, where over half of the trials returned produced an explanation that was likely due to chance. Since we *know* that StayAliveJitter does not use any of atomic attributes under MoveOpposite in Table 5.4 to make its decisions, we know that this outcome is fairly sensitive to random actions. Future extensions of this system might perform preliminary diagnostics to estimate a prior on the outcome being true.

Since the results of this system are exploratory, and since agents may be stochastic, we will run only confirmatory analysis for those counterfactual explanations that have frequency higher than one. For StayAliveJitter, this means that there is only one candidate explanation: `paddle.position.x` for AimRight. Since we know the agent's policy is to attempt to align the paddle under the ball, it makes sense that intervening on the paddle's x position would change the outcome. We are now left to decide whether this is a meaningful intervention, which we can do either by inspection of the states or by, e.g., computing the probability of observing an outcome under control and intervention. Table 5.5 gives the outcome frequencies for each of the control and intervention states. Figure 5.12 depicts the treatment and control states, for visual comparison.

Considering the data of Table 5.5, as well as the large number of unfinished trials and observed-only-once counterfactual explanations of Table 5.4, we may want to try to narrow down the search space. We know that there is only a handful of variables used by the StayAliveJitter agent—fewer than 1% of all possible atomic attributes. We can reduce the search space with a small amount of effort. Furthermore, we might note that the specific values of the paddle x position may not matter; it could be the case that a composite variable such as distance between ball and paddle is more meaningful. Therefore, we re-run the system for StayAliveJitter with one composite

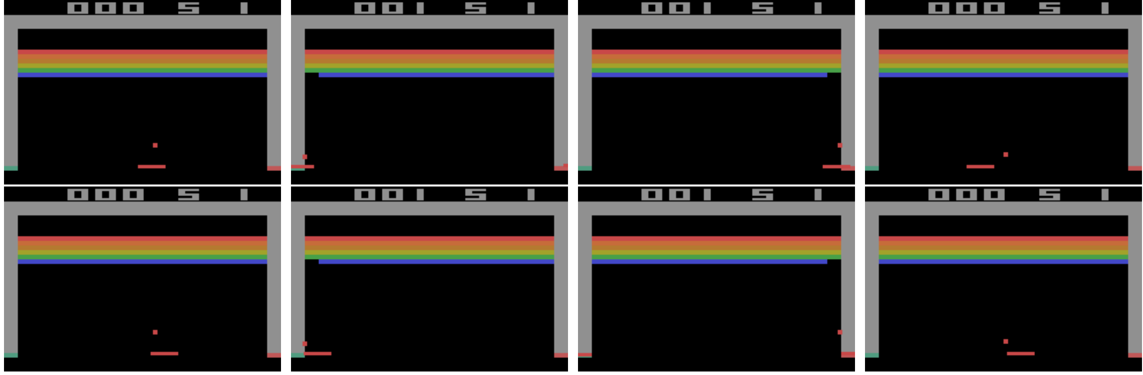


Figure 5.12: **Top:** Control states at intervention time for aim right. **Bottom:** Treatment states at intervention time; after forward simulating for three steps, the agent is detected as aiming left. All paddle x positions values were automatically chosen by AutoExp.

variable (L2 distance between ball and paddle) and exclude several brick attributes. Thus, the total number of attributes drops from over 1500 to below 1000. We then run automated experimentation for the Aim right outcome with a 6h timeout.

After running our constrained set, we found that only 2 of the 30 trials terminated within the 6h window. The two trials that found counterfactual explanations selected `paddle.position.x` and `l2dist_ball_paddle`.

Finally, we note that, even with the reduced attribute set, the system does not increase its look-back. This is because the outcome was identified early in game play, so the window did not allow further look-back. Since the counterfactual was frequently identified later in game play, to illustrate the efficacy of our look-back mechanism, we swapped the factual and counterfactual and ran the system on a further reduced attribute set and added two more composite variables. Table 5.6 describes our results.

5.6.1.2 Target

Since the results of the Target agent are quite similar to the StayAliveJitter, we will not discuss them here; see Appendix F. The primary result we note is that automatically detecting "interesting" explanations for the Target agent is quite challenging. We

believe this is due to two reasons: (1) the agent uses “interesting” composite variables internally that depend on “uninteresting” atomic variables, and (2) the agent only uses these variables in very specific (and possibly low-frequency) contexts. Therefore, the probability of discovering an explanation is low, and when we do discover explanations, they are either Level 2 (due to outcome dynamics) or Level 1 (due to randomness).

5.6.1.3 Performance

Figures 5.13 and 5.14 give timing information for the number of interventions attempted for both the StayAliveJitter and Target agents. Given the size of the attribute set and the fact that a large number of runs timed out, we would generally recommend an iterative search cycle, where researchers run more concurrent trials for a shorter period of time and winnow down the set of attributes.

5.6.2 Deep Agents

We ran our evaluation on 10 agents trained using OpenAI's second proximal policy optimization algorithm implementation (PPO2 [117, 98]). Since we are evaluating AutoExp as an exploratory tool, the choice of algorithm family was arbitrary.[17] We ran 30 trials for each of these agents from our ToyBox experiments on the complete set of atomic attributes. 195 agents timed out after 12 hours. Fifteen found counterfactual explanations via `paddle.position.x`, three found counterfactual explanations via `ball[0].position.x`, and another three found counterfactual explanations via `ball[0].velocity.x`. Forty-three trials found explanations that were not replicated (i.e., frequency of 1), and all but one of these were brick attributes (the lone exception being `paddle_speed`). We were surprised that none of the visual attributes of the bricks (e.g., size, color, position, etc.) had a frequency higher than one.

[17] There are several families of deep RL algorithms that differ in the functions they use to evaluate the agent's policy. Policy-gradient is one such family; training algorithms from this family treat the policy as a differentiable function of state and reward, and seek to find the policy that maximizes reward.

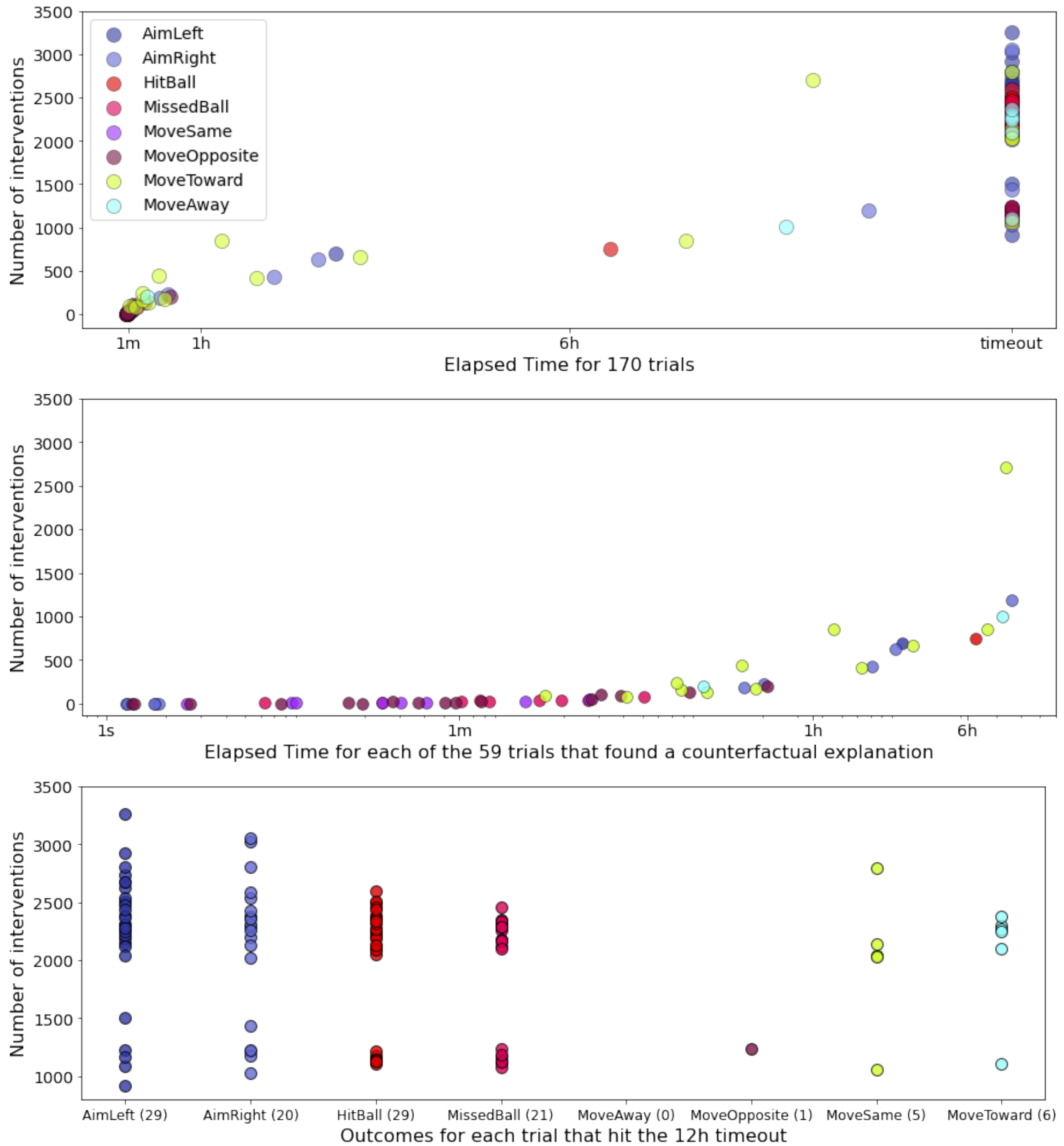


Figure 5.13: **Top:** Scatter plot of time in seconds vs. number of interventions performed for StayAliveJitter. Most searches did not terminate; at the 12h timeout, there was high variance in the number of interventions performed so far. The number of interventions performed is likely inversely proportional to the number of real-valued/floating point attributes sampled. **Middle:** Scatter plot on a log-scale for time, for the trials that successful found counterfactual explanations for the StayAliveJitter agent within the 12h timeout. **Bottom:** Scatter plot of the number of interventions performed during the trials that timed out, for StayAliveJitter.

5.7 Findings and Conclusions

For strong counterfactual explanation, we were pleased that AutoExp was able to find reasonable Level-3 (environment dynamics) and Level-2 (outcome) explanations. Weak counterfactual explanation produced far fewer explanations; however we designed these agents to test AutoExp's sensitivity to random actions. We will continue to improve AutoExp's capabilities at ruling out Level-4 (random) explanations.

Unfortunately, none of the agents we designed were appropriate for Level-1 (policy) explanations (with the possible exception of our deep agents). Future testing of the AutoExp system would include designing agents and outcomes in tandem, such that the only appropriate explanations are Level-1 (policyI).

Aim left		Aim right		Missed ball		Hit ball	
* * *	(14)	balls[0].position.y	(20)	* * *	(12)	* * *	(30)
balls[0].position.y	(7)	* * *	(7)	balls[0].velocity.x	(2)		
paddle.position.x	(5)	paddle.position.x	(2)	bricks[0].size.y	(1)		
balls[0].position.x	(2)	bricks[3].size.y	(1)	bricks[7].color.r	(1)		
bricks[64].position.y	(1)			bricks[14].color.g	(1)		
bricks[94].row	(1)			bricks[30].position.x	(1)		
				bricks[33].position.y	(1)		
				bricks[34].size.x	(1)		
				bricks[37].position.x	(1)		
				bricks[60].position.x	(1)		
				bricks[60].size.x	(1)		
				bricks[71].points	(1)		
				bricks[80].position.x	(1)		
				bricks[83].color.g	(1)		
				bricks[90].position.x	(1)		
				bricks[98].position.x	(1)		
				bricks[100].color.b	(1)		
				bricks[105].size.y	(1)		

Move same		Move opposite		Move toward		Move away	
**	(18)	*	(23)	**	(23)	*	(26)
*	(2)	bricks[3].color.b	(1)	*	(1)	bricks[13].size.x	(1)
bricks[9].size.y	(1)	bricks[5].size.y	(1)	bricks[16].position.y	(1)	bricks[54].size.x.	(1)
bricks[30].size.x	(1)	bricks[29].row	(1)	bricks[35].position.y	(1)	bricks[77].alive	(1)
bricks[40].color.a	(1)	bricks[61].size.x	(1)	bricks[84].size.x	(1)	bricks[82].position.x	(1)
bricks[55].size.x	(1)	bricks[63].size.y	(1)	* * *	(1)		
bricks[91].size.y.	(1)	bricks[80].size.x	(1)				
bricks[94].size.y	(1)	bricks[107].position.y	(1)				
bricks[77].position.x	(1)						
bricks[83].size.x	(1)						
bricks[84].size.x	(1)						
bricks[107].size.y	(1)						

Table 5.3: Strong counterfactual explanations for the StayAliveJitter agent, with agent randomness controlled. Experiments run over atomic attributes only, for 30 random seeds for each of the eight outcomes. * This outcome was not observed during normal game play (i.e, no evidence found for behavior, violation of $P(t)$ in sentence 5.7). ** This outcome was not observed during normal game play (i.e, no evidence found for behavior, violation of $Q(t)$ in sentence 5.7). *** The search timed out without producing any counterfactual explanations after 12h of search.

Aim left		Aim right		Missed ball		Hit ball	
†	(28)	†	(21)	bricks[104].size.x	(2)	†	(30)
bricks[39].color.g	(1)	bricks[0].size.x	(1)	bricks[45].size.x	(1)		
bricks[70].position.y	(1)	bricks[3].position.x	(1)	bricks[58].position.y	(1)		
		bricks[24].size.y	(1)	bricks[74].color.b	(1)		
		bricks[37].color.r	(1)	bricks[76].size.x	(1)		
		bricks[75].size.x	(1)	bricks[80].size.x	(1)		
		bricks[90].size.x	(1)	bricks[84].size.y	(1)		
		bricks[106].position.x	(1)	bricks[85].size.x	(1)		
		bricks[106].position.y	(1)	bricks[91].position.x	(1)		
		paddle_speed	(1)	bricks[101].size.y	(1)		

Move same		Move opposite		Move toward		Move away	
**	(11)	*	(12)	**	(21)	*	(22)
***	(5)	**	(1)	***	(6)	bricks[23].position.x	(1)
*	(2)	***	(1)	*	(1)	bricks[36].color.r	(1)
bricks[4].size.x	(1)	bricks[4].color.g	(1)	bricks[29].position.x	(1)	bricks[59].size.y	(1)
bricks[5].size.y	(1)	bricks[11].size.x	(1)	bricks[107].color.g	(1)	bricks[64].size.y	(1)
bricks[7].size.x	(1)	bricks[20].position.y	(1)			bricks[69].position.x	(1)
bricks[9].position.x	(1)	bricks[25].position.x	(1)			bricks[85].row	(1)
bricks[20].size.x	(1)	bricks[26].color.a	(1)			bricks[87].position.y	(1)
bricks[35].size.x	(1)	bricks[35].position.y	(1)			bricks[92].size.y	(1)
bricks[45].size.y	(1)	bricks[45].size.y	(1)				
bricks[49].size.y	(1)	bricks[46].size.x	(1)				
bricks[53].position.x	(1)	bricks[54].size.x	(1)				
bricks[59].size.x	(1)	bricks[76].size.x	(1)				
bricks[69].row	(1)	bricks[76].size.y	(1)				
bricks[76].position.x	(1)	bricks[84].position.x	(1)				
		bricks[90].position.x	(1)				
		bricks[101].size.y	(1)				
		bricks[107].color.b	(1)				
		bricks[107].size.y	(1)				

Table 5.4: Counterfactual explanations for the StayAliveJitter agent. Experiments run over atomic attributes only, for 30 random seeds for each of the eight outcomes. * This outcome was not observed during normal game play (i.e, no evidence found for behavior, violation of $P(t)$ in sentence 5.7). ** This outcome was not observed during normal game play (i.e, no evidence found for behavior, violation of $Q(t)$ in sentence 5.7). *** The search timed out without producing any counterfactual explanations after 12h of search. † No single change in action at any time in the window could produce the counterfactual.

Seed	Control	left	right	up	Treatment	left	right	up
1350332	128.0	0	5%	60%	139.8	6%	0	5%
6879583	8.0	0	19%	55 %	23.8	56%	0	0
7634392	224.0	0	11%	56%	240.3	91%	0	0
9660110	100.0	0	16%	0	136.0	60%	0	0

Table 5.5: Outcome frequencies computed over 100 random seeds, for each of the possible aiming outcomes, for the trials that found `paddle.position.x` to be a counterfactual explanation. Although we tested only the Aim right/Aim left outcome pairs, we included Aim up for reference.

Atomics	Composites
balls[0].velocity.x	l2_dist_ball_paddle
balls[0].velocity.y	top_row_color
paddle.velocity.x	board_config
paddle.velocity.y	
ball_radius	
is_dead	
level	
lives	
paddle_speed	
paddle_width	
reset	

Property	Count	Iteration
l2dist_ball_paddle	9	1
paddle_width	8	1
ball_radius	20	1
top_row_color	5	1
balls[0].velocity.x	7	1
level	1	1
paddle_speed	17	1
balls[0].velocity.y	4	1
lives	1	1
l2dist_ball_paddle	1	2

Table 5.6: Outcome AimLeft interventions, run for StayAliveJitter, over a constrained set of interventions to illustrate the look-back feature described in Section 5.4.1. **Left**: The intervention set. **Right**: Summary information for the interventions actually performed and analyzed. This run completed in about 5 minutes and produced a counterfactual explanation via the composite variable that defines the L2 distance between the paddle and the ball.

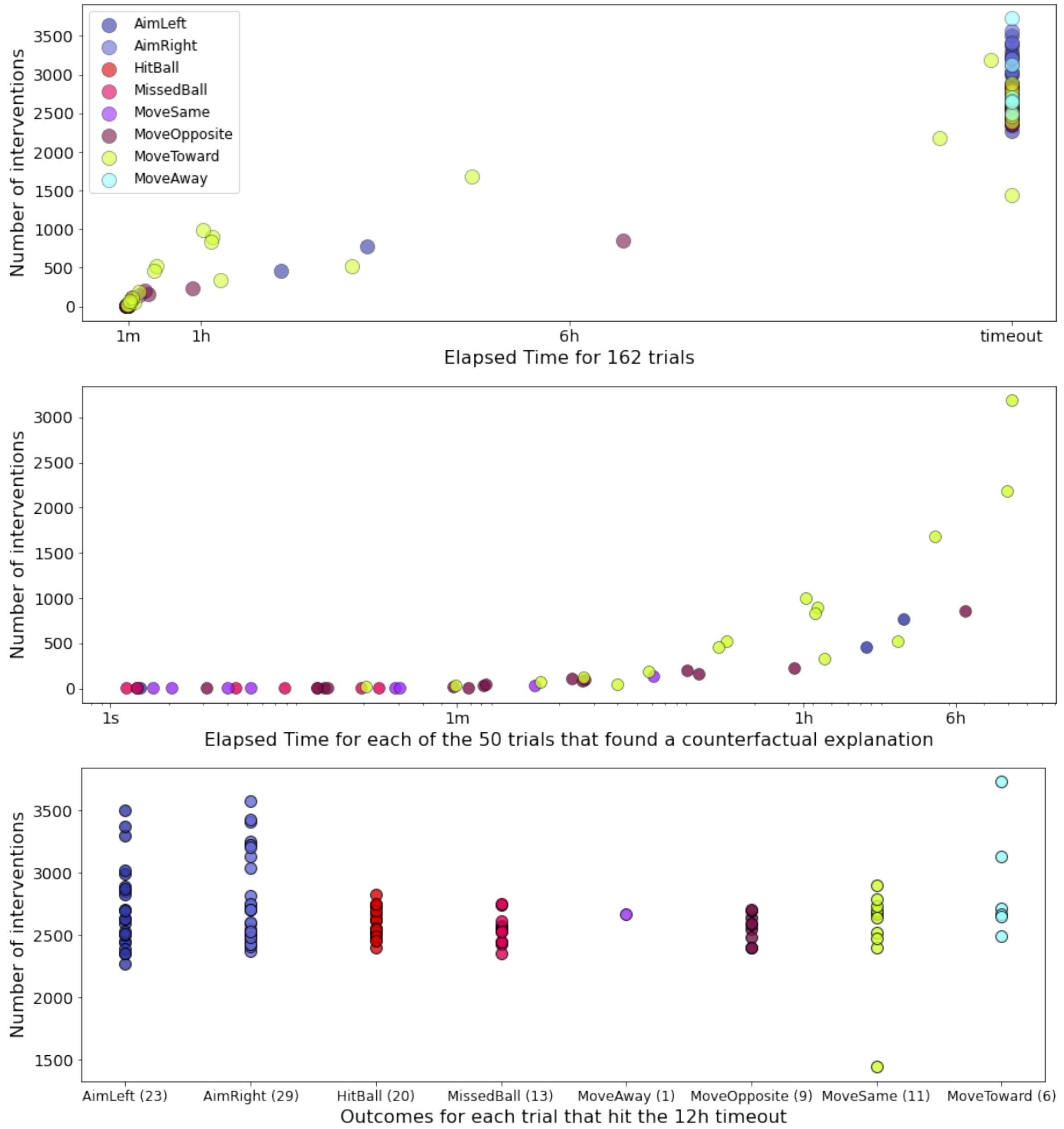


Figure 5.14: **Top:** Scatter plot of time in seconds vs. number of interventions performed for the Target agent. **Middle:** Log-scale Scatter plot of number of interventions performed before our system found a counterfactual explanation, for the trials that completed before the 12h time limit. **Bottom:** Scatter plot of the number of interventions performed at the time that these trials were canceled due to timing out.

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

In this dissertation, we characterized experiments as computer programs that can be analyzed statically, defined a framework for testing autonomous agents as software, and presented a method for using automated experimentation for generating explanations. We have shown how experimentation traditions that were previously disparate (e.g., field experiments vs. simulation experiments) begin to converge when the subject under test is effectively a black box, such as a person or an autonomous software agent, and when that subject interacts with a complex, but intervenable or configurable software system.

6.1 Findings

We designed a static analysis tool (PlanAlyzer) that correctly identifies threats to the validity of statistical conclusions for a subset of between-subjects experiments. We analyzed PlanAlyzer on a large corpus of real-world experiments conducted at Facebook. We found that that the types of mistakes that real-world users make are characteristic of experts in experimental design who may not be programmers, due to an abundance of errors related to type mismatches, dead code, and similar code smells. We used fault-injection to evaluate PlanAlyzer's ability to catch errors that programmers who are not experts in experimental design might make and find that PlanAlyzer identifies a large percentage of these injected errors (92% precision and recall).

PlanAlyzer also produces *contrasts* for the analyses of causal effects, for a subset of experimental designs. The state of the art for identifying contrasts is manual inspection. Using a corpus of manually-identified contrasts, we found that PlanAlyzer is able to identify a large subset of manually identified contrasts (82%).

We then turned our attention to within-subjects experiments, where the experimental subject is a black-box autonomous agent. In order to test hypotheses about such agents, we developed a white-box testing environment: ToyBox. ToyBox clones standard deep reinforcement learning benchmarks, allowing for unprecedented intervention capabilities with no performance cost. ToyBox facilitates intervention by modeling each game fully and making this state transparently readable and writable by users via a JSON ontology. Additionally, we designed a testing framework on top of ToyBox and Python's standard `unittest` framework in order to facilitate rapid behavioral testing for deep autonomous agents. We used our framework to test a broad range of hypotheses and prior assertions about deep agents' behaviors and find that some popular assertions about intelligent behavior are unfounded. ToyBox has been used to support research into the utility of saliency maps for explaining deep agents' behavior [5].

We have also used ToyBox as an example simulation environment for the development of an automated experimentation system (AutoExp) which helps to explain agent behavior. Our AutoExp system leverages ToyBox's high performance capabilities, as well as its automatically exported ontology, to navigate a large search space of interventions. AutoExp differs from prior work in the automated experimentation space in three major ways: (1) its purpose is exploratory, for finding counterfactual explanations for potentially temporally-extended outcomes, rather than for optimization; (2) it operates over time-systems where the experimenter has flexibility in choosing the time to intervene; and (3) it can operate in a relational environment, where the

most useful variables for explanation may be functions of attributes, computed over a collection of objects.

We first test AutoExp adversarially on scripted agents with known causal mechanisms for their decision making. We show that AutoExp makes progress in localizing causes for outcomes, even when the search space is very large. We characterize a hierarchy of explanations and evaluate AutoExp on agents that have been designed with particular explanatory variables in mind. Finally, we highlight how AutoExp can be used interactively for exploratory data analysis by iteratively analyzing the behavior of agents, and by searching for explanations of apparently high-level behavior.

6.2 Conclusions

Through our findings we have shown that there are opportunities for system and programming language design for digital experimentation. We have shown how processes that were previously only manual can be automated (PlanAlyzer), or semi-automated (AutoExp), and have discussed how designing systems with experimentation in mind can not only increase the throughput of experimentation, but also answer questions that cannot be answered otherwise (ToyBox, Atrey et al [5]).

6.3 Future Work

This dissertation introduces novel perspectives on experimentation in software systems. There are several topics where further inquiry would be fruitful.

We evaluated PlanAlyzer using a corpus of real PlanOut scripts. Ideally, PlanAlyzer would be: provably correct, extended to work on a broader range of experimental designs, and empirically evaluated on a larger corpus of real-world scripts.

In order for PlanAlyzer to be provably correct, we would first need a soundness theorem for software-defined experiments. One possibility is to use *identifiability*—i.e., the property that, for some true property of the population (e.g., ATE), it is possible to learn the true value of that property from a potentially infinite dataset[1]— as our soundness theorem. One major challenge for using identifiability as a soundness theorem is that the study of identifiability, like the study of experimental designs and causal inference, arises from parallel research traditions. Recent work on developing a unified theory of identification could aid in the development of a soundness theorem for PlanOut programs [10].

PlanAlyzer could be extended to consider estimators beyond ATE or CATE, as well as provide more precise contrast sets, if the PlanOut framework were to include both hypotheses and models of the population being testing. Evaluation of such models would require human-verified ground-truth data for evaluation. We face two major challenges in the acquisition of such data: (1) *security* considerations from firms donating their scripts, and (2) *privacy* considerations for any user data that might be used to validate the resulting statistical inference. We have thus far not considered statistical inference because we have not had access to user data, but automating statistical inference is a logical extension of PlanAlyzer, especially over iterated experimentation [6].

Future work on and with ToyBox. ToyBox's core suite contains three games, chosen for their difference in game play style, with the expectation that agents that learn to play different styles, learn different strategies. Recent work has shown that the complete Atari ALE benchmark contains redundancies [8]. While we have not

[1] From Manski [84]: "Studies of identification seek to characterize conclusions that could be drawn if one could use the sampling process to obtain an unlimited number of observations. Studies of statistical inference seek to characterize the generally weaker conclusions that can be drawn from a finite number of observations." Given the use-case of PlanOut as an experimentation language for large Internet firms, where we expect large sample sizes for experiments, identifiability is the obvious choice for a soundness theorem.

advocated using ToyBox for benchmarking purposes, its ability to produce out-of-sample game states means that it could be used to generate such a suite. In the future, we look forward to designing such a benchmark for the purpose of evaluating behavioral properties of agents [100].

The ToyBox testing system is currently integrated with the Python `unittest` framework. The behavioral testing of ToyBox differs from traditional unit tests insofar as it is statistical in nature. Consequently, statistical suites and counterfactual testing offer a better analogy. ToyBox environments have been designed as a model of more general *intervenable* and *relational* environments. These intervenable environments differ from classic notions of testable white-box software because the variables of interest that we wish to test may not be directly manipulable (due to software limitations). Future work might run a continuous data collection program, and use causal inference to test hypotheses that would otherwise rely on the intervention of variables that cannot be directly manipulated in ToyBox or comparable environments (e.g., Starcraft II [138]).

The AutoExp system is a prototype for automated experimentation for explanation. Future work would apply AutoExp to environments beyond ToyBox. Other complex game environments (such as Starcraft II) would be suitable, but introduce challenges, due to their greater environmental complexity and the inability to intervene at arbitrary time points.

Another direction of research arising from the AutoExp system would be on the encoding of composite variables. Rather than have them be written as Python subclasses, composite variables could instead be encoded in a probabilistic programming language, serving as a bridge between experimentation and observational models of causal inference.

Experimentation, particularly digital experimentation, will be at the heart of many research fields in the future, and the findings in this dissertation can help guide the development of systems to further future research.

APPENDIX 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 [40]. 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.

factor A variable that may be manipulated or intervened upon for the purposes of experimentation.

factor level The value that a factor takes on.

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: $Y^{(\text{treatmentA=a, treatmentB=b, etc.})}$.

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 variables (factors) or collection of variables (factors) being manipulated, or the value that that variable takes on. In Figure 3.1, this corresponds to non-empty subsets of `dynamic_policy`, `max_bitrate` , either variable individually, or the values that either of these may take on, e.g. { `dynamic_policy` = false, `max_bitrate` = 400 } or { `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.

APPENDIX B

PLANOUT SYNTAX

Figure B.1 gives PlanOut's concrete syntax. We highlight important language features and execution details below.

Important Language Features. As a DSL built by domain experts, PlanOut implements functionality relevant only 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) [104, 141, 87, 49, 50]. 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 2.1. 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 [75, 73]. Readers interested in the specifics of PlanOut's hashing method for scaling concurrent experiments can refer to the paper [7]; it is not relevant to PlanAlyzer's analyses.

$$prog ::= \langle stmtl \rangle \tag{B.1}$$

$$stmtl ::= \epsilon \mid \langle stmt \rangle \langle stmtl \rangle \tag{B.2}$$

$$stmt ::= \langle ite \rangle \mid \mathbf{id_i} = \langle expr \rangle; \mid \mathbf{return}\ \langle expr \rangle; \tag{B.3}$$

$$ite ::= \mathbf{if}\ (\langle expr \rangle)\ \{\ \langle stmtl \rangle\ \} \tag{B.4}$$

$$\mid \mathbf{if}\ (\langle expr \rangle)\ \{\ \langle stmtl \rangle\ \}\ \mathbf{else}\ \langle ite \rangle \tag{B.5}$$

$$\mid \mathbf{if}\ (\langle expr \rangle)\ \{\ \langle stmtl \rangle\ \}\ \mathbf{else}\ \{\ \langle stmtl \rangle\ \} \tag{B.6}$$

$$bop ::= + \mid * \mid \% \mid / \mid == \mid < \mid > \mid \&\& \mid || \tag{B.7}$$

$$expr ::= \langle value \rangle \tag{B.8}$$

$$\mid \mathbf{null} \tag{B.9}$$

$$\mid \mathbf{id_i} \tag{B.10}$$

$$\mid \mathbf{id_i}(\mathbf{id_i} = \langle expr \rangle, \ldots, \mathbf{id_i} = \langle expr \rangle) \tag{B.11}$$

$$\mid [] \mid [\langle expr \rangle, \ldots] \mid @\{\} \mid @\{\mathbf{id_i} : \langle expr \rangle, \ldots\} \tag{B.12}$$

$$\mid \langle expr \rangle [\langle expr \rangle] \tag{B.13}$$

$$\mid !\ \langle expr \rangle \mid \mathbf{length}(\langle expr \rangle) \mid \langle expr \rangle\ \langle bop \rangle\ \langle expr \rangle \tag{B.14}$$

$$\mid \mathbf{coalesce}(\langle expr \rangle, \langle expr \rangle) \tag{B.15}$$

$$\mid \mathbf{randomInteger}(\mathbf{min} = \langle expr \rangle, \mathbf{max} = \langle expr \rangle, \mathbf{unit} = \langle expr \rangle) \tag{B.16}$$

$$\mid \mathbf{randomFloat}(\mathbf{min} = \langle expr \rangle, \mathbf{max} = \langle expr \rangle, \mathbf{unit} = \langle expr \rangle) \tag{B.17}$$

$$\mid \mathbf{bernoulliTrial}(\mathbf{p} = \langle expr \rangle, \mathbf{unit} = \langle expr \rangle) \tag{B.18}$$

$$\mid \mathbf{uniformChoice}(\mathbf{choices} = \langle expr \rangle, \mathbf{unit} = \langle expr \rangle) \tag{B.19}$$

$$\mid \mathbf{weightedChoice}(\mathbf{weights} = \langle expr \rangle, \mathbf{choices} = \langle expr \rangle, \mathbf{unit} = \langle expr \rangle) \tag{B.20}$$

$$\mid \mathbf{sample}(\mathbf{n} = \langle expr \rangle, \mathbf{choices} = \langle expr \rangle, \mathbf{unit} = \langle expr \rangle) \tag{B.21}$$

Figure B.1: PlanOut's concrete syntax. Although not depicted here, all random operators have an additional optional argument **salt** for manipulating random assignment. Production B.1 defines a PlanOut program. Production B.7 defines the binary operators. Production B.8 defines a number, boolean, or string value. Production B.10 defines a reference, which can be external. Production B.11 defines external function calls. Production B.12 defines list and map lookup. In production B.15, if the first argument is not null, then return it; return the second argument otherwise.

APPENDIX C

DETAILED CHARACTERIZATION OF THE PLANOUT-A CORPUS

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. 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 [120, 131]. 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=userid)` 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 `userid * 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.

APPENDIX D

BREAKOUT SCRIPTED AGENT GRAPHS

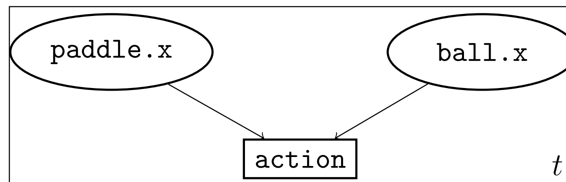


Figure D.1: The data dependency graph at time t for the StayAlive agent captures the causal mechanism of the agent's decision making as depicted in Equation 5.2.

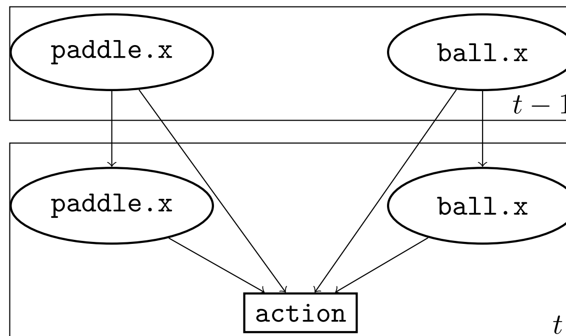


Figure D.2: Data dependency graph for agent StayAliveJitter. Not pictured: action is sometimes via an external source of randomness.

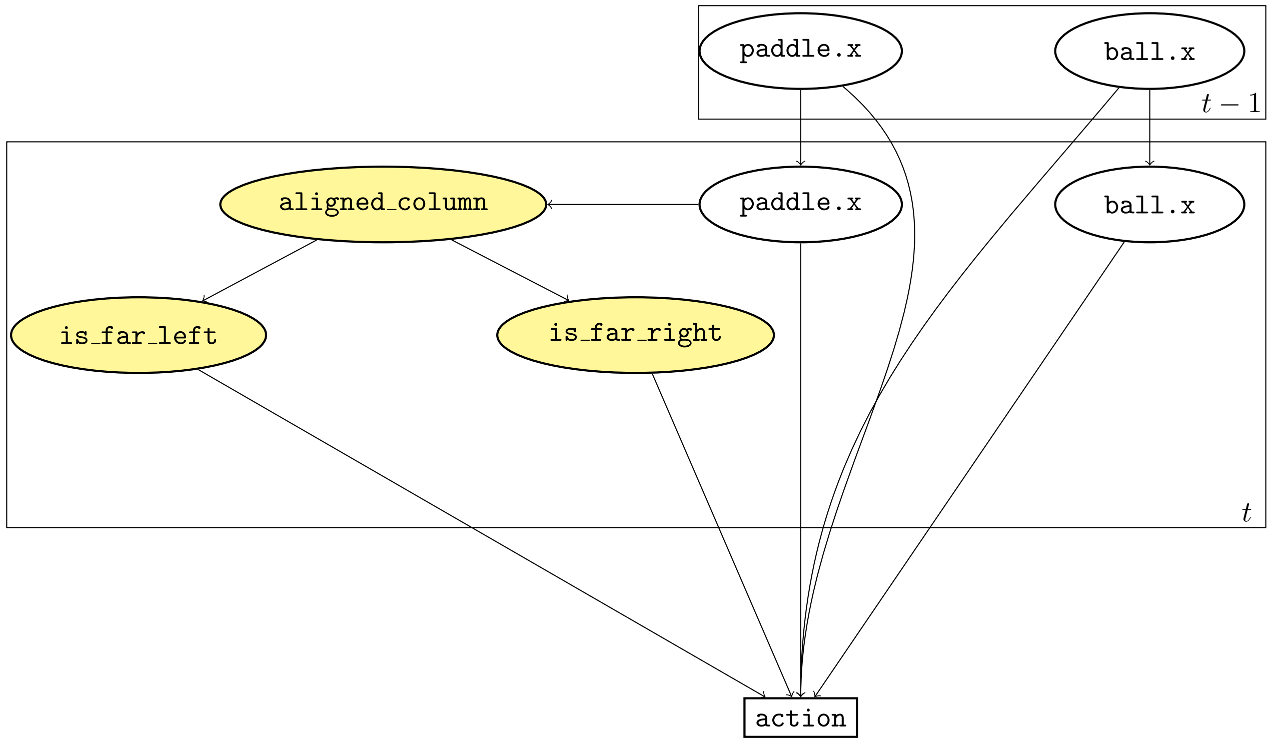


Figure D.3: The data dependency graph for the SmarterStayAlive agent. The shaded yellow circles denote internal higher-level concepts that the agent uses to decide how to act. These variables may be useful for explanation. Note that, if we did not know that these existed, but guessed, our measurement/calculation of them from state information would be faithful to the underlying representation (i.e., there is no ambiguity), thus avoiding some of the issues with higher level variables discussed in [**?**].

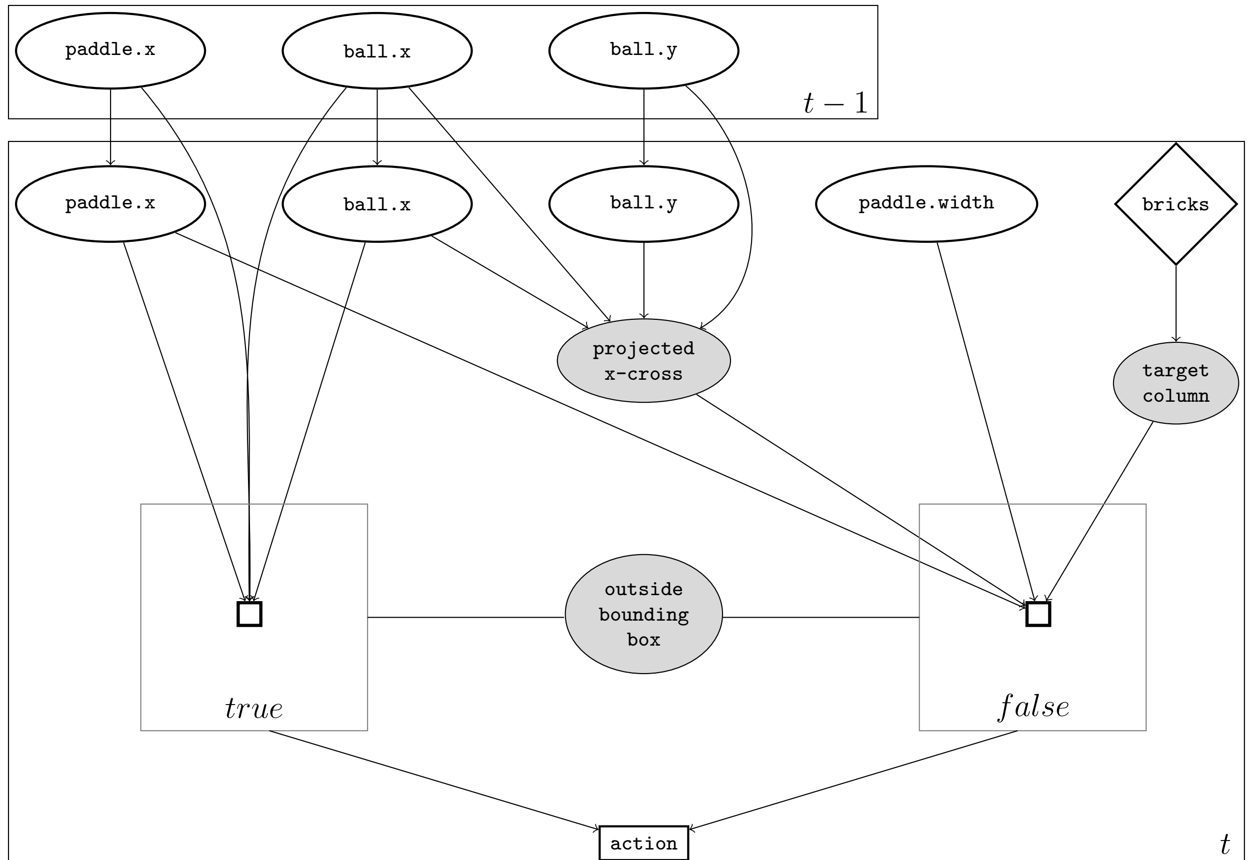


Figure D.4: A causal model of the Target agent's decision making. We use notation from gates that encode context-specific independence (or even determinism) [86, **?**] to indicate the "gating" mechanism of the bounding box.

APPENDIX E

SAMPLING CONTINUOUS ATOMIC ATTRIBUTES

Input : X, the attribute we want to mutate and x, its value before mutation
Data : interventions_tried, a map of attributes to a list of their values already tried.
Parameters : const $\in \mathbb{Z}$, the cutoff for testing whether a value is effectively constant; $\epsilon \in \mathbb{Q}$, a tolerance for deciding equality over the reals; dc $\in \mathbb{Z}$, the discretization cutoff
Output : A new value for X or $\perp$

```
1  Call x' ← sample(X) until x ≠ x' or const trials
2  if x' = x then
3  |   remove X from possible_interventions and return ⊥
4  else if X ∉ interventions_tried then
5  |   return x'
6  else
7  |   tried ← interventions_tried[X]
8  |   if x' is not a float then
9  |   |   if x' ∉ tried then
10 |   |   |   return x'
11 |   |   else
12 |   |   |   return ⊥
13 |   |   end
14 |   else
15 |   |   if ∀v ∈ tried, |x' − v| < ε then
16 |   |   |   return ⊥
17 |   |   else if |tried| > dc then
18 |   |   |   n ← |tried|
19 |   |   |   μ̂ ← sample mean of tried
20 |   |   |   σ̂ ← sample variance of tried
21 |   |   |   h ← 3.49σ̂ / n^(1/3)
22 |   |   |   low ← min(tried)
23 |   |   |   high ← h + low
24 |   |   |   while low < max(tried) do
25 |   |   |   |   if x' ∈ (low, high) then
26 |   |   |   |   |   if |{v|v ∈ tried ∧ v ∈ (low, high)}| = 0 then
27 |   |   |   |   |   |   return x'
28 |   |   |   |   |   end
29 |   |   |   |   end
30 |   |   |   |   low ← high
31 |   |   |   |   high ← high + h
32 |   |   |   end
33 |   |   return ⊥
34 |   end
35 end
```

Algorithm 3: Sampling algorithm for AutoExp; resembles space-filling experimental designs.

APPENDIX F

AUTOEXP RESULTS FOR TARGET AGENT

Figures F.1 and F.2 give the results for strong and weak counterfactual explanations of the Target agent, respectively.

Aim left		Aim right		Missed ball		Hit ball	
†	(16)	†	(14)	†	(13)	†	(22)
balls[0].position.y	(7)	balls[0].position.y	(11)	∗	(8)	∗∗	(8)
balls[0].position.x	(5)	paddle.position.x	(3)	ball_radius	(2)		
paddle.position.x	(1)	balls[0].position.x	(2)	balls[0].velocity.x	(2)		
bricks[63].color.g	(1)			balls[0].position.y	(1)		
				bricks[57].size.y	(1)		
				bricks[66].color.g	(1)		
				bricks[102].position.y	(1)		
				bricks[103].color.g	(1)		

Move same		Move opposite		Move toward		Move away	
†	(7)	paddle.position.x	(9)	∗∗	(18)	∗	(18)
balls[0].position.x	(4)	balls[0].position.y	(7)	†	(9)	balls[0].position.x	(1)
paddle.position.x	(2)	balls[0].velocity.x	(3)	bricks[57].position.y	(1)	balls[0].velocity.x	(1)
∗∗	(1)	balls[0].position.x	(2)	bricks[93].size.x	(1)	bricks[2].position.y	(1)
balls[0].position.y	(1)	bricks[18].position.y	(1)	bricks[85].points	(1)	bricks[24].position.x	(1)
bricks[17].size.x	(1)	bricks[33].position.x	(1)			bricks[63].row	(1)
bricks[23].points	(1)	bricks[36].size.x	(1)			bricks[73].size.y	(1)
bricks[23].position.x	(1)	bricks[53].size.y	(1)			bricks[76].row	(1)
bricks[30].size.y	(1)	bricks[67].position.y	(1)			bricks[76].size.x	(1)
bricks[38].size.x	(1)	bricks[79].points	(1)			bricks[78].color.b	(1)
bricks[46].size.y	(1)	bricks[81].position.y	(1)			bricks[85].position.y	(1)
bricks[47].position.y	(1)	bricks[100].size.x	(1)			bricks[90].position.x	(1)
bricks[49].size.x	(1)	∗	(1)			bricks[101].position.x	(1)
bricks[80].position.x	(1)						
bricks[80].size.y	(1)						
bricks[87].position.x	(1)						
bricks[97].col	(1)						
bricks[104].position.x	(1)						
bricks[105].size.x	(1)						

Table F.1: Strong counterfactual explanations for the Target agent. Experiments run over atomic attributes only, for 30 random seeds for each of the eight outcomes. † The experiment timed out after 12hrs with no counterfactual explanations found.

Aim left		Aim right		Missed ball		Hit ball	
* * *	(26)	* * *	(30)	* * *	(16)	* * *	(24)
balls[0].position.y	(1)			*	(5)	**	(5)
bricks[14].color.a	(1)			balls[0].velocity.x	(1)		(1)
bricks[82].size.x	(1)			bricks[11].alive	(1)		
paddle.position.x	(1)			bricks[13].position.y	(1)		
				bricks[35].position.y	(1)		
				bricks[35].size.y	(1)		
				bricks[40].color.g	(1)		
				bricks[63].position.x	(1)		
				bricks[94].color.b	(1)		
					(1)		

Move same		Move opposite		Move toward		Move away	
* * *	(12)	* * *	(10)	**	(21)	*	(21)
**	(2)	paddle.position.x	(3)	* * *	(8)	* * *	(1)
bricks[3].size.x	(1)	* * *	(2)		(1)	bricks[7].position.x	(1)
bricks[4].position.y	(1)	bricks[38].size.y	(2)			bricks[20].color.g	(1)
bricks[9].position.x	(1)	balls[0].position.x	(1)			bricks[56].color.g	(1)
bricks[12].position.x	(1)	balls[0].position.y	(1)			bricks[64].size.x	(1)
bricks[26].size.y	(1)	bricks[10].size.y	(1)			bricks[84].position.x	(1)
bricks[29].position.y	(1)	bricks[19].color.r	(1)			bricks[98].position.y	(1)
bricks[35].position.y	(1)	bricks[24].size.y	(1)			bricks[98].size.y	(1)
bricks[36].size.y	(1)	bricks[26].position.y	(1)			bricks[105].depth	(1)
bricks[53].size.y	(1)	bricks[38].position.x	(1)				
bricks[58].position.y	(1)	bricks[58].depth	(1)				
bricks[72].color.a	(1)	bricks[60].position.y	(1)				
bricks[81].size.x	(1)	bricks[62].position.y	(1)				
bricks[84].size.x	(1)	bricks[73].size.y	(1)				
bricks[87].size.x	(1)	bricks[78].size.x	(1)				
bricks[101].size.y	(1)	bricks[79].alive	(1)				
bricks[107].size.y	(1)						

Table F.2: Weak counterfactual explanations for the Target agent. Experiments run over atomic attributes only, for 30 random seeds for each of the eight outcomes. Three runs are missing (for Missed ball, Hit ball, and Move toward) and are currently being re-run. * This outcome was not observed during normal game play (i.e, no evidence found for behavior, violation of $P(t)$ in sentence 5.7). ** This outcome was not observed during normal game play (i.e, no evidence found for behavior, violation of $Q(t)$ in sentence 5.7). *** The search timed out without producing any counterfactual explanations after 12h of search.

BIBLIOGRAPHY

[1] Association for Computing Machinery. Artifact review and badging. April 2018.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques.* Addison Wesley, 75 Arlington Street, Suite 300 in Boston, Mass., 1986.

[3] J. D. Angrist, G. W. Imbens, and D. B. Rubin. Identification of causal effects using instrumental variables. *Journal of the American Statistical Association*, 91(434):444–455, 1996.

[4] A. Asperti, C. De Pieri, and G. Pedrini. Rogueinabox: an environment for roguelike learning. *International Journal of Computers*, 2, 2017.

[5] A. Atrey, K. Clary, and D. Jensen. Exploratory not explanatory: Counterfactual analysis of saliency maps for deep reinforcement learning. In *International Conference on Learning Representations*, 2020.

[6] E. Bakshy, L. Dworkin, B. Karrer, K. Kashin, B. Letham, A. Murthy, and S. Singh. AE: A domain-agnostic platform for adaptive experimentation. In *2018 NeurIPS Workshop on Systems for ML*, 2018.

[7] E. Bakshy, D. Eckles, and M. S. Bernstein. Designing and deploying online field experiments. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 283–292, New York, NY, USA, 2014. ACM.

[8] D. Balduzzi, K. Tuyls, J. Perolat, and T. Graepel. Re-evaluating evaluation. In *Advances in Neural Information Processing Systems*, 2018.

[9] E. Bareinboim and J. Pearl. Causal inference from big data: Theoretical foundations and the data-fusion problem. Technical report, DTIC Document, 2015.

[10] G. Basse and I. Bojinov. A general theory of identification. *arXiv preprint arXiv:2002.06041*, 2020.

[11] C. Beattie, J. Z. Leibo, D. Teplyashin, T. Ward, M. Wainwright, H. Küttler, A. Lefrancq, S. Green, V. Valdés, A. Sadik, et al. Deepmind Lab. *arXiv preprint arXiv:1612.03801*, 2016.

[12] K. Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[13] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. Manifesto for agile software development. 2001.

[14] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, Jun 2013.

[15] Y. Bengio. *Learning deep architectures for AI.* Now Publishers Inc, 2009.

[16] A. Beygelzimer, E. Fox, F. d'Alché Buc, and H. Larochelle. Call for papers. April 2019.

[17] G. E. Box, W. G. Hunter, J. S. Hunter, et al. *Statistics for experimenters.* John Wiley and Sons, New York, 1978.

[18] A. Braylan, M. Hollenbeck, E. Meyerson, and R. Miikkulainen. Frame skip is a powerful parameter for learning to play Atari. In *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[19] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Gym, 2016.

[20] P. S. Castro, S. Moitra, C. Gelada, S. Kumar, and M. G. Bellemare. Dopamine: A research framework for deep reinforcement learning, 2018.

[21] K. Clary. Proposed Formalism for Generalization in Reinforcement Learning. *KDL Technical Memorandum TM-2019-003*, June 2019.

[22] K. Clary, E. Tosch, J. Foley, and D. Jensen. Let's Play Again: Variability of Deep Reinforcement Learning Agents in Atari Environments. In *NeurIPS 2018 Workshop on Critiquing and Correcting Trends in Machine Learning*, 2018.

[23] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman. Quantifying generalization in reinforcement learning. *arXiv preprint arXiv:1812.02341*, 2018.

[24] D. Cohen, S. M. Jordan, and W. B. Croft. Distributed evaluations: Ending neural point metrics. In *SIGIR; LND4IR*, 2018.

[25] T. Crook, B. Frasca, R. Kohavi, and R. Longbotham. 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*, pages 1105–1114, New York, NY, USA, 2009. ACM.

[26] J. Cunha, J. P. Fernandes, P. Martins, J. Mendes, and J. Saraiva. Smellsheet detective: A tool for detecting bad smells in spreadsheets. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 243–244. IEEE, 2012.

[27] J. Cunha, J. P. Fernandes, H. Ribeiro, and J. Saraiva. Towards a catalog of spreadsheet smells. In *International Conference on Computational Science and Its Applications*, pages 202–216. Springer, 2012.

[28] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[29] A. D'Amour, H. Srinivasan, J. Atwood, P. Baljekar, D. Sculley, and Y. Halpern. Fairness is not static: deeper understanding of long term fairness via simulation studies. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, pages 525–534, 2020.

[30] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Germany, 2008. Springer.

[31] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[32] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. OpenAI Baselines. `https://github.com/openai/baselines`, 2017.

[33] F. Doshi-Velez and B. Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[34] R. Dubey, P. Agrawal, D. Pathak, T. L. Griffiths, and A. A. Efros. Investigating human priors for playing video games. *arXiv preprint arXiv:1802.10217*, 2018.

[35] J. Engel, B. Rice, and R. Jones. Behave. `https://github.com/behave/behave`, 2011. Last updated: 2020.

[36] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[37] R. A. Fisher. Design of experiments. *Br Med J*, 1(3923):554–554, 1936.

[38] R. A. Fisher and J. Wishart. *The arrangement of field experiments and the statistical reduction of the results*. Number 10. HM Stationery Office, 1930.

[39] J. Foley, E. Tosch, K. Clary, and D. Jensen. Toybox: Better Atari Environments for Testing Reinforcement Learning Agents. 2018.

[40] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Professional, 2018.

[41] M. Fredrikson and S. Jha. 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)*, page 42, New York, NY, USA, 2014. ACM.

[42] A. A. Freitas. Comprehensible classification models: a position paper. *ACM SIGKDD explorations newsletter*, 15(1):1–10, 2014.

[43] Y. Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[44] J. Garcıa and F. Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.

[45] A. Gelman and G. King. Estimating incumbency advantage without bias. *American Journal of Political Science*, 34(4):1142–1164, 1990.

[46] A. S. Gerber and D. P. Green. *Field experiments: Design, analysis, and interpretation*. WW Norton, 2012.

[47] J. Goodman, A. G. Greenberg, N. Madras, and P. March. Stability of binary exponential backoff. *Journal of the ACM (JACM)*, 35(3):579–602, 1988.

[48] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: A language for generative models. In *Proc. 24th Conf. Uncertainty in Artificial Intelligence (UAI)*, pages 220–229, Online, 2008. JMLR: W&CP.

[49] A. D. Gordon, T. Graepel, N. Rolland, C. V. Russo, J. Borgström, and J. Guiver. Tabular: a schema-driven probabilistic programming language. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 321–334, New York, NY, USA, 2014. ACM.

[50] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181, New York, NY, USA, 2014. ACM.

[51] S. Greydanus, A. Koul, J. Dodge, and A. Fern. Visualizing and understanding atari agents. *arXiv preprint arXiv:1711.00138*, 2018.

[52] U. Grönmping. R package frf2 for creating and analyzing fractional factorial 2-level designs. *Journal of Statistical Software*, 56(1):1–56, 2014.

[53] U. Grönmping. Frf2: Fractional factorial designs with 2-level factors. `http://CRAN.R-project.org/package=FrF2`, 09 2016.

[54] U. Grönmping. CRAN Task View: Design of Experiments (DoE) & Analysis of Experimental Data. `http://CRAN.R-project.org/view=ExperimentalDesign`, 10 2017.

[55] D. Gunning. Explainable artificial intelligence (XAI). *Defense Advanced Research Projects Agency (DARPA) Technical Report*, 2016.

[56] W. H. Guss, C. Codel, K. Hofmann, B. Houghton, N. Kuno, S. Milani, S. Mohanty, D. P. Liebana, R. Salakhutdinov, N. Topin, et al. The MineRL Competition on Sample Efficient Reinforcement Learning using Human Priors. *arXiv preprint arXiv:1904.10079*, 2019.

[57] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep Reinforcement Learning that Matters. In *AAAI Conference on Artificial Intelligence (AAAI)*. arXiv preprint 1709.06560, 2017.

[58] F. Hermans, M. Pinzger, and A. van Deursen. Detecting code smells in spreadsheet formulas. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 409–418. IEEE, 2012.

[59] F. Hermans, M. Pinzger, and A. van Deursen. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, 20(2):549–575, 2015.

[60] M. A. Hernán and J. M. Robins. *Causal Inference*. Chapman & Hall/CRC, Forthcoming, Boca Raton, 2016.

[61] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[62] D. Jensen. Using causal models to generate explanations. *KDL Technical Memorandum TM-2019-004*, August 2019.

[63] D. D. Jensen, A. S. Fast, B. J. Taylor, and M. E. Maier. 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*, pages 372–380, New York, NY, USA, 2008. ACM.

[64] Joelle Pineau. Building reproducible, reusable, and robust machine learning software. `https://2019.icse-conferences.org/details/icse-2019-Plenary-Sessions/20/Building-Reproducible-Reusable-and-Robust-Machine-Learning-Software`, May 2019.

[65] S. C. Johnson. *Lint, a C program checker*. Citeseer, 1977.

[66] S. M. Jordan, D. Cohen, and P. S. Thomas. Using Cumulative Distribution Based Performance Analysis to Bechmark Models. In *NeurIPS 2018 Workshop on Critiquing and Correcting Trends in Machine Learning*, 2018.

[67] Jordan, Scott and Chandak, Yash and Cohen, Daniel and Zhang, Mengxue and Thomas, Philip S. Evaluating the performance of reinforcement learning algorithms. *Proceedings of the 37th International Conference on Machine Learning*, 119, 2020.

[68] A. Juliani, V.-P. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*, 2018.

[69] Y. Kanagawa and T. Kaneko. Rogue-Gym: A New Challenge for Generalization in Reinforcement Learning, 2019.

[70] K. Kansky, T. Silver, D. A. Mély, M. Eldawy, M. Lázaro-Gredilla, X. Lou, N. Dorfman, S. Sidor, S. Phoenix, and D. George. Schema networks: Zero-shot transfer with a generative causal model of intuitive physics. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

[71] A.-H. Karimi, G. Barthe, B. Balle, and I. Valera. Model-agnostic counterfactual explanations for consequential decisions. In *International Conference on Artificial Intelligence and Statistics*, pages 895–905, 2020.

[72] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski. Vizdoom: A Doom-based AI research platform for visual reinforcement learning. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2016.

[73] R. Kohavi, A. Deng, B. Frasca, T. Walker, Y. Xu, and N. Pohlmann. Online controlled experiments at large scale. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1168–1176, New York, NY, USA, 2013. ACM.

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

[75] R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne. Controlled experiments on the Web: survey and practical guide. *Data Mining and Knowledge Discovery*, 18(1):140–181, 2009.

[76] R. Korf. Depth-first iterative deepening: an optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[77] S. S. Krishnan and R. K. Sitaraman. Video stream quality impacts viewer behavior: Iing causality using quasi-experimental designs. *IEEE/ACM Transactions on Networking (TON)*, 21(6):2001–2014, 2013.

[78] C. Kuang. Can A.I. Be Taught to Explain Itself? *New York Times Magazine*, Nov. 21, 2017.

[79] P. Langley, G. L. Bradshaw, and H. A. Simon. Rediscovering chemistry with the BACON system. In *Machine Learning*, pages 307–329. Springer, 1983.

[80] J. Leike, M. Martic, V. Krakovna, P. A. Ortega, T. Everitt, A. Lefrancq, L. Orseau, and S. Legg. AI safety gridworlds. *arXiv preprint arXiv:1711.09883*, 2017.

[81] Lenat, Douglas B. and Seely Brown, John. Why AM and Eurisko Appear to Work. *AAAI-83 Proceedings*, 1983.

[82] R. J. Little and D. B. Rubin. Causal effects in clinical and epidemiological studies via potential outcomes: Concepts and analytical approaches. *Annual Review of Public Health*, 21(1):121–145, 2000.

[83] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. J. Hausknecht, and M. Bowling. Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents. *CoRR*, abs/1709.06009, 2017.

[84] C. F. Manski. *Identification problems in the social sciences*. Harvard University Press, 1999.

[85] R. Metcalfe and D. R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19:395–404, July 1976.

[86] T. Minka and J. Winn. Gates. In *Advances in Neural Information Processing Systems*, pages 1073–1080, 2009.

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

[88] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on Machine Learning*, pages 1928–1937, 2016.

[89] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level Control through Deep Reinforcement Learning. *Nature*, 518:529, 02 2015.

[90] J. Mockus, V. Tiesis, and A. Zilinskas. The application of Bayesian methods for seeking the extremum. *Towards Global Optimization*, 2, 1978.

[91] A. W. Moore. Efficient memory-based learning for robot control. Technical report, 1990.

[92] S. L. Morgan and C. Winship. *Counterfactuals and causal inference*. Cambridge University Press, Cambridge, UK, 2014.

[93] S. S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, Burlington, MA, USA, 1997.

[94] W. J. Murdoch, C. Singh, K. Kumbier, R. Abbasi-Asl, and B. Yu. Definitions, methods, and applications in interpretable machine learning. *Proceedings of the National Academy of Sciences*, 116(44):22071–22080, 2019.

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

[96] A. Nichol, V. Pfau, C. Hesse, O. Klimov, and J. Schulman. Gotta Learn Fast: A New Benchmark for Generalization in RL. *arXiv preprint arXiv:1804.03720*, 2018.

[97] OpenAI. Infrastructure for Deep Learning. `https://openai.com/blog/infrastructure-for-deep-learning/`, 2016.

[98] OpenAI. Proximal Policy Optimization. `https://blog.openai.com/openai-baselines-ppo/`, 2017.

[99] OpenAI. `https://openai.com/blog/openai-five/#rapid`, 2018.

[100] I. Osband, Y. Doron, M. Hessel, J. Aslanides, E. Sezener, A. Saraiva, K. McKinney, T. Lattimore, C. Szepezvari, S. Singh, et al. Behaviour suite for reinforcement learning. *arXiv preprint arXiv:1908.03568*, 2019.

[101] D. C. Parkes, A. Mao, Y. Chen, K. Z. Gajos, A. Procaccia, and H. Zhang. TurkServer: Enabling synchronous and longitudinal online experiments, 2012. Fourth Workshop on Human Computation (HCOMP'12).

[102] J. Pearl. *Causality: Models, Reasoning and Inference*. Cambridge University Press, New York, NY, USA, 2nd edition, 2009.

[103] D. Pedreschi, F. Giannotti, R. Guidotti, A. Monreale, S. Ruggieri, and F. Turini. Meaningful explanations of black box ai decision systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 9780–9784, 2019.

[104] A. Pfeffer. *Practical probabilistic programming*. Manning Publications Co., Shelter Island, NY, USA, 2016.

[105] F. Provost. Iterative weakening: Optimal and near-optimal policies for the selection of search bias. *Proceedings of the Eleventh National Conference on Artificial*, pages 749–755, 1993.

[106] V. Raychev, P. Bielik, M. Vechev, and A. Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, pages 761–774. ACM, 2016.

[107] P. R. Rosenbaum and D. B. Rubin. The central role of the propensity score in observational studies for causal effects. *Biometrika*, 70(1):41–55, 1983.

[108] D. B. Rubin. Estimating causal effects of treatments in randomized and nonrandomized studies. *Journal of Educational Psychology*, 66(5):688, 1974.

[109] D. B. Rubin. Causal inference using potential outcomes: Design, modeling, decisions. *Journal of the American Statistical Association*, 100(469):322–331, 2005.

[110] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003.

[111] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.

[112] J. Sacks, W. J. Welch, T. J. Mitchell, and H. P. Wynn. Design and analysis of computer experiments. *Statistical Science*, pages 409–423, 1989.

[113] J. Sall. Jmp: Design of experiments. `https://www.jmp.com/en_us/about.html`.

[114] R. Scheines. Causal reasoning: Disseminating new curricula with online courseware, 2003. Presented at the American Education Research Association.

[115] Schmidt, Michael and Lipson, Hod. Distilling Free-Form Natural Laws from Experimental Data. *Science*, 324:81–85, 2009.

[116] U. Schöning. *Logic for computer scientists*. Springer Science & Business Media, 2008.

[117] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

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

[119] W. R. Shadish, T. D. Cook, and D. T. Campbell. *Experimental and quasi-experimental designs for generalized causal inference*. Houghton Mifflin Company, Boston, MA, USA, 2002.

[120] A. Sherman, P. A. Lisiecki, A. Berkheimer, and J. Wein. Acms: The akamai configuration management system. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 245–258. USENIX Association, 2005.

[121] J. Shin, A. Paepcke, and J. Widom. 3X: A Data Management System for Computational Experiments (Demonstration Proposal). Technical report, Stanford University, 2013.

[122] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354, 2017.

[123] B. Smucker, M. Krzywinski, and N. Altman. Optimal experimental design. *Nature Methods*, 15(8):559–560, 2018.

[124] C. Solis and X. Wang. A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 383–387. IEEE, 2011.

[125] S. Spangler, A. D. Wilkins, B. J. Bachman, M. Nagarajan, T. Dayaram, P. Haas, S. Regenbogen, C. R. Pickering, A. Comer, J. N. Myers, et al. Automated hypothesis generation based on mining scientific literature. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1877–1886, New York, NY, USA, 2014. ACM.

[126] P. Spirtes. Variable definition and causal inference. 2009.

[127] J. Splawa-Neyman, D. Dabrowska, T. Speed, et al. On the application of probability theory to agricultural experiments. essay on principles. section 9. *Statistical Science*, 5(4):465–472, 1990. [Updated 1900].

[128] R. S. Sutton, A. G. Barto, et al. *Reinforcement Learning: An Introduction*. MIT press, 1998.

[129] J. Sybrandt, M. Shtutman, and I. Safro. Moliere: Automatic biomedical hypothesis generation system. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 1633–1642, New York, NY, USA, 2017. ACM.

[130] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations*, 2014.

[131] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 328–343. ACM, 2015.

[132] D. Tang, A. Agarwal, D. O'Brien, and M. Meyer. Overlapping experiment infrastructure: More, better, faster experimentation. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 17–26, New York, NY, USA, 2010. ACM.

[133] S. Tikka and J. Karvanen. Identifying causal effects with the r package. *Journal of Statistical Software*, 76:1–30, February 2017.

[134] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A Physics Engine for Model-based Control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

[135] J. Togelius, S. Karakovskiy, and R. Baumgarten. The 2009 mario ai competition. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[136] E. Tosch, E. Bakshy, E. D. Berger, D. D. Jensen, and J. E. B. Moss. PlanAlyzer: Assessing threats to the validity of online experiments. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30, 2019.

[137] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

[138] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, et al. Starcraft II: A New Challenge for Reinforcement Learning. *arXiv preprint arXiv:1708.04782*, 2017.

[139] W. Wang, D. Rothschild, S. Goel, and A. Gelman. Forecasting elections with non-representative polls. *International Journal of Forecasting*, 31(3):980–991, 2015.

[140] S. Witty, J. K. Lee, E. Tosch, A. Atrey, M. Littman, and D. Jensen. Measuring and Characterizing Generalization in Deep Reinforcement Learning. 2018.

[141] F. Wood, J. W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics*, volume 33, pages 1024–1032, Online, 2014. JMLR: W&CP.

[142] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba. Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation. In *Advances in Neural Information Processing Systems*, pages 5279–5288, 2017.

[143] S. Yegge. Stevey's Google platforms rant. `https://gist.github.com/chitchcock/1281611`, 2011.

[144] T. Zahavy, N. Ben-Zrihem, and S. Mannor. Graying the black box: Understanding DQNs. In *International Conference on Machine Learning*, pages 1899–1908, 2016.