Causal Inference for Theory Building in Software Evolution
Work in Progress

Quinten Leidekker
University of Amsterdam
Amsterdam, The Netherlands
quintenleidekker@hotmail.com

Ana-Maria Oprescu
University of Amsterdam
Amsterdam, The Netherlands
a.m.oprescu@uva.nl

Lodewijk Bergmans
Software Improvement Group
Amsterdam, The Netherlands
l.bergmans@sig.eu

Graduate School of Informatics, University of Amsterdam

Abstract

Most academic disciplines have a strong focus on theories. In software engineering and software evolution however, there is a distinct lack. We propose a new methodology to construct and validate theories of software evolution. This methodology utilises causal inference to define and test causal relations. We explore this iterative methodology by exercising it and reflect on the utility.

1 Introduction

Software engineering is a complex discipline of engineering. It involves the process of creating and maintaining complex and evolving products of software. This is done using different tools and processes, but also involves extensive social and cognitive processes [ESSD08, ZR96]. This means that empirical software engineering research, is about both technical products and human activities.

A significant amount of exploratory research is done into these processes. This type of research generally focuses on finding correlations in data and analysing these using statistical inference techniques. Jørgensen and Sjøberg find that only 3 out of 47 studies in the Journal of Empirical Software Engineering which applied a statistical inference technique, were able to base their statistical testing on well-defined populations and random samples from those populations [SDAH08]. Additionally, when using correctly but narrowly defined populations, it may be difficult to generalise. An example of this is research done by researchers at automotive company Daimler. A 78.3% accuracy is reached in software fault prediction by analysing static code measures. This is based on 8 software projects of automotive head unit control systems [OMS18]. An issue arises when we attempt to aggregate knowledge, for example for software systems that are not automotive head unit control systems. Aggregation of knowledge in science is generally done through theories. Isolated, exploratory studies do not commonly result in new theories.

Early works by Lehman aimed to established a theory of software evolution based on qualitative data [Leh79, Leh80]. Not much work in theory building has been done since. This is described by Johnson et al. in their 2012 paper on the lack of consensus on any theories in software engineering [JEJ12]. In recent years, there has been significant interest in theory building for software engineering; we refer to the workshop series 'General Theories of Software Engineering' (GTSE) in 2012-2015 as a major forum for this interest [RJJ13, HW13, REN+14, SGJ16]. This workshop se-

In: A. Editor, B. Coeditor (eds.): Proceedings of the SaTToSE Workshop, Amsterdam, The Netherlands, 01-07-2020, published at http://ceur-ws.org

ries has extensively motivated the need for, and benefits of, theory building for software engineering. There are several types of theories, but most interest exists in theories that both explain and predict, where the aspect of explanation includes the notion of causality [SDAH08].

A change of focus, away from specific contexts and into more generalised theories is required. Lehman had very limited access to software systems and their evolution data to analyse back in his day. Nowadays, source code evolution data in the form of source code repositories and issue tracking systems is available for many systems. In other fields, such as the epidemiological [RGL08], medical [SS19, YB15] and econometric [SK15], observational data is used to establish causal effects through causal inference [Pea09]. Recent early stage work is investigating methodological issues to consider when performing observational studies in software engineering [Saa19]. This paper presents ideas on the practical use of observational data for theory construction in software engineering.

We propose and investigate a new methodology. This methodology is intended to provide a practical and actionable framework to iteratively create, explore, test and expand theories. We show some preliminary experiences and share results from the first experiments.

We define one main research question.

Research Question: How can causal inference be used as a methodology to create and validate theories on software evolution?

2 Background

We discuss theory building and causal inference with observational data in empirical software engineering.

2.1 Theory Building in Software Engineering

A large part of current research is based on statistical hypothesis testing [Jor04], a form of statistical inference. In this method a null hypothesis is tested against an alternative hypothesis [LR05]. A key step in this process is the selection of random sample from a population. The nature of conclusions drawn from these tests can only be as solid as the sample upon which they are based.

In many scientific disciplines it is the norm to produce theories and test these. As described earlier a strong focus lies on exploratory research in software engineering [SDAH08, JEJ12]. Jørgensen and Sjøberg propose a stronger focus on theories, specifically a change in focus from *"generalizing from a random sample to a larger population"* to *"generalizing across populations through theory-building"* [Jor04]. Theories are well-confirmed, before they are we refer to them as hypotheses. Some argue that a long deferred beginning of theorising is worse than any number of failures, because (1) it encourages the blind accumulation of information that may turn out to be mostly useless, and (2) a large bulk of information may render the beginning of theorising next to impossible [BUN12, SDAH08].

Johnson et al. describe theories as sharing three characteristics: they attempt to generalise local observations and data into more abstract and universal knowledge; they typically represent causality (cause and effect); and they typically aim to explain or predict a phenomenon [JEJ12]. Sjøberg et al. define a framework for theories with four archetypes: Actor, Technology, Activity and Software system [SDAH08]. This reflects the notion that software engineering research involves social processes, as well as technical data. They also references three levels of complexity of theories. Because theory construction is such an early stage we consider the first two: *"Minor working relationships that are concrete and based directly on observations"* and *"Theories of the middle range that involve some abstraction but are still closely linked to observations"*.

In this research we limit the scope to theories of software evolution.

2.2 Causal Inference & Observational Data

Statistical inference is the process of using data analysis to learn about distributions and associations between variables. Causal inference differs from statistical inference in that it defines causal relations between variables [Pea09, PGJ16]. A simple initial definition causality is proposed by Pearl: A variable X (exposure) is a cause of a variable Y (outcome) if Y in any way relies on X for its value [PGJ16]. We see that causal relations are directed. An example of an exposure-outcome pair in medicine could be smoking and lung cancer; an example in software engineering could be code smells and bugs [Saa19].

Causal inference considers the potential outcomes of an exposure. The potential outcomes describe what can happen or could have happened. The fundamental problem of causal inference is that we cannot observe the counterfactual; the potential outcome that did not happen. The best type of data to use for causal inference is data from a randomised trial. Using observational data we can simulate randomised experiments, under certain conditions. We can then use this to calculate the average causal effect of an exposure [Her04].

Covariates, like exposures and outcomes, are characteristics of a unit in the population. A covariate is considered a confounder if it influences both the exposure and the outcome [PBV12]. Confounders are

an essential part of causal inference. By controlling for confounders, which can be elements in our theories, we improve exchangeability, with which we strengthen our randomisation [Her04, SSC+02]. Selection of exposures, outcomes and confounders is not at all trivial. Causality is directional, but also temporal. The outcome happens after the exposure. To ensure the validity of research using observational data, this temporal aspect has to be taken into consideration.

There are several requirements for validity in this type of research. In a book on experimental design for causal inference several methods are described to increase the power of relationships [SSC+02]. Confounders are of high importance, reiterating the importance of well-considered selection of variables. Using larger sample sizes is also mentioned. This paper does not explicitly list guidelines for observational studies in empirical software engineering, but such research is being done [Saa19].

Since we aim to generalise across populations through theory building, population selection is important. We propose using conventional data extraction methods to extract code metrics at a large scale. Instead of looking at one software repository, we can look at a large set. Observational data mimics a randomised trial, we control for confounders to improve exchangeability. Large populations aid in this effort; the more group membership is randomised, the more "comparable" they are [Her04].

Causal effects can be visualised through causal diagrams. Causal diagram are directed acyclic graphs (DAGs). These diagrams facilitate theory building by offering an intuitive and easy-to-understand overview of involved concepts. Because this visual representation also allows for easy extension of the ideas under investigation, theories can iteratively be expanded. This incremental and iterative approach strikes a balance between pure theory construction and pure exploratory research. This in combination with a workable terminology emphasises research design and theory building over correlation in data.
Data collection for this type of research should be done with care. When extracting source code data from repositories or commits for example, it is important to clearly define how data is extracted and consider this for the relevant theory.

3 Experiment Design

3.1 Theory Construction

The theory construction process is made actionable through causal diagrams. We list some ways to go about theory construction. Experts in the field of software evolution might have ideas about how software evolves. This information can be extracted through surveys, formal plenary sessions and interviews. Another source of initial theories can be literature. A literature study of the topic of interest can yield initial theories. Finally, the reader may have intuitions or ideas that could form theories of software engineering. This approach means that theories can be of low granularity initially. The process of theory building becomes incremental. This low granularity means initial complexity and costs are low which may promote theory investigation.

Theories are generally presented in natural language. A problem that can be encountered when using natural language is ambiguity. It is therefore important to clearly define terms used in these theories. Additionally, great care should be taken when choosing confounders. Consider what the effect of a confounder is, why it may influence something. Selection of these is of high importance to the validity of final results.

3.2 Data Collection & Causal Inference

To collect data from software repositories we use PyDriller, a Python framework that facilitates repository mining [SAB18]. PyDriller has functionality to look at commits and changes made in those commits. Because time is explicit it aids in considering the temporal element of what we researching. We do not yet cover any variables that are not extractable from source code.

Causal diagram construction is facilitated through DAGgity, an environment forcreating, editing, and analysing causal diagrams. It is also available as a R package.

Data analysis is done in R. For the second experiment, the Match library is used to match the two groups with different exposures. The tableone package is used to estimate the causal effect [AS15]. Finally we t-test the results. Further work on what other methods to apply causal inference exist and analysis of these methods is still in progress.

4 Intermediate Results and Discussion

This research consist of several investigations into software evolution phenomena. The first case is largely exploratory and serves as a learning instrument to become familiar with the method. The second case is ongoing and preliminary results are presented.

4.1 Case 1: CC & SLOC

The first experiment is based on research by Landman et al. in which they investigate the notion that the code metrics Cyclomatic Complexity (CC) and Source Lines of Code (SLOC) are linearly correlated in Java methods [LSV14]. We chose this research because it

has very clear definitions on how to prepare the corpus, which is the population, for analysis. Additionally, the corpus is large. How to measure the variables that are used in the research is also clearly defined.

Landman et al. investigate the hypothesis that there is a strong linear (Pearson) correlation between CC and SLOC metrics. Additionally they investigate if the inclusion of the %% and || operators influence this correlation. The third hypothesis looks into whether or not a higher level of aggregation of CC (over multiple units) correlates with aggregated SLOC. The fourth hypothesis investigates what the effect is of a power transform on the CC and SLOC data. Finally, the effect of zooming in on quantiles of data is investigated.

We started investigation into what happens here and came up with fig. 1, the causal diagram for this case. For an explanation of the variables we refer to Landman et al. [LSV14].

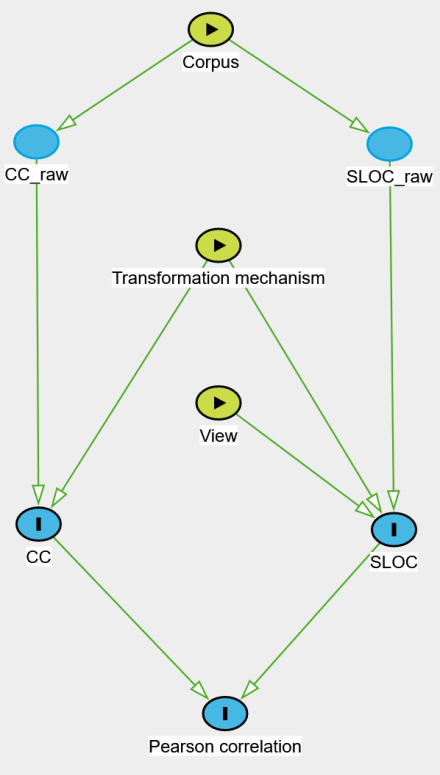

Figure 1: Causal diagram describing the CC & SLOC case

4.1.1 Discussion

The goal of the first case is to learn about the process of theory- and causal diagram construction in practice. Initially this case seems to have many aspects that are required when we want to apply causal inference: a well-defined, large population and clear definitions of what variables are relevant and how to measure them. Without much prior understanding, it was possible to draw up a causal diagram. We can make several observations.

We see that this diagram does not describe any particular evolution of software metrics. Although CC and SLOC both are valid software metrics, this diagram tells us nothing about how software evolves. In this diagram, the exposure dictates a way data is transformed. The diagram describes what influences the correlation between the metrics, it shows us that the manner in which data is processed can impact how data is interpreted. This observation aligns with the original study [LSV14].

Taking a step back we consider that this experiment was not theory oriented. The causal diagram shows us this. Instead, we want the code metrics to be exposure variables. We want to learn about the effect of these exposures.

4.2 Case 2: Co-evolution of code clones

The second case explores the effect of clone co-evolution on future changes to those clones. Code clones are a common occurrence. There are different ways to classify code clones, to start out with a small scope we only look at type 1 clones. A type 1 code clone is defined as: *"code fragments [that] are identical except for variations in white space, layout, and comments"* [RCK09]. Krinke explores the evolution of code clones and defines two different types of clone evolution, consistent and inconsistent. We refer to clone co-evolution as a consistent change to a group of code clones [Kri07]. Co-evolution does not occur if the clones change inconsistently. Krinke finds that roughly half of the code clone groups are changed consistently. He also finds that code clone groups rarely become consistent after inconsistently changing.

Through informal discussion with experts on software evolution from the Software Improvement Group we came up with the notion that some clone groups do in fact become consistent after changing inconsistently. To explore this we create a causal diagram to represent the evolution of code clones, shown in Fig. 2.

We choose the evolution of a clone group as exposure variable. Evolution of a clone is either consistent or inconsistent. By analysing commits, we can find commits that touch on any instance of a clone group.

The outcome variable, Correction, is a binary variable that is counted when, after an inconsistent change to a clone group, this change is applied to the other clone(s) in that clone group. We introduce Test as confounder, a binary variable that shows us if the clone group is a combination of test classes and non-test

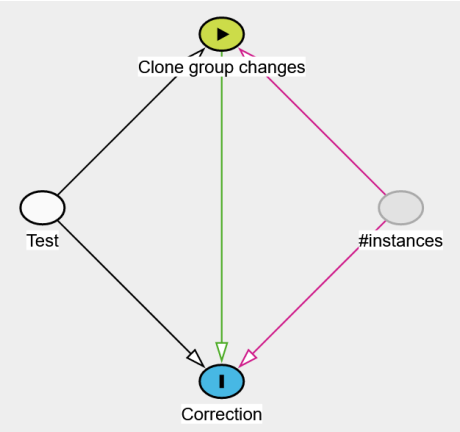

Figure 2: Causal diagram describing effect of a changed clone group on a later correction

classes. Additionally we introduce #instances, a variable that counts the number of instances in a clone group. In Fig. 2 Test has been adjusted for, instances is unobserved.

We start out with one repository, Apache commons-collection package from version 4.2 onward. The initial results on evolution distribution are: 116 clone groups do not evolve, 31 clone groups consistently evolve and 37 evolve inconsistently. This supports data by Krinke that there is roughly a 50/50 split in the way clones evolve. In our data we also see 3 instances of a Correction.

4.2.1 Discussion

This case covers a much simpler example than the first case. The initial population is very small: it consists of the clones in one repository. For reasons mentioned before it is important to use a bigger population for this type of research. However, we still find some notion that this causal diagram may provide interesting information.

A correction can only happen when a clone changes inconsistently, never when a clone has changed consistently. This means that if a correction happens, the fact that a clone changed inconsistently must have an effect. However, as Krinke suggest, this effect may be negligible. In our limited data we find occurrences where such a correction happens. Although the limited amount of data means we cannot conclusively say anything about the effect of a clone group evolving, it warrants further investigation. When analysing the data we find that the clones that require later correction are often clone groups that have both test and regular classes. Seeing this, we introduce it as a potential confounder. We also consider that the number of instances in a clone group might affect the final results.

The next steps in this research are to gather a larger population to select for, and run tests with these different confounders. This second experiment shows us that we can start to incrementally build theories with causal diagrams. We are investigating what the effect of inconsistently changing clones is on a later correction. Further work is needed to determine if we can start to conclusively define causal effects.

5 Related Work

There are other fields where causal inference is applied to observational data to determine effects of exposure [SS19, Her04]. In these fields the usage of causal inference is more mature and has been tried out for various use cases. In their work, Saarimäki describes working towards a methodology for applying observational studies in empirical software engineering [Saa19]. In this paper we explore a practical way to work with observational data. If the field of empirical software engineering is to adapt observational studies and causal inference, guidelines are required. Causal inference is clearly still new in the field of empirical software engineering and more work is needed to determine its utility.

6 Conclusion & Next Steps

These initial findings do not conclusively say if this methodology is an effective way to build theories for software evolution. Although the first case did not end up proving adequate for this type of research, visualising what we are doing with a causal diagram allowed us to understand why that was the case. The second case is simple, but we do see that iteratively adding information to a diagram as we learn from our data is possible.

We reflect on the **Research Question**. We learn that the visual aspect of causal diagrams can help us quickly understand what we are looking at and iteratively build on that. It may be possible to construct theories in this manner, or at least help to facilitate a new type of thinking about theories. To conclusively say anything more investigation is required. The current case will have to be extended and tested extensively. Additionally, new cases will have to be investigated for more definite results. This researched has so far been limited in scope to large scale technical data. Causal inference allows for data to be added from different sources, including qualitative. Further investigation into how this type of data can be included for software engineering research is required.

7 Acknowledgements

The work in this paper has been partially conducted within the scope of the FASTEN project. The FASTEN project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825328.

References

[AS15] Peter Austin and Elizabeth Stuart. Moving towards best practice when using inverse probability of treatment weighting (iptw) using the propensity score to estimate causal treatment effects in observational studies. *Statistics in medicine*, 34, 08 2015.

[BUN12] M. BUNGE. *Scientific Research I: The Search for System*. Studies in the Foundations, Methodology and Philosophy of Science. Springer Berlin Heidelberg, 2012.

[ESSD08] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. *Selecting Empirical Methods for Software Engineering Research*, pages 285–311. 01 2008.

[Her04] M A Hernán. A definition of causal effect for epidemiological research. *Journal of Epidemiology & Community Health*, 58(4):265–271, 2004.

[HW13] Johanna Hunt and Xiaofeng Wang. Report on the Second SEMAT Workshop on General Theory of Software Engineering (GTSE 2013). *ACM SIGSOFT Software Engineering Notes*, 38(5):43–46, 2013.

[JEJ12] Pontus Johnson, Mathias Ekstedt, and Ivar Jacobson. Where's the theory for software engineering? *IEEE Software*, 10 2012.

[Jor04] M. Jorgensen. Generalization and theory-building in software engineering research. pages 29–35, 01 2004.

[Kri07] J. Krinke. A study of consistent and inconsistent changes to code clones. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 170–178, 2007.

[Leh79] M.M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213 – 221, 1979.

[Leh80] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sep. 1980.

[LR05] E. L. Lehmann and Joseph P. Romano. *Testing statistical hypotheses*. Springer Texts in Statistics. Springer, New York, third edition, 2005.

[LSV14] D. Landman, A. Serebrenik, and J. Vinju. Empirical analysis of the relationship between cc and sloc in a large corpus of java methods. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 221–230, Sep. 2014.

[OMS18] Safa Omri, Pascal Montag, and Carsten Sinz. Static analysis and code complexity metrics as early indicators of software defects. *Journal of Software Engineering and Applications*, 11:153–166, 01 2018.

[PBV12] Mohamad Amin Pourhoseingholi, Ahmad Baghestani, and Mohsen Vahedi. How to control confounding effects by statistical analysis. *Gastroenterology and hepatology from bed to bench*, 5:79–83, 03 2012.

[Pea09] Judea Pearl. Causal inference in statistics: An overview. *Statist. Surv.*, 3:96–146, 2009.

[PGJ16] J. Pearl, M. Glymour, and N.P. Jewell. *Causal Inference in Statistics: A Primer*. Wiley, 2016.

[RCK09] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470 – 495, 2009.

[REN+14] Paul Ralph, Iaakov Exman, Pan-Wei Ng, Pontus Johnson, Michael Goedicke, Alper Tolga Kocata, and Kate Liu Yan. How to Develop a General Theory of Software Engineering. *ACM SIGSOFT Software Engineering Notes*, 39(6):23–25, 2014.

[RGL08] K.J. Rothman, S. Greenland, and T.L. Lash. *Modern Epidemiology*. Wolters Kluwer Health/Lippincott Williams & Wilkins, 2008.

[RJJ13] Paul Ralph, Pontus Johnson, and Howell Jordan. Report on the first SEMAT workshop on general theory of software engineering (GTSE 2012). *ACM SIGSOFT Software Engineering Notes*, 38(2):26–28, 2013.

[Saa19] Nyyti Saarimaki. Methodological issues in observational studies. 2019.

[SAB18] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. *PyDriller: Python Framework for Mining Software Repositories*. 2018.

[SDAH08] Dag Sjøberg, Tore Dybå, Bente Anda, and Jo Hannay. *Building Theories in Software Engineering*, pages 312–336. 01 2008.

[SGJ16] Klaas-Jan Stol, Michael Goedicke, and Ivar Jacobson. Introduction to the special section—General Theories of Software Engineering: New advances and implications for research. *Information and Software Technology*, 70:176–180, 2016.

[SK15] Sapra SK. Causality inference with observational data in economics. *Business and Economics Journal*, 7, 01 2015.

[SS19] Steven D Stovitz and Ian Shrier. Causal inference for clinicians. *BMJ Evidence-Based Medicine*, 24(3):109–112, 2019.

[SSC+02] WS SHADISH, W.R. Shadish, E. Chelimsky, T.D. Cook, D.T. Campbell, and Cengage Learning. *Experimental and Quasi-experimental Designs for Generalized Causal Inference*. Houghton Mifflin, 2002.

[YB15] Afsaneh Yazdani and Eric Boerwinkle. Causal inference in the age of decision medicine. *Journal of data mining in genomics proteomics*, 6, 2015.

[ZR96] Hadar Ziv and Debra Richardson. The uncertainty principle in software engineering. 09 1996.