What Challenges Do Developers Face When Using Verification-Aware Programming Languages?

1st Francisco Oliveira
Faculty of Engineering
University of Porto
Porto, Portugal

2nd Alexandra Mendes
INESC TEC, Faculty of Engineering,
University of Porto
Porto, Portugal

3rd Carolina Carreira
Carnegie Mellon University
INESC-ID & IST, University of Lisbon
Lisbon, Portugal

***Abstract*—Software reliability is critical in ensuring that the digital systems we depend on function correctly. In software development, increasing software reliability often involves testing. However, for complex and critical systems, developers can use Design by Contract (DbC) methods to define precise specifications that software components must satisfy. Verification-Aware (VA) programming languages support DbC and formal verification at compile-time or run-time, offering stronger correctness guarantees than traditional testing. However, despite the strong guarantees provided by VA languages, their adoption remains limited. In this study, we investigate the barriers to adopting VA languages by analyzing developer discussions on public forums using topic modeling techniques. We complement this analysis with a developer survey to better understand the practical challenges associated with VA languages. Our findings reveal key obstacles to adoption, including steep learning curves and usability issues. Based on these insights, we identify actionable recommendations to improve the usability and accessibility of VA languages. Our findings suggest that simplifying tool interfaces, providing better educational materials, and improving integration with everyday development environments could improve the usability and adoption of these languages. Our work provides actionable insights for improving the usability of VA languages and making verification tools more accessible.**

***Index Terms*—Formal verification, Verification-Aware programming languages, Software reliability, Usability, Topic modeling, Developer survey, Programming languages**

I. Introduction

In today's world, software is integral to almost every aspect of our lives, from personal communication and entertainment to critical systems like air traffic control and IoT devices. As software becomes increasingly prevalent, ensuring its reliability is essential. The importance of reliability is even greater when considering critical applications where failures can be life-threatening and cause extensive damage [38], such as financial damage and risk users' privacy.

Verification-aware (VA) languages and systems, such as Dafny [34], VCC [13], Why3 [20], and Verus [32], go beyond traditional programming languages by integrating formal verification capabilities directly into the development process. These languages enable the verification of program correctness against user-defined specifications or contracts, using formal methods to mathematically prove that a program adheres to its specification. This built-in verification makes it possible to detect bugs early in the development cycle, offering a higher degree of assurance about software correctness and safety.

Despite their importance, the adoption of VA languages is still low. The reason usually attributed to this is the level of expertise required from the developers and challenges that they often face when trying to prove a program correct. Previous work identifies some challenges when using these tools, such as the lack of counter-examples [47], lack of clarity of the error messages [42], and a general difficulty in understanding why proofs fail [24]. However, to the best of our knowledge, no previous studies have been made to systematically collect the challenges and struggles that VA language developers face.

In this paper, we aim to identify challenges usually encountered when using VA languages with the aim of supporting VA language, verifier, and IDE designers to improve the developer experience and, hopefully contribute to a wider adoption.

We conducted a literature review to identify challenges in the use of VA languages that are documented in the literature and used topic modeling, in particular Latent Dirichlet Allocation (LDA) [9], to extract challenges from developer posts on Stack Overflow. This was followed by a user study to validate the identified challenges and improvements that can be made to VA languages to improve adoption and user experience.

We address the following research questions:

***RQ1**:* What challenges do verification-aware language practitioners face?

***RQ2**:* What is the context of the first contact with verification-aware languages?

***RQ3**:* What improvements can be made to existing tools to overcome those challenges?

Contributions. The main contributions of this work are:

- A collection of challenges that affect VA language developers, obtained through a literature review and the analysis of 1,420 questions from Stack Overflow, validated and complemented by a user study with 31 practitioners.
- An overview of practitioners' first contact with VA languages.
- A collection of suggested improvements, validated by beginner and experienced VA language practitioners, that can inform language, verifier, and IDE designers.

II. Study Design

We used a three-step approach as shown in Figure 1. In *Step 1*, we performed a literature review to understand the context in which practitioners apply VA languages and the

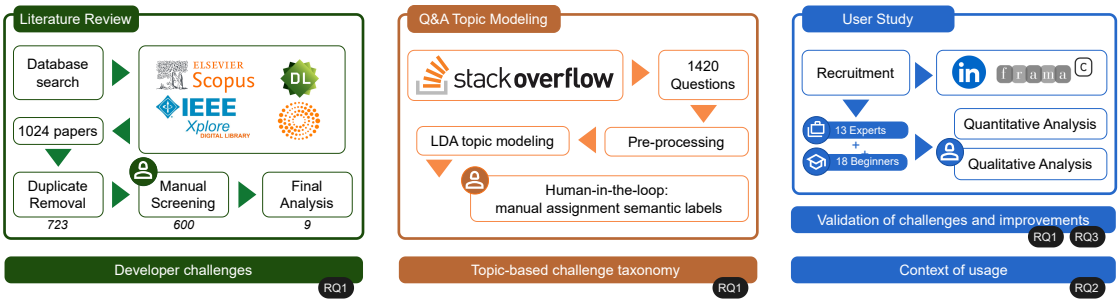


Fig. 1. Overview of the three-step research methodology.

challenges that arise from their use. *Step 2* consisted of topic modeling, in particular using Latent Dirichlet Allocation (LDA) [9], to extract challenges from developer posts on Stack Overflow. Finally, *Step 3* consisted of a user-study to validate the challenges collected in the previous two steps, to explore first contexts of users' contact with these languages, and to explore potential improvements. Steps 1, 2, and 3 contribute to answering RQ1. Step 3 contributes to answering questions RQ2 and RQ3.

A. Verification-Aware Languages Considered

We consider as VA programming languages those that natively support constructs for automated program verification, such as pre-conditions, post-conditions, and invariants. For our search queries, we defined a set of VA languages to be considered, which resulted from a search of Verifiable Programming Languages used in industry taken from the Formal Methods Europe Industry Committee's website,[1] consulted on 20/11/2023. The languages that did not comply with our definition of VA languages were removed.

Following a three-person analysis of the technologies listed, we arrived at the following set: Bandera [14], Dafny [34], Escher C Verifier [19], Eiffel [36], Frama-C [15], Java PathFinder [26], JML [33], mbeddr [49], Spec# [7], Spark Ada [6], VeriFast [28], VCC [13], Whiley [39], and Why3 [20]. Although KeY [2] was considered a VA language, it was later removed as it introduced significant noise due to the name's generality, including, e.g., results from security.

B. Literature Review

To understand from existing work what challenges are faced by practitioners when using VA languages, we performed a literature review. Next, we present all the steps followed.

We queried the ACM Digital Library, IEEE Xplore, Scopus, and Web of Science. The research query was developed following an iterative methodology. Firstly, we identified the following relevant articles that we would like to find with our research query: [27], [37], [42], [47], [50]. Then, we developed an initial search query based on keywords from the collected articles. The resulting query was then tailored to find as many articles as possible, while also containing the previously identified ones. We only accepted articles with ten or fewer years because, considering we are trying to understand developer practices and challenges, older articles could contain issues already addressed or outdated. We also included in the query the set of VA languages identified in Section II-A. The resulting query is as follows:

```
("theorem proving" OR "theorem proof"
 OR "deductive verification" OR "deductive methods"
 OR "formal verification" OR "formal specification"
 OR "contract programming" OR "design by contract"
 OR "programming by contract" OR "Bandera" OR "Dafny"
 OR "Escher C Verifier" OR "Eiffel" OR "Frama-C"
 OR "Java PathFinder" OR "JML" OR "mbeddr" OR "Spec#"
 OR "Spark Ada" OR "VeriFast" OR "VCC" OR "Whiley"
 OR "Why3")
AND industr*
AND (challeng* OR issue* OR drawback? OR constraint?
     OR limitation? OR obstacle? OR difficult*)
```

The query only searches in the field *Abstract* because it is the field that all databases support. The queries were conducted on 26/12/2023, resulting in 1024 articles. After 729 duplicates were removed, 600 articles were screened.

The following exclusion criteria were applied. These aim to include the maximum number of relevant articles. When in doubt, the full-text screening was applied. *Title Screening:* 1) the title is not related to formal verification (or derived terms) — e.g. "Solidification Principle in Large Vertical Steel Casting Under the EMS Effect"; 2) the title mentions formal verification tools that are not VA languages — e.g. Alloy, SysML. *Abstract Screening:* 1) the term formal methods or related is used in contexts not related to our research — e.g. formal verification of startup models; 2) the tools or formal verification methods mentioned are related to the area of software engineering, but the application area is not aligned with the our research — formal verification of system models. *Full-Text Screening:* 1) the article does not mention any challenges; 2) the challenges are not related to the application of VA languages (e.g., the article talks about the challenges of integrating Dafny with Solidity).

After this process, nine of the 600 articles were considered relevant and were analyzed (see III-A), highlighting a significant research gap and reinforcing the relevance of this study.

C. Stack Overflow Q&A Study

This section details how we extract challenges from developer posts on Stack Overflow, a widely used Q&A plat-

[1] https://fme-industry.github.io/

form where developers from diverse backgrounds openly discuss real-world programming problems at scale, making it a rich and authentic source for empirical studies [1], [3], [4], [8], [18], [25], [40]. However, manually analyzing the vast amounts of data available in these sources may become infeasible, particularly as the dataset size increases. Therefore, we use topic modeling, a popular analytical tool for evaluating data, as an approach to identify groups of similar words, or topics, within the text [48]. One of the most popular algorithms for topic modeling is *Latent Dirichlet Allocation* (LDA) [9], an unsupervised machine learning algorithm that models automatically extracted topics in corpora. It is based on the idea that each document is a mixture of topics, and each topic is a mixture of words. LDA aims to discover how much of each topic is in each document and how much of each word belongs to a particular topic. The algorithm outputs probability distributions that quantify the proportion of each topic within a document and the association of each word with a given topic [45]. Several previous studies have used unsupervised machine learning, in particular LDA, for categorizing forum posts [1], [3], [4], [8], [18], [25], [40].

Our forum post analysis with topic modeling follows prior work and involves four steps: a) collecting Stack Overflow data on VA languages, b) pre-processing (cleaning, stop-word removal, stemming/lemmatization), c) training an LDA model, and d) labeling tags based on associated words and topics.

a) **Data Collection:** Three popular Stack Overflow post datasets are: SO official data dumps [21], SOTorrent [5], and SE Data Explorer.[2] The first is distributed as a very large XML file, which we found difficult to process; the second is not updated since 2020; and the third is available through a human-centered interface, making it difficult to automate data collection. Therefore, we used the official Stack Exchange API[3] as, similarly to SE Data Explorer, it provides the newest data at the time of querying, but in a computer-friendly format.

We identified the tags we would use to query the API endpoint and settled with a variation of a list of VA languages presented in Section II-A. From our observations of the Stack Overflow tag list,[4] only hyphenated tags are supported, so "Escher C Verifier", "Java PathFinder", and "Spark Ada" had to be transformed to accommodate this requirement. In the final mapping between VA language names and the tag names, "Escher C Verifier", "Java PathFinder", and "Spark Ada" became "escher", "pathfinder", and "spark-ada", respectively.

Moreover, we decided only to query the */questions* endpoint because we are looking for challenges commonly found in questions but not in the answers and comments.

We ended up with 1420 questions, most of which were about Dafny and Frama-C. The distribution of questions was Dafny: 530, Frama C: 474, Eiffel: 289, Spark Ada: 55, Why3: 52, JML: 44, and Spec#: 16.

b) **Data Pre-processing:** Since we gathered data from the Stack Exchange API, the question bodies contain code snippets and tags (similar to [21] and [5]). With that in mind, we built four pre-processing pipelines, with each pipeline combining a subset of the following nine steps.

Pre-processing steps. Based on prior research, we considered the following nine pre-processing steps:

- *__remove_code:* Removes all tags and code snippets from the body of a given question, wrapping it into a *div* tag, and then performing a depth-first search into its children, removing all *code* and *pre* tags. Only the inner text of the other tags, such as *anchor*, *strong*, *paragraph*, etc, and text that is directly a child of a given tag is kept. The resulting text is then trimmed and joined with a white space. Based on [8], [4], [25], [3], [18], and [40].
- *__gensim_preprocessing:* Performs common pre-processing tasks using the functions *strip_numeric*, *strip_multiple_whitespaces*, and *strip_punctuation* from the Gensim[5] library. This step removes digits from the initial string and multiple consecutive white spaces, leaving just one separating the words. Finally, it strips all the punctuation from the input string. Based on [4], [25], [3], [18], and [40].
- *__lemmatize_body:* Starts by tokenizing the input string using NLTK's *word_tokenize* function. After tokenizing, we apply stemming using NLTK's WordNet lemmatizer. This lemmatizer needs to receive the part of speech of each word. Since identifying specific parts of speech is out of the scope of our work, we decided to consider every word as a verb since we found that verbs and plural nouns were transformed into their stems. Based on [25].
- *__remove_stop_words:* Filters out stop words using the NLTK stop words corpus (e.g., *a*, *an*, *the*, *of*, *and*). This is an approach used by [25], [1], and [40].
- *__transform_bigrams:* Takes a list of words and returns a list of bigrams using NLTK *bigrams* function. This step was also used by [1], although the authors used Gensim to build the bigrams.
- *__make_lower_case:* Transforms all words into their lowercase version.
- *__remove_handpicked:* Removes words that, after examination, we considered as not adding value to the trained model — e.g., *also*, *always*, *etc*, *like* (full list available in the supplementary material). Based on [3] and [40].
- *__remove_uncommon:* Removes words with less than three uses since these words can not form patterns.
- *__remove_small:* Removes words with less than two characters since we considered that words with that size could not express meaningful value.

Pre-processing pipelines. After collecting the data, we process the questions using the four pre-processing pipelines listed below, developed as a combination of the nine steps listed previously. All the pipelines are based on [25] because they lemmatize the words before removing the stop words.

1) *baseline:* This pipeline is the simplest and produces a list of individual words. The exact sequence of the

[2]https://data.stackexchange.com/

[3]https://api.stackexchange.com/

[4]https://stackoverflow.com/tags

[5]https://radimrehurek.com/gensim/

steps is: *__remove_code* → *__gensim_preprocessing* → *__lemmatize_body* → *__make_lower_case* → *__remove_stop_words*.

2) *baseline_bigram:* Create bigrams from the list of words resulting from the baseline pipeline. Based on the work by [1]. The steps are: *baseline* →*__transform_bigrams*.
3) *rem_words_baseline*: Similar to the regular baseline pipeline but incorporates the innovation of [3] and [40] of removing domain-specific words as well as an additional removal of words with less than three uses — since these words can not possibly form patterns — and less than two characters — since we considered that words with that size could not express meaningful value. The sequence is: *baseline*→*__remove_handpicked*→*__remove_uncommon* → *__remove_small*.
4) *rem_words_bigram:* Based on the rem_words_baseline but, similarly to baseline_bigram, ends with a step that transforms the resulting list of words in a list of bigrams. Steps: *rem_words_baseline* →*__transform_bigrams*.

c) **Model Training:** To configure the LDA algorithm, several parameters, known as hyperparameters, must be set: the number of topics (K); the number of iterations (i.e., optimization steps); the number of passes over the training corpus; to what extent questions mix topics (α) — a high value means that questions cover a wide range of topics; and to what extent questions mix words (β) — a high value means that topics are likely to contain a broad range of words, rather than being focused on a small, specific set of terms.

Each of the four pre-processing pipelines presented in the previous subsection was tested against each combination of the following hyperparameters: K ranging from 2 to 40 in steps of 2 to capture all identified challenges, and from 500 to 3000 in steps of 500 to align with prior work; we also tried 50 iterations, the Gensim default. We set the number of passes at 1 (Gensim default) and 5 (as per [25]). For β, we tested 0.1, 0.01, and None (Gensim default), while α followed the common formulation $\alpha = 50/K$ from prior studies [3], [4].

Final Model. We select the final model configuration through an evaluation across multiple training runs and criteria, which, due to space limitations, we briefly detail below. More details can be found in the supplementary material.[6]

This process began with an *initial training run* using the full dataset of collected Stack Overflow questions. This run aimed to evaluate the performance of different pre-processing pipelines and model configurations. This produced: 4 model runs, one for each pipeline; 840 model configurations resulting from the combination of LDA hyperparameters; and 3360 models, four for each configuration. To account for the non-deterministic nature of the LDA algorithm, we trained four models for each combination of the hyperparameters.

Model performance was evaluated using two primary metrics: topic coherence (C_{UMass}) [41] and perplexity score (logarithm of perplexity) [9]. We decided to use C_{UMass} to evaluate our models since it is a standard metric that can be quickly calculated. We also analyzed the logarithm of the perplexity, which measures how likely an outside observer is to predict the values drawn from a model. The values used are the output of the function log_perplexity[7] in Gensim.

We trained a model for each combination of pre-processing pipelines and hyperparameters. The performance of each combination (pipeline × combination of hyperparameters × 4 models) was assessed by calculating the average coherence and perplexity scores, as well as their standard deviations across the four models. This allowed to identify configurations that were not only high-performing but also consistent. Configurations with high standard deviations were considered unstable. Taking into consideration LDA's non-deterministic nature, this strategy allows us to detect if a score is exceptionally good or if the hyperparameters are able to make LDA consistently output good models.

The *baseline* and *baseline_bigram* pipelines resulted in configurations with significantly high standard deviations in coherence, indicating instability. The *rem_words_baseline* pipeline showed better overall performance and, unlike *baseline*, it removed words that were not helpful for identifying challenges. The *rem_words_bigram* pipeline generally showed higher degradation in coherence compared to *rem_words_baseline*. Based on this initial analysis, the *rem_words_baseline* was chosen. A configuration with 16 topics within this pipeline was initially selected for further consideration.

As further explained in II-C0d, questions with low probability of belonging to a topic were discarded and open card sorting [46] was performed based on the top 30 words and top 20 documents for each topic. Looking at the composition of the topics, we obtained 209 unique questions from the 320 that were assigned to the 16 topics. Most questions appeared only once, indicating diverse topics.

After applying open card sorting to the clustered questions, we obtained the tag distribution depicted in Figure 2. The most frequent tag is the *not-suited* tag with 79 questions, meaning that around 38% of questions were considered inadequate to answer RQ1. Of the 16 resulting topics, in 15, the most frequent tag was *not-suited*, indicating that the topics are not representative of any challenge.

To overcome this issue, we decided to filter out these questions and run the training in a dataset without the 79 inadequate questions. A *second training run* was then done using the filtered dataset. Again, we applied the four pre-processing pipelines and explored various hyperparameter configurations and used the same evaluation metrics as in the *first training run*. Similar performance patterns to those obtained in the *first training run* were observed across the pipelines, with *baseline*, *baseline_bigram*, and *rem_words_bigram* showing high errors. The *rem_words_baseline* pipeline again demonstrated consistent performance for configurations filtered by lowest error. Considering the similar performance, we again chose the *rem_words_baseline* pipeline to pick the best configuration

[6]https://figshare.com/s/bc220b6c300b46c97d89

[7]https://radimrehurek.com/gensim/models/ldamodel.html#gensim.models.ldamodel.LdaModel.log_perplexity [last access 31st August 2024]

Fig. 2. Frequency of each tag (initial training)

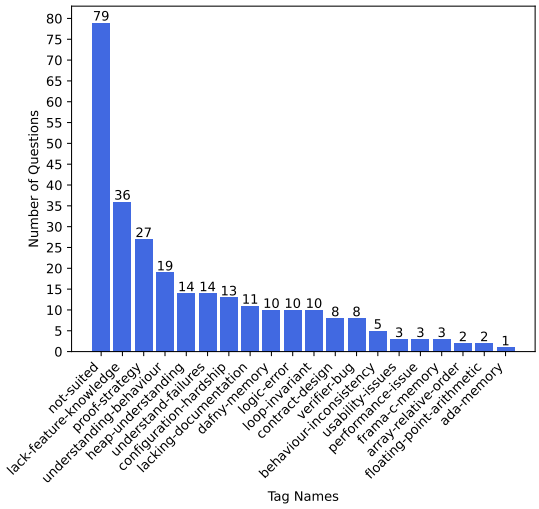
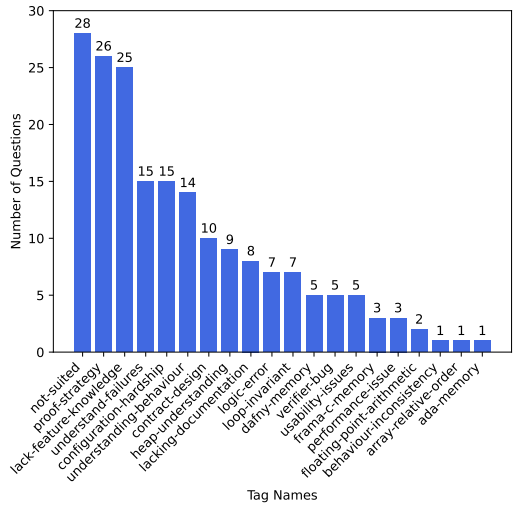


Fig. 3. Frequency of each tag (second training)



and model. We chose the configuration with eight topics because it resulted in a number of topics that allowed for the identification of challenges consistent with prior literature, as discussed in Section III-A. Looking at the topics, we observe these are composed of 160 questions, from which 145 are unique. This indicates that this model is diverse.

After performing open card sorting, we end up with the tag distribution shown in Figure 3. Despite being the most popular tag, the *not-suited* questions only comprise 19% of the overall questions against the 38% obtained in the *initial training run*. Based on our analysis, we decided that this model was suited for naming and challenge identification.

d) **Topic Labelling:** This process involved analyzing the chosen model configuration's output. Questions were filtered, discarding those where the probability of belonging to a topic was less than $\frac{1}{n}$ (where n is the number of topics). The reasoning for this was that $\frac{1}{n}$ is the probability of randomly assigning a document to a given topic, thus we only accepted probabilities greater than the random assignment.

After filtering the questions with low probabilities, we sorted them and performed open card sorting [46] based on the top 30 words and top 20 documents for each topic. The labeling for each topic was discussed among the team members until a consensus was reached.

For the final chosen model configuration, which had eight topics, the labeling process involved analyzing the frequency and distribution of the tags that were created during the open card sorting process.

For *Topic 0*, even though the most common tag is *not-suited*, the label **Memory Model Understanding** was assigned, influenced by the second most common tag, *heap-understanding*. *Topic 1* was labeled **Language Understanding**, driven by the frequency of the *lack-feature-knowledge* and *understand-behaviour* tags, and the occurrences of *logic-error*, *loop-invariant*, *understand-failures*, and *lacking-documentation*. *Topic 2* was labeled **Not Suited** because no clear relationship between the clustered questions was found. *Topic 3* was labeled **Environment Setup** as most of the questions are related to to the *configuration-hardship* tag. *Topics 4* and *5* were labeled **Non Relevant** as no clear relationships were found between the top question label. *Topics 6* and *7* were labeled **Proof Elaboration**, because both contain many questions regarding the label *proof-strategy* and some questions regarding *loop-invariant*. These labeled topics were then used to identify broader categories of challenges, which are presented in the Section III-B.

D. User Study

To better understand the challenges faced by VA language users and assess the relevance of the issues identified in our prior qualitative analysis, we conducted an online survey study. We designed the study to capture both quantitative and qualitative feedback, drawing on the experiences of participants with different levels of familiarity with VA languages. We provide the full protocol in the supplementary material.

1) Survey Design: We start the survey with a section about background information, where we ask about familiarity with VA languages and years of experience. We also ask about participants' initial contact with VA languages and how they learned to program with them. We then attempt to validate the answers to RQ1. We use matrix-style questions with a 5-point Likert scale to ask questions about challenges related to **Proof Elaboration** and **Environment Setup**. We also ask participants about a subset of **language-agnostic challenges** identified in Section III-B and **language-specific challenges** with Dafny and Frama-C. We only showed these sections to participants who mentioned having experience with these languages. This part of the survey mainly targets Dafny and Frama-C practitioners because over 70% of the questions found in Stack Overflow were related to these specific languages and some challenges are language dependent.

We ask two open-ended questions about additional challenges not identified in the previous parts of the study and suggestions for improving the adoption of VA languages.

Then, we use a matrix-style closed-ended question to ask about the suggestions we developed *a priori*. We asked open-ended questions before the closed-ended to prevent bias and allow the participants to think more freely without being conditioned by the options they had already seen.

2) Recruitment: Our sample had a total of 31 participants. We recruited participants using our social media, professional network, and snowball sampling. We began by distributing the survey to 15 students of the 2023/2024 edition of the Formal Methods for Critical Systems course, an optional first-year Master's level course from the Faculty of Engineering, University of Porto. Using Proxycurl,[8] we gathered 28 profiles of LinkedIn users from the US with software and formal verification skills, contacted 16 industry professionals from our professional network, and 26 Dafny contributors on GitHub, whose contact information was public. Additionally, we shared our survey on social media through *X*.[9] We used snowball sampling by asking our contacts to distribute the survey to relevant members of their professional and student networks. We also distributed the survey through Frama-C's mailing list. We divide participants by years of experience: more than three are considered *experienced*, all others are *beginners*.

All participants were over 18 years old and provided informed consent after reviewing a form with the study's purpose, procedures, and their rights. Participants were informed that they were not required to participate and could withdraw at any time without penalty. No personal or identifiable data were collected.

Sample. Most of our sample identify as male and are employed in the computing field with formal education. Most experienced participants have some graduate or professional degree, while most beginners only have a bachelor's degree. See Table I for complete demographic information.

3) Quantitative Analysis: For our quantitative data, we use the *Mann-Whitney U* test to check whether the data from two groups, beginners and experienced, differ significantly. We chose a non-parametric test because our data does not follow a normal distribution. We use a Friedman test to compare answers to statements within the same group. If the Friedman test is significant ($p < 0.05$), we then use the Wilcoxon signed-rank test to make pairwise comparisons of each statement and understand which particular statements are significantly different from the others. Finally, we analyze the frequency of each response to understand if the participants encountered the challenge. To analyze Likert scale data, we remove the *Not Sure* responses from the statistical test, assuming that these were due to the participant not understanding the statement.

4) Qualitative Analysis: Most of our survey consisted of closed-ended questions. However, we asked two open-ended questions about challenges and possible solutions to challenges. We used deductive and inductive coding to analyze the qualitative data derived from these questions. Two coders did two independent rounds of coding. In the first round, the initial coder designed the first version of the codebooks that both coders used. This first codebook was developed with prior knowledge of the types of challenges identified in an earlier portion of the study. After completing the first pass, the codebook was refined, and the coders met to discuss the added and modified codes. Following these updates, both coders independently applied the revised codebooks to the dataset in a second round of coding. After this, the coders met to resolve discrepancies in their coding and reached a consensus. The process resulted in two finalized codebooks: one for suggested improvements, consisting of 13 codes, and another for challenges, consisting of 14 codes. Each participant's response could have up to 5 different codes.

[8]https://nubela.co/proxycurl/

[9]https://x.com/

TABLE I
OVERVIEW OF PARTICIPANT DEMOGRAPHICS AND EXPERIENCE

	Beginners	**Experienced**	**Total (%)**
Gender			
Male	14	11	80.6%
Female	4	1	16.1%
Prefer not to say	–	1	3.2%
Age			
18–24 years	15	1	51.6%
25–34 years	3	2	16.1%
35–44 years	–	7	22.6%
45–64 years	–	3	9.7%
Highest Education Achieved			
Some college, no degree	2	–	6.5%
Associate's or technical degree	–	1	3.2%
Bachelor's degree	14	1	48.4%
Graduate/professional degree	2	11	41.9%
Race			
Asian	1	1	6.5%
Black	4	–	12.9%
White	12	8	64.5%
Prefer not to say	1	4	16.1%
Currently Employed in CS			
Yes	7	13	64.5%
No	9	–	29.0%
No, but have been in the past	2	–	6.5%
Formal CS Education			
Yes	18	11	93.5%
No	–	2	6.5%
Total Participants	**18**	**13**	**100%**

III. WHAT CHALLENGES DO VERIFICATION-AWARE LANGUAGE PRACTITIONERS FACE? (RQ1)

A. Findings from Literature Review

From the existing literature, we identified several challenges, such as: *behavior dependent on previous executions*, which include challenges related to the verification of properties where the behavior is influenced by previous executions [17], [23]; *behaviour inconsistency*, as verification success changes depending on the solver [47]; *cost of applying VA languages* [50]; *feedback quality*, which includes the lack of counter-examples by some tools [47], the lack of clarity of the

error messages [42], and a general difficulty in understanding why proofs fail [24]; *handling boolean values*, i.e. challenges related to verifying properties about boolean values [16], [23]; *imprecise specifications* [23], [30]; *increasing the scale of verification*, which includes challenges related to the integration of multiple verified functions [30], the scaling of verification time [16], the difficulty of tracking and maintaining annotations [16], [27], and general claims about scalability [37]; *integration with hardware*, including difficulties related to the verification of functions that interact with hardware, often resulting in side effects [17]; *integration with other tools* [27]; *lack of features* to support verifying some behaviors [30], [43]; *lack of verified APIs* [27]; *lacking support for specific tasks*, such as lack of support for floating-point arithmetic [27], [37], concurrent software [27], and reflection [27] verification; *loop invariant discovery* [24], [42]; *need for automation*, which includes the need for automation so that deductive verification can be widespread [37]; *insufficient [27], [42], [47] and costly [27], [50] support and documentation*; *switching costs*, which includes authors that considered the lack of interoperability between tools and the cost to switch them an obstacle for the adoption of VA languages [50]; and *type conversions*, including challenges related to the verification of implicit [23] or heterogeneous [16] type conversions.

B. Findings from Stack Overflow Q&A Study

In this section, we group the identified challenges by the four topics identified in II-C0d. Due to space restrictions, full details can be found in the supplementary material.

a) **Proof Elaboration:** The challenges in this class are related to the programming paradigm, so these are language agnostic. The first challenge is related to creating a proof strategy. In many questions, authors could not even start the proof, not knowing what proof strategy to choose. This challenge was observed in nine questions.

The second challenge is related to completing a proof. In contrast with the previous challenge, these authors could start to design a proof strategy that then proved impractical in verifying the properties of the code. This challenge was observed in three questions.

The third challenge is related to authors who chose a non-ideal strategy to prove the properties of the code; for example, choosing a proof by induction when a proof by contradiction would be easier. This was observed in one question.

The fourth challenge concerns identifying and expressing the necessary loop invariants to verify loops. This challenge was observed in five questions.

The fifth challenge, observed in one question, is related to the fourth and concerns identifying and expressing a valid decrease clause that proves that loops or recursion functions terminate. This challenge was observed in one question.

The sixth challenge concerns not being able to gather feedback, i.e., whether the contracts are respected due to the verifier timing out. This was observed in two questions.

b) **Environment Setup:** The challenges in this topic are language agnostic. The first class of challenges is related to dependency installation, which were identified in four questions. It includes challenges related to missing dependencies, incompatible dependency versions, and dependencies not being found. The other two classes of challenges are related to unsupported use cases and missing installation instructions. These challenges are represented by one question each.

c) **Memory Model Understanding:** The challenges in this topic are language-specific, as we observed that languages where the user manually manipulates the memory, like C, have different challenges than garbage-collected ones, like Dafny.

We identified three challenges among Dafny programmers, all related to the *modifies* clause. The *modifies* clause specifies the set of memory locations that a method, iterator, or loop body may modify [35].

The first challenge is related to the user applying a *modifies* clause to a given memory location, holding a reference to an object within the scope of a method, but only changing the properties of the given object inside a loop. Dafny assumes that the loop might change the object to which the memory location is pointing, thus making verification based on the previously set object properties impossible. The programmer must create an invariant saying that the memory location always points to the same object.

The second challenge is related to programmers not understanding which variables are affected by the *modifies* clause, identified in two questions. In both, we see that the authors fail to acknowledge what is the right target of the *modifies* clause when the target property is a memory location to an object: one author is missing a *modifies this* clause; the other tries to modify a property of the class while using *modifies this* instead of including *modifies this.property_name*.

In another question, we observed a similar challenge, but the clause the author is using is the *reads* clause. The *reads* clause marks all the heap memory locations the function is allowed to read [35].

d) **Language Understanding:** Similarly to the previous topic, most of the challenges in this topic are related to features that are language-specific, related to Dafny, and to the Frama-C WP Plugin, which uses ACSL as an annotation language.

For Dafny, in one of the questions, the author seemed to lack knowledge about partial functions. The second question is about the *fuel* attribute in Dafny using an example function. Another question concerns an author that fails to understand the error message provided by the verifier about a type definition. Upon further inspection, we considered the message self-explanatory for developers who knew about subset types and witnesses. In the final question, the author fails to acknowledge what constructs are opaque. In Dafny, when a construct is opaque, the verifier does not introspect it to derive pre-conditions and post-conditions — the developer must define the contract. In this specific case, the developer forgot the constructor's post-condition.

Regarding the Frama-C questions, in one of the questions, the author is trying to use a feature defined in the ACSL Reference Manual but not implemented by the Frama-C WP Plugin. In another, the author wants to prove that, given

the same arguments, the function always returns the same values using ACSL. The accepted answer claims that this is impossible in ACSL. In the third question, the author is trying to use a C11 specification feature not implemented by the Frama-C pre-processor at the time of writing. In the final Frama-C question, the accepted answer claims that the author fails to acknowledge the need for *loop assigns* in the presented loops. The question has some *loop assigns* expressions and was edited 8 hours after creation, presumably to add them.

The following questions are considered language agnostic. In the first, the author wants to prove a lemma in Dafny. The original code of the author causes an internal error in Dafny 4.7.0. According to the accepted answer, an assertion was missing to make the verifier accept the proof. Because the need for the assertion is probably tied to the internal verification mechanism, we considered that the proof elicits a lack of language understanding, namely, the underlying verification mechanisms. In the second question, the author is trying to understand the translation of mapping WhyML code into the SMT encoding to understand how the proofs for ACSL annotated programs are provided. This question is not a challenge but introduces a need for understanding the underlying verification mechanisms. The last question contains a simple syntax error where the author does not use the correct syntax to write comments in ACSL.

C. Findings from User Study

a) **Proof Elaboration Challenges:** In this section, we validate seven challenges related to *Proof Elaboration*. Of the seven challenges, presented in Table II, only PE_7, which addresses difficulties in expressing the necessary clause to prove loop/recursion termination, had a significant p-value ($p = 0.02666$), this means that PE_7 is the only PE challenge significantly worse for beginners than experienced users. All other challenges had a non-significant p-value ($p > 0.05$) when comparing beginners and experienced users.

To assess differences between the seven conditions across all participants, we conducted a Friedman test. The results indicate a significant difference between the conditions, $chi^2(6) = 27.79$, $p = 0.0001$. To assess these differences, we conducted pairwise comparisons with a Wilcoxon signed-rank test with Bonferroni correction (see Table III). The results indicate that there were significant differences between PE_1 and PE_7 ($p = 0.023$) and between PE_2 and PE_7 ($p = 0.032$). No other comparisons reached statistical significance after the Bonferroni correction (all $p > 0.05$). This means that PE_7, the challenge of expressing loop/recursion termination clauses, is a significantly less important problem than PE_1 — designing proof strategies — and PE_2 — completing proof strategies.

Our results show that most developers struggled with designing proof strategies (PE_1, 64.5%). The challenge related to difficulty completing proof strategies (PE_2) was more split with 48.4% agreeing that this is an issue for them and 22.6% disagreeing. When it comes to searching for the right proof strategies (PE_3), we saw a similar split with 48.4% agreeing and 22.6% disagreeing it is a challenge. Identifying (PE_4) and expressing (PE_5) loop invariants and termination clauses (PE_6 and PE_7) does not seem to be a problem for most.

TABLE II
PROOF ELABORATION CHALLENGES

ID	Challenge Description
PE_1	Difficulty designing proof strategies accepted by the validator.
PE_2	Difficulty completing designed proof strategies.
PE_3	Tendency to choose non-ideal proof strategies.
PE_4	Inability to identify necessary loop invariants.
PE_5	Inability to express necessary loop invariants.
PE_6	Inability to identify clauses proving termination.
PE_7	Inability to express clauses proving termination.

b) **Environment Setup Challenges:** While we do not see a significant difference between beginners and experts, most developers did not encounter dependency conflicts when installing verification tools with 67.7% agreeing this is not a problem. Similarly, the challenge of finding the correct version of a verification tool compatible with a preferred IDE was not common (77.4% did not consider this a challenge).

c) **Feature Knowledge Challenges:** We do not see a significant difference among experienced users. However, most developers, particularly beginners, struggled to understand how their high-level code is translated to the underlying SMT code, while more experienced developers felt confident in this area. When it came to translating proofs from other mediums (e.g., paper) to the VA language, most developers did not report significant difficulties. Experienced developers showed relatively higher agreement with this challenge than beginners.

d) **Dafny Challenges:** Most participants did not face significant issues understanding the "modifies" and "reads" clauses or distinguishing between opaque and non-opaque statements. However, some developers still struggled with expressing the immutability of variables, and many participants were unfamiliar with critical concepts like the fuel attribute, partial functions for reasoning about high-order functions, and using witnesses to prove the non-emptiness of type definitions.

e) **Frama-C WP Plugin Challenges:** None of these challenges had a significant p-value ($p < 0.05$) when comparing beginners and experienced users. Most developers felt confident in their understanding of the different memory models and their ability to choose the correct model for proving code properties and in selecting variables for "loop assigns" statements. Some developers also identified missing features in Frama-C's C specifications and noted that certain features available in other plugins were absent in the WP Plugin.

f) **Additional Challenges:** The most frequently reported challenges were "Tool Performance Issues" and "Poor Error Messages and Feedback," each mentioned seven times. "Complex Language Features and Low-Level Operations" was reported six times, followed by "Specification and Annotation Challenges" with five occurrences. Both "Understanding Verifier Capabilities" and "Learning Curve" were identified four times each. Additional challenges included "Mindset Shift," "Scalability and Large Project Integration," and "IDE Integration," each reported twice. The following challenges were identified once each: "Lack of Documentation," "Conflicts

TABLE III
PAIRWISE WILCOXON SIGNED-RANK TEST RESULTS WITH BONFERRONI CORRECTION FOR MULTIPLE COMPARISONS. *SIGNIFICANT AT ADJUSTED $p - value < 0.05$.

	PE_1	PE_2	PE_3	PE_4	PE_5	PE_6	PE_7
PE_1	-	95.5, 1.000	144.0, 1.000	172.5, 0.237	150.5, 0.093	207.5, 0.172	263.5, 0.023*
PE_2	-	-	80.5, 1.000	143.5, 1.000	139.0, 0.368	169.0, 0.336	189.5, 0.032*
PE_3	-	-	-	178.5, 1.000	148.5, 0.611	205.5, 0.811	184.5, 0.058
PE_4	-	-	-	-	26.5, 1.000	60.0, 1.000	86.0, 1.000
PE_5	-	-	-	-	-	104.5, 1.000	88.0, 1.000
PE_6	-	-	-	-	-	-	49.5, 1.000
PE_7	-	-	-	-	-	-	-

with Performance Optimization," "Proof Stability," "Translation of Algorithms," and "Lack of Proof Automation."

RQ1: VA language practitioners most commonly face challenges in designing and completing proof strategies. The only challenge different between beginners and experienced users was proving loop/recursion termination, though this issue was uncommon and significantly less important than the challenges of designing and completing proof strategies. Language-specific semantics (e.g., frame specifications, termination clauses) appeared on Stack Overflow but were not confirmed as challenges in our user study. While environment setup is less problematic, feature knowledge challenges were confirmed, with beginners struggling more with SMT translation, and experienced developers with translating proofs to the VA language. We identified fourteen new challenges, the most reported ones being "Tool Performance Issues", "Poor Error Messages and Feedback", "Complex Language Features and Low-Level Operations", and "Specification and Annotation Challenges".

IV. WHAT IS THE CONTEXT OF THE FIRST CONTACT WITH VERIFICATION-AWARE LANGUAGES? (RQ2)

To understand the educational and professional pathways that lead individuals to work with VA languages, we asked all participants two questions, one about where they first **learned about** VA languages and another about where they **learned to program** with VA languages. In our sample, we have 18 beginners and 13 experienced programmers. Most beginners had their first contact and learned VA languages in an *optional university course* (16 participants, 89%), with the exception of two participants who had their first contact in a work context. On the other hand, most experienced developers had contact with VA languages in a *work context* (12 participants, 92%). Overall, the most frequent response for both groups was a university course (77%) and work context (48%). Three participants mentioned online discussions, one mentioned "university and enterprise colleagues" (P17), another reported "self-learning" (P18), and finally, one participant said their first encounter was due to being "bored in high school" (P28). From our observation, it seems that most participants learn about VA languages as a choice of education or career path. Table IV summarizes the results.

TABLE IV
CHARACTERIZATION OF PARTICIPANTS BASED ON THE CONTEXT IN WHICH THEY FIRST LEARNED ABOUT VERIFICATION-AWARE LANGUAGES.

Context	**Frequency**
Optional University Course	22
Work Context	15
Online Discussions	3
Mandatory University Course	2
With university and enterprise colleagues	1
Self-learning	1
Was bored in high school	1

When it comes to their experience programming with VA languages, their responses were consistent with their first contact. Most participants reported learning how to program in a university course (22 participants, 71%) and being self-taught (14 participants, 45%). When considering only experienced users, 11 participants (85%) are self-taught and 10 participants (77%) learned in a work context.

These results suggest that beginners usually first encounter VA languages at university, while experienced developers do so in professional settings and increasingly through self-teaching. These insights highlight the need for differentiated learning resources: universities should integrate VA languages into curricula for early exposure, while practitioners could benefit from self-guided resources and community support.

RQ2: The context of the first contact with VA languages seems to vary by experience level. Most beginners first encounter VA language in university courses, while most experienced programmers are introduced to VA languages in a work context. Moreover, as users gain experience, self-teaching emerges as a key mechanism through which they acquire knowledge.

V. WHAT IMPROVEMENTS CAN BE MADE TO EXISTING TOOLS TO OVERCOME THOSE CHALLENGES? (RQ3)

The user survey combined closed and open-ended questions. The closed questions aimed to validate the following specific proposals derived from our earlier analysis: 1) Create user-readable counter-examples to complement error messages.

TABLE V
CODEBOOK WITH CODE FREQUENCY FOR THE PROPOSED SOLUTIONS SUGGESTED BY PARTICIPANTS

#	Code
5	Better Documentation and Learning Materials
4	Integration with Mainstream Languages and Tools
4	Improved Usability
3	Better Error Messages and Feedback
3	Improved Automation
3	Increased Awareness and Education
3	Better Debugging and Profiling Tools
3	Industry Demand and Investment
2	Easier Installation and Better Tool Support
2	Use of Common Specification Methodologies
2	Skepticism about Practical Use
2	Reusable Blocks
1	Exposing Internal Proof State

From our analysis, counter-examples do not always seem helpful and their use was even discouraged in some cases; 2) Develop official tutorials on common verification exercises (e.g., Fibonacci sequence, greatest common divisor). Neither Dafny nor Frama-C provides guided tutorials on their official pages; 3) Implement a search feature in the documentation. Neither Dafny nor Frama-C include a search feature, and Frama-C's documentation is only available in PDF format. Dafny's online documentation lacks a persistent table of contents, unlike popular tools like Jest,[10] and Pytest;[11] 4) Expand the themes covered in the documentation based on the gaps identified in our analysis; and 5) Promote VA languages in mandatory courses, as these languages often appear only in optional courses, unlike testing, which is frequently taught.

The open-ended responses complemented these results with additional, user-driven suggestions not previously captured. In this subsection, we present both the quantitative results and the qualitative themes that emerged from participants' feedback.

Closed-ended Questions. The first improvement addresses the inclusion of user-readable counter-examples that could allow users to "debug" the proofs. Most participants (83.9%) agreed on its importance. The suggestion to improve tutorials with common projects was also well-received by the majority of participants (64.5%). Improving documentation search mechanisms saw a notable divide, with beginners (66.7%) placing more emphasis on this improvement than experienced developers (15.4%), and this difference was statistically significant ($p = 0.02822$). Expanding documentation themes received mixed responses, with 41.9% considering it very important. Finally, including VA languages in undergraduate programs was considered highly impactful by most participants (50.1%).

Open-ended Questions. We identified 13 categories of improvement for VA languages (see Table V). The most frequently mentioned area was **improvements in documentation and learning materials**, suggested by five participants. These requests included better tutorials, guides, and documentation to assist both newcomers and experienced developers in using VA languages effectively. For example, one beginner developer, P28, noted the need for "more documentation on coming up with invariants". Four participants mentioned **improved usability**, and three better **education**. When facing errors, two beginners and one experienced developer mentioned that **better debugging** would help them understand why the verification fails. P23 mentioned directly that "To be really useful in practice, a verification-aware language needs to be familiar, easy to write proofs in, predictable in terms of what can be verified, well-integrated with common tools, interoperable with other languages, and able to generate efficient code." (P23). This aligns with the broader theme of **integration with mainstream languages**, mentioned by four participants.

Another suggestion by two participants is related to **reusable blocks**, P19 proposed that tools should "provide more automation to help 'easy tasks', so the programmer/prover can focus on complex tasks", and P26, mentioned directly they would like to see "reusable bricks for common verification patterns". Three participants advocated for increased **investment in verification tools**, suggesting that funding and institutional support are necessary for widespread adoption. While the majority of participants maintained a positive outlook, **skepticism** regarding practical adoption was expressed by two participants. For instance, P13 remarked, "verification is expensive, and tends not to produce immediate value, something which corporate doesn't like very much."

RQ3: Our findings point to four key improvements for addressing challenges in VA languages: better debugging tools; richer educational resources like tutorials and improved documentation; greater automation of routine verification tasks; and tighter integration with mainstream development tools.

VI. RELATED WORK

To the best of our knowledge, this is the first work to address challenges faced by developers using VA languages. In this section, we focus on related work on analysis and topic modeling of developer discussions, and on studies on challenges faced when using different programming languages.

Developer Discussion Analysis and Topic Modeling. Prior studies have applied topic modeling to analyse developer discussions on platforms like Stack Overflow and GitHub, focusing on areas different from VA languages [10]. Barua et al. [8] analysed posts from a Stack Overflow data dump to understand the main general discussion topics on the platform. Bagherzadeh and Khatchadourian [4] used the official Stack Overflow data dump to study the topics of questions about big data. Han et al. [25] studied what issues deep learning framework practitioners discuss, gathering data from Stack Overflow and GitHub. Abdellatif et al. [1] used the official Stack Overflow data dump to identify the topics chatbot developers were asking about. They analysed heuristics to discover relevant tags, using only post titles for training data, claiming it was representative and reduced noise. El aoun et al. [18]

[10]https://jestjs.io/docs/getting-started [last access 16th July 2024]

[11]https://docs.pytest.org/en/8.2.x/contents.html [last access 16th July 2024]

investigated challenges in quantum software engineering using data from GitHub and the Stack Exchange Data Explorer (for Stack Overflow). Al Alamin et al. [3] conducted an empirical study of developer discussions on low-code software development challenges. Peruma et al. [40] used the SOTorrent dataset to study the topics around Software Refactoring being asked by developers. These studies demonstrate a well-established methodology for using topic modeling, primarily on online forum data, to study developer discussions across various software engineering areas. Our work builds on these studies and applies them to the specific context of VA languages.

Programming Language Challenges. Other studies have examined the challenges developers face when using programming languages, suggesting that even experienced developers struggle to learn new programming languages [44]. Ko et al. [31] identified six learning barriers in programming systems, which include understanding, design, use, and information barriers. These barriers align closely with the challenges reported by VA language users in our study, such as difficulty interpreting tool feedback and expressing proof strategies. Other studies focused on particular languages. For instance, Zhu et al. [51] focused on Rust. They analyzed 15,000 Stack Overflow posts and surveyed 101 Rust developers to find that Rust safety features impose a steep learning curve and that compiler messages often lack actionable information. This is in line with other work that identified onboarding and usability issues as barriers to the adoption of languages [22], [29]. Chakraborty et al. [11] examined the support ecosystem for newer languages such as Go, Swift, and Rust by analyzing both Stack Overflow and GitHub data. They identified gaps in documentation, which often drove developers to rely on community Q&A sites. Finally, Christakis and Bird [12] studied developer expectations regarding program analysis tools by surveying engineers at Microsoft. They found that high false-positive rates, poor tool integration, and unclear feedback were the main reasons developers rejected static analysis tools in practice. These findings echo the feedback-related problems reported by VA language users in our study and suggest that if tools fail to align with developers' workflows or generate opaque error messages, they are unlikely to be adopted.

VII. Conclusion

This work analyzes the challenges developers face with VA languages, incorporating feedback from both beginners and experienced users. We trained an ML model on 1,420 questions, with 145 analyzed post-clustering. From this, 24 distinct challenges were identified and rated by 31 study participants. Additionally, five improvement suggestions were evaluated, and 12 more were uncovered via open-ended responses.

Our findings suggest that VA language adoption is primarily blocked by lack of documentation, and usability and ecosystem limitations, highlighting key areas for improvement. For example, a challenge across all experience levels lies in designing and completing proof strategies, underscoring the need for better tool support and guidance. Additionally, the mismatch found between online discussions and reported challenges (e.g., frame specifications and proof termination clauses, were identified in Stack Overflow, but not confirmed as challenges), points to the importance of empirical user studies, as research based solely on online discussions may not fully capture the real experience of practitioners (or vice-versa).

This work can pave the way for future research on challenges and improvements, while also offering valuable insights to guide the design and/or enhancement of VA languages to support broader adoption. Future work includes expanding the dataset with further sources and carry out a large-size user study. In addition, newly identified challenges should be ranked based on relevance, in order to direct development towards support that can have a greater impact.

Threats to Validity. For *topic modeling*, using question bodies provides more context but also introduces noise from off-topic discussions, error messages, and troubleshooting attempts, which may obscure the actual challenges. We mitigated this by reviewing both topic keywords and question bodies during classification. Additionally, Stack Overflow may lack sufficient coverage for some tools, limiting the generalizability of our results. Our mapping between languages and questions may also have missed relevant posts, reducing completeness. In the *survey*, the number of participants limited the ability to detect differences between experience levels. Also, the translation of challenges into affirmations may have lost important context, affecting participants' understanding. We mitigated this by providing open-ended questions where participants could provide their opinions.

Acknowledgments

Alexandra Mendes was supported by an Amazon Research Award, Fall 2024. Carolina Carreira was financed by National Funds through the FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the project VeriFixer, with reference 2023.15557.PEX (DOI: 10.54499/2023.15557.PEX), and by funding from FCT under grant PRT/BD/153739/2021. We thank João F. Ferreira for his valuable comments on this work.

References

[1] Ahmad Abdellatif, Diego Costa, Khaled Badran, Rabe Abdalkareem, and Emad Shihab. Challenges in Chatbot Development: A Study of Stack Overflow Posts. In *MSR'20*, pages 174–185. ACM, 2020.

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. Deductive software verification - the KeY book. *Lecture notes in computer science*, 10001, 2016.

[3] Md Abdullah Al Alamin, Sanjay Malakar, Gias Uddin, Sadia Afroz, Tameem Bin Haider, and Anindya Iqbal. An empirical study of developer discussions on low-code software development challenges. In *2021 IEEE/ACM 18th MSR*, pages 46–57, 2021.

[4] Mehdi Bagherzadeh and Raffi Khatchadourian. Going big: a large-scale study on what big data developers ask. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 432–442, New York, NY, USA, August 2019. ACM.

[5] Sebastian Baltes, Lorik Dumani, Christoph Treude, and Stephan Diehl. Sotorrent: reconstructing and analyzing the evolution of stack overflow posts. In *MSR*, MSR '18, page 319–330, New York, NY, USA, 2018. Association for Computing Machinery.

[6] John Barnes. *SPARK: The Proven Approach to High Integrity Software*. Altran Praxis, 2012.

[7] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2004.

[8] Anton Barua, Stephen W. Thomas, and Ahmed E. Hassan. What are developers talking about? An analysis of topics and trends in Stack Overflow. *Empirical Software Engineering*, 19(3):619–654, June 2014.

[9] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[10] Carolina Carreira, Nuno Saavedra, Alexandra Mendes, and João F Ferreira. From "Worse is Better" to better: Lessons from a mixed methods study of ansible's challenges. *arXiv preprint arXiv:2504.08678*, 2025.

[11] Partha Chakraborty, Rifat Shahriyar, Anindya Iqbal, and Gias Uddin. How do developers discuss and support new programming languages in technical Q&A site? An empirical study of Go, Swift, and Rust in Stack Overflow. *Information and Software Technology*, 137:106603, 2021.

[12] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *31st IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 332–343, 2016.

[13] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[14] James C Corbett, Matthew B Dwyer, John Hatcliff, Shawn Laubach, Corina S Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. In *22nd ICSE*, pages 439–448, 2000.

[15] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. pages 233–247, 2012.

[16] Adel Djoudi, Martin Hána, and Nikolai Kosmatov. Formal Verification of a JavaCard Virtual Machine with Frama-C. In *Formal Methods*, pages 427–444. Springer International Publishing, 2021.

[17] Frank Dordowsky. An experimental Study using ACSL and Frama-C to formulate and verify Low-Level Requirements from a DO-178C compliant Avionics Project. *Electronic Proceedings in Theoretical Computer Science*, 187:28–41, 2015.

[18] Mohamed Raed El aoun, Heng Li, Foutse Khomh, and Moses Openja. Understanding quantum software engineering challenges an empirical study on stack exchange forums and github issues. In *2021 IEEE ICSME*, pages 343–354, 2021.

[19] Escher Technologies. Escher C verifier (ECV). https://web.archive.org/web/20250115124219/https://eschertech.com/products/ecv.php, 2017. Accessed via Internet Archive on January 15, 2025.

[20] Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where programs meet provers. In *Programming Languages and Systems: 22nd European Symposium on Programming 2013*, pages 125–128, 2013.

[21] David Fullerton. Stack Exchange Creative Commons data now hosted by the Internet Archive, 2014. [Online]; accessed 03 June 2024.

[22] Kelsey R Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L Mazurek. Benefits and drawbacks of adopting a secure programming language: Rust as a case study. In *17th Symposium on Usable Privacy and Security*, pages 597–616, 2021.

[23] Dilian Gurov, Christian Lidström, Mattias Nyberg, and Jonas Westman. Deductive Functional Verification of Safety-Critical Embedded C-Code: An Experience Report. In Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti, editors, *Critical Systems: Formal Methods and Automated Verification*, pages 3–18. Springer International Publishing, 2017.

[24] Matthias Güdemann. Online Teaching of Verification of C Programs in Applied Computer Science. In João F. Ferreira, Alexandra Mendes, and Claudio Menghi, editors, *Formal Methods Teaching*, 2021.

[25] Junxiao Han, Emad Shihab, Zhiyuan Wan, Shuiguang Deng, and Xin Xia. What do Programmers Discuss about Deep Learning Frameworks. *Empirical Software Engineering*, 25(4):2694–2747, July 2020.

[26] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2:366–381, 2000.

[27] Reiner Hähnle and Marieke Huisman. Deductive software verification: From pen-and-paper proofs to industrial tools. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10000:345 – 373, 2019.

[28] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA formal methods symposium*, pages 41–55. Springer, 2011.

[29] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th ICSE*, pages 672–681. IEEE, 2013.

[30] Alexander Knüppel, Thomas Thüm, Carsten Pardylla, and Ina Schaefer. Experience Report on Formally Verifying Parts of OpenJDK's API with KeY. *Electronic Proceedings in Theoretical Computer Science*, 284:53–70, 2018. Publisher: Open Publishing Association.

[31] Amy J Ko, Brad A Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*, pages 199–206. IEEE, 2004.

[32] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jay Lorch, Oded Padon, and Bryan Parno. Verus: A practical foundation for systems verification. In *SOSP*, pages 438–454, November 2024.

[33] Gary T Leavens, Albert L Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[34] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010.

[35] K. Rustan M. Leino, David R. Cok, and Dafny contributors. Dafny Reference Manual, 2024. [Online]; accessed 06th January 2025.

[36] Bertrand Meyer. The Eiffel programming language. *http://www. eiffel. com*, 1992.

[37] Mattias Nyberg, Dilian Gurov, Christian Lidström, Andreas Rasmusson, and Jonas Westman. Formal verification in automotive industry: Enablers and obstacles. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11247 LNCS:139 – 158, 2018.

[38] David L. Parnas, A. John van Schouwen, and Shu Po Kwan. Evaluation of safety-critical software. *Commun. ACM*, 33(6):636–648, 1990.

[39] David J Pearce and Lindsay Groves. Whiley: a platform for research in software verification. In *SLE'13*, pages 238–248. Springer, 2013.

[40] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D. Newman, Mohamed Wiem Mkaouer, and Ali Ouni. How do I refactor this? An empirical study on refactoring trends and topics in Stack Overflow. *Empirical Software Engineering*, 27(1), January 2022.

[41] Michael Röder, Andreas Both, and Alexander Hinneburg. Exploring the Space of Topic Coherence Measures. In *Proc. of WSDM'15*, pages 399–408. ACM, 2015.

[42] Marc Schoolderman, Sjaak Smetsers, and Marko Van Eekelen. Is Deductive Program Verification Mature Enough to be Taught to Software Engineers? In *Proceedings of the 8th Computer Science Education Research Conference*, pages 50–57. ACM, 2019.

[43] Yuvaraj Selvaraj, Wolfgang Ahrendt, and Martin Fabian. Verification of Decision Making Software in an Autonomous Vehicle: An Industrial Case Study. In *Formal Methods for Industrial Critical Systems: 24th International Conference*, pages 143–159. Springer-Verlag, 2019.

[44] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. Here we go again: Why is it difficult for developers to learn another programming language? In *ICSE*, pages 691–701, 2020.

[45] Julia Silge and David Robinson. Text Mining with R: A Tidy Approach, 2024. [Online]; accessed 30 May 2024.

[46] Donna Spencer. *Card Sorting: Designing Usable Categories*. Rosenfeld Media, 2009.

[47] Vassil Todorov, Frederic Boulanger, and Safouan Taha. Formal verification of automotive embedded software. In *6th Conference on Formal Methods in Software Engineering (FormaliSE '18)*, page 84 – 87, 2018.

[48] Ike Vayansky and Sathish AP Kumar. A review of topic modeling methods. *Information Systems*, 94:101582, 2020.

[49] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proc. of SPLASH'12*, pages 121–140, 2012.

[50] Neil White, Stuart Matthews, and Roderick Chapman. Formal verification: Will the seedling ever flower? *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104), 2017.

[51] Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, and Linhai Song. Learning and programming challenges of Rust: A mixed-methods study. In *ICSE*, pages 1269–1281, 2022.