Experience Report: Studying the Readability of a Domain Specific Language

Johann Thor Mogensen Ingibergsson
University of Southern Denmark / CLAAS E-Systems
Nivå, Denmark
jomo@mmmi.sdu.dk

Stefan Hanenberg
University of Essen
Essen, Germany
stefan.hanenberg@uni-due.de

Joshua Sunshine
Carnegie Mellon University
Pittsburgh, PA, USA
sunshine@cs.cmu.edu

Ulrik Pagh Schultz
University of Southern Denmark
Odense, Denmark
ups@mmmi.sdu.dk

ABSTRACT

Domain-specific languages (DSLs) are commonly expected to improve communication with domain experts compared to general-purpose programming languages (GPLs). However, there is a huge gap in the literature concerning how evidence can be given for this expected improvement—a phenomenon that is not only known from DSLs, but also from GPLs in general. This paper presents an experience report of applying an iterative process for evaluating DSL readability for a given DSL in the context of safety-critical software in robotics. The goal of this process is to conduct a randomized controlled trial that gives evidence for the better readability of the DSL in comparison to the readability of a GPL. In this experience report, we describe common pitfalls we identified and possible solutions to overcome these problems in the future.

CCS CONCEPTS

• **General and reference** → **Evaluation**; • **Software and its engineering** → **Domain specific languages**; • **Computing methodologies** → *Vision for robotics*; *Image processing*; • **Hardware** → *Safety critical systems*; • **Human-centered computing** → User studies; Natural language interfaces;

KEYWORDS

DSL, readability, safety, functional safety, computer vision

ACM Reference Format:
Johann Thor Mogensen Ingibergsson, Stefan Hanenberg, Joshua Sunshine, and Ulrik Pagh Schultz. 2018. Experience Report: Studying the Readability of a Domain Specific Language. In *SAC 2018: SAC 2018: Symposium on Applied Computing , April 9–13, 2018, Pau, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3167132.3167436

SAC 2018, April 9–13, 2018, Pau, France

ACM ISBN 978-1-4503-5191-1/18/04.
https://doi.org/10.1145/3167132.3167436

1 INTRODUCTION

Domain-specific languages (DSLs) are programming languages dedicated to a particular problem domain with the goal to decreasing program complexity and increasing programmer productivity within the domain [13]. The use of a domain-specific notation is commonly expected to improve communication with domain experts compared to general-purpose programming languages (GPLs). However, there is relatively little evidence for the benefit of concrete domain-specific languages in general—and it seems as if there is a lack of systematic approaches for evaluating and improving DSLs. On the other hand, a number of hints can be found in literature about how to evaluate software using empirical methods.

This paper presents an experience report, where guidelines from literature were applied to run a randomized control trial. The language used for the study is the Vision Safety Language (ViSaL) DSL for safety-critical computer vision systems for autonomous robots [9]. It turned out that following these guidelines did not lead to satisfying experimental results – which are from our perspective indicators that advice in literature must be weighed carefully and at times even clearly prioritized when designing an empirical study.

2 BACKGROUND AND ASSUMPTIONS

2.1 Background and Related Work

There has been a significant number of investigations of programming language readability in the literature [18], but "What makes computer programming language easy to use for people of all skill levels remains elusive" [18]. There have been several studies dealing with the syntax and constructs of programming languages [18]. In addition, a central research topic has been to make programming seem more natural [14]. Developers spend significant parts of their time trying to read and understand programs [2], and as a result readability of programming languages is critical [1]. Readability is according to Buse et al. "a human judgment of how easy a text is to understand" [5]. It is so critical that it has been suggested that code should be reviewed and developed for this as a goal [6]. However, reliable data from human studies are required in order to judge whether improvements in readability have been achieved [15].

Quantitative empirical evaluation tends not to be applied in software engineering in general [12], and the same is the case in the field of language evaluation [10]. Even so, a number of authors argue for the need for such quantitative evaluations based on randomized

controlled trials [8, 17] – and some have performed randomized controlled trials to evaluate their languages [20]. However, there are hardly guidelines for what the process for language evaluation should look like, although a few textbooks on empirical methods in software engineering are available (see [21]).

Nevertheless it is possible to extract from the literature specific advice in order to define such a process. The work by Feigenspan et al. [7] indicates that it is possible to estimate the performance of subjects upfront based on their their own, subjective self-estimation. Myers et al. [14] proposes natural programming which aims for developers to formulate solutions for issues in natural language, and thereby not being constrained. The work by Ko et al. [12] gives a number of suggestions with respect to the task design in experiments, the execution of pretests, etc. Post studies have been administered in studies to gauge concept understanding of completed tasks [19]. We are however not investigating the use of a tool or the language explicitly, because readability is an intrinsic value of the language. Rosenthal et al. [16] observed the learning effect, that when exposed to a language participants gradually learn more about the language. Furthermore, Rosenthal et al. states that task designs has to be aware of *ceiling* and *flooring effects*.

Taking into account this literature, we would simply want to extract the advice from this literature as a guideline for running our controlled experiments.

2.2 Experimental Process: Assumptions

Based on the previously described literature, we had the following assumptions in mind when evaluating the ViSaL DSL: **(1)** The learning effect is an important issue: participants doing different tasks probably learn from previous tasks—especially if we take into account that the applied DSL was previously hardly known to them. Thus we decided not to apply counter-balanced designs. **(2)** As a consequence of the first assumption, and since different tasks depend on one another, randomization of tasks is probably impossible. **(3)** A common issue for study designs in readability is that the tasks become less representative of practice [20]. Thus the chosen task should be close to real-word examples. **(4)** When dealing with human participants it is critical that the human psyche is taken into account. Participants working in groups or in close proximity can be impacted by "following the crowd" mentality [12]. Thus our study should be conducted individually. **(5)** Recruitment for such studies can however be difficult, which is also a commonly perceived barrier. The study can be conducted online, which lowers the barrier for recruiting participants [4] but weakens the control of the study [12]. Thus we use a monitored remote study by running an experiment in a Skype session. **(5)** The length of the study can be based on similar studies where work-time was between 60-90 minutes [12] but this can result in *ceiling* and *floor effects* [16]. Thus we conduct pilot studies to investigate the task design. **(6)** A critical step in designing an empirical study is to run *pre-pilots* [12]. Thus we use a pilot study to evaluate both the questionnaire and the DSL under investigation, conducted in a controlled environment to establish a baseline for assessing the material with as few variations as possible [11]. **(7)** If specific areas have been identified as particularly difficult, then it can potentially be addressed or optimized using natural programming [14]. Thus applying natural programming seems quite important. **(8)** So-called dry runs seems to be a key element in running experiments. Thus the study should be split up into a pilot study for iterating and dry runs for evaluating the readiness of the material. **(9)** Asking participants about the development skills seems to be important [7]. Thus we give participants the ability to express their own programming performance upfront. **(10)** Post studies that consider language preferences are important [19]. Hence, we explicitly ask which language is preferred, along with enquiries about attitude towards the study, and if there were changes that should be made.

3 STUDY AND RESULTS

We demonstrate and conduct an empirical study on the ViSaL DSL. The study is conducted based on the assumptions found in literature as described in Section 2. The goal is to understand the implications of following the assumptions for an empirical study.

3.1 Research Questions and Approach

The following initial research questions (RQ) deal with comparing readability of ViSaL and C++. **RQ1**: For which language can programmers better connect textual descriptions to program fragments? **RQ2**: Is one language easier to debug than the other? **RQ3**: Do correct-by-construction features of ViSaL target relevant parts of C++?

We aim to answer **RQ1** using questions where participants have to match code and and a description, referred to as "matching" questions. Matching tasks include a prose description of a piece of functionality, and participants are asked to match these to one of several code snippets, written in either DSL or GPL. This is used to examine the time used and solution percentage as an indicator for readability. For **RQ2** we look at two-way bug-finding to examine the debugging performance of participants with a GPL versus the ViSaL DSL. Two-way bug-finding refers to questions where the participants are tasked with finding bugs in programs that can be written in either the DSL or the GPL. For **RQ3** we look at one-way bug-finding to examine issues specific to C++ and which by design of the ViSaL language are not possible in ViSaL. This addresses specific readability for issues in GPL programs, and allows the developer to focus attention to create DSL features to accommodate these issues, thereby improving the resulting DSL usability. These **RQs** are seen as sub-questions to evaluate the overall readability of the DSL.

3.2 Pre-Study

During pre-study piloting, we investigated the DSL design. Certain constructs of the language posed an issue for the participants, which was revised other constructs was perceived to be beneficial for readability. We updated the questions and conducted new "pilot studies." We kept iterating and conducted the pilot studies with new participants, resulting in updates to tooltip descriptions for language constructs. We created a natural programming task with previous participants from the pilot study, as to eliminate training requirements. The question was asked along with a small introduction to ViSaL, and the overall metamodel of the languages, along with a code snippet.

3.3 Main Study

With ViSaL and the material reaching a mature state based on the pilot studies and natural programming, we conducted two subsequent successful dry runs. After the completion, we conducted the studies by alternating the 10 questions between C++ and DSL.

We recruited 22 participants, including industrial programmers, professors, and students. All participants were required to have some experience with C++. Two were removed based on inadequate programming knowledge during the pre-study and the first questions, ending with a total of 20 participants for the evaluation.

3.4 Post-Study

After the study people were asked to answer some questions, and from the discussion that followed, certain improvements for the DSL was suggested. Some participants felt that they had issues with adapting to the different languages, where others were unsure which languages they used at which stage, resulting in them having a hard time rating C++ versus the DSL. However five participants stated explicitly that they felt they got a better handle on the DSL as the study progressed.

3.5 Quantitative Results

We assess the overall readability by looking at the percentage of correctly answered questions. We observe that the matching-type questions had a good performance for ViSaL; these questions tasked the participants with deducting code from safety goals, and the reverse match code to description. The next four questions were bug finding tasks, where the DSL performed worse than C++: there is bad performance in the DSL case for one question, resulting in C++ having a better performance in the two-way bug-finding case. For this specific question it can nevertheless be seen that participants who solved the question correctly in ViSaL did it faster than people using C++. Looking at the timings for all questions it seems like a weak overall trend is that the DSL is using less and less time, while the C++ answers are more volatile. Last the one-way bug-finding question solely concerned C++, with two errors in the code. ViSaL ensures by design that these errors do not exist in generated code. Only half of the participants caught the first error, and only 3 people the second error, and none found both. However, a statistical analysis of the results did not reveal clear effects.

3.6 Qualitative Results

From a qualitative point of view, the overall result is that developers with no experience in ViSaL apparently can solve tasks just as well as in C++. From our perspective, this indicates that the basic concepts in the language are acceptable. Moreover the post-study revealed no clear preference for either language, indicating a potential for the language to be considered acceptable by developers in the domain (although it should be noted that participants with a high experience in C++ were unsure which language they preferred at the end, often ending with choice by chance). Some aspects of ViSaL were however consistently emphasized as being superior to C++. In combination, all of the phases of the study have already generated a number of immediate indicators for the improvements for the language and have moreover identified a number of specific issues to be addressed in future revisions of the language. Therefore, our interpretation is that the study provides first indicators that ViSaL has the potential to improve the readability and that the evaluation process can yield substantial improvement to the DSLs under investigation.

3.7 Assessment

From the results and analysis it is not clear whether C++ or ViSaL is easier or quicker to read. One interpretation is that a null result indicates that ViSaL is easy to read, since participants had prior knowledge of C++ and no prior training or knowledge of ViSaL. The fact that participants after the study still seem to be able to have similar performance is an indicator that the readability is good in ViSaL. Nevertheless it was not possible to find any statistically significant results for the individual tasks. The only candidate for a statistical result was the sum of time used to solve the tasks, which is not a good measure because we have restrictions on time for the individual tasks. Stronger results would be preferred, but we are conducting a hard study when we compare participants knowledge of a known programming language with respect to an unknown language.

Regarding the RQ asked initially, **RQ1** has a weak positive result, i.e., that the DSL has a better connection to textual written requirements than C++. **RQ2** is not in favour of ViSaL in the case of debugging, since C++ outperformed ViSaL initially. However the performance should be viewed in context of the participants not having received any training. **RQ3** relates to the results from one-way bug-finding. Here it can be seen that the built in constraints are an important aspect of utilizing DSLs in new contexts.

4 LESSONS LEARNED

We based our empirical study on assumptions identified in literature, as stated in Section 2. Our initial assumption was that counter-balanced study design could not be applied, because of our choice of not training participants. This assumption can have had a negative impact on the statistical significance of the results, because counter-balancing attempts to address systematic issues for human participants. Counter-balancing could nevertheless have been done within the different task groups, and thereby still present participants with matching tasks before bug-finding tasks, to take advantage of the participants not knowing the language as a means to identify the readability of the language. This implicitly negates our second assumption that randomization cannot be done, since we have argued it could be done within the different types of tasks.

The third assumption was that our examples should be close to reality, however any attempt of representing real practice will always be a representation. Despite the perceived simplicity of using example code from ViSaL use cases, the use of real-world examples could potentially have increased the noise and diversity in the questions, because the question cannot be as focused as artificial examples. Thereby the measurements of the real-world examples become more uncertain and decrease the validity of the overall study results.

The pilot study was a critical point for the empirical study and allowed us to improve the DSL, while evaluating the study material. We saw a benefit on using ViSaL versus C++ in the pilot study,

which was conducted with Post-Docs and Ph.D.'s. When we then moved to the main study, we however had issues of attracting participants, despite our choice to utilize Skype sessions as a means to overcome distances. The main study as a result "only" had 9 out of 20 participants from the same category. We thus introduced additional uncertainty in the main study, where we instead should have had a varied group for the pilot-studies to introduce noise early and focus the study with recruitment for the main study. Thereby the pilot studies could have been utilized better.

In connection with the randomization of the tasks, the task design itself is critical, especially the ceiling and floor effects. Nevertheless our focus did not properly manifest itself in the resulting design, which could be a result of the recruitment. While we from the pilot studies saw that participants could complete the questions within the time frame, we saw that this was not the case in the main study. The result is that we have two independent variables: time and if the question was solved correctly. This implies that the statistical properties deteriorate further.

We saw natural programming and dry-runs as a benefit for the overall study. Natural language allowed us to improve critical constructs of the language that we were not able to address through the pilots and our internal iterations. The dry-runs allowed us to have static material to verify the readiness of the material.

Asking people to rate themselves, while it has been shown that there are some correlation between rating oneself and programming skills [7], the effects are small, meaning that the correlation may not have been strong in our case. It can therefore be beneficial to have introductory programming tasks to ascertain the skills of the participants to better correlate the results.

The inconclusive statistical results could be a result of the difficult goal of the case study to compare a completely unknown language with a known language. The result from the case study was that it seemed that the DSL could be read as easily as the known GPL. The result is a reminder that DSLs can not be assumed to be more readable than a GPL based upon the fact that they are DSLs. Overall the main study was time consuming because only one participant at a time could be complete the study. Nevertheless when the participants got "stuck" the monitoring allowed the researcher to ask the participants to describe the issue leading them to find solutions on their own. The concrete result was several iterations of the ViSaL DSL, while decreasing the needed time for answering question by approximately 50%.

From this section it is evident that relying on assumptions from literature for conducting an empirical study is not a guarantee of a successful study. While we strongly believe that the DSL is a benefit, and saw certain improvements of the language design during the pilots, the current study and its assumption may have hurt so much, that the quantitative results do not match our impression – or it could be the case that our impressions are wrong, because of a given bias.

5 CONCLUSION

In this paper we presented an initial case study of a readability process on a DSL for safety-critical vision software. The case study was conducted based on guidance extracted from literature as a means to ensure that the process could be relied upon. We found that elements from the guidance were beneficial for the study as a whole: pilot studies; dry-runs; and natural programming. Other guidance did not improve the study and may have hampered the statistical significance of the results, as an example by not using counter-balancing or randomization. Based on the inconclusive results, the next steps for this work are to conduct additional studies based on the lessons learned, comparing our approach to that of Barišić et al. [3].

Our current study captures lessons learned from conducting an empirical study based on extracted assumptions from literature regarding how a study should be conducted. The assumptions from literature does not guarantee that the process is beneficial for the overall study results.

REFERENCES

[1] K.K. Aggarwal, Y. Singh, and J.K. Chhabra. 2002. An integrated measure of software maintainability. In *Reliability and maintainability symposium, 2002. Proceedings. Annual.* IEEE, 235–241.

[2] M. Akour and B. Falah. 2016. Application domain and programming language readability yardsticks. *2016 7th International Conference on Computer Science and Information Technology (CSIT)* (2016), 1–6.

[3] A. Barišić, V. Amaral, and M. Goul ao. 2018. Usability driven DSL development with USE-ME. *COMLAN* 51, Supplement C (2018), 118 – 157.

[4] A. Bruun, P. Gull, L. Hofmeister, and J. Stage. 2009. Let your users do the testing: a comparison of three remote asynchronous usability testing methods. In *Proc. SIGCHI Conf. Human Factors Computing Systems.* ACM, 1619–1628.

[5] Raymond P. L. Buse and Westley R. Weimer. 2010. Learning a Metric for Code Readability. *IEEE Trans. Softw. Eng.* 36, 4 (July 2010), 546–558.

[6] J.L. Elshoff and M. Marcotty. 1982. Improving Computer Program Readability to Aid Modification. *Commun. ACM* 25, 8 (Aug. 1982), 512–521.

[7] J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. 2012. Measuring programming experience. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on.* IEEE, 73–82.

[8] S. Hanenberg. 2010. Faith, Hope, and Love: An Essay on Software Science's Neglect of Human Factors. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10).* ACM, New York, NY, USA, 933–946. https://doi.org/10.1145/1869459.1869536

[9] Johann Thor Mogensen Ingibergsson, Dirk Kraft, and Ulrik Pagh Schultz. 2017. Safety Computer Vision Rules for Improved Sensor Certification. In *The First IEEE International Conference on Robotics Computing (ICRC-17).*

[10] K. Kaijanaho. 2015. *Evidence-based programming language design : a philosophical and methodological exploration.* University of Jyväskylä, Finnland.

[11] A. Kittur, E.H. Chi, and B. Suh. 2008. Crowdsourcing user studies with Mechanical Turk. In *Proc. SIGCHI Conf. Human Factors Computing Systems.* ACM, 453–456.

[12] A.J. Ko, T.D. Latoza, and M.M. Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (2015), 110–141.

[13] M. Mernik, J. Heering, and A.M. Sloane. 2005. When and how to Develop Domain-Specific Languages. *Comput. Surveys* 37, 4 (2005), 316–344.

[14] Brad A. Myers, John F. Pane, and Andy Ko. 2004. Natural Programming Languages and Environments. *Commun. ACM* 47, 9 (Sept. 2004), 47–52.

[15] D. Posnett, A. Hindle, and P. Devanbu. 2011. A Simpler Model of Software Readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11).* ACM, New York, NY, USA, 73–82.

[16] R. Rosenthal and R.L. Rosnow. 1991. *Essentials of behavioral research: Methods and data analysis.* McGraw-Hill Humanities Social.

[17] A. Stefik and S. Hanenberg. 2014. The Programming Language Wars: Questions and Responsibilities for the Programming Language Community. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014).* ACM, New York, NY, USA, 283–299. https://doi.org/10.1145/2661136.2661156

[18] A. Stefik and S. Siebert. 2013. An empirical investigation into programming language syntax. *ACM TOCE* 13, 4 (2013), 19.

[19] J. Sunshine, J.D. Herbsleb, and J. Aldrich. 2014. Structuring documentation to support state search: A laboratory experiment about protocol programming. In *ECOOP.* Springer, 157–181.

[20] P.M. Uesbeck, A. Stefik, S. Hanenberg, J. Pedersen, and P. Daleiden. 2016. An empirical study on the impact of C++ lambdas and programmer experience. In *Proc. 38th ICSE.* ACM, 760–771.

[21] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, and B. Regnell. 2012. *Experimentation in Software Engineering.* Springer.