1.anonymize authors

2.CCSXML

3.figure out keywords

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

Jeffrey Young
Oregon State University
School of EECS
Corvallis, Oregon, USA

Eric Walkingshaw
Oregon State University
School of EECS
Corvallis, Oregon, USA

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. **4.I want to say something here about using formal qualitative methods to expand the reach of computer science**

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

ACM proceedings, LaTeX, text tagging

ACM Reference format:
Jeffrey Young and Eric Walkingshaw. 2017. A Domain Analysis of Data Structure and Algorithm Explanations in the Wild. In *Proceedings of SIGCSE, Seattle, Washington USA, March 2017 (Seattle'2017),* 6 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* 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,

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

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, 5.cite Strauss and Corbin on first grounded theory paper 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 Section6.create grounded theory overview section.

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.

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 **??**).

C3. 7.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*.

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

2 BACKGROUND

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 **??**, 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 nd 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 expected 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, in addition to providing a summary of a qualitative research method that has seen little use inside computer science.

2.2.1 Main Idea. Grounded theory was discovered by sociology researchers, Glaser and Strauss in the late sixties. Its central idea 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 with the theory [14]. As Glaser and Strauss state:

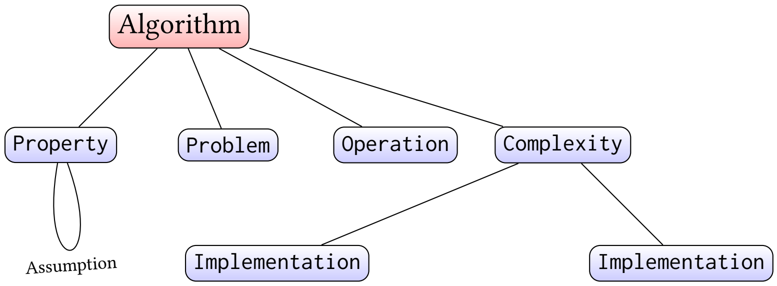
"A grounded theory is one that is inductively derived from the study of the phenomena it represents".

2.2.2 Terminology. Like any methodology, grounded theory employs a specific vocabulary to refer to stages and practices of research. The first such term is called *coding*, coding is the act of taking some qualitative data e.g lecture notes, an interview, personal letters and assigning tags that *captures the essence* of the data.

The act of coding consists of three stages: First, a researcher performs what is known as *open coding*. In open coding, one writes down *any* term or terms that describes the data. This first set of tags is often varied, numerous, and will typically consist of many concepts that are expressed in duplicate, or related tags. The second stage of coding is called *Axial Coding*, in axial coding the researcher tries to identify the relationships between the tags that were developed in open coding. The goal of this step is to develop a *coding paradigm*, which is a theoretical model that visually displays the inter-relationship between codes [1]. *Selective Coding* is the last and final stage of coding, in this stage the researcher tries to identify one or two central tags that forms the basis for their theory. After identifying these core tags the researcher then seeks to relate the core tags to the remaining tags.

2.2.3 How it is done. In contrast to quantitative research methods, grounded theory does not proceed linearly through its methodology. For instance, quantitative methods are typically formulated into broad steps e.g. form hypothesis, design experiment, gather data, analyze data, draw conclusions. Rather a grounded theory researcher will proceed in parallel through the In this sense the theory is *grounded* in the data.

2.3 How it was used



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

Figure 1: Explanation Tree for Dijkstra's 009

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.