Aalto University

OPEN ACCESS

This is an electronic reprint of the original article.
This reprint may differ from the original in pagination and typographic detail.

Tilanterä, Artturi; Mariani, Giacomo; Korhonen, Ari; Seppälä, Otto

Towards a JSON-based Algorithm Animation Language

Published in:
Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021

DOI:
10.1109/VISSOFT52517.2021.00026

Published: 13/11/2021

Document Version
Peer reviewed version

Please cite the original version:
Tilanterä, A., Mariani, G., Korhonen, A., & Seppälä, O. (2021). Towards a JSON-based Algorithm Animation Language. In *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021* (pp. 135-139). IEEE. https://doi.org/10.1109/VISSOFT52517.2021.00026

This material is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorised user.

Towards a JSON-based Algorithm Animation Language

Artturi Tilanterä, Giacomo Mariani, Ari Korhonen, Otto Seppälä
School of Science
Aalto University, Espoo, Finland
Email: {artturi.tilantera,giacomo.mariani,ari.korhonen,otto.seppala}@aalto.fi

***Abstract*—Visual algorithm simulation (VAS) is a method used in teaching data structures and algorithms. In a VAS exercise a learner simulates the steps of an algorithm by interacting with data structure visualisations and receives feedback on the correctness of steps taken. A data format for storing VAS simulation traces would allow for later inspection of the process by instructors and researchers. In this study we describe the development of a prototype language for this purpose. The initial version was tested in a research project where the language was used for recording traces, which were later analyzed. The results were positive but also instructed some revisions in the data format and the requirements. We describe an iterative development process for extending and improving the language and the tooling. This is a work in progress: we will proceed with the data format specification, as well as further develop the technologies needed to use the data format in conjunction with VAS exercises.**

***Index Terms*—Algorithm Visualization, Algorithm Animation, Visual Algorithm Simulation, Tracing.**

I. Introduction

Data structures and algorithms is a common topic on undergraduate computer science curriculum. It can be taught with the help of algorithm visualizations that employ images to show all the states of the data structures during the execution of an algorithm. Visual algorithm simulation (VAS) exercises utilize algorithm visualization methods to make the process interactive: the student simulates, or traces, the steps of an algorithm and receives automatic feedback. Solving a VAS exercise is analogous to *code tracing*, which is a method used to manually follow the execution of a program, i.e., simulating code execution in which a learner step through program code and track the values of the variables. Code tracing is a skill that any programmer needs to master when learning how to program. Similarly, we consider the skill of tracing algorithms also important.

The theory of Engagement Taxonomy states that algorithm visualization is educationally more efficient when it engages the student in simulating an algorithm or responding to questions compared to passive viewing of an animation [1], [2], [3] [4]. VAS exercises enable very high level of engagement in this sense. Figure 1 shows an example of a VAS exercise in which the learner is supposed to simulate Dijkstra's algorithm by clicking the graph edges in the correct order. For brevity, the corresponding code (algorithm) is omitted from the figure. The learner is supposed to study the algorithm in order to determine, for example, which node is selected in case of two equal length paths (i.e., how the priority queue is implemented). When the student has completed the simulation, they click a Grade button and receive immediate, automatic feedback on the correctness of their simulation trace. Tracing with feedback allows therefore efficient practise.

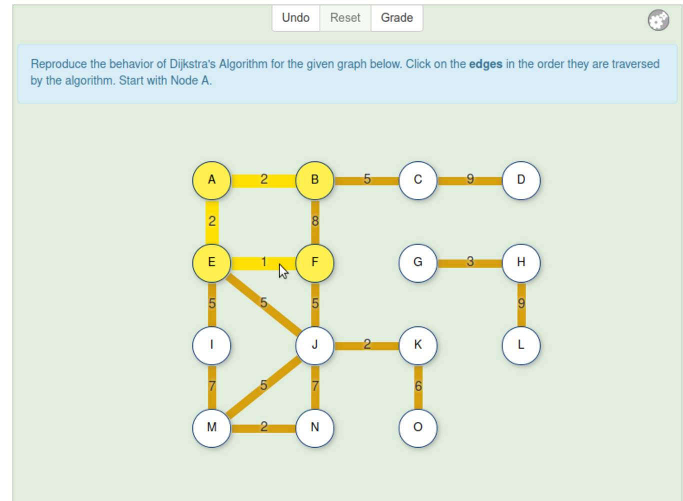

Fig. 1. A VAS exercise on Dijkstra's algorithm, in progress. The software displays a weighted graph with nodes A–O. The learner has clicked three edges, and in result, those edges and the related vertices are highlighted in yellow.

To our knowledge, the only actively maintained collection of VAS exercises is the OpenDSA e-textbook utilising the JavaScript Algorithm Visualization Library (JSAV) [5], [6], [7]. The exercise in Figure 1 is made with JSAV. First, JSAV generates an unique instance (initial state) of the exercise for each solution attempt. Second, JSAV grades student's trace automatically by counting correct steps from the initial state. However, JSAV does not support reviewing students' simulation traces.

We foresee a number of use cases for such saved simulation traces. First, a student can review their graded solutions. Second, a student can ask instructor on their incorrect simulation trace. Third, a researcher could study students' misconceptions in the traces. [8], [9], [10]. Hypothetically, this allows more elaborate automatic feedback in the exercises, which would also support the instructor. All these use cases require that the simulation traces can be saved to some data format in terms

of *algorithm animation language*. We elaborate the use cases more in Section IV-A1.

We approach the problem with a design science method using the terminology introduced by Wieringa [11]. We have formulated the following *knowledge questions* (KQ) and *design problems* (DP) in chronological order.

- **KQ1.** Which existing algorithm animation languages are relevant for our case?
- **DP1.** Design a data format for recording students' traces in visual algorithm simulation exercises.
- **KQ2.** How does our design meet the goals when it is used in a real context?

A complete solution consists of: (i) an algorithm animation language, (ii) a VAS exercise recorder which automatically records a student's trace, and (iii) a record player. However, this article discusses only the language.

Our solution is a proposal for a JSON-based Algorithm Animation Language (JAAL). The language is based on another algorithm animation language, XAAL, which was designed for data interchange between algorithm animation systems. However, XAAL was never used with JSAV; there has not been a standardised way to record students' traces in JSAV exercises. The initial design of JAAL used JSON as the base language for the following two reasons. First, the animation must be based on well-supported technologies in wider context than just JSAV. Second, the language must be able to describe objects and events. [12]

The main reason for choosing JSON over XML is that JSON has become more popular in web development [13]. For our purpose, the base language could have equally been XML.

II. Related Work and Background

XAAL [14], the eXtensible Algorithm Animation Language, is an XML-based format for data exchange between algorithm animation systems. It was developed in 2005–2009 to facilitate the use of algorithm animations in education. The aim is to allow any algorithm animation software to display available animations made with another software.

Based on common properties of algorithm animation languages [15], XAAL supports semantic representation of data structures (e.g. list, array, and tree), data structure operations (e.g. swap, insert, remove), and vector graphics. Several prototype implementations exist for XAAL. JSXaal is an animation viewer written in HTML and JavaScript [16], [17], [18]. There is a XAAL generator for MatrixPro algorithm visualization software. Moreover, XSL stylesheets can be employed to convert XAAL to other algorithm animation formats and SVG. [19, pp. 32–33] Currently XAAL is not in active use; relevant literature is from years 2005–2010. However, XAAL is the state of the art on the field of algorithm animation languages and therefore a valuable starting point for us.

III. Methodology

We have applied the design science method following Wieringa [11] to design and implement an artifact—an algorithm animation language—to interact with a problem context. Design science is an iterative process with the following phases. (i) *Problem Investigation*, (ii) *Treatment Design*, (iii) *Treatment Validation*, (iv) *Treatment Implementation*, and (v) *Implementation Evaluation*. Phases (i)–(iii) form the Design Cycle, while adding (iv) and (v) results in an Engineering cycle. The Design Cycle may iterate multiple times inside an Engineering Cycle. [11]

A. Design Cycle

1) Problem Investigation: The aim of the problem investigation phase is to inspect the context where the artifact will be applied. Such context can be divided into *social context*, which represents users, developers, and maintainers of the artifact, and *knowledge context*, which is the field of existing knowledge within the research area. We commenced by identifying the context: what are the stakeholders and their goals, and what concepts and phenomena are related to the problem.

First, we inspected the social context through the user roles and use cases. Second, the knowledge context was covered by a review of the existing literature on algorithm animation (AA) and VAS, an exploration of existing AA languages, and a detailed analysis of the JSAV library. This phase answers KQ1: *Which existing algorithm animation languages are relevant for our case?* We have already summarised this in Section II. A more rigorous literature survey and the overall problem investigation is elaborated in [12]. We only summarise the latter in Section IV-A1.

2) Treatment Design: In Section IV-A2, we specify requirements to meet the goals and choose a *treatment*, which is our attempt to solve DP1. We defined the requirements together with an university instructor representing the application stakeholders. JAAL was implemented iteratively based on the functional and qualitative requirements specified for a web application that collects, stores, and plays user interaction data from JSAV VAS exercises in a standardized manner. The web application, which leverages the JAAL format to save the exercise trace, is not covered in this paper; we focus on the language hereafter.

3) Treatment Validation: As a part of the iterative process, we tested the designed treatment in a laboratory context and investigated how it contributes to the goals. Based on this testing, we refined the requirements, implemented them in an application prototype able to produce a JAAL recording from a JSAV-based VAS exercise, and validated how the implementation satisfies the requirements.

B. Engineering Cycle

1) Treatment Implementation: This is the first step towards developing an artifact based on a validated design to be used in real context. In practise, we integrated the Recorder and Player applications developed in the Treatment Design into a learning management system. This required developing some integration code and testing it manually. In addition, we used the Recorder to save hundreds of traces submitted by students for the Dijkstra's algorithm in JAAL format. We tested the

implementation with other JSAV VAS exercises as well, but for brevity, we report only the experiment with this single exercise.

2) Implementation Evaluation: The final step is to observe how the treatment interacts with the problem context and to evaluate the results. We observed the JAAL recordings produced by the Recorder in conjunction of the data analysis for another research and reported practical issues. Based on discussion on the found issues, we re-evaluated the current requirements of JAAL and added additional requirements.

IV. Results

A. Design cycle

1) Problem Investigation: To define the user roles we conducted bi-weekly iterations where an university instructor represented the application stakeholders. As a result, the main roles we obtained were (i) learner, (ii) instructor, and (iii) researcher. We identified also other roles such as application and exercise developer, but those are omitted for brevity. See [12] for more details.

The main goal for JAAL is to save the traces emerging from solving VAS exercises into a form that can be reviewed later. With the same iterative approach, we identified the use cases from each stakeholder's point of view. The main use cases for JAAL are:

- A student reviews their solution. A student may retry a VAS exercise several times. They might learn better about their mistake if they can compare their earlier solution with a model solution. This requires recording the solution in detail.
- A student can ask an instructor about their solution which was automatically graded as incorrect.
- A researcher may study the simulation traces to improve computing education. In large courses students submit hundreds of traces for VAS exercises. A branch of research is to study these traces to identify misconceptions related to algorithms and data structures [8], [9].

2) Treatment Design: Regarding KQ1 related to existing treatments, we have described existing algorithm animation languages in Section II.

To answer DP1, *Design a data format for recording students' traces in visual algorithm simulation exercises*, we specified requirements and developed software. The essential initial requirements for JAAL are the first six rows in Table I. For comprehensive list of initial requirements and JAAL 1.0 specification, see [12]. Based on the requirements, a prototype software was developed to implement JAAL. The source code and a working demonstration can be found at Github[1].

Figure 2 illustrates how the structure of JAAL is based on XAAL. XAAL has the following structure of four sections: (i) the `<metadata>` section has information on the animation creator and software; `<defs>` specifies animation options, graphical styles, and shapes; `<initial>` contains initial graphical objects; and `<animation>` contains animation operations [14]. Similarly, JAAL contains `metadata`, `definitions`, `initialState`, and `animation`. However, JAAL stores two animations: the model answer of a VAS exercise in `definitions.modelAnswer` (Figure 2 (b), lines 5–9) and student's simulation in `animation` (Figure 2 (b), lines 22–34).

The data in Figure 2 (b) is an actual JAAL recording of the trace for the Dijkstra's algorithm VAS exercise shown in Figure 1 with some details omitted. The object `initialState.dataStructures[0]` on lines 12–19 describes the nodes and weighted edges of a graph semantically. The object `initialState.animationHTML` on line 20 corresponds to the visual representation of the initial view. JSAV has created it using HTML, CSS, and SVG. The object `animation[0]` on lines 22-32 includes the first step after a student has clicked an edge in the GUI. Line 29 shows how JSAV has added a style attribute `marked` for the clicked edge. The `type` of the animation step on line 23 indicates whether the student has proceeded in the simulation, clicked the Undo button, or requested grading. The Undo action allows the student to retry a step if they think they made a mistake. Recorded Undo actions provide the instructor with knowledge on difficult steps in the exercise. Further animation steps would placed from line 34, but they are omitted from the figure.

Figure 2 (a) represents manually crafted XML data which has been validated against the XAAL XML Schema [20]. XAAL supports similar semantic definitions of data structures and step sequences than JAAL: the correspondence of the `marked` attribute in Figure 2 (b), line 29, is the `change-style` object with attribute `style-ref="selected-edge"` and reference to the edge with `id` of `edgeAB` in Figure 2 (a), lines 29–32. The vector graphic representations could be added using the XML-based graphical primitives of XAAL.

3) Treatment Validation: The validation phase of the design cycle grew incrementally over the bi-weekly iterations. First, validation consisted of discussing the developed JAAL structure with the instructor representing the stakeholders. Later, as the web application prototype that produced the JAAL data developed, we iteratively validated the JAAL design by storing traces from JSAV VAS exercises as JAAL recordings. Finally, we used the same application to play the exercise traces, starting from the stored JAAL data. Column *Validation* in Table I shows the results of validation of the language against its functional requirements. Again, in the following, we focus only on JAAL and not the player application. Thus, the following engineering cycle will describe how we proceeded from here, after a prototype implementation for JAAL 1.0 was achieved.

B. Engineering Cycle

1) Treatment Implementation: The Player and Recorder prototypes were integrated into the A+ Learning Management System (LMS)[2] to collect data for another research project

[1]https://github.com/atilante/JAAL

[2]https://apluslms.github.io/

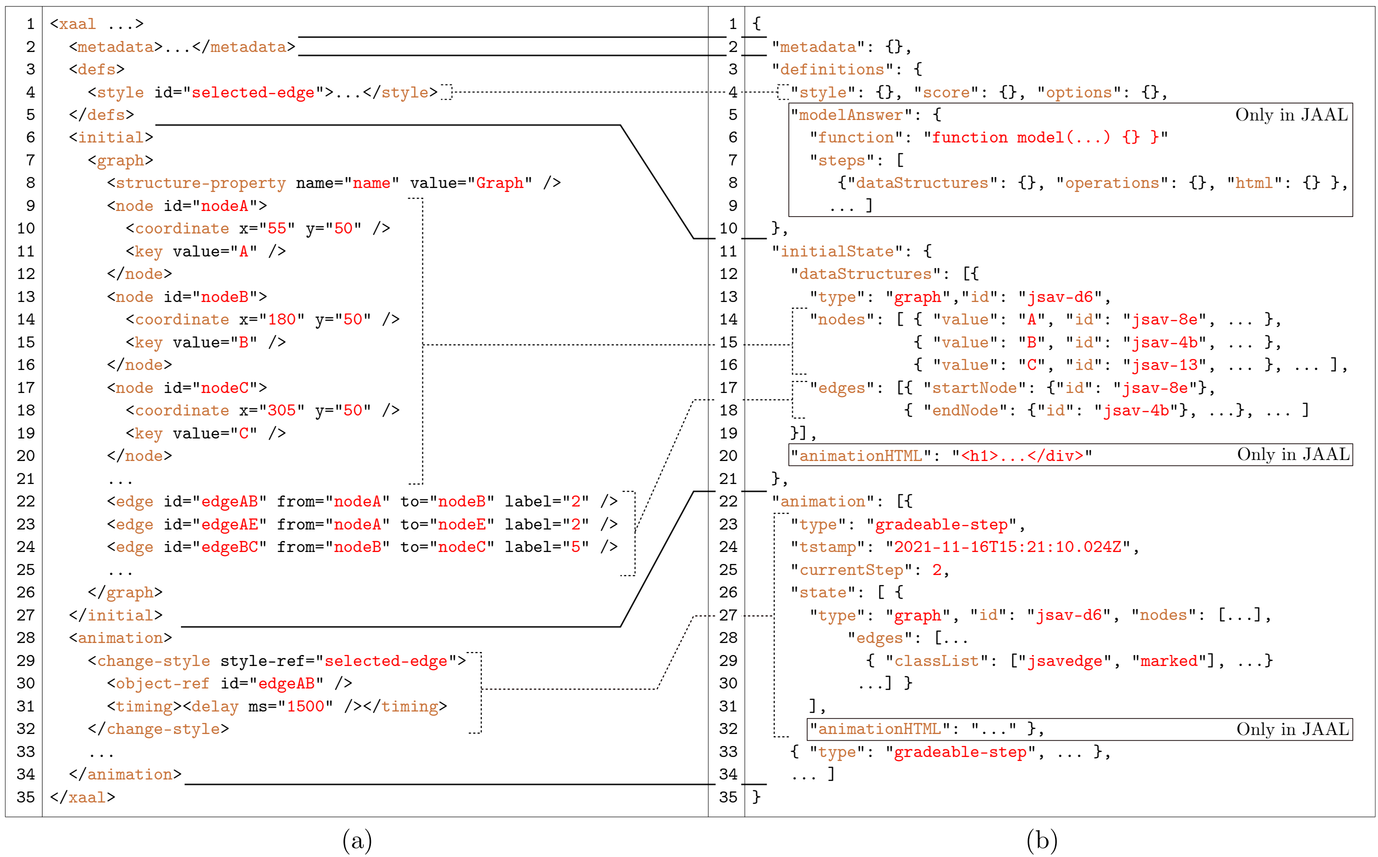

Fig. 2. XAAL (a) has influenced JAAL (b). The languages have similar top level blocks: `metadata`, `defs`, `initial`, and `animation`.

TABLE I
THE ESSENTIAL REQUIREMENTS OF JAAL AFTER VALIDATION (**V**) AND EVALUATION (**E**) ACTIVITIES.

Functional Requirements	**V**	**E**
All relevant GUI actions are saved.	1	1
All relevant exercise data is recorded.	2	*1*
The model answer is stored.	1	1
The JAAL recording contains the application [and JAAL language] version.	0	0
All the actions that can be performed on a data structure are recorded.	1	1
The trace can be viewed step-by-step as an AA.	2	2
Minimize the number of keywords in the language: represent visualizable data structures with generic data structures, such as graph and matrix.	–	0
In the semantic representation, each data structure should have a reference to another data structure, including the contents of the nodes.	–	0
Save vector graphics in one standard format, e.g. SVG.	–	0
Qualitative Requirements		
Design a general-purpose language which is not bound to OpenDSA/JSAV.	–	0
Specify the algorithm animation language independently of the Recorder.	–	1
The recording is compact, containing no redundant data.	–	0

0: missing, 1: partial, 2: ready
[added after evaluation]

related to the Modality Effect [21]. The JAAL integration process involved (i) modifying the HTML template for a JSAV/OpenDSA exercise; (ii) designing the communication process between OpenDSA, JSAV, the Recorder, and the LMS; (iii) implementing and manually testing the communication; and (iv) integrating the JSAV Exercise Player into the LMS. In the last part, the LMS stored each exercise submission as HTML data that containing the JAAL recording as Base64 encoded JSON string, plus necessary HTML, CSS and JavaScript start the Player. Finally, we had the Dijkstra's algorithm VAS exercise shown in Figure 1 which worked inside the course material provided by the A+ LMS.

2) Implementation Evaluation: This sections answers to KQ2: *How does our design meet the goals when it is used in a real context?* The scope here are the exercise traces recorded as JAAL recordings in the LMS; we excluded the inspection of the JSAV Exercise Recorder and Player.

We observed the structure of JAAL during the data analysis in the Modality Effect research [21]. The JSAV Exercise Recorder produced 553 JAAL recordings, each having an average size of 1 MB as a JSON string. When the data was downloaded from the LMS, the JSON decoding of 21 files (3.8%) failed for an unknown reason. The structure of the recordings seemed to make automated data analysis cumbersome, as the JAAL recordings for Dijkstra's algorithm lacked compact, semantic representation of the choices for a

new edge at each step. Furthermore, some recordings did not have the data for marked edges.

To summarise the evaluation, in the best case, the JAAL recording contained all necessary information to support research, but the recording was too verbose and complex for the purpose. Column *Evaluation* in Table I shows the evaluated requirements. Additional requirements were added based on authors' discussion together. First, the support for numerous data structures in JAAL should be implemented with generic data structures such as graph and matrix, their generic attributes, and nesting of data structures. This allows great freedom in declaring new data structures when necessary without needing to extend the language specification. Second, the combination of HTML, CSS, and SVG in the `animationHTML` blocks is rather complex; for simplicity, the graphical representation should rely on one language. Third, the language must be independent of JSAV and OpenDSA to allow software-independent use. Finally, redundancies in the graphical representation must be reduced to decrease the size of the recordings.

V. Discussion

Thus far, no VAS exercise software has had support for recording students' traces. Furthermore, current algorithm animation languages do not support the exercise recording activity. Therefore we have designed a new language, JAAL, for that purpose. We followed the path set by Naps et al. [22] and Karavirta [15], when they proposed and designed an XML-based intermediate language for integrating different algorithm animation systems. JAAL defines a set of rules for encoding algorithm animations in human and machine readable form. Its topmost document structure is based in XAAL.

JAAL has two additional features over XAAL. First, JAAL supports saving the model answer for a VAS exercise instance. Second, JAAL represents data structures with more generic language constructs than XAAL. For example, XAAL's XML Schema has separate `list`, `tree` and `graph` structures (see *xaal-structures.xsd* at [20]), while JAAL's recent JSON Schema incorporates all of these in a `graph` schema.

We have demonstrated a prototype language and related software in collecting data from a JSAV-based VAS exercise in another research setting. Based on the evaluation of the implementation, we have updated the requirements of JAAL. Further software development is required to overcome the deficiencies found in the prototype. We will continue with the design of the language and related software in the future.

References

[1] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, "A meta-study of algorithm visualization effectiveness," *Journal of Visual Languages & Computing*, vol. 13, no. 3, pp. 259–290, 2002. doi: https://doi.org/10.1006/jvlc.2002.0237. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1045926X02902375

[2] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. A. Velázquez-Iturbide, "Exploring the role of visualization and engagement in computer science education," *SIGCSE Bull.*, vol. 35, no. 2, p. 131–152, Jun. 2002. doi: 10.1145/782941.782998. [Online]. Available: https://doi.org/10.1145/782941.782998

[3] S. Grissom, M. F. McNally, and T. Naps, "Algorithm visualization in cs education: Comparing levels of student engagement," in *Proceedings of the 2003 ACM Symposium on Software Visualization*, ser. SoftVis '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 87–94. doi: 10.1145/774833.774846. ISBN 1581136420. [Online]. Available: https://doi.org/10.1145/774833.774846

[4] M.-J. Laakso, N. Myller, and A. Korhonen, "Comparing learning performance of students using algorithm visualizations collaboratively on different engagement levels," *Educational Technology & Society*, vol. 12, pp. 267–282, 2009. [Online]. Available: https://drive.google.com/open?id=1O_1LgGqpdz4m9Z6X2y_mkWQQ-Z8zWiG_

[5] V. Karavirta and C. A. Shaffer, "Creating engaging online learning material with the jsav javascript algorithm visualization library," *IEEE Transactions on Learning Technologies*, vol. 9, no. 2, pp. 171–183, 4 2016. doi: 10.1109/TLT.2015.2490673

[6] V. Karavirta et al., "Jsav: The javascript algorithm visualization library. documentation for v1.0.x," 2015. [Online]. Available: http://jsav.io/

[7] "Opendsa." [Online]. Available: https://opendsa-server.cs.vt.edu/

[8] V. Karavirta, A. Korhonen, and O. Seppälä, "Misconceptions in visual algorithm simulation revisited: On ui's effect on student performance, attitudes, and misconceptions," in *2013 Learning and Teaching in Computing and Engineering*. New York, NY, USA: IEEE, 3 2013, pp. 62–69. doi: 10.1109/LaTiCE.2013.35

[9] A. Korhonen, O. Seppälä, and J. Sorva, "Automatic recognition of misconceptions in visual algorithm simulation exercises," in *2015 IEEE Frontiers in Education Conference (FIE)*. New York, NY, USA: IEEE, 8 2015. doi: 10.1109/FIE.2015.7344046

[10] O. Seppälä, L. Malmi, and A. Korhonen, "Observations on student misconceptions—a case study of the build – heap algorithm," *Computer Science Education*, vol. 16, no. 3, pp. 241 – 255, 2006. doi: 10.1080/08993400600913523

[11] R. J. Wieringa, *Design science methodology for information systems and software engineering*. Berlin, Heidelberg: Springer, 2014. ISBN 978-3-662-43839-8

[12] G. Mariani, "Design of an Application to Collect Data and Create Animations from Visual Algorithm Simulation Exercises," Master's thesis, Aalto University. School of Science, 2020. [Online]. Available: http://urn.fi/URN:NBN:fi:aalto-202005313418

[13] S. Safris, "A deep look at json vs. xml, part 1: The history of each standard," n.d. [Online]. Available: https://www.toptal.com/web/json-vs-xml-part-1

[14] V. Karavirta, "Xaal - extensible algorithm animation language," 2005, master's thesis. Helsinki University of Technology, Espoo. [Online]. Available: http://www.cs.hut.fi/Research/SVG/publications/karavirta-masters.pdf

[15] V. Karavirta, A. Korhonen, L. Malmi, and T. Naps, "A comprehensive taxonomy of algorithm animation languages," *Journal of Visual Languages & Computing*, vol. 21, no. 1, pp. 1–22, 2010. doi: https://doi.org/10.1016/j.jvlc.2009.09.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1045926X09000664

[16] V. Karavirta, "Seamless merging of hypertext and algorithm animation," *ACM Trans. Comput. Educ.*, vol. 9, no. 2, Jun. 2009. doi: 10.1145/1538234.1538237

[17] V. Karavirta, "Jsxaal example." [Online]. Available: http://demo.villekaravirta.com/jsxaal/doc/example.html

[18] V. Karavirta, "JsXAAL," Exported from code.google.com/p/jsxaal. [Online]. Available: https://github.com/vkaravir/jsxaal

[19] V. Karavirta, "Facilitating algorithm animation creation and adoption in education," 2007, licentiate's thesis. Helsinki University of Technology, Espoo. [Online]. Available: http://www.cs.hut.fi/Research/SVG/publications/karavirta-lis.pdf

[20] "Xaal schemas," 2009. [Online]. Available: http://xaal.org/schemas/index.html

[21] A. Zavgorodniaia, A. Tilanterä, A. Korhonen, O. Seppälä, A. Hellas, and J. Sorva, "Algorithm visualization and the elusive modality effect," in *Proceedings of the 2021 ACM Conference on International Computing Education Research*, ser. ICER '20. New York, NY, USA: Association for Computing Machinery, 2021, (in press).

[22] T. Naps, G. Rößling, P. Brusilovsky, J. English, D. Jarc, V. Karavirta, C. Leska, M. McNally, A. Moreno, R. J. Ross, and J. Urquiza-Fuentes, "Development of XML-based tools to support user interaction with algorithm visualization," *ACM SIGCSE Bulletin*, vol. 37, no. 4, pp. 123–138, 2005. doi: 10.1145/1113847.1113891