Causal Modeling, Discovery, & Inference for Software Engineering

Authors Name/s per 1st Affiliation *(Author)*
line 1 (of *Affiliation*): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: e-mail: name@xyz.com

Authors Name/s per 2nd Affiliation *(Author)*
line 1 (of *Affiliation*): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: e-mail: name@xyz.com

***Abstract*—Empirical studies in software engineering frequently rely on correlation data in an effort to demonstrate that a process or tool affects an important or meaningful outcome, with the ultimate goal of improving software engineering practice. But all students of statistics know that "correlation does not imply causation," and so causal conclusions (using traditional methods) from observational studies are inevitably highly constrained. These studies thus have limited impact on real world practice. In this paper, we use methods beyond mere correlation, and apply techniques of causal discovery to software engineering data as a means of unambiguously determining cause and effect. We apply causal discovery techniques to a set of observational data on open source projects; use the results to determine some consequences of architectural flaws; and show how causal inference may be applied to software engineering data in the future.**

Keywords-correlation studies; causal inference; empirical software engineering

I. Why causation?

Software engineering, as with any engineering discipline, is grounded in science [8]. In the case of software engineering, this broad-ranging scientific basis consists of purely technical or mathematical theories of computation, programming languages, data structures, algorithms, modularity, and many other areas.

However, the *practice* of software engineering necessarily involves human effort: people collect requirements, and design, code, test, analyze, deploy, and maintain software. There have been decades of research on how to use this human capital efficiently. Empirical studies have been conducted to determine the costs and benefits of the many techniques, methods, tools, and languages that have been proposed and deployed. Of course, more research is needed, as seen by the many researchers, workshops, conferences, and journals that have emerged [6].

Controlled experiments in software engineering, when they have occurred, have largely been met with skepticism. The prevailing wisdom is that the tradeoffs needed to make such experiments tractable—the limited size and experience of the group of experimental subjects, and the limited size and realism of the experimental task—make it difficult to translate those results to the broader software engineering profession. Controlled experiments tend to be conducted with relatively small populations of (relatively inexperienced) undergraduates, and with experimental tasks that constrain the messiness, ambiguity, and complexity that face practitioners in the real world. The software engineering practitioner community has therefore quite rightly questioned whether the results of such experiments can be profitably generalized to professional software engineers working on less constrained problems in real-world contexts.

Of course, controlled experiments are not the only type of empirical studies. Increasingly, empirical research in software engineering has focused on case studies, action research, and studies of "naturalistic" phenomena (e.g., [3]). But these studies collect only observational data, and so traditional analysis techniques yield only correlations between project practices and characteristics (on the one hand) and measurable outcomes (on the other hand). And as every software project manager knows and will tell you, correlation does not mean causation. Without knowing the causal effects, it is difficult for that manager to act upon correlational evidence. For example, as source files in a software project increase in size, they tend to have more bugs and be touched by more developers. That is, file size, bugs, and number of developers are all strongly positively correlated. If one mistakes correlation for causation, then one might be tempted to conclude that bug rates could be lowered just by reducing the number of developers who are working on that file!

While this example is simplistic, it captures the essence of the problems associated with relying on correlation data. While some correlations are spurious, others clearly are not. As Tufte said: "Correlation is not causation but it sure is a hint." [11] Certainly, if one project characteristic or practice causes another, it will not only be statistically correlated with it, but also provide a means to change it. Our goal, therefore, is to understand the causal relations between project characteristics and practices and the outcomes that we desire. In doing so, we can confidently create concrete, actionable recommendations for software project managers: adopt this language, or tool, or practice and something that you care about—code quality, bug rate, productivity, developer satisfaction—will improve.

In what follows, we explain our framework for reasoning about causality, and a case study to which we applied such methods. We show that one can, indeed, make justifiable causal claims based on data collected from naturalistic phenomena, and that this can greatly increase our understanding of, and hence recommendations for, best software engineering practices.

II. A FRAMEWORK FOR CAUSAL MODELS

Over the past thirty years, a robust framework for causal modeling—*causal graphical models*—has been developed, with algorithms for discovery of causal structure from observational, experimental, and mixed datasets. There have been multiple case studies, across domains ranging from genetics to ecology to computer hardware, in which these methods have been used on purely observational data, and the outputs (i.e., learned causal structures) have subsequently been experimentally confirmed. We thus have reason to causally interpret the graphical models learned by these algorithms, even though we have not yet conducted follow-up confirmatory experiments.

A. Causal modeling & inference

We employ the standard framework of causal graphical models (CGMs); causal Bayesian networks (CBNs) and causal structural equation models (SEMs) are two common types of CGMs, but not the only ones. At a high level, a CGM has two distinct components: (1) a graph for qualitative causal relations, and (2) a joint probability distribution or density for quantitative causal strengths. The graph is over nodes for the variables—$V_1, \ldots, V_n$—with $V_c \rightarrow V_e$ if V_c is a (qualitative) cause of V_e. The notion of causation here is instrumental: $V_c \rightarrow V_e$ means that external interventions on V_c will probabilistically lead to changes in V_e, but not vice versa. The intensity of that causal connection is represented in the joint distribution or density $P(V_1, \ldots, V_n)$. These two components are connected through a pair of assumptions, commonly called *Markov* and *Faithfulness*, each of which uses one component to constrain the other. These assumptions encode standard, domain-general ways in which causal relations manifest in data. For a given CGM, there are fast, computationally efficient algorithms for inference and prediction given observations or interventions (including policy changes), many implemented in standard software packages [5], [10].

B. Causal discovery

Perhaps surprisingly, every CGM implies a determinate pattern of (un)conditional independencies and associations over $V_1, \ldots, V_n$. Causal discovery is difficult because the reverse is not true: any particular pattern of independencies can be generated by multiple different CGMs. The CGM $\rightarrow$ Independence Pattern map is many $\rightarrow$ one. Nonetheless, given an observed pattern of (un)conditional independences, one can determine the set of CGMs (if any) that could have produced that pattern. Sometimes that set will be a singleton, in which case the causal structure can be uniquely identified. More frequently, the set has a few different members that share many causal/structural features.

Numerous algorithms have been developed over the past thirty years for this problem, including Bayesian, score-based [1], constraint-based [10], and more sophisticated [9] methods. And these different types of algorithms have been generalized for situations with unobserved common causes, sample selection bias, time series data, proxy measurements, millions of variables, and many other conditions. As noted above, these algorithms have been applied, with confirmatory follow-up experiments, in a wide range of domains; there is a track record of successful causal discovery using these methods.

III. A CASE STUDY IN CAUSAL DISCOVERY

The dataset that we examine in this paper resulted from an exploration of architectural design flaws in a set of large, primarily open source software systems. Those systems were identified in two separate studies ([4] and [2]) that explored the relationship between design flaws and multiple outcomes that were (potentially) caused by design flaws. However, those studies—like the vast majority of studies in software engineering—drew only correlational conclusions, and so cannot be used to confidently justify policy or practice changes. In this paper, we aim to determine whether architectural design flaws *caused* higher rates of bugs, higher rates of changes, and higher amounts of churn (committed lines of code).

A. Data

We analyzed both sets of projects, and project versions, as listed in Table 1. These two studies analyzed a total of 15 distinct projects, and 20 distinct project versions, across a wide variety of application domains, and with greatly varying ages and sizes: the commercial project had just 56,000 lines of code, while the Google Chrome browser had over 5 million lines of code. We focused on nine systems for the present analysis, indicated in bold in Table 1.

Table 1: Subject Projects and Versions

Project	*Study 1 [4]*	*Study 2 [2]*
Avro	**1.7.6**	**1.6.3**
Camel	**2.11.1**	**2.8.4**
Cassandra	**1.0.7**	
CXF	**2.7.10**	**2.5.2**
Derby		10.9.1.0
Hadoop	**2.2.0**	**0.94.0**
HBase	**0.94.16**	**0.94.0**
Httpd		2.0.58
Ivy	**2.3.0**	
OpenJPA	**2.2.2**	
PDFBox	**1.8.4**	
PHP		4.4.6
Tomcat		6.0.0
Commercial	N/A	
Chrome		17.0.963.46

To collect the raw data, the original studies extracted the source code, revision histories, and issue-tracking databases from each of these projects. Using the revision histories and issue-tracking databases, they calculated the number of bugs and changes (non-bugs) associated with each file in each project, as well as the churn associated with each bug or change. These two studies relied on each project's issue tracking system (and their own internal conventions) to determine what was a bug and what was a change. Next, each project's source code was reverse-engineered to analyze all of the static relationships between its files (e.g. calling, inheritance, typing, etc.). This information was used to build and analyze a DRSpace (Design Rule Space) representation of each project's modular structure (as described in [12] and [13]). By analyzing the resulting DRSpaces, the design flaws associated with each file in each project were calculated, as described in [4].

Using this collected data, those two studies calculated a number of correlations: between the number of design flaws that a file was implicated in, on the one hand, and four different *extrinsic* measures of a file's goodness—number of bugs, number of changes, bug churn, and change churn—on the other hand. To do this the Pearson Correlation Coefficient (PCC) was calculated between the four pairs of data sets. An example of the correlation results that were reported in the original studies ([4] and [2]) is presented in Table 2, for the Apache Avro project.

Table 2: Correlations for Apache Avro 1.7.6

# Flaws	Avg. Bug Freq.	Avg. Bug Churn	Avg. Change Freq.	Avg. Change Churn
0	0.1	3.7	0.5	29.0
1	0.4	3.9	0.9	26.2
2	1.6	12.6	5.2	376.7
3	7.9	124.5	21.6	628.5
4	16.5	255.0	33.5	1220.0
PCC	**0.91**	**0.89**	**0.94**	**0.95**

It is obvious from a cursory examination of the data that as the number of design flaws per file increases, so too do the average numbers of bugs, changes, bug churn, and change churn. The PCC values shown in the last row of Table 2 bear out this observation: these variables are strongly correlated. And such correlations were similarly observed for all 20 of the projects in the two prior studies. But the question remains: do the observed design flaws *cause* the higher rates of bugs, changes, and churn, or is there some other explanation, such as a third (unexplored) variable that accounts for the observed correlation? This is the crucial question that we seek to understand in the present study.

These strong correlations present a complication for causal discovery, as they increase the number of variables without substantially increasing the amount of information or signal. Moreover, highly correlated variables can create spurious independences—factors that are causally connected can appear independent, due to the particular mathematical definition of statistical independence. There were two triples of highly intercorrelated variables; in both cases, we retained one variable as a proxy for the set. Thus, our final analyses focused on six variables:

Age [age in months]	*Devs* [# of developers]
LOC [lines of code]	*Violations* [total # of violations]
Churn_bugs [bug churn]	*Bugs* [# of bugs]

B. *Causal discovery*

We applied the PC algorithm [10] to the datasets for the nine systems shown in bold in Table 1. The PC algorithm is an asymptotically reliable, constraint-based causal discovery method that efficiently determines the set of structures over $V_1, \ldots, V_n$ that imply the observed pattern of (un)conditional independencies. The PC algorithm has a free parameter—the alpha level used in independence tests—that controls the relative proportions of Type I and Type II errors. For each of the nine datasets, as well as the concatenation of all nine, we applied the PC algorithm at α=0.05. The PC algorithm does not explicitly search for causal structures with unobserved common causes, in contrast with algorithms such as FCI that do have this ability. Thus, a causal edge in the output could potentially be explained in other ways. Importantly, though, *absences* of causal edges are robust against the possibility of unobserved common causes: if there is no $A \rightarrow B$ edge, then we can reliably conclude that *A* does not directly cause *B* (subject to the usual errors due to statistical noise).

As an example of the algorithm output, the causal graph for all nine concatenated datasets is shown in Figure 1. This causal graph includes a bidrected edge $Dev \leftrightarrow LOC$, indicating the likely existence of an unobserved common cause of the two (plausibly, something like project size). Although the PC algorithm does not explicitly search for such unobserved factors, it does sometimes include them when required to explain the observed data.

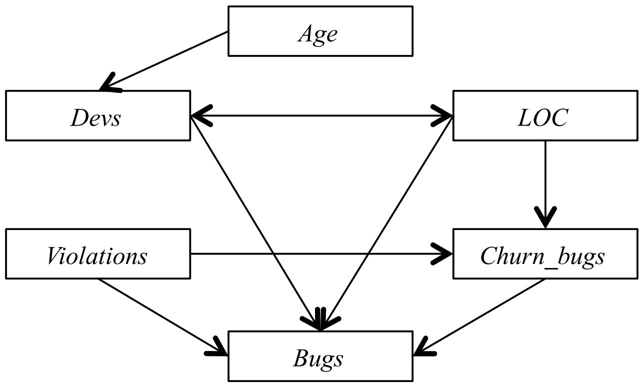

Figure 1: Output graph for all nine datasets

Some aspects of Figure 1 provide useful "sanity checks." For example, $LOC \rightarrow Bugs$ is predicted by essentially every model (and experience) of software development. It is also sensible that *Age* only influences other variables through the mediating factor of *Devs*. The age of a project should not directly cause problems, but only because older projects tend to have had more people working on them.

This causal graph provides a suggestive answer to our key question, as we see directed edges from *Violations* to *Bugs* and *Churn_bugs*. However, it is important not to over-interpret Figure 1. The software projects studied are quite diverse, and so the concatenated dataset may be a mixture distribution. Mixtures are known to give misleading results for causal discovery algorithms [7]. We thus did an additional analysis, focusing on the shared causal structures across multiple individual projects. Interestingly, only one edge (*Devs* → *Bugs*) appeared in almost every output, which further suggests *causal diversity* between projects.

We thus *post hoc* identified groups of projects based on shared causal structure. Apache Avro was quite different from all the others, so we exclude it from this analysis. The projects fell into two rough groups: {Camel, Cassandra, Hadoop, OpenJPA, PDFBox} vs. {CXF, HBase, Ivy}. Table 3 lists the "characteristic" causal structures for each cluster (i.e., the edges that appear in most graphs for those projects).

Table 3: Characteristic edges for project-groups

Camel, Cassandra, Hadoop, OpenJPA, PDFBox	CXF, HBase, Ivy
Devs → *LOC*	*Age* → *Devs*
LOC → *Violations*	*Devs* → *Bugs*
Violations → *Churn_bugs*	*Bugs* → *Churn_bugs*
Age → *Bugs*	*LOC* → *Churn_bugs*
Devs → *Bugs*	
LOC → *Bugs*	
Violations → *Bugs*	
Churn_bugs → *Bugs*	

These two groups exhibit quite different types of causal structures. Most obviously, the first group of projects simply has more causal connections than the second. The details also vary between the groups. For example, they both posit a causal connection between *Bugs* and *Churn_bugs*, but the directions differ. More generally, *Bugs* is the "causal sink"—the variable that is caused by many other things in the system—in the first group of projects, but *Churn_bugs* is the sink for the second.

With regards to our key question, we find different answers in the two groups of projects. In the first group, *Violations* is a direct cause of both *Bugs* and *Churn_bugs*; that is, we predict that focusing on violation reduction should lower the number of bugs, and their churn. In the second group, though, *Violations* is not causally connected with any of the other variables. We thus predict that a focus on violation reduction would not have a corresponding impact on bugs in those three projects.

Why are there these differences? At this point, we do not know. We focused on a limited set of variables, and presumably there are others, perhaps capturing project practices and characteristics, that would explain these results if they were included in our datasets. Nonetheless, causal discovery and analysis has provided us with a powerful new tool in our toolbox to ask and examine such questions.

IV. Conclusions and Next Steps

This causal discovery analysis is intended as an initial step, and is certainly not the final word. For example, one could apply multiple causal discovery algorithms to measure the sensitivity of the learned structures to the use of the PC algorithm. Moreover, software projects exhibit significant dynamics over time, as code is written, refined, refactored, and so forth. We used static datasets that provide snapshots of the projects at particular moments in time. If we collect longitudinal data about similar variables, then we could start to uncover the underlying causal dynamics. One might also suspect that those dynamics could shift over time, as the software practices and philosophies change, as project members enter and leave, etc. Longitudinal data could also enable us to test for this type of causal non-stationarity. The key point that we have established here, however, is the first demonstration of the applicability and usefulness of causal discovery algorithms applied to observational software engineering datasets.

Acknowledgment

Removed for blinding.

References

[1] D. M. Chickering. "Optimal Structure Identification with Greedy Search", *Journal of Machine Learning Research*, 2002, 3, 507–554.

[2] Q. Feng, R. Kazman, Y. Cai, R. Mo, L. Xiao, "An Architecture-centric Approach to Security Analysis", *Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA 2016)*, (Venice, Italy), April 2016.

[3] R. Kazman, D. Goldenson, I. Monarch, W. Nichols, G. Valetto, "Evaluating the Effects of Architectural Documentation: A Case Study of a Large Scale Open Source Project", *IEEE Transactions on Software Engineering*, 2016, 42:3, 220-260.

[4] R. Mo, Y. Cai, R. Kazman, L. Xiao, "Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells", *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA 2015)*, (Montreal, Canada), May 2015.

[5] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco: Morgan Kaufmann. 1988.

[6] D. Perry, A. Porter, L. Votta, "Empirical studies of software engineering: a roadmap", *Proceedings of the Conference on the future of Software Engineering*, 345-355, 2000.

[7] J. D. Ramsey, P. Spirtes, C. Glymour, "On Meta-analyses of Imaging Data and the Mixture of Records", 2011, *NeuroImage*, 57, 323-330.

[8] M. Shaw, "Prospects for an Engineering Discipline of Software", *IEEE Software*, 7 (6), 15-24, 1990.

[9] S. Shimizu, P. O. Hoyer, A. Hyvärinen, A. Kerminen, "A Linear Non-Gaussian Acyclic Model for Causal Discovery", *Journal of Machine Learning Research*, 2006, 7, 2003–2030.

[10] P. Spirtes, C. Glymour, R. Scheines. *Causation, Prediction, and Search* (2nd ed.). Cambridge, MA: The MIT Press. 2000.

[11] E. Tufte, *The Cognitive Style of PowerPoint: Pitching Out Corrupts Within*, 2nd ed, Graphics Press, 2006.

[12] L. Xiao, Y. Cai, R. Kazman, "Titan: A Toolset That Connects Software Architecture with Quality Analysis", *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, (Hong Kong), November 2014.

[13] L. Xiao, Y. Cai, R. Kazman, "Design Rule Spaces: A New Form of Architecture Insight", Proceedings of the International Conference on Software Engineering (ICSE) 2014, (Hyderabad, India), June 2014.