A Domain Analysis of Data Structure and Algorithm Explanations in the Wild

Anonymized for Review

ABSTRACT

Explanations of data structures and algorithms are complex interactions of several notations, including natural language, mathematics, pseudocode, and diagrams. Currently, such explanations are created ad hoc using a variety of tools, and the resulting artifacts are static, reducing explanatory value. We envision a domain-specific language for developing rich, interactive explanations of data structures and algorithms. In this paper, we analyze this domain to sketch requirements for our language. We perform a grounded theory analysis, to generate a qualitative coding system for explanation artifacts collected online. We show that grounded theory provides a robust methodology for analyzing qualitative objects and that the resultant coding system forms the skeleton of a domain-specific language. This work is part of our effort to develop the paradigm of explanation-oriented programming, which shifts the focus of programming from computing results to producing rich explanations of how those results were computed.

ACM Reference format:
Anonymized for Review. 2017. A Domain Analysis of Data Structure and Algorithm Explanations in the Wild. In *Proceedings of SIGCSE, Seattle, Washington USA, March 2017 (Seattle'2017),* 10 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

Data structures and algorithms are at the heart of computer science and must be explained to each new generation of students. A pressing question is: How can we do this effectively?

In this paper, we focus on the *artifacts* that constitute or support explanations of data structures and algorithms (hereafter just "algorithms"), which can be shared and reused. For verbal explanations, such as a lecture, the supporting artifact might be the associated slides. For written explanations, the artifact is the explanation as a whole, including the text and any supporting figures. Explanation artifacts associated with algorithms are interesting because they typically present a complex interaction among many different notations, including natural language, mathematics, pseudocode, executable code, various kinds of diagrams, animations, and more.

Currently, explanation artifacts for algorithms are created ad hoc using a variety of tools and techniques, and the resulting explanations tend to be static, reducing their explanatory value. Although there has been a substantial amount of work on algorithm visualization [6–10, 13], and tools exist for creating these kinds of supporting artifacts, there is no good solution for creating integrated, multi-notational explanations as a whole. Similarly, although some algorithm visualization tools provide a means for the student to tweak the parameters or inputs to an algorithm to generate new visualizations, they do not support creating cohesive interactive explanations that correspondingly modify the surrounding explanation or that allow the student to respond to or query the explanation in other ways. To fill this gap, we envision a *domain-specific language* (DSL) that supports the creation of rich, interactive, multi-notational artifacts for explaining algorithms. The development of this DSL is part of a larger effort to explore the new paradigm of *explanation-oriented programming*, briefly described in Section 2.1.

The intended users of the envisioned DSL are CS educators who want to create *interactive artifacts* to support the explanation of algorithms. These users are experts on the corresponding algorithms, and also trained and skilled programmers. The produced explanation artifacts might supplement a lecture or be posted to a web page as a self-contained (textual and graphical) explanation. The DSL should support pedagogical methods directly through built-in abstractions and language constructs. It should also support a variety of forms of student interaction. For example, teachers should be able to define equivalence relations enabling users to automatically generate variant explanations [5], to build in specific responses to anticipated questions, and to provide explanations at multiple levels of abstraction.

This paper represents a formative step toward this vision. We conduct a *qualitative analysis* of our domain in order to determine the form and content of the explanation artifacts that educators are already creating. We base our analysis on an established qualitative research method called *grounded theory*[14] in order to better understand how well existing artifacts explain complex topics

More specifically, we collect 15 explanation artifacts from the internet, consisting of only lecture notes that explain two algorithms and one data structure commonly covered in undergraduate computer science courses: Dijkstra's shortest path algorithm [11, pp. 137–142], merge sort [11, 210–214], and AVL trees [12, pp. 458–475]. We analyze these artifacts through the application of grounded theory[14], a formal method for analyzing qualitative data that originated in sociological research. Through the application of grounded theory, we develop a coding system that captures the structure of explanation for each document. An overview of the coding system is given in Section 4.1.

This paper makes the following contributions:

C1. We provide a case study on analyzing *qualitative data* through the application of a formal research method *grounded theory*, that is not part of the computer science parlance.

Seattle'2017, March 2017, Seattle, Washington USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

C2. We provide a coded qualitative data set of explanation artifacts, using the system defined in C3 applied to our sample of 15 collected explanation artifacts (Section 3).

C3. 1.decide if we need to split this contribution up We provide a coding system for analyzing *explanation artifacts* in the form of lecture notes, and we show that through the application of the coding system, each such artifact forms a tree structure, which we have termed a *explanation tree*

2.tie this in One of the central dilemmas of analyzing explanations in the wild is that they vary widely and do not directly map onto any perceivable shared, abstract structure. If explanation-oriented programming is to become a viable paradigm then this formative problem must be solved. A major step forward in this direction is the concept of *explanation trees*. In this paper, use a coding system grounded in the data to observe a shared structure between explanations of like content and dissimilar content, lengths, authors, and institutions. We believe that such a structure forms the building blocks of 3.something (Section 4.2).

C4. We describe how a coding system grounded in data can provide a semantics basis for a DSL and argue for the advantages of such an approach (Section **??**).

2 BACKGROUND AND RELATED WORKS

In this section, we put the present work into context by the describing the paradigm of *explanation-oriented programming* in Section 2.1, the exploration of which is an underlying motivation of our work, and by introducing the methodology of grounded theory [14] in Section 2.2, which is the theoretical foundation of our coding system.

2.1 Explanation-oriented programming

Explanation-oriented programming (XOP) is a new programming paradigm where the primary output of a program is not a set of computed values, but an *explanation of how* those values were computed [2–5, 15]. A high-level goal of this work is to further develop the paradigm of XOP through the development of a specific DSL.

Programming languages for XOP should not merely produce explanations as a byproduct, but should provide abstractions and features specific to the creation of interactive explanation artifacts. For example, they should provide facilities for creating application-specific notations and visualizations (which are widespread in explanations of algorithms), and for describing alternative explanations produced in response to user input, for example, at different levels of abstraction, by parameterization, or generated by explanation equivalence laws [5]. Additionally, languages for XOP should help guide the programmer toward the creation of *good* explanations.

The need for interactive explanation artifacts is motivated by the observation that there is a trade-off between personal explanations and traditional explanation artifacts, which can be partially bridged by XOP programs viewed as rich, interactive explanation artifacts. A good *personal explanation* is useful because the explainer can *respond* to the student, adjusting the pace and strategy as necessary. For example, the teacher can answer questions, rephrase parts of an explanation, and provide additional examples as needed. Unfortunately, good personal explanations are a scarce resource. First, there are limited number of people who can provide high quality personal explanations on a topic. Second, a personal explanation is usually ephemeral and so cannot be directly shared or reused. Since personal explanations are hard to come by, many students learn from *impersonal explanation artifacts*, such as recorded lectures, textbooks and online written and graphical resources. These impersonal explanations lack the interaction and adaptability of personal explanations, but have the advantage of being easy to massively share and reuse via libraries and the internet.

In-person lectures, such as those covering algorithms in most undergraduate computer science programs, exist at a midway point between impersonal and personal explanations, perhaps closer to the personal end of the spectrum. These *classroom explanations* are adaptable—students can ask questions in class, the teacher can respond, and explanations can be adapted on the fly if students are confused—but they are not as adaptable as personal explanations since the teacher must accommodate many students at once. Classroom explanations are more efficient than personal explanations since they are shared amongst many students, but not as efficient as impersonal explanations since they are still ephemeral and therefore difficult to reuse.

We target another midway point, a bit closer to the impersonal end of the spectrum, of *interactive explanation artifacts* that provide as much of the responsiveness and adaptability of personal explanations as possible, but which can still be massively shared and reused online. Such an explanation artifact would be quite expensive to produce with current tools since an *explanation designer* must not only construct a high quality initial explanation and corresponding visualizations, but also anticipate and explicitly program in responses to queries by the student. We expect that DSLs for XOP can help alleviate this burden.

2.2 Grounded Theory

In this section we present an overview of the research method known as grounded theory. This section is meant to help orient the reader to understand the process by which the coding system was formed.

The central idea behind grounded theory is to generate, or discover theory, inductively, *based on data* rather than to use data to evince a hypothesis of a theory. Grounded theory is rooted in a *pragmatist* view of theory i.e. that theory should be purposed and suited towards its intended uses vis-a-viz logico-deductive theories which are concerned with what can be expressed by the theory [14]. As Glaser and Strauss state:

"A grounded theory is one that is inductively derived from the study of the phenomena it represents".

Like any methodology, grounded theory employs a specific vocabulary to refer to stages and phases of research. The rest of this section will introduce grounded theory terminology in concert with an operational example that will show how one may actually perform a grounded theory analysis.

Imagine a researcher who wants to find out why students drop out of Computer Science programs. The first task for a grounded theorist would be to collect data from the subject they wish to study. This data would most likely be in the form of interviews with the students, their schedule, and example homework assignments. Once some data is collected the research will begin *coding*, coding is the act of taking qualitative data e.g lecture notes, an interview, personal letters and assigning tags that describes the data.

Coding consists of three stages[1]: First, a researcher performs *open coding*. In open coding, one writes down *any* term or terms that describes the data, regardless of duplicate or related concepts. In our example these tags could be things such as the year of a student, nicknames that people call one another, things about their life, family, or activities. Open coding should make use of *in vivo* terms, these are terms that people immersed in the culture use. For example, a computer scientist may discuss the "recurrence relation", while a factory worker may use the term "floor boss" to mean the manager.

The second stage of coding is called *axial coding*, in axial coding the researcher identifies the relationships between the tags developed in open coding. The goal is to develop a *coding paradigm*, which is a model that displays the inter-relationship between codes [1]. An axial code for our example may coalesce sets of open codes to several axial codes. For example, if one had the open codes "tired, exhausted, no time for other activities, stressed from homework", these might inform the axial code "burn out".

selective coding is the final stage of coding, in this stage the researcher tries to identify a few central tags that form the basis for the theory. After identifying these core tags the researcher then seeks to relate the core tags to the remaining tags. In our example, we might have the axial codes "burn out, no support, everyone is perceived better, anxiety over schoolwork" and conclude that the selective code would be "impostor syndrome".

All phases of research proceed in parallel to data collection and analysis. That is to say that the researcher in our example will be simultaneously collecting data, coding, forming their theory, and moving between each activity at the same time . This is a marked departure from quantitative methods where phases are distinct and disjunct[14].

Through coding, the researcher interacts with their data. But how does that researcher know when to stop coding? Or how much data to code? Such questions are the motivation for three central tenets in grounded theory.

The first is called *constant comparison*, the idea is that during coding, the researcher must think back and ask if they are being consistent. Researchers should perform this check constantly and at every stage of coding as it is pivotal to the validity of the method[14].

The second tenet is *theoretical sampling*; theoretical sampling, focuses on filling the *perceived* gaps in data based on current theory. To put this into context, if the researcher in our example were a quantitative researcher they would determine the total student population of the CS program, and then use statistics to generate a representative data set. The grounded theory researcher, would look at their data and determine that they do not have any data points from people who have taken the operating systems class, thus they will collect more data to fill that gap *based on their current results*.

The last tenet describes how to reach an end point in coding; grounded theorists call it *saturation*. Saturation occurs when new data is added to the researcher's data set and no new tags or modifications to the coding system are required. This is when the researcher's system is able to express data that was *not* used to generate it. The requisite number of data to reach this point may vary drastically based on factors such as researcher, topic, and motivating research questions.

3 EXPERIMENTAL SETUP

In this section we describe how the data was collected, any preprocessing the data may have undergone, and the system used to apply tags to the data.

We restrict the scope of data collection to include only artifacts that provide an explanation of common computer science algorithms from computer science departments at accredited universities. Restricting the scope of artifacts in this manner provides two benefits: 1) All explanatory artifacts have a stated, intrinsic goal to communicate the mechanics, application, or implementation of some common computer science algorithm. 2) There are bountiful and varied examples of different approaches to explain *the same* algorithm, and numerous examples of *like* approaches to explain different algorithms. All data collected was either in PDF file format, or an html file that was converted to PDF. No powerpoint lecture notes were considered because we feared that such documents would be information scarce; powerpoints are used to structure a lecture and thus should be considered with the concomitant lecture.

All data was coded by hand with the aid of Atlast.ti software 4.reference for software here by a single researcher. The use of the Atlast.ti software was primarily for organizational and representational benefit, such as generating the code list and performing constant comparison. The use of such software is contingent with respect to grounded theory. If a document drifted substantially into a different algorithm then that section was not considered. For example, if a document explaining Dijkstra's algorithm, discussed shortest-path problems, then this was considered as it is directly relevant to explaining Dijkstra's algorithm, but if that document included a section on explaining Bellman-Ford's algorithm, then this section was not considered as it's purpose is not to *explain* Dijkstra's algorithm.

Explanatory artifacts were indexed according to the algorithm that was to be explained, and a simple counter. For example, the first document regarding AVL Trees would be named "AVT01" while the sixth would be "AVT06" and so on. All of the data in this paper was collected and analyzed by a single researcher.

We made no attempt to restrict the size of the data collected, in order to collect a diverse set of data. The smallest document collected has only a page and a half of relevant material while the longest consists of eighteen pages. All data can be found the projects github repository[1] Fifteen documents were coded in total, with five documents each per algorithm.

[1] anonymized for review

4 RESULTS

In this section we present our findings. We make three major observations from the data. First, we observe that the data is able to be expressed in the coding system and provide an overview of the coding system in Section 4.1. In Section 4.2 we observe that when the data is expressed through the lens of the coding system, the result is an explanation tree, and that the content of each document is expressed with a pre-order traversal of the corresponding explanation tree. **5.any more observations? That gt is well suited to a task like this?**

4.1 The Coding System

In this section we present an instructive example of the coding system developed through grounded theory. There are several instances of each type of tag that will not be shown or discussed here. For an exhaustive list please see this projects github repository[1]:

The coding system consists of 4 finite sets of tags, these were modeled as "groupings" in Atlast.ti, they are:

[1] anonymized for review

Tag	Description
Structure	Structure tags control how the context in the artifact is set
→	Push: takes one content tag and sets the context to the conjunction of the input tag and parent context
←	Pop: takes any number of action or expression tags as input, sets context up one level, and then applies input tags.
|	Swap: takes one content tag as input, synonymous with the sequence ←, → <input tag>
↞	return: takes no input, pops until top level is reached
Content	Content tags denote *what* is being discussed in a snippet of text
Advantages	The text is discussing the pros, the upside or the advantages of the parent context.
Algorithm	the text is discussing an algorithm in general
Application	the text is discussing the use cases, or applications of the parent context
Class	denotes the explicit discussion of a group, set or class of some thing
Complexity	the text is discussing the computational complexity of the parent context.
Condition	the text is discussing a condition that whatever the context was set to has, that must be satisfied.
Constituent	the text is discussing some constituent part of the parent context
Data Structure	the text is discussing a Data Structure, this is an analog to the Algorithm code.
Design	the text is talking about the design, or design considerations of the parent context
Disadvantages	the text is discussing the downsides, the cons or shortfalls of the parent context
Goal	the text is discussing the goal, the end game, that which is the desired outcome, of the parent context
History	the text is discussing the history of the parent context
Implementation	the text is discussing implementation details of the parent context
Motivation	the text is discussing the motivation for the parent context
Operation	the text is discussing an Operation that is a requisite and central part of the parent context
Problem	the text is discussing a problem that may or may not be solved later in the document
Property	the text is discussing some property of the parent context context.
Solution	the text is discussing a solution to some prior introduced problem, which is typically in the parent context
State	the text is now discussing something related to the state or the state of the parent context
Actions	Action tags denote *how* the document or author is discussing the context.
Abstraction	The text is abstracting or generalizing that which the context is set to
Assumption	The text is giving the reader or telling the reader an assumption about the context
Base Case	The text is giving an explicit base case in an inductive procedure in relation to the context
Cases	The text is breaking down the context into chunks of information, or cases.
Comment	a dummy code whose use is just to provide a binding for an Aside, Caveat, or Meta modifier
Conclusion	The text is making a conclusion about the context
Definition	The text is defining some term about the context.
Derivation	The text is making a derivation about something in relation to the context
Description	This code is the most general Action code. It denotes that the text is describing the context in some manner.
Example	The text is giving, or providing an example in relation to the context
Implication	The text is giving an implication about the context. This could be anything that fits the logical connective if..then..else..
In Vivo	The text is defining a term that practitioners of the context would be familiar with
Legend	The text is giving a legend to understand something
Observation	The text making a general observation about the context
Outline	The text is giving a bulleted list of the content the document will go through
Proof	The text giving a mathematic or logical proof about the context
Proposal	The text suggesting a path forward.
Solicitation	The text is explicitly asking something of the reader
Summary	This code denotes a concluding block of text that summarizes the previous contexts
Expression	Expression tags denote *how* the content is actually expressed. If not otherwise stated, assumed to be text
Cartoon	The content is represented in a drawn or animated graphic
Code	The content is represented as a block of code from some programming language
Mathematic	The content is represented using Mathematic formulae, variables, or equations
PseudoCode	The content is represented in pseudocode; a programming-like language that is not a executable programming language.
Sequence	The content is being expressed in a ordered, bullted or punctuated way
Table	The content is explicitly displayed in a Table
Modifiers	each modifier takes any number of tags of any type as input and alters the meaning of the input tags in a specified way
Aside	Denotes that the input tags describe text that is not directly related to the scope the input tags refer to
Caveat	Denotes that the purpose of the text is to further clarify a point, provide extra detail
Meta	Denotes text that is about the current context but not directly explaining it
Pedagogical	Denotes that the specific purpose of a statement is pedagogical in nature
Related	Denotes that the input tags are substantially related to the parent context in some manner
Review	Denotes that the purpose of the input tags is to provide a pedagogical review of the material to the reader.

A typical coding for a paragraph might be:

1	2.2 Mergesort	Algorithm
2	The algorithms that we consider in this section is based on a simple operation known as merging: combining two ordered arrays to make one larger ordered array.	Description In Vivo
3	This operation immediately lends itself to a simple recursive sort method known as mergesort: to sort an array, divide it into two halves, sort the two halves (recursively), and then merge the results.	Description
4	<cartoon of list>	Cartoon
5	Mergesort guarantees to sort an array of N items in time proportional to N log N, no matter what the input.	-> Motivation Description
6	Its prime disadvantage is that it uses extra space proportional to N.	| Disadvantage Description
7	Abstract in-place merge.	| Operation

Table 2: Sample text and codings for beginning of MS03, bolded text are headers

6.fix location and formatting, ask eric what this should look like

Where "Algorithm", "Motivation", "Problem", are Context tags; →, |, and ← are modifiers; "Description" and "Example" are Actions and "Cartoon" is a Content Expression tag. The ... are used here to denote several lines or paragraphs of text or images. Shown in the example above, modifiers determine the scoping of their input tags. When a context tag is *orphaned*, like "Algorithm" above, then that is interpreted as returning to the top level scope and then sub scoping in to the "Algorithm" context. The first tag in every document is interpreted as setting the *root* node of that particular document. So if the example above was the first line in a document and "Algorithm", the first context tag, then it would be interpreted as setting the root node for that document. The modifier →, read as *push*, takes a content tag as input, and creates a sub-scoping of that content tag *with respect to the parent*. So in the example above, we have "Algorithm" as the parent, and then we push into "Motivation", thus the document would now be discussing the motivation for the algorithm at the parent node.

The modifier ←, read as *pop*, denotes the opposite, it moves up one level of scoping. The pop operator takes no content tags and instead may be orphaned or may take any number of action any content expression tags. In the example above this modifier is used to exit the "Problem" scope, return to the "Algorithm" scope and then give an example. The pop modifier may be orphaned to denote an exiting of scope without any action or context expression occurring.

The modifier |, read as *swap*, takes a single content tag as input, and is equivalent to the coding sequence "← → <tag>". That is to say we pop out of the current sub-scoping, and then push into the new sub-scoping of the input tag. Note that the pop modifier has precedence over the push modifier.

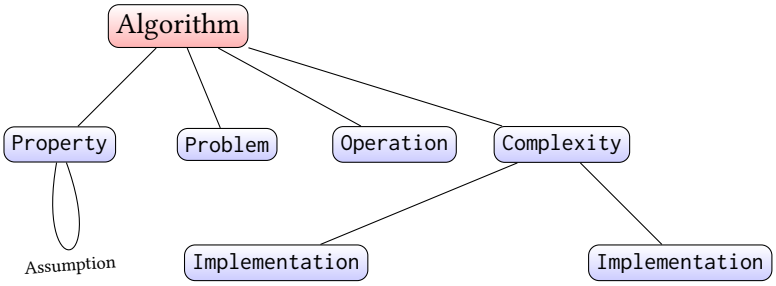
Context tags define, in abstract, the topic that is being discussed in a particular section of the document. These are abstract descriptors such as: Problem, to denote that the text is discussing a problem which may or may not be solved later in the document. Motivation, to denote that the text is discussing the reason for discussing or analyzing whatever chain of content the scoping is set to. These tags, with the modifiers, form the skeleton of an explanation tree; where the content tags are the nodes and the modifiers dictate the structure. In essence, these tags dictate what the document is discussing.

Action tags are used to denote the explicit action that is being undertaken by the document. These tags must be concrete things that are being done in the text, such as: Giving an example, providing an outline, describing something, concluding a point, or giving a proof. These tags denote, in abstract, what the document is doing.

Lastly, Expression tags are used to denote how the content is being expressed in the current context. Some examples are: Cartoon, PseudoCode, Code, and Table. These tags denote, in abstract, what form the content is being expressed in.

4.2 Explanation Trees

The coding system borne out of the data is, in essence, one used to describe a tree structure. This conclusion was born out of necessity during axial coding and became the *coding paradigm* for this study. The central problem was to keep track of the different topics each document would discuss throughout its' duration. This led directly to the creation of Content and Modifier tags, which upon retrospection clearly formulate the document into a tree structure. An example tree is given below for document **7.pick good example explanation tree**.



8.Fix this caption, and figure, this was just proof of concept

Figure 1: Explanation Tree for Dijkstra's 009

As shown in figure 1, the document ... ... **9.finish this description of the explanation tree**

5 DISCUSSION OF RESULTS

REFERENCES

[1] J. Corbin, A. Strauss, and A.L. Strauss. 2014. *Basics of Qualitative Research*. SAGE Publications. https://books.google.com/books?id=0RlElwEACAAJ

[2] Martin Erwig and Eric Walkingshaw. 2008. A Visual Language for Representing and Explaining Strategies in Game Theory. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing (VL/HCC)*. 101–108.

[3] Martin Erwig and Eric Walkingshaw. 2009. A DSL for Explaining Probabilistic Reasoning. In *IFIP Working Conf. on Domain-Specific Languages (DSL) (LNCS)*, Vol. 5658. 335–359.

[4] Martin Erwig and Eric Walkingshaw. 2009. Visual Explanations of Probabilistic Reasoning. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing (VL/HCC)*. 23–27.

[5] Martin Erwig and Eric Walkingshaw. 2013. A Visual Language for Explaining Probabilistic Reasoning. *Journal of Visual Languages and Computing (JVLC)* 24, 2 (2013), 88–109.

[6] Peter Gloor. 1997. Animated Algorithms. In *Elements of Hypermedia Design: Techniques for Navigation & Visualization in Cyberspace*. Birkhäuser, Boston, 235–241.

[7] Peter A. Gloor. 1992. AACE – Algorithm Animation for Computer Science Education. In *IEEE Workshop on Visual Languages*. 25–31.

[8] Steven Hansen, N.Hari Narayanan, and Mary Hegarty. 2002. Designing Educationally Effective Algorithm Visualizations. *Journal of Visual Languages and Computing (JVLC)* 13, 3 (2002), 291 – 317. https://doi.org/10.1006/jvlc.2002.0236

[9] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. 2002. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing (JVLC)* 13 (2002), 259–290.

[10] Charles Kann, Robert W. Lindeman, and Rachelle Heller. 1997. Integrating algorithm animation into a learning environment. *Computers and Education (CE)* 28, 4 (1997), 223 – 228. https://doi.org/10.1016/S0360-1315(97)00015-8

[11] Jon Kleinberg and Eva Tardos. 2006. *Algorithm design*. Pearson Education, Boston.

[12] Donald E. Knuth. 1998. *The Art of Computer Programming: Sorting and Searching* (2nd ed.). Pearson Education, Boston.

[13] Clifford A Shaffer, Matthew L Cooper, Alexander Joel D Alon, Monika Akbar, Michael Stewart, Sean Ponce, and Stephen H Edwards. 2010. Algorithm visualization: The state of the field. *ACM Transactions on Computing Education (TOCE)* 10, 3 (2010), 9.

[14] Anselm Strauss and Juliet Corbin. 1967. Discovery of grounded theory. (1967).

[15] Eric Walkingshaw and Martin Erwig. 2011. A DSEL for Studying and Explaining Causation. In *IFIP Working Conf. on Domain-Specific Languages (DSL)*. 143–167.