Towards Automatic Extraction of Cartographic Metadata from the Code of Online Maps

Florian Ledermann
Technische Universität Wien
Research Group Cartography
Erzherzog-Johann-Platz 1/120-6
Vienna, Austria
florian.ledermann@tuwien.ac.at

Abstract

Many maps on the web do not provide any formal information on the cartographic processes at work in them, despite existing elaborate metadata standards. The key idea of the research presented in this paper is to use the program code of online maps to analyse the transformational processes applied to render them, and to infer cartographic metadata, such as projection or symbology specifications, from such an analysis. By using runtime code instrumentation, the data flow graph of each output element on the map is reconstructed and can be matched against a library of cartographic patterns. Using this method, cartographic processes implemented in informal program code can be annotated with appropriate metadata, and novel applications for retrieving and analysing online maps can be envisioned.

Keywords: web cartography, metadata, transformations, reverse engineering, web engineering.

1 Introduction

Standards for services and metadata have improved tremendously over the past decade, giving cartographers powerful tools to publish maps and geographic information online in ways that are interoperable and discoverable. However, only a fraction of all maps made available online adhere to such standards. Many maps today are made by "neocartographers" (Cartwright, 2012), amateurs or professionals from outside the geosciences who are not aware of the established standards or have little incentive to adhere to them. Some of the innovative experiments in interactive cartography and geovisualization today come from such actors – programmers, designers or journalists – and may in format, technique or creativity go beyond what current map publishing standards can cover. Also, commercial actors may have the resources to drive innovation in online maps, but are not always incentivised to publish their maps using open and interoperable standards – their strategic interest lies in driving and keeping traffic to their own sites and solutions.

From the point of view of standardization, there exist two fundamental "camps" of online geographic information: those who adhere to public standards for data and metadata, and can therefore be linked, mixed and aggregated, and those who do not, and are therefore destined to remain separate, inaccessible to any standards-based approach for retrieval and aggregation. The only way to move into the former camp is to publish the information in the required format. However, as has been pointed out, this is not always possible for the involved actors, or maybe even not in their perceived best interest. Even with intensive lobbying for standards, parts of the geoweb will remain proprietary or informal if we rely on publishers of data and maps to provide suitable metadata.

A potential solution is to have the metadata for a map or dataset (such as: map extent, spatial reference systems and projections used, symbology specification etc.), assigned by third parties. For example, a map can be analysed by a human expert, possibly with the assistance of technical tools, to reconstruct its projection (Jenny & Hurni, 2011; Bayer, 2014) or symbology. Appropriate metadata would be defined in standards such the OGC specifications for Web Map Services (WMS) or the Symbology Encoding specification (Bocher & Ertz, 2018). This information can be assigned to the map and then be published, allowing the creation of, for example, web map services for historical maps, although the original creator of the map has not provided the required metadata herself – it has been assigned, manually, by an external actor.

While such detailed manual inspection of geographic artefacts may be feasible for corpora of historical maps and small numbers of artefacts of specific scientific or cultural interest, it is certainly not an approach that scales up to the volume of maps published on the web in general. What would be needed are automatic approaches that can assign semantics to informally published geographic information, to make that information interoperable with standards-based infrastructures.

2 Related work and overview of proposed approach

The need for reconstructing metadata for maps, especially historic maps, is not new. Often, the map is all that is available to an investigator, and documentation about the processes of its creation is not accessible or does not exist. Therefore, existing tools for cartographic analysis focus on manual or semi-automatic approaches to reconstruct the projection parameters from the map image. Both the *detectproj* (Bayer, 2014) and the *MapAnalyst* (Jenny & Hurni, 2011) tools work by having a human operator place control points on (scanned) map images, in order to reconstruct their cartographic projection.

A potentially fully automatic retrieval of projection parameters for shapefiles with missing or wrong projection metadata is presented by Egger (2016). In Egger's approach, the point correspondences are automatically retrieved based on matching place names in the shape files. The tool is therefore limited to shape files covering overlapping geographic regions and containing the necessary administrative units.

Few projects deal with automated metadata reconstruction beyond projection analysis. An ongoing software project, *Rainbowbot* (Niccoli, 2016), sets as its goal to automatically detect maps which use a "rainbow" color scheme. It is unclear whether the software is operational and works satisfactorily, as no results or other details have been published so far.

In the wider field of information visualization, Poco and Heer (2017) apply computer vision and machine learning algorithms to retrieve the visualization parameters of simple chart images. It remains unclear whether such a purely image-based approach could work with considerably more complex and less constrained visualizations such as maps.

All these approaches, except Egger's, have in common that they focus on trying to infer properties of the transformations used to create a map or visualization solely from the resulting image. Egger presents a fully automatic approach, which is very limited in its domain (assigning point correspondences) and works only on input data requiring very specific properties.

An approach entirely different from image-based or data-based methods is presented in this paper. Many maps are posted online not as static images, but as JavaScript program code that recreates the map image from geospatial data. The key idea of the research presented here is to use the program code of the map to analyse the transformational processes applied to the data to yield the map image, and to infer cartographic metadata from such an analysis.

Program code follows a strict syntax, and is therefore free of the ambiguities of human language. However, reasoning over the computational semantics of a given program, sometimes called *static analysis*, is provably limited – "most questions about the behaviour of a program are undecidable or infeasible to compute" (D'silva, Kroening, & Weissenbacher, 2008). While for strongly typed languages like Java, C++ or Haskell, it is possible to successfully employ static analysis techniques to answer a range of specific questions about the behaviour of a program (Nielson, Nielson, & Hankin, 1999; Ayewah, Penix, Morgenthaler, Pugh, & Hovemeyer, 2008), for loosely typed scripting languages like JavaScript, static analysis is further impeded by the properties of the language (see Jensen, Møller, & Thiemann (2009) for a discussion of the specific difficulties of static analysis of JavaScript programs). On the web, programs to be run in the browser are necessarily delivered in JavaScript, so these limitations further impede the successful application of static analysis techniques to gain insight into the cartographic properties of web maps. An alternative approach for manual analysis of cartographic code by "close reading" has been presented at AGILE 2016 (Ledermann, 2016), but the manual study of cartographic code is prohibitively labour-intensive to be employed on a larger scale.

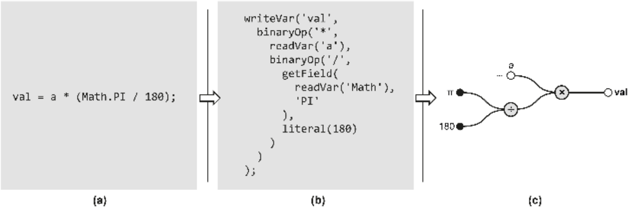
In the light of these theoretical limitations and practical difficulties, instead of trying to directly infer the transformations applied by a given cartographic program from static analysis of its source code, this project investigates a different approach: the *dynamic analysis* of the program's behaviour, meaning: running the program while simultaneously observing and recording every step of the program for later investigation.

3 Analyzing cartographic transformations using runtime program analysis

Several options were evaluated for doing such a "play and record" session of cartographic programs in a potentially automated manner, including creating a custom JavaScript interpreter or tapping into the debugging interface of modern web browsers. The requirements were best met by using the *Jalangi* framework for code instrumentation and dynamic analysis (Sen, Kalasapur, Brutch, & Gibbs, 2013). Jalangi provides a *transpiler* for JavaScript that wraps every operation of a given program in additional code that provides hooks to trace calculations, variable changes, conditionals etc. throughout the run of the program (see Figure 1 for an example of source code instrumented with Jalangi).

A custom analysis module has been implemented that uses Jalangi to trace all mathematical operations performed by the scripts on a web page. This allows us to reconstruct the *data flow graph* for all variables of the script and, ultimately, the program's visual output.

Figure 1 shows a simple example of such automated reconstruction of a value's data flow graph. A line of JavaScript code, containing a simple calculation (1a), is transpiled by Jalangi to instrumented code of equivalent functionality, containing the hooks mentioned above to track each operation (1b). Our analysis module connects to Jalangi to reconstruct the data flow graph (1c) and associates it with the variable.



```
val = a * (Math.PI / 180);
```

```
writeVar('val',
  binaryOp('*',
    readVar('a'),
    binaryOp('/',
      getField(
        readVar('Math'),
        'PI'
      ),
      literal(180)
    )
  )
);
```

Figure 1: A simple line of code, analysed with Jalangi. Part (a) shows the original code, (b) the instrumented source code produced by Jalangi, and (c) the data flow graph extracted with our module. (Part (b) simplified for illustration purposes)

A sufficiently educated observer will be able to guess what the line of code shown in Figure 1a represents: a conversion of input value a from degrees to radians! While in this simple example, the semantics of the calculation can still be directly inferred by reading the source code, for more complex transformations, the number of possible syntactic variations of expressing the underlying calculations makes a direct static analysis of the code impossible (as has been pointed out above). The data flow graph, however, contains a representation of the transformation's computational semantics which is much more robust to syntactic and structural variations of the underlying implementation. Because the data flow graph is also available

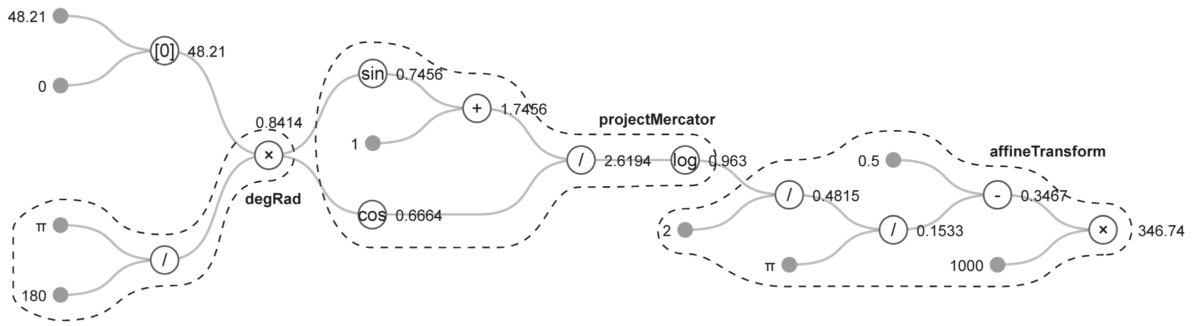


Figure 2: Data flow graph of the value "346.74" used as the y-coordinate of a plotted point (rightmost node). The proposed system identifies a chain of transformations in the data flow graph, involving conversion to radians, cartographic projection using the Mercator projection, and an affine transformation to screen space. This information can be provided as metadata for the map.

as a machine-processable data structure, automated methods can be applied to identify patterns in the graph.

Once such *transformational patterns* are identified, this information can be used to assign additional semantics to otherwise generic values in the program. Having access to the data flow graph, our system would, in the example presented in Figure 1, not only know the value and basic type of the variable **`val`**, but can with reasonable certainty assume that, because of the identified transformational pattern, *the value represents an angle in radians that has been converted from a degree value*!

Again, the simple example of identifying a degrees-to-radians conversion served to illustrate the proposed method and could easily be replicated with alternative methods. But once the complete data flow graph of each value in the program is available, one can attempt to identify more complex transformational patterns: different cartographic projections, geometric transformations, classification, calculation of visual variables (line width, dot size etc.) and classes of generalization algorithms, which may all be applied in the program. We are currently in the process of building such a library of *cartographic transformation patterns* for detection in the extracted graphs.

Figure 2 presents a simple real-world cartographic example. In the transformation graph of a value used as the y-coordinate of a plotted point, three transformation patterns could be identified by our system: a degree-to-radians conversion applied on the first entry of a coordinate array, followed by a sequence of calculations characteristic for the Mercator projection, followed by an affine transformation (translation and scale) to yield the plotted screen coordinate. The patterns and the values of their nodes can be used to unambiguously reconstruct the parameters of the spatial transformation from geographic datum to screen.

The instrumented source code provides access to the data flow graphs of *all* variables used in the program, no matter whether they are ever involved in creating visual output. For example, a program may pre-compute a range of values using different cartographic projections without ever using them to actually render a map. In this case, the transformation patterns of these projections would be identified in the overall data flow graph, although they are not contributing to the actual output of the program. To capture only those transformations relevant for the cartographic output, the variables involved in actually generating such output must be identified.

In a browser environment, to create any visual output, a script invokes some method(s) of the browsers APIs (since no direct access to operating system methods is permitted). To capture the relevant operations, our system augments all methods of the browser APIs that actually generate visual output with additional code to capture the previously generated data flow graphs of the methods' parameters and stores them for subsequent analysis. Modern browsers provide a plethora of API methods to modify the contents on a page: scripts can create HTML or SVG elements dynamically, modify the attributes of existing elements, use the canvas API to dispatch drawing operations etc. – we so far identified a total of 128 methods in the Chrome browser API that need to be captured to cover the various options.

With such a *capturing layer* in place, the system can be run in a fully autonomous way to analyse the transformational processes of maps on the web. A "headless" version of the chrome browser is controlled by a script to load the pages and run the analysis scripts. Pages are loaded through a proxy server that instruments each page's JavaScript code in real time using Jalangi and injects the API capturing scripts. The headless browser loads the page, runs all code on the page (including the instrumentation and capturing) and stores a screenshot of the page and the data flow graphs of all output operations in a database. Analysis scripts can then be run offline on this database of captured visual output and associated transformations. Figure 3 gives an overview of the complete technical setup.

4 Discussion and outlook

To this author's knowledge, the presented method for runtime analysis of cartographic programs is the first system that supports the fully automatic reconstruction of cartographic transformations from program code and extraction of metadata by analysing the transformation patterns employed. It is therefore hoped that it may contribute to connecting the proprietary and informal parts of online geoinformation to standards-based infrastructures. One of the strengths of the proposed method is that it works independent of different coding styles, APIs used or even code obfuscation, allowing for the analysis of a wide range of real-world cartographic programs.

The proposed method requires access to the program code performing the cartographic transformations, which is not always available. Many maps are simply loaded as images, or rendered completely on the server, out of reach for code analysis. For such maps, other methods of analysis, such as those discussed in section 2, need to be used. However, current trends in web mapping lead to more and more maps being rendered in the browser: the desire to provide fully interactive

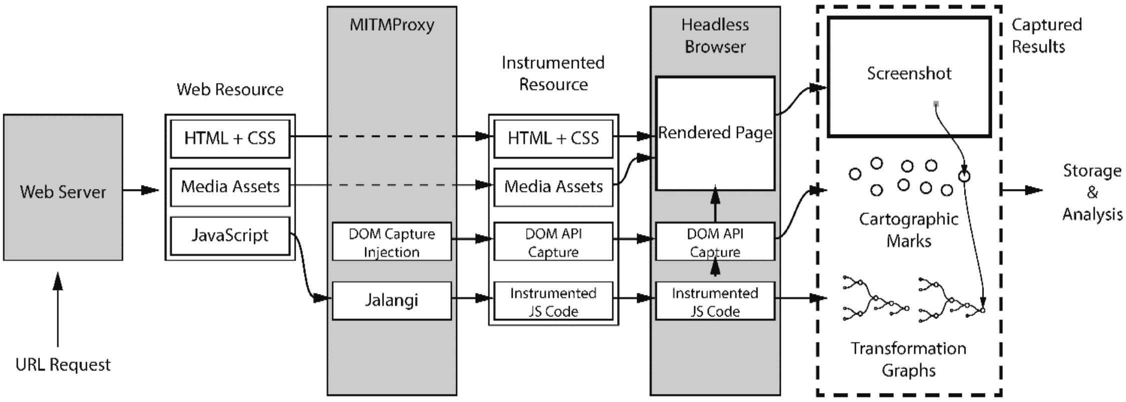


Figure 3: Overview of a system for automated cartographic analysis using the proposed method (see previous page for discussion)

maps, reacting in real time to the users actions and context; the evolution from web mapping to web GIS solutions, providing entire GIS pipelines in the browser; and the technical advantages of delivering raw data instead of map images to the client, for example in the form of vector tiles. We are hopeful that in the face of these developments, the proposed method will be of increased relevance to cartographic analysis in the near future. (Note, however, that there may be additional ethical and legal concerns to be considered before applying such reverse engineering techniques to third-party software!)

Mathematical transformations can appear in the data flow graph in various forms (for example, when converting from degrees to radians, one may perform the multiplication or, alternatively, the division first). Our system currently uses a fixed set of alternative patterns for detecting such permutations. In the future, it is planned to look into algorithms for normalizing the mathematical operations represented in the graph by analytical methods (e.g. see Shatnawi & Youssef (2007)) for more robust pattern identification.

Even in a normalized graph, an amount of uncertainty about the *precise* semantics of mathematical operations must remain. Multiplying by π and dividing by 180 would *usually* indicate a conversion from degrees to radians, but maybe this sequence of calculations could occur in a different context as well. The reliability of our approach of matching transformational patterns against the data flow graph has to be verified against real-world corpora of cartographic programs to see if this is an issue of wider relevance.

Going forward we can envision innovative applications for the proposed method of connecting the informal and the standards-based geoweb. The automatic system sketched out would allow the construction of a *cartographic search engine*, allowing users to retrieve online maps according to cartographic search concepts (projection, symbology, classification etc.). The screenshots captured by the system, together with the extracted metadata, would even allow to provide screenshots of web maps as a Web Map Service (WMS) to browse the collected maps in a standardized way.

On a smaller scale, an application for cartographic analysis could be envisioned to support experts or inform novices about the cartographic principles at work in individual maps.

The engineering methods presented here are not limited to JavaScript or web technologies in principle. Similar approaches could be applied in other technological contexts, such as server software or even historic examples of cartographic programs, to isolate, preserve and study the transformations applied.

References

Ayewah, N., Penix, J., Morgenthaler, J. D., Pugh, W., & Hovemeyer, D. (2008). Using Static Analysis to Find Bugs. *IEEE Software*, *25*(5), 22–29.

Bayer, T. (2014). Estimation of an unknown cartographic projection and its parameters from the map. *GeoInformatica*, *18*(3), 621–669.

Bocher, E., & Ertz, O. (2018). A redesign of OGC Symbology Encoding standard for sharing cartography. *PeerJ Computer Science*, *4*, e143. https://doi.org/10.7717/peerj-cs.143

Cartwright, W. (2012). Neocartography: Opportunities, issues and prospects. *South African Journal of Geomatics*, *1*(1), 14–31.

D'silva, V., Kroening, D., & Weissenbacher, G. (2008). A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *27*(7), 1165–1178.

Egger, M. (2016). Shapefile Projectionfinder - A new way to find and define the coordinate system of GIS data automatically. In *Poster at FOSS4G 2016*. Bonn, Germany.

Jenny, B., & Hurni, L. (2011). Studying cartographic heritage: Analysis and visualization of geometric distortions. *Computers & Graphics*, *35*(2), 402–411.

Jensen, S. H., Møller, A., & Thiemann, P. (2009). Type Analysis for JavaScript. In *Proceedings of the 16th International Static Analysis Symposium (SAS 2009)*. Los Angeles, CA: Springer.

Ledermann, F. (2016). Initial Findings from Close Reading of Cartographic Programs. In *Workshop „Code Loves Maps", AGILE 2016*. Helsinki, Finland.

Niccoli, M. (2016). Rainbowbot. Retrieved January 18, 2018, from https://github.com/mycarta/rainbowbot

Nielson, F., Nielson, H. R., & Hankin, C. (1999). *Principles of Program Analysis*. Berlin, DE: Springer.

Poco, J., & Heer, J. (2017). Reverse-Engineering Visualizations: Recovering Visual Encodings from Chart Images. *Computer Graphics Forum*, *36*(3).

Sen, K., Kalasapur, S., Brutch, T., & Gibbs, S. (2013). Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (pp. 488–498). Saint Petersburg, Russia: ACM.

Shatnawi, M., & Youssef, A. (2007). Equivalence detection using parse-tree normalization for math search. In *Proceedings of the 2nd International Conference on Digital Information Management (ICDIM '07)*. https://doi.org/10.1109/ICDIM.2007.4444297