Small-step pipelines reduce the complexity of XSLT/XPath programs

Marcel Schaeben*

m.schaeben@uni-koeln.de

Cologne Center for eHumanities

Köln, Germany

Gioele Barabucci*

gioele.barabucci@ntnu.no

Norwegian University of Science and Technology

Trondheim, Norway

ABSTRACT

As code is adapted to deal with unclean external data, it tends to grow in size and complexity.

The use of *small-step pipelines* (a programming style in which data is manipulated through small, independent steps; a variation on the point-free and concatenative programming paradigms) has been suggested as a way to reduce this complexity, resulting in simpler programs.

Our preliminary quantitative results show that writing data-curation and data-analysis XSLT/XPath programs as small-step pipelines leads to a significant reduction of the peak McCabe cyclo-matic complexity. This reduction of complexity is associated with a parallel increase in readability of the resulting code.

CCS CONCEPTS

• **Software and its engineering** → **Error handling and recovery**; *Software design techniques*; *Software development techniques*; *Software functional properties*; • **Applied computing** → *Document management and text processing*.

KEYWORDS

Pipelines, cyclomatic complexity, data curation, point-free style

ACM Reference Format:

Marcel Schaeben and Gioele Barabucci. 2021. Small-step pipelines reduce the complexity of XSLT/XPath programs. In *ACM Symposium on Document Engineering '21* (DocEng '21), August 24–27, 2021, Limerick, Ireland. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3469096.3474922>

1 INTRODUCTION

Code that analyzes user-provided data tends to become more complex as it is modified to deal with imperfections, exceptions and edge cases in the input data. [10]

Take for example this collection of expenses encoded in XML:

```
<expenses>
  <!-- all expenses in EUR -->
  <expense id="1" person="gb" value="10" />
  <expense id="2" person="ms" value="1000" />
  [...]
```

*Both authors contributed equally to this research.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

DocEng '21, August 24–27, 2021, Limerick, Ireland

© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8596-1/21/08.

<https://doi.org/10.1145/3469096.3474922>

Summing up the expenses using XSLT/XPath is trivial:

```
<xsl:value-of select="sum(//expense/@value)"/>
```

However, as we add more features (e.g., support for other currencies, support for default currencies, normalization of currency names), the code becomes much more complex:

```
<xsl:value-of select="sum((
  //expense[upper-case(@currency)='EUR' or
  not(@currency)]/@value,
  //expense[upper-case(@currency)='USD']/
  (@value * $usd-to-eur)
)"/>
```

In particular, the main intent of the code is now hard to grasp, buried in additional code that deals with fixing imperfections in the data.

The *small-step pipeline* programming style [3] has been proposed as a solution to the problem of secondary code that deals with imperfections in the data complicating and obfuscating the core code that represents the main intent of the analysis.

The main idea of *small-step pipelines* is that each analysis should be decomposed into a pipeline of individual programs, each of which changes a single aspect of the input data. The final step of the pipeline, the core analysis, can then be implemented assuming perfect, flawless data, eliminating the need to mix cleanups and business logic. The principles behind small-step pipelines are similar to that of point-free programming (UNIX pipelines) or concatenative languages, but with key differences (e.g., constraints on the operations carried out by the intermediate steps).

While practitioners report [1, 3, 5, 7–9] that small-step pipelines simplify writing programs that deal with data imperfections, such improvements are yet to be proved or studied in a quantitative way.

Our main contribution in this paper is the study of the complexity of XSLT/XPath programs written in the small-step pipeline style. Our results (Section 5) show that the peak cyclomatic (McCabe) complexity is reduced on average by a factor of 1.5, and up to 2.5 for programs dealing with more complex data imperfections.

2 BACKGROUND: CODE COMPLEXITY GROWS AS DATA EVOLVES

Practitioners know that code that deals with external data has a tendency to grow in size and complexity. As Alan Perlis put it: "*In the long run every program becomes rococo – then rubble*".[16]

When code evolves organically, one fix at a time, to address and smooth out issues in the data, it tends to lead to problems such as shotgun parsing (where "input recognition is intermixed with input processing throughout the code base" [15, 20]) or redundant defensive programming ("check once more, have many assertions" [13]).

Fixing one issue at a time is, however, a necessity in most concrete situations, where incoming data (or data *in the wild*) is often ridden with small imperfections that must be accepted to maintain interoperability. Even in the best designed closed systems, the natural evolution of data formats forces programs to deal with different versions of the data.

We see five main issues that lead to an increase of complexity in code: 1) default values (e.g., attributes that have a default when not present), 2) values that require normalization (e.g., values that must be treated as case-insensitive), 3) values that require conversions (e.g., amounts in different currencies), 4) exceptions (e.g., elements that must be ignored if preceded by another element with a certain value) and 5) edge cases (e.g., all items are separated by a comma except the last one).

For the sake of brevity, in this paper we will refer to all of the above problems as *data imperfections*.

Because the aforementioned issues are unavoidable, it is of paramount importance that programs are structured in a way in which future extensions, corrections or adaptations will not only be possible, but will fit naturally into the architecture of the program without making its complexity explode.

In particular, for the sake of readability of the produced code, it is essential to make sure that the core business logic, the "intent of the code", remains clear and uncluttered by all the extra details needed to handle edge cases. In this paper, we refer to this core business logic as *data analysis* or *core analysis*.

3 USING SMALL-STEP PIPELINES TO REDUCE COMPLEXITY

The use of *small-step pipelines* has been suggested as an approach to constrain the growth in complexity of programs that must deal with imperfect input data or evolving data formats. [3]

The small-step pipeline approach is characterized by the following properties:

1. **Curation before analysis** Data is curated (i.e., imperfections are smoothed out, edge-cases are handled) before it is analyzed;
2. **Small steps** The data-curation phase is deconstructed into small steps;
3. **One problem at a time** Each data-curation step fixes only one specific problem;
4. **Either read or fix** A data-curation step can only use a piece of data or fix it, but not both.

Property 1 (curation before analysis) makes sure that the core analysis never runs on the raw original data, but on a refined version in which all problems have already been fixed. This allows the code of the analysis to avoid checks and focus on the core task, greatly improving its readability.

Properties 2 (small steps) and 3 (one problem at a time) suggest that, much in the style of UNIX pipelines, the complex task of curating the data should be split into a pipeline of smaller, independent tasks, each of which receives as input the output of the previous one and focuses on fixing one particular problem.

Property 4 (either read or fix) is the key property that differentiates the small-step pipelines from similar approaches. We will use an example to summarize how it works. Suppose an attribute has

a default value. A conventional program would have to check for the presence of that attribute before reading its value and, if not present, use the default value. In a program written as a small-steps pipeline this would not be allowed. Instead, two steps would be required: 1) a step that checks the presence of the attribute and, if missing, adds it with the default value, and 2) another step that assumes the presence of the attribute and just reads it.

All these properties work together to reduce the complexity of the whole program. In addition, and perhaps most importantly, these properties ensure that the code performing the core analysis is more concise and thus more readable.

Past experiences by various practitioners [1, 3, 5, 7–9] have shown that programs written in this style are perceived as easier to write and to maintain over time.

In the rest of the paper we provide quantitative evidence showing that:

- (1) The complexity of the code implementing the core analysis in a small-step pipeline (the last step) is significantly lower than that of an equivalent program written in a conventional way.
- (2) The peak complexity of the whole pipeline (i.e., the data-curation steps and the last step with the core analysis) is lower than the complexity of a conventional program that carries out the core analysis while handling all data imperfections.

4 METHODS AND COMPARATIVE ANALYSIS

To evaluate in quantitative terms the complexity of XSLT/XPath programs, we implemented a series of tasks in two versions: the conventional version and the small-step pipeline version. We then measured and compared their cyclomatic complexity (following McCabe's definition [14]).

We devised in total 8 tasks that act on 2 sets of input data.

The sets of input data are:

Expenses A record-oriented list of expenses in various currencies, each carried out by a specific user.

Paragraphs A literary text with XML mixed content and hyphenated words split across multiple lines.

The tasks are:

expenses-1-all-paid-in-eur Find all expenses that have been paid in EUR.

expenses-2-sum-of-eur-expenses Sum all expenses that have been paid in EUR.

expenses-3-sum-of-all-expenses Report the total in EUR of all expenses, including those made in other currencies.

expenses-4-big-expenses Report which expenses are above a certain threshold.

expenses-5-big-spenders List all users that have spent at least once above a certain threshold.

paragraphs-1-count-long-words Count how many words longer than two letters there are in each line of the poem.

paragraphs-2-count-after-joining Same as the previous task, but taking into account hyphenated words.

paragraphs-3-lines-start-with-vowel Report all lines whose first whole word starts with a vowel.

In our analysis we focused on what we call *peak cyclomatic complexity*. For conventional programs, the peak cyclomatic complexity is simply their overall cyclomatic complexity. For programs written as pipelines, the peak cyclomatic complexity is the highest of the cyclomatic complexities of all the steps or, in other words, the cyclomatic complexity of the most complex step.

Calculating the cyclomatic complexity of XSLT/XPath programs is, however, no trivial task. To the best of our knowledge, no tools to compute the complexity of XSLT/XPath programs currently exist.

To work around this lack of tools, we manually transpiled (i.e. converted) all versions of the programs that implement the tasks into equivalent JavaScript programs and measured their complexity using the tool *escomplex*¹.

A detailed description of the transpilation process is out of scope for this paper, but we would like to illustrate the mapping that we used to ensure a fair representation of the XSLT/XPath expressions. We mapped four kinds of XPath expressions. 1) Element selection has been implemented via `getElementsByTagName` lookups. 2) Predicates have been implemented as chains of `if` blocks in which the innermost block contains the code to run on the element sequence to which the predicate has been applied. 3) Function applications have been implemented as calls to equivalent JavaScript functions (whose complexity is assumed null). 4) If-then-else and boolean expressions have been converted 1:1 into JavaScript code. The XSLT template matching has been implemented via loops over the results returned by XPaths converted using previously mentioned techniques. In practice, this means that the cyclomatic complexity of XSLT/XPath expressions is increased only by predicates, if-then-else and boolean expressions.

An automated validation procedure ensures the functional equivalence between the original and the transpiled programs.

The test input data, the source code of the programs, as well as the validation scripts are available at <https://gioele.io/p/doceng2021/code>. [18]

5 RESULTS

Our results show that programs implemented as small-step pipelines consistently have lower peak cyclomatic complexity than their conventional counterparts.

This reduction in cyclomatic complexity is more pronounced the more exceptions in the data the code has to deal with.

This proportionality can be seen in Figure 1. The code for *1-all-paid-in-eur* must handle only one data imperfection. For this reason the peak complexity of the pipeline is only marginally lower.

In contrast, to accomplish the task *5-big-spenders*, the code needs to handle more kinds of data variations, in particular data variations in the selection of the elements to be considered for the final analysis. This results in a peak cyclomatic complexity that is 2.5 times higher for the conventional implementation.

Figure 2 illustrates that writing programs as small-step pipelines does not always lead to a strict reduction in peak complexity, but it provides a way to incrementally get rid of complexity, allowing for a simpler final step.

Figure 1: Expenses dataset. Comparison of the computed cyclomatic complexity of tasks implemented as conventional programs and as small-step pipelines. The bars indicate the complexity of, respectively: "conventional", the conventional program; "step-inter", the intermediate steps in the pipeline; "step-final", the core analysis. For these tasks, conventional programs are 1.3x to 2.5x more complex.

Figure 2: Paragraphs dataset. Comparison of the computed cyclomatic complexity of tasks implemented as conventional programs and as small-step pipelines. The bars indicate the complexity of, respectively: "conventional", the conventional program; "step-inter", the intermediate steps in the pipeline; "step-final", the core analysis. For these tasks, conventional programs are up to 2.5x more complex than the pipeline's final step.

The initial task, counting words in lines, can be carried out without handling any data variation. For this reason the graph for task *1-count-long-words* shows that the conventional and the small-step pipeline implementations have the same peak cyclomatic complexity.

Conversely, the graph for task *3-lines-start-with-vowel* shows that the core analysis performed in the final step of that task has less than half of the complexity of its conventional counterpart. This reduction in complexity is possible because the most complicated aspect of that analysis (handling split words on different lines) is

¹<https://github.com/escomplex/escomplex>

dealt with by a specific task, and thus no longer an issue by the time the final step with the core analysis runs.

6 RELATED WORK

6.1 Using cyclomatic complexity to compare approaches

There is little research on the use of cyclomatic complexity to compare the application of different programming approaches or paradigms within the same language. In particular, we are not aware of any studies related to XSLT, XPath or XQuery.

For Java, Bexell [4] compares imperative and functional versions of the same programs, and finds that the functional version has a lower complexity and a comparable level of readability.

On a similar note, Turhan et al [19] study the cyclomatic complexity of programs written using a test-first approach (writing the tests before the program) or a test-last approach (writing the tests once the program is in place). They report no significant differences in the cyclomatic complexity of programs written using these two approaches.

6.2 Cyclomatic complexity and readability

The real goal of small-step pipelines is to make programs that handle imperfect data leaner and easier to understand. Reductions in cyclomatic complexity have been associated to increases in the understandability and readability of code.[2]

6.3 Small-step pipelines

The small-step pipelines approach draws from similar approaches and techniques, but focuses on different aspects.

UNIX pipelines [17], for instance, provide ways to decompose a complex computation into a series of smaller computations performed by programs that operate on a (mostly textual) stream. These programs are, however, generic in their nature and operate on a wide (almost infinite) variety of possible input formats. By contrast, small-step pipelines are composed of application-specific steps that operate on a single data format.

Concatenative programming (also known as *stack programming*, for instance Forth, Joy, PostScript) [11] focuses on creating programming languages where the concept of data filtering is not only a first-class concept, but also the main construct to operate on data. In contrast, the small-step pipelines approach is just a methodology and, as such, can be applied in any language.

It is possible to classify the small-step pipeline approach as a variation of point-free/tacit programming (where programs are "expressed as combinations of simpler functions [without arguments]" [6, 12]) with a precise restriction: the *either read or fix* property.

Small-step pipelines lend themselves well to data-curation tasks, in particular as part of *replicable functional data-curation pipelines* [3] in which data is never modified manually and all desired adjustments are expressed as code.

7 CONCLUSIONS AND FUTURE WORK

These preliminary results show that writing XSLT/XPath programs as small-step pipelines can reduce their peak cyclomatic complexity up to 2.5 times.

The main advantage brought by this reduction in cyclomatic complexity is that the code implementing the core part of the program is separated from the code that deals with imperfections in the data, making it more concise and able to show the "intent of the programmer" without ambiguities and distractions.

In the future, we intend to automate the transpiling process and extend our comparison approach to study the advantages of small-step pipelines in other programming languages and for large-scale programs.

REFERENCES

- [1] AdW Mainz. 2021. The School of Salamanca. <https://www.salamanca.school/en/project.html>. Last accessed 2021-06-08.
- [2] Duaa Alawad, Manisha Panta, Minhaz F. Zibran, and Md Rakibul Islam. 2019. An Empirical Study of the Relationships between Code Readability and Software Complexity. *CoRR* abs/1909.01760 (2019). arXiv:1909.01760
- [3] Gioele Barabucci. 2018. Funktionale und deklarative Programmierung-basierte Methode für nachhaltige, reproduzierbare und verifizierbare Datenkuration. In *DHd Konferenz 2018: Kritik der digitalen Vernunft*. Köln, 26. Februar bis 2. März 2018, Georg Vogeler (Ed.). 214–219.
- [4] Andreas Bexell. 2017. *Comparing functional to imperative Java*. Master's thesis. Linnaeus University, Sweden.
- [5] Elisa Cugliana and Gioele Barabucci. 2021. Signs of the times: medieval punctuation, diplomatic encoding and rendition. *Journal of the Text Encoding Initiative* 14 (2021). In print.
- [6] Manuel Alcino Pereira da Cunha. 2005. *Point-free Program Calculation*. Ph.D. Dissertation. Universidade do Minho.
- [7] Peter Daengeli and Christian Forney. 2019. Referencing annotations as a core concept of the hallerNet edition and research platform. In *TEI 2019: What is text, really? TEI and beyond. 19th annual Conference of the TEI Consortium.. September 16-20, University of Graz, Austria*, Georg Vogeler (Ed.). ZIM.
- [8] Peter Daengeli, Thomas Roesler, Dieter Hagedorn, and Klaus Maresch. 2020. cceh/papyri-wl-data: v24; 24. Fassung vom 30.09.2020.
- [9] Peter Dängeli, Martin Stuber, and Christian Forney. 2019. Vom Stellenkommentar zum Netzwerk und zurück: grosse Quellenkorpora und tief erschlossene Strukturdaten auf hallerNet. In *DHd 2019*.
- [10] Julia Flanders and Trevor Muñoz. 2012. An Introduction to Humanities Data Curation. In *DH Curation Guide*. <https://guide.dhcurator.org/intro/index.html>
- [11] Dominikus Herzberg and Tim Reichert. 2009. Concatenative Programming - An Overlooked Paradigm in Functional Programming. In *ICSOF 2009 - Proceedings of the 4th International Conference on Software and Data Technologies, Volume 1, Sofia, Bulgaria, July 26-29, 2009*, Boris Shishkov, José Cordeiro, and Alpesh Ranchordas (Eds.). INSTICC Press, 257–263.
- [12] Roger K. W. Hui, Kenneth E. Iverson, and Eugene E. McDonnell. 1991. Tacit Definition. In *Proceedings of the International Conference on APL 1991, August 4-8, 1991, Palo Alto, CA, USA*, Charles A. Schulz (Ed.). ACM, 202–211.
- [13] Rushikesh K. Joshi. 2008. Implementing Assertions in Distributed Object Systems. <https://www.cse.iitb.ac.in/~rkj/talks/assertions.pdf>
- [14] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Trans. Software Eng.* 2, 4 (1976), 308–320.
- [15] Falcon Momot, Sergey Bratus, Sven M. Hallberg, and Meredith L. Patterson. 2016. The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them. In *IEEE Cybersecurity Development, SecDev 2016, Boston, MA, USA, November 3-4, 2016*. IEEE Computer Society, 45–52.
- [16] Alan J. Perlis. 1982. Special Feature: Epigrams on Programming. *SIGPLAN Not.* 17, 9 (Sept. 1982), 7–13.
- [17] Dennis M. Ritchie. 1984. The UNIX system: The evolution of the UNIX time-sharing system. *AT&T Bell Lab. Tech. J.* 63, 8 (1984), 1577–1593.
- [18] Marcel Schaebe and Gioele Barabucci. 2021. *Small-step pipelines reduce the complexity of XSLT/XPath programs (auxiliary material)*. <https://doi.org/10.5281/zenodo.5115768>
- [19] Burak Turhan, Ayse Basar Bener, Pasi Kuvaja, and Markku Oivo. 2010. A Quantitative Comparison of Test-First and Test-Last Code in an Industrial Project. In *Agile Processes in Software Engineering and Extreme Programming, 11th International Conference, XP 2010, Trondheim, Norway, June 1-4, 2010. Proceedings (LNBP, Vol. 48)*, Alberto Sillitti, Angela Martin, Xiaofeng Wang, and Elizabeth Whitworth (Eds.). Springer, 232–237.
- [20] Katherine Underwood and Michael E. Locasto. 2016. In Search of Shotgun Parsers in Android Applications. In *2016 IEEE Security and Privacy Workshops, SP Workshops 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 140–155.