FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Identifying Challenges and Practitioner Techniques in the Use of Verification-Aware Programming Languages

Sofia Sousa Moura

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Alexandra Mendes

June 30, 2026

Identifying Challenges and Practitioner Techniques in the Use of Verification-Aware Programming Languages

Sofia Sousa Moura

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. João Pascoal Faria
Examiner: Prof. José Creissac Campos
Supervisor: Prof. Alexandra Mendes

June 30, 2026

Resumo

Garantir a fiabilidade do software tornou-se uma preocupação crescente à medida que os sistemas de software aumentam de complexidade e expandem-se para domínios de sistemas críticos, tais como a saúde, as finanças e a mobilidade. Entre as técnicas desenvolvidas para enfrentar este desafio, a Verificação Formal oferece garantias matemáticas de correção que vão além daquilo que os testes tradicionais conseguem fazer. As linguagens de programação *Verification-Aware* integram a verificação formal no fluxo de trabalho do desenvolvimento de software, permitindo que os programadores definam especificações em conjunto com o código de implementação, o que possibilita a deteção precoce de erros e garantias de correção mais robustas. Este trabalho centra-se especificamente nas fases de especificação e implementação do desenvolvimento de software, nas quais os programadores escrevem e verificam especificações formais em paralelo com o código, em vez de se focar nas atividades iniciais de planeamento ou definição de requisitos.

Apesar destas vantagens, a adoção de linguagens de verificação continua a ser limitada. Os profissionais enfrentam grandes desafios ao trabalhar com estas ferramentas, incluindo a dificuldade em construir e fazer o *debugging* de provas, interpretar mensagens de erro pouco informativas e gerir a sobrecarga cognitiva do raciocínio formal. Embora investigações anteriores tenham começado a documentar estes obstáculos, as estratégias que os programadores utilizam para os ultrapassar não foram ainda exploradas.

Este trabalho aborda esta lacuna através da recolha e análise de conteúdos multimédia nos quais os utilizadores interagem diretamente com linguagens *Verification-Aware* enquanto resolvem problemas de verificação. Ao contrário de estudos anteriores baseados em discussões de texto, este trabalho capta os processos de raciocínio e as práticas dos programadores à medida que surgem. Utilizando um *pipeline* que combina *semantic similarity filtering* e *topic modeling*, extraímos uma taxonomia de técnicas de verificação, organizada em categorias temáticas.

O conjunto de técnicas resultante foi validado através de um questionário que incluiu participantes com experiência em verificação formal, avaliando a sua frequência, complexidade e aplicabilidade.

Os resultados da validação revelam práticas comuns de verificação e as técnicas mais frequentemente adotadas pelos programadores, identificando áreas onde continuam a sentir dificuldades significativas. Estes resultados fornecem evidência sobre o *workflow* de verificação na prática e sugerem perspetivas que têm o potencial de apoiar o design de ferramentas mais utilizáveis, recursos educativos e investigação futura em verificação formal.

Palavras-chave: Linguagens de Programação Orientadas à Verificação • Métodos Formais • Experiência do Desenvolvedor • Modelagem de Tópicos • Engenharia de Software

Abstract

Ensuring software reliability is becoming a greater concern as software systems grow in complexity and expand into safety-critical domains, such as healthcare, finance, and transportation. Among the techniques developed to address this challenge, Formal Verification offers mathematical guarantees of correctness that go further than what traditional testing can provide. Verification-Aware programming languages bring formal verification into software development workflows by allowing programmers to define specifications along with the implementation code, enabling early detection of errors and stronger correctness assurances. This work focuses specifically on the specification and implementation stages of development, where developers write and prove formal specifications alongside code, rather than earlier planning or requirements activities.

Despite these advantages, the adoption of Verification-Aware (VA) languages remains limited. Practitioners face significant challenges when working with these tools, including the difficulty of constructing and debugging proofs, interpreting uninformative error messages, and managing the cognitive overhead of formal reasoning. While prior research has begun to document these obstacles, the strategies that developers employ to overcome them are still largely unexplored.

This work addresses this gap, by collecting and analyzing multimedia content in which practitioners interact directly with verification-aware languages while solving verification problems. Unlike previous studies based on text discussions, this work captures developers' reasoning processes and practices as they come up. Using a pipeline that combines semantic similarity filtering and topic modeling, we extract a taxonomy of verification techniques, organized into thematic categories.

The resulting set of techniques was validated through a survey including participants with experience in formal verification, evaluating their frequency, perceived complexity, and applicability.

The validation results reveal common verification practices and the techniques most frequently adopted by practitioners, identifying areas where developers continue to experience significant difficulty. These results provide evidence about the verification workflow in practice and suggest insights that have the potential to support the design of more usable tools, educational resources, and future research on formal verification.

Keywords: Verification-Aware Programming Languages • Formal Methods • Developer Experience • Topic Modeling • Software Engineering

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework for achieving a better and more sustainable future for all. It includes 17 goals (https://sdgs.un.org/goals) to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice. In software engineering, failures in critical domains can lead to significant economic losses and life-threatening situations. Considering this, to demonstrate the relevance of formal methods in stability and the research on challenges and strategies in Verification-Aware programming languages, this section focuses on the SDGs to which the dissertation contributes most directly, considering UNESCO's vision of engineering for sustainable development: **SDG 4 (Quality Education)**, **SDG 8 (Decent Work and Economic Growth)** and **SDG 9 (Industry, Innovation and Infrastructure)**.

SDG 4 Ensure inclusive and equitable quality education and promote lifelong learning opportunities for all

SDG 8 Promote sustained, inclusive and sustainable economic growth, full and productive employment and decent work for all

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

SDG	Target	Contribution	Performance Indicators and Metrics
4	4.4	Identification of practical techniques in Verification-Aware languages, contributing to advanced technical education in the area.	Qualitative assessment of time reduction in debugging verification based on identified strategies.
8	8.2	Analysis of practitioner practices supports productivity improvements by reducing cognitive effort and time required for proof validation.	Qualitative assessment of time reduction in debugging verification based on identified strategies.
9	9.1	Proposed improvements contribute to the development of reliable software infrastructure in sectors like healthcare and mobility.	Number of identified usability barriers and insights for tool evolution.
	9.5	Research on developer techniques advances scientific knowledge in formal verification and safety-critical systems.	Number of research outputs produced. Validation of results through the proposed survey.

Acknowledgements

This dissertation is the result of years of hard work and personal growth, and reaching this milestone would not have been possible without the support of many people who have been by my side along the way. First, I would like to express my gratitude to my supervisor, Professor Alexandra Mendes, for her guidance, encouragement, and continuous support throughout this entire academic year. I also express my thanks to Carolina Carreira for sharing her expertise and for her willingness to always make time to help. Their knowledge and constructive feedback were fundamental in the development of this work and my research skills.

I would like to extend my thanks to the people in my personal life who have supported me throughout this entire journey and made this achievement possible. To my parents, to whom I owe my deepest gratitude, for providing me with the education and support that allowed me to reach the position I am in today, for constantly worrying about my future and for being the voice of wisdom I always turn to. To my sister, who has been my role model for as long as I can remember and whose strength has always inspired me to become a better version of myself, for being my ultimate reassurance in every single step I take.

To Gonçalo, for believing in me during the moments when I doubted myself the most, for knowing exactly when I needed encouragement or simply a reason to take my mind off things, and especially for always being the person I can turn to after a long day.

To my friends, Leonor and Teresa, for being a constant presence and support throughout all these years, even without understanding a word of what I was doing, they were always there with some kind of advice and much-needed moments away from work. To Eva, for always being one of my biggest supporters, celebrating my successes as if they were her own. To Maria, for sharing and navigating this challenging chapter with me, no matter how many miles separate us. To Miguel, for all the deep talks, for his constant support, and for always pushing me to step away from my desk for even just a coffee.

I could not forget to mention BEST Porto, a community that shaped my academic journey in ways I never expected. It helped me grow as a student, step outside my comfort zone, and become the person I am today, and for that, I will always be grateful.

Thank you to all of you for everything.

This work was partially funded 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 (https://doi.org/10.54499/2023.15557.PEX) and by an Amazon Research Award, Fall 2024.

fct Fundação para a Ciência e a Tecnologia

Sofia

Declaration of honour

I declare that this dissertation is my original work and has not previously been submitted or assessed in any other course or curricular unit, whether at this or any other institution. References to other authors (statements, ideas, thoughts), as well as the use of published works fully complies with the rules of attribution and are appropriately cited in the text and bibliography, in accordance with reference standards. I am aware that plagiarism and self-plagiarism constitute academic misconduct.
I further declare that I have not used Artificial Intelligence tools to produce any part(s) of this dissertation or, when such tools were used, their use and results are clearly indicated and were carefully reviewed for the detection of errors, inaccuracies, unfounded attributions, or other forms of content and argument bias, as well as for the determination of authorship.
I further declare that I have included studies carried out with multiple co-authors, having specified, with transparency and rigour, the contribution of each co-author, provided information on the possible existence of any other dissertation in which the same article has been included, reproduced the work in full with the permission of the journal in which it was published, and obtained the explicit written consent of all co-authors for the inclusion of the article.

Sofia Sousa Moura

"This for everybody going through tough times
Believe me, been there, done that
But every day above ground is a great day,
remember that"

Armando Christian Pérez

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface

AI Artificial Intelligence

ANOVA Analysis of Variance

ASR Automatic Speech Recognition

AWS Amazon Web Services

CI/CD Continuous Integration and Continuous Deployment

CLMM Cumulative Link Mixed Model

IDE Integrated Development Environment

IoT Internet of Things

ITP Interactive Theorem Prover

JML Java Modeling Language

LDA Latent Dirichlet Allocation

LLM Large Language Model

PRISMA Preferred Reporting Items for Systematic Reviews and Meta-Analyses

SMT Satisfiability Modulo Theories

SDG Sustainable Development Goal

TAM Technology Acceptance Model

VA Verification-Aware

VCC Verifier for Concurrent C

Chapter 1

Introduction

1.1 Context

Software plays a central role in modern society, where it powers the vast majority of sectors. From finance, healthcare, transportation, and communications, much of everyday activity is dependent on reliable digital solutions to operate, solve problems, optimize complex tasks, and drive innovation. As software systems grow in complexity and their use in critical areas increases, ensuring their correctness and reliability becomes a pressing priority. Failures and poor quality in such systems can lead to life-threatening situations, significant economic losses, and security breaches [20].

Among the methods used to enhance software reliability, formal methods stand out as an approach to developing software that is correct by construction, using rigorous specifications throughout the development process. These methods use mathematical models for specification, analysis, and verification [40], offering guarantees about software behavior that go beyond what is achievable with traditional testing. While testing explores only a finite set of execution scenarios, formal verification provides mathematical guarantees that a program satisfies its specification. For this reason, formal methods have gained attention and are promoted for use in critical domains such as aerospace, automotive, healthcare, and finance [23], where software failures can have severe consequences, including deaths, environmental damage, critical financial losses, and threats to public safety.

In software engineering practice, formal methods have often been seen as a post-development activity, especially for validation rather than early specification. For example, Mashkoor, Leuschel, and Egyed [24] observe that formal validation activities have been frequently postponed until the latest stages of development, meaning that requirement errors are often discovered near the end of the lifecycle rather than during early design or implementation. However, Verification-Aware (VA) programming languages have emerged as a means of helping to integrate formal methods more directly into everyday software development, by allowing specification and verification constructs to be expressed alongside implementation code. Languages such as Dafny [22], Verifier for Concurrent C (VCC) [7], Why3 [11], KeY [1], and Verus [21], integrate specification and verification

mechanisms directly into the development workflow, allowing programmers to detect errors early in the process, going beyond traditional testing.

Despite their promise, the effective use of VA languages in real-world development remains challenging, motivating increasing research efforts that examine their technical capabilities and developers' interactions with them in practice.

1.2 Motivation

Although verification-aware programming languages offer significant advantages by enabling early detection of errors and stronger correctness guarantees, their adoption in both academic and industrial contexts remains limited [23, 30].

The primary obstacles to adoption are no longer purely theoretical or algorithmic. Both beginners and experienced users face challenges that prevent their effective use. Common difficulties include the need to formulate correct proofs, the interpretation of uninformative error messages, and the complexity of integrating these tools into existing development workflows. These challenges result in a high level of expertise required from developers and a steep learning curve [30], reducing productivity and contributing to the low adoption of these languages [23].

Understanding the challenges faced by users and how they try to solve them is therefore crucial to support the use and development of more effective VA frameworks and to further strengthen the reliability of critical software. Developers actively exchange knowledge in online tutorials, where they discuss practical techniques and workarounds to overcome these challenges. While prior research has only begun to identify common difficulties, the strategies employed by practitioners to successfully work with verification-aware languages remain largely unexplored and insufficiently documented.

By examining common difficulties and the strategies practiced to overcome identified challenges in the use of VA languages, this study aims to gather actionable insights that can guide towards concrete solutions, aiming to promote their broader adoption.

1.3 Problem

Despite the strong guarantees that verification-aware programming languages provide, their adoption remains low.

Among the existing research, the work by Oliveira, Mendes, and Carreira [30] is the most related to our work, as it has started to analyze the challenges faced by users of verification-aware languages through techniques such as topic modeling applied to practitioner discussions on public platforms. However, a clear gap remains in the literature, since this work focuses on a single text-based data source and on identifying problems, without investigating the strategies developers use to overcome them. Online discussions about VA tools extend far beyond and are where experienced users often share workarounds and techniques to overcome their obstacles. Video

content represents a potentially valuable, yet still scarcely unexplored source of knowledge, as experienced users often document their approaches to solving work through verification challenges in real time and might share the techniques they have developed.

Our work aims to address these gaps by analyzing online video content and observing the problems users face as well as the techniques they use to overcome them. The insights gathered may be strong enough to provide a basis for identifying potential measures towards the development of more accessible verification-aware programming languages and supporting tools.

We will first analyse the existing literature to map out the challenges already documented in the topic. Building on this, we will focus on exploring video content in which practitioners interact directly with VA languages to discover the specific techniques users employ to prove software correctness and overcome obstacles. To the best of our knowledge, this methodology has not been used in the past in this context.

In this work, we focus specifically on the specification and implementation phase of the software development process, in which developers write formal specifications alongside the implementation code and attempt to formally verify their correctness.

1.4 Research Questions

Based on the identified research gap, this study addresses the following research questions:

RQ1: What challenges do practitioners face when using verification-aware languages in different contexts?

This question aims to categorize the barriers that constrain both novice and expert practitioners. While the literature identifies general difficulties, the goal of this study is to understand how these obstacles manifest across diverse industrial and academic environments.

RQ2: What strategies and practices do practitioners employ to overcome these challenges when working with VA frameworks?

As far as we are aware, strategies and practices used by developers to overcome challenges in the use of VA frameworks have not been studied in the past. By analyzing relevant online video content, we aim to identify reusable patterns for successful verification.

RQ3: What insights can be derived to improve the usability and adoption of VA programming languages and their supporting tools?

This final question aims to synthesize the findings identified in RQ1 and RQ2 into concrete recommendations for tool evolution.

The following chapters will try to address these research questions by analyzing existing research and practitioner experiences with verification-aware tools, focusing on the challenges faced in practice and the strategies used to overcome them. The insights obtained from this analysis will contribute to the identification of guidelines for improving verification-aware languages and supporting their adoption.

1.5 Dissertation Structure

This dissertation is organized into six chapters.

- **Chapter 1** presents the motivation, research problem, defines the research questions, and outlines the overall objectives of the dissertation.
- **Chapter 2** presents a systematic literature review, discussing the current state of the art, the challenges reported in previous research, and recent advances in developer experience, safety-critical applications, and Artificial Intelligence (AI)-assisted verification.
- **Chapter 3** describes the research methodology adopted in this work. It details the data collection process, the preprocessing pipeline, the semantic similarity filtering approach used to identify relevant transcript excerpts, and the topic modeling methodology designed to extract verification techniques.
- **Chapter 4** presents the results obtained from the analysis of practitioner videos. It introduces the taxonomy of verification techniques, analyzes their relationships through cooccurrence analyses, and compares the identified techniques across different verification paradigms.
- **Chapter 5** describes the validation methodology based on a practitioner survey. It presents the survey design, participant profile, and the analysis of the collected responses, assessing the frequency, perceived complexity, and applicability of the identified verification techniques.
- **Chapter 6** concludes the dissertation by answering the research questions, discussing threats to validity, and outlining directions for future research.

Chapter 2

Literature Review and State of the Art

2.1 Introduction

This chapter provides a comprehensive overview of the current landscape of research on challenges in the use of verification-aware programming languages, combining a systematic literature review with an analysis of the state of the art. We focus on work that discusses the practical use of these languages, identifying the obstacles encountered by users and the strategies proposed to overcome them.

The first part of this chapter details the methodology adopted for this review, outlining a replicable process for assessing and including relevant studies according to the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) guidelines. Subsequently, we present the state of the art by synthesizing the findings from the selected literature, categorizing the main challenges and advancements in the field to provide a clear picture of the current research and industrial context.

2.2 Search String

The definition of the search string was driven by the research questions of this work, which served as the basis for identifying the key concepts and terms to be included in the search.

To build the search string, we expanded the core concepts *verification-aware*, *languages*, and *challenges* into several synonyms and related expressions and tested them to capture the diversity of expressions used in the literature and ensure a broad coverage.

In addition, specific verification-aware languages and tools were included in the query, as part of the studies discuss challenges within the scope of a particular language rather than using broader terminology. The list of languages was based on those referenced by the Formal Methods Europe industry committee, which serves as a curated reference of formal verification frameworks known to have practical usage in the industry. Throughout several refinement iterations, some of the languages were removed because they mostly returned results that were not related to the scope of this work. The languages and the reasons for their exclusion are:

- **Eiffel**: generated a high volume of irrelevant results due to strong association with unrelated content (e.g., *Eiffel Tower*)
- **Idris**: the articles mostly referred unrelated entities, including *Universiti Pendidikan Sultan Idris* and an open-source virtualization platform also named *IDRIS*. As a result, no article was found that specifically focused on the verification language.
- **KeY**: produced noise due to being a common English word, making it impractical to isolate relevant studies.
- **Java Modeling Language (JML)**: generated a dominant volume of papers referring to the *Joint Maximum Likelihood* method or focusing on the Java modeling theory without addressing practitioner challenges using VA languages.
- **P**: not feasible as a search term due to being a single character, producing an excessive number of irrelevant matches.
- **Spec#** and **TLA+**: database engines systematically interpreted these languages as the substrings *spec* and *tla*, retrieving papers unrelated to the intended verification languages.

More general terms such as *formal methods* were also avoided. Although these terms were conceptually related to the topic, they retrieved an excessive number of papers discussing verification in a general approach rather than the practical challenges encountered by developers when using VA programming languages.

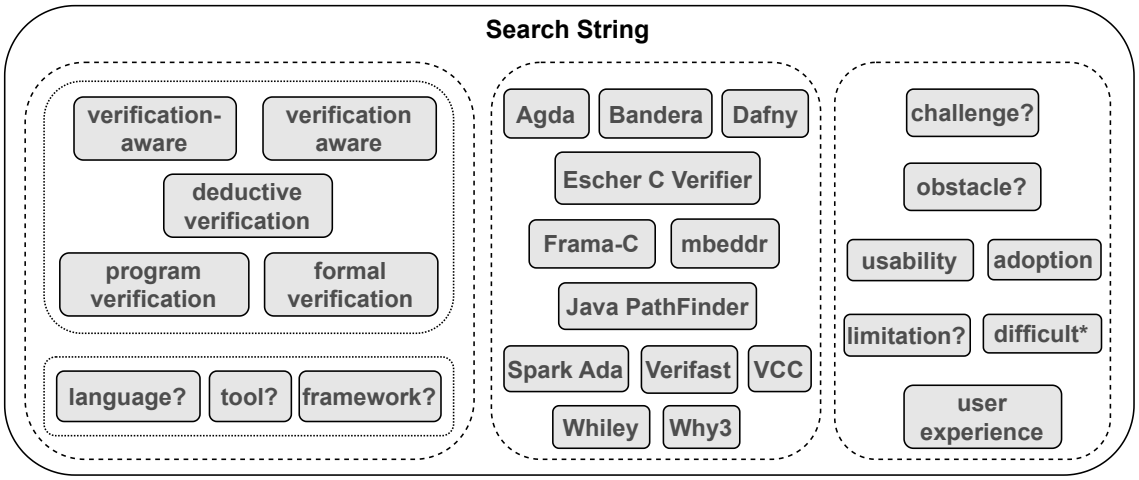
The final query contained three main concept groups combined using Boolean logic. The first group covers verification-aware programming languages and tools, including both general terms and specific language names. The second group targets software development artifacts, such as languages, tools, and frameworks, to restrict the search to practical programming contexts. The third group captures challenges related to usability, adoption, and developer experience, focusing on the difficulties practitioners face when using verification-aware languages.

A conjunction (Boolean `AND`) was applied between the groups to ensure that the retrieved papers included terms from all of them. Within each group, a disjunction (Boolean `OR`) was used to account for synonyms and related expressions. Additionally, wildcards (characters `?` and `*`) were also included to match any word ending, broadening the search. For instance, searching the term “language” captures variations such as “language” and “languages”.

The resulting search query, structured according to the description above, is illustrated in Figure 2.1 and was formulated as follows:

```
(((“verification-aware” OR “verification aware” OR “deductive
verification” OR “program verification” OR “formal verification”)
AND (language? OR tool? OR framework?)) OR Agda OR Bandera OR
Dafny OR “Escher C Verifier” OR “Frama-C” OR “Java PathFinder”
OR mbeddr OR “Spark Ada” OR VeriFast OR VCC OR Whiley OR Why3)
AND (challenge? OR obstacle? OR usability OR adoption OR
limitation? OR difficult* OR “user experience”)
```

Figure 2.1: Visual representation of the keywords included in the search string



2.3 Databases

Our criteria for choosing the databases was that we wanted sources that:

- could support complex Boolean queries, allowing the use of the search string defined in the previous section without limitations
- could provide filtering options to select specific types of publications
- contained peer-reviewed articles, ensuring the scientific rigor of the studies included in the review.

Taking these factors into account, and focusing only on databases either specifically related to engineering or providing broader coverage across several disciplines, the selected data sources were ACM Digital Library, IEEE Xplore, Scopus, and SpringerLink. Our search was conducted in the abstract of the publication, as this is the only common field supported by all data sources, which nonetheless effectively reflects the core topics of the papers.

Table 2.1: Scientific research data sources consulted.

Source	Access Date
ACM Digital Library	Nov 4, 2025
IEEE Xplore	Nov 4, 2025
Scopus	Nov 4, 2025
SpringerLink	Nov 4, 2025

2.3.1 Inclusion Criteria

The following inclusion criteria were applied to select relevant and up to date studies:

- Published within the last five years
- Written in English
- Peer-reviewed journal article or conference paper

However, two highly relevant papers by Oliveira, Mendes, and Carreira [30] and Brugger, Denis, and Müller [5] were included from other sources, as they were not retrieved by the search string. Both articles address key aspects directly related to our investigation and help capture the most recent developments in this research area.

2.4 Screening

After retrieving the initial set of publications from each selected data source, a screening phase was carried out to ensure that only relevant papers were retained. This process involved removing duplicates, examining titles, and abstracts.

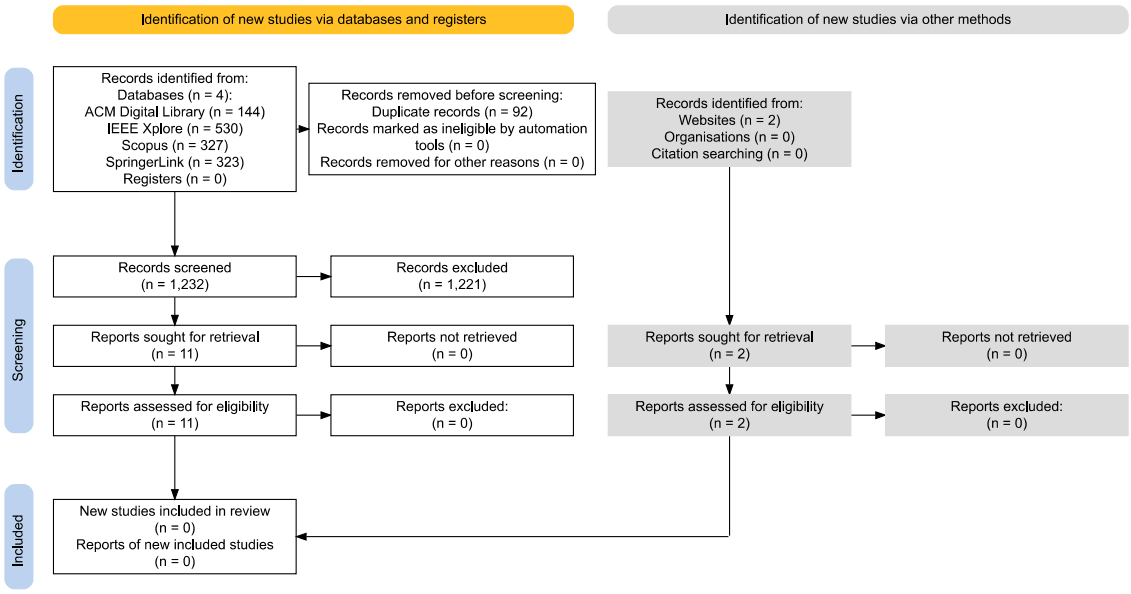
The inclusion criteria for this phase were kept simple by only considering articles related to the objectives of this work. As for the exclusion criteria, we created a set designed to exclude irrelevant studies from title and abstract inspection, followed by full-text screening if necessary.

2.4.1 Exclusion Criteria

- The title explicitly refers to a domain that is unrelated to software verification or formal methods.
- The title does not focus on verification-aware programming languages or model checking and SMT-based tools.
- The abstract focuses solely on the technical development of a verification approach and does not mention any analysis of the difficulties experienced by users.
- The study does not analyze usability issues or general practitioner experience.

After using the search string in each data source, the search returned 1324 records, of which 1232 remained after duplicate removal. Following the title and abstract screening phases, 11 studies were considered relevant and 2 were added through other sources, totaling 13 identified studies, which will be analyzed in the following section. The complete screening and selection process is summarized in Figure 2.2.

Figure 2.2: PRISMA 2020 Flow Diagram illustrating record screening and selection



2.5 Content Analysis and Main Findings

As part of the literature review, we conducted a content-driven analysis of the selected papers. Although formal methods have been an active research area for many decades, the study of the practical challenges hindering their adoption remains relatively limited [23]. We found 13 relevant papers and categorized them into 3 main clusters that emerged from their similarities in focus and contributions to VA programming languages challenges. Table 2.2 describes each cluster and summarizes its main focus.

Table 2.2: Included studies.

ID	Title	Author	Year
S01	A manifesto for applicable formal methods	Gleirscher, Pol, and Woodcock [12]	2023
S02	A Study on Formal Verification of Smart Contracts in Distributed Ledger Technology	Gokulnath and Kizhakkethottam [13]	2023
S03	Formal methods to comply with rules of the road in autonomous driving: State of the art and grand challenges	Mehdipour et al. [25]	2023
S04	From Scientific Texts to Verifiable Code: Automating the Process with Transformers	Wang, Scazzariello, and Chiesa [38]	2025
S05	Laurel: Unblocking Automated Verification with Large Language Models	Mugnier et al. [27]	2025

ID	Title	Author	Year
S06	Leveraging Large Language Models to Boost Dafny's Developers Productivity	Silva, Mendes, and Ferreira [34]	2024
S07	On the Impact of Formal Verification on Software Development	Mugnier et al. [28]	2025
S08	Pinpointing the Learning Obstacles of an Interactive Theorem Prover	Juhošová, Zaidman, and Cockx [18]	2025
S09	Seamless Interactive Program Verification	Grebing, Klamroth, and Ulbrich [14]	2020
S10	Systematic Evaluation and Usability Analysis of Formal Methods Tools for Railway Signaling System Design	Ferrari et al. [10]	2022
S11	Toward Practical Deductive Verification: Insights from a Qualitative Survey in Industry and Academia	Brugger, Denis, and Müller [5]	2025
S12	Toward Secure and Reliable IoT Systems: A Comprehensive Review of Formal Methods Applications	Haddou-Oumouloud et al. [16]	2024
S13	What Challenges Do Developers Face When Using Verification-Aware Programming Languages?	Oliveira, Mendes, and Carreira [30]	2025

Table 2.3: Article clusters, their main focus, and associated studies

Name	Summary	Studies
Software Engineering and Developer Experience	How developers interact with verification tools, the usability challenges and learning barriers they face, and the impact of verification on productivity. The focus is on understanding why verification is hard for practitioners and which factors hinder or promote adoption.	S01, S07, S08, S09, S11, S13
Formal Methods in Safety-Critical Domains	Challenges of applying formal methods in safety-critical systems, specifically in reliability requirements, scalability, integration with industrial development workflows, and practical obstacles faced when adopting formal verification in real-world environments.	S12, S03, S10, S12

Name	Summary	Studies
Automated Reasoning and AI-assisted Verification	Technical advances aimed to improve the automation of verification tools. Works explore techniques from automated reasoning and AI-based assistance to reduce proof effort, support invariant generation, improve solver performance, or guide developers during verification.	S04, S05, S06

This distribution shows that the majority of recent research in the field is concerned with understanding how developers interact with verification-aware languages and which factors influence usability and adoption. This prominence is highly relevant to our work, which aims to identify the challenges practitioners face (RQ1) and the strategies they use to overcome them (RQ2). The remaining clusters complement this perspective with emerging automation techniques and domain-specific constraints in critical contexts, where formal verification should be predominant [10, 23]. The three clusters identified above are explored in detail in the following subsections.

2.5.1 Software Engineering and Developer Experience

The six studies included in this cluster analyze how developers interact with verification-aware tools, with particular focus on their usability, integration, learning curve, and overall user experience. Although each of them focuses on different contexts, ranging from industrial deployments of verification-aware languages to cognitive analyses of novice proof developers, the result is a common outcome scenario: VA technologies bring unique engineering benefits but also introduce significant challenges that shape developer experience and must be addressed. The literature suggests that the main challenges are not only theoretical but stem from a lack of *applicability* and maturity in the developer experience.

2.5.1.1 Usability and Learning Curve

A frequent aspect mentioned in recent studies is the **steep learning curve** associated with VA languages, which requires a significant cognitive shift from traditional programming. In a study involving students learning the Agda [29] Interactive Theorem Prover (ITP), Juhošová, Zaidman, and Cockx [18] mention that the new concepts and paradigms (e.g. dependent types) are not intuitive and the way that users need to think about the program is much more abstract, often requiring users to invest in more advanced knowledge.

This barrier is manifested in the **difficulty of developing a proof strategy**, which Oliveira, Mendes, and Carreira [30] identify as one of the most significant challenges faced by both novice and experienced users, based on their analysis of online discussions and survey data. Mugnier et al. [28] even establish a distinction between developers with a formal methods background, who tend to acquire knowledge in real time as they interact with evolving systems, and those with a

software engineering background, who complain about the steep learning curve and struggle to justify the extreme effort required to use a verifier.

Mugnier et al. [28] study reveals that users' difficulties lie not only in completing a proof strategy but, more fundamentally, in designing a valid strategy from the very beginning. Their analysis of Stack Overflow[1] discussions show a tendency for users to select non-ideal strategies, such as proof by induction when proof by contradiction would be more appropriate, leading to avoidable complexity and time waste. The study shows that users are not able to gather feedback from VA tools, due to a lack of necessary guidance to assist users in the phase of proof construction. In an industrial context, Brugger, Denis, and Müller [5] also manifests that developers find the process of structuring a proof as one of the most challenging aspects of verification due to the amount of detail to be considered. They state that **scalability**, as in verifying a single method or isolating a block of code within a method, could have a crucial role if splitting up a proof was not so manual and high expertise requiring. Mugnier et al. [28] indicate that proof complexity often snowballs as developers prove higher-level properties, making the proof-contexts ever larger and obligations harder to discharge.

One might expect that **automation** in VA languages would mitigate these barriers. While automation reduces manual effort, it complicates debugging when verification fails and Brugger, Denis, and Müller [5] report that developers describe automation as "both a blessing and curse", as it obscures the internal state of the verifier, depriving the user of fine-grained control. Mugnier et al. [28]' work shows that the challenge is to make the verifier's context more explicit, creating an interactive environment where one could assess the state of the solver. Automated verifiers act as opaque systems, making users struggle to build a mental model of the implicit context to provide the correct hints, a process that is often non-intuitive accompanied by **unclear error messages**. This lack of transparency creates an ambiguity where the developer cannot easily determine whether the error stems from an actual invalid property in the code or solely from the solver's inability to verify a valid property.

2.5.1.2 Tool Maturity and Interaction Models

The low adoption of VA languages is also frequently attributed to the lack of maturity of the supporting ecosystem, meaning the auxiliary tools and infrastructure are underdeveloped. Gleirscher, Pol, and Woodcock [12] compile the requirements they found crucial in their manifesto, arguing that for a method to be widely adopted, it must satisfy principles like **integration**, **explainability**, and **ease of use**. They link these factors directly to the Technology Acceptance Model (TAM), suggesting that if a tool is not perceived as useful and easy to use, it will be rejected. Current tools often violate these principles and Juhošová, Zaidman, and Cockx [18] found obstacles for learners in VA ecosystems, pointing to error-prone and incomplete tools, incomplete supporting material and frustrating installation processes. Oliveira, Mendes, and Carreira [30] complement the description of the environment setup with the mention of dependency installation process, alerting

[1] https://stackoverflow.com

for missing or not found dependencies and incompatible versions. Furthermore, Juhošová, Zaidman, and Cockx [18] note that the language design itself exacerbates these issues by forcing users to rely on custom and often unstable IDE plugins instead of robust development environments.

The **lack of explainability and ease of use** raised a focus shift from automation to enhanced interaction and visualization, as manifested by practitioners in the study by Brugger, Denis, and Müller [5]. Grebing, Klamroth, and Ulbrich [14] proposed a *Seamless Interaction Concept*, where users can access solver contexts, inspect proof states, and interactively guide proof automation. Their concept is based on synchronized views that allow the user to interact with different levels of abstraction without losing the connection to the original code, aiming to make the relation between the code and the proof artifacts explicit and addressing the fundamental challenge of tool explainability.

2.5.1.3 Integration into the Software Development Workflow

The adoption of verification-aware languages needs imperative changes to the standard software development lifecycle. Contrary to the theoretical ideal of "correct-by-construction" software replacing traditional Quality Assurance (QA), practitioners overwhelmingly view verification as a complementary assurance method rather than a total replacement for testing. Juhošová, Zaidman, and Cockx [18] report that new users often struggle to perceive the practical relevance of tools because they fail to understand this supplementary role, viewing them as isolated theoretical exercises rather than integrated engineering tools.

Mugnier et al. [28] argue that VA languages introduce two phases to the standard engineering workflow, which are not anticipated by new developers.

- **Proof Debugging**: actively probing the verifier's opaque state using probe assertions to understand why a property that appears valid is being rejected by the solver.
- **Proof Hardening**: addresses proof brittleness, an occurrence where valid proofs break due to minor code changes and require major refactoring and visiting steps thought to be done. Developers must *harden* proofs by optimizing context visibility and managing resource limits to ensure long-term stability in the Continuous Integration and Continuous Deployment (CI/CD) pipeline .

Brugger, Denis, and Müller [5] confirms that developers are often forced to verify only non-evolving code or stable libraries, effectively contradicting the agile principle of iterative development. They further emphasize the lack of mature proof repair techniques, which currently forces developers to manually reconstruct proofs after refactoring, a process described as tedious and counterproductive.

2.5.1.4 Cost-Benefit Analysis

The decision to adopt verification is heavily influenced by the cost-benefit analysis. Gleirscher, Pol, and Woodcock [12] states that a method is only applicable if the benefit from using it in a task

justifies the cost efforts of applying it. However, Brugger, Denis, and Müller [5] found that for many practitioners, verification still fails to pass this cost-benefit relation outside of safety-critical domains.

2.5.2 Formal Methods in Safety-Critical Domains

Safety-critical systems are defined by the unacceptability of failure, where software errors can lead to loss of life, significant economic loss, or critical infrastructure failure. Gokulnath and Kizhakkethottam [13] and Mehdipour et al. [25] explicitly argue that testing cannot cover the infinite edge cases present in such complex systems, necessitating rigorous mathematical proofs to ensure correctness. Consequently, formalization has emerged as a foundational requirement in critical domains and the 4 works in this cluster analyze the application of these methods across distinct sectors, exposing the technical barriers of uncertainty and scalability in dynamic, real-world environments.

2.5.2.1 Domain-Specific Challenges

In the **railway domain**, the adoption of formal methods is driven by strict regulatory pressure, which mandates rigorous verification for safety-critical systems. The real barrier identified in this area is **process integration**, where engineers must not only verify that the logic is correct but also maintain a strict chain of evidence linking every line of the model back to a specific safety requirement. Ferrari et al. [10] found that this is where most formal tools fail, by acting as **isolated independent ecosystems** that perform verification well but lack the necessary development functionalities, like **automated report generation** and **requirements linking**, required to fit into the strict industrial workflows. The tools solve the mathematical problem but fail to solve the problem of safety certification.

For vehicles in **autonomous driving**, the challenge changes for a fundamental difficulty of formalizing the environment. Mehdipour et al. [25] argue that the primary barrier is translating ambiguous natural language traffic laws (e.g., “maintain a safe distance”) into rigorous logic formulas. This leads to the complex problem of **rule prioritization**. Under uncertainty and conflict, autonomous vehicles must choose between violating a minor rule to satisfy a higher-priority one to avoid a catastrophic failure. Current formal methods struggle to handle these conflicting objectives and to mathematically encode this hierarchy of violations. Mehdipour et al. [25] also mention that verification algorithms are computationally too expensive for the millisecond-level decision loops required by autonomous vehicles, indicating a need for **online verification**. Online verification methods struggle to keep a real-time pace systems often resort to conservative "fail-safe maneuvers" (e.g., coming to a stop) when computation takes too long. While this guarantees safety, it degrades driving efficiency and traffic flow

In the Internet of Things (IoT), Haddou-Oumouloud et al. [16] identify **scalability** as the paramount challenge. The exponential growth of IoT applications leads to a state space explosion

that makes traditional exhaustive verification techniques like model checking computationally unfeasible. The author mentions abstraction has been proposed as a solution but that further research is needed to develop a scalable algorithm. As concluded by Mehdipour et al. [25] in the field of autonomous driving, safety-critical IoT applications operate under strict timing constraints and Haddou-Oumouloud et al. [16] note that a major gap in current research is the inability to verify real-time performance while maintaining system efficiency in these environments.

Finally, Gokulnath and Kizhakkethottam [13] highlight that in the **smart contracts** domain, the fundamental challenge is the architectural constraint of **immutability**. Unlike other software, smart contracts cannot be modified after deployment and this irreversibility transforms verification from a continuous quality assurance process into a high-risk, one-time requirement where any overlooked vulnerability results in permanent financial loss.

2.5.2.2 Hybrid Approach

Across all domains, there is a consensus towards the future of safety-critical verification. The literature suggests that it lies in the convergence of complementary formal methodologies to address the limitations of individual techniques. Haddou-Oumouloud et al. [16] show that while model checking provides automation, it struggles with scalability, whereas theorem proving handles complexity but requires manual effort. The integration of varied tools offers a comprehensive verification solution. Similarly, Gokulnath and Kizhakkethottam [13] highlight the emergence of hybrid workflows that combine formal verification with static analysis and runtime monitoring to provide assurance for blockchain systems.

2.5.3 Automated Reasoning and AI-assisted Verification

The integration of Large Language Models (LLMs) into formal verification represents a paradigm shift from manual proof engineering to AI-assisted proof development. The drive to integrate artificial intelligence into formal verification derives from a critical limitation in current tools. The literature identifies three main barriers where traditional automation fails and where AI techniques are proved to be transformative, addressing the semantic gap between informal scientific knowledge and formal code, the guidance gap where solvers require manual intervention, and the contextual gap regarding how to effectively prompt these models.

2.5.3.1 Semantic Gap

A central challenge in formal methods is the translation of high-level algorithmic concepts into mechanically verifiable code. Wang, Scazzariello, and Chiesa [38] identify a semantic gap where proofs in scientific literature rely on intuition and omit low-level details that are mandatory for formal verifiers. To overcome this, they propose a two-stage transformer approach, their tool *PROMETHEUS*, where the first stage utilizes the LLM's natural language processing capabilities to translate the textual proof into a high-level verification model. The second stage leverages the

model to autonomously generate the tedious low-level proof steps, such as induction tactics, which are intellectually trivial for humans but syntactically demanding to encode.

2.5.3.2 Guidance Gap

While auto-active verifiers like Dafny automate the mechanical checking of proofs, they usually suffer from incomplete automation. The reliance on Satisfiability Modulo Theories (SMT) solvers is characterized by Mugnier et al. [27] as a "double-edged sword", while it significantly reduces the manual burden of proof checking, the solver's opaque failures require expert manual intervention to resolve, as discussed in Section 2.5.1.1. To resolve this limitation, the literature proposes AI-driven solutions that operate at two levels of abstraction:

1. **Inferring Missing Lemmas**: Silva, Mendes, and Ferreira [34] address the challenge where verification fails because a necessary intermediate theorem, a lemma, is entirely missing. In this context, the LLM functions as an assistant, hypothesizing missing properties that the verifier cannot derive on its own. The creative but potentially unreliable suggestions of the LLM are subjected to the rigorous check of the formal verifier, ensuring that only mathematically valid lemmas are accepted.

2. **Synthesizing Assertion Hints**: Even when the correct lemmas are present, solvers often fail due to the complexity of the reasoning chain. Mugnier et al. [27] focus on the tedious task of debugging these failures by inserting inline assertions that decompose the proof state for the solver. This automation directly addresses the steep learning curve and the difficulty in constructing proof strategies identified by Juhošová, Zaidman, and Cockx [18] and Oliveira, Mendes, and Carreira [30] in Section 2.5.1.1, as it assists users in identifying the necessary intermediate steps that are often non-intuitive. Mugnier et al. [27] argue that generic LLM prompting is ineffective for this task and introduce *LAUREL*, which works with two specialized techniques to improve accuracy:

   (a) **Error Localization**: Analyzes the verifier's error message to strictly constrain where the assertion placeholder should be inserted.

   (b) **Context-Aware Retrieval**: Uses a Proof Similarity Metric to retrieve structurally similar examples from the codebase, feeding them to the LLM to guide the synthesis of domain-specific assertions.

2.5.3.3 Contextual Gap

A common finding across the studies is that, despite their general capabilities, LLMs lack the specific logical context required for formal verification. Consequently, the field is converging on two complementary methodologies to ground the model's output.

Mugnier et al. [27] demonstrate that naive prompting fails because the LLM lacks knowledge of the codebase's specific definitions, and invariants. To overcome this, they introduce a **Proof**

Similarity Metric, a hierarchical algorithm based on edit distance that measures structural similarity between proofs rather than just textual overlap. This metric enables the retrieval of highly relevant few-shot examples from the codebase, effectively teaching the LLM the local verification variant. The authors found that this targeted context is critical, since using structurally similar examples boosted performance significantly compared to using random examples.

Complementing context is the shift from one-shot generation to dynamic interaction. Both Silva, Mendes, and Ferreira [34] and Wang, Scazzariello, and Chiesa [38] treat the formal verifier not as a final decision, but as a conversational assistant. Silva, Mendes, and Ferreira [34] describe a "Continuous Feedback Loop", where the verifier's opaque error messages are fed back into the LLM to correct syntactic or logical errors in generated lemmas, essentially using the verifier to debug the AI's erroneous outputs. Similarly, Wang, Scazzariello, and Chiesa [38] implement a Top-Down Refinement strategy in *PROMETHEUS*. Instead of attempting to generate a perfect proof in one pass, the system first generates a high-level model and then iteratively fills in and verifies the low-level gaps, ensuring that each step is sound before proceeding. In this approach, the formal verifier evolves from a passive checker into an oracle that guides and constrains the generative process.

2.6 Conclusion

The reviewed literature reveals that the main barriers to the adoption of VA languages are not purely theoretical, but largely related to usability, learning curve, tool maturity, and integration into existing software development workflows. Difficulties such as designing proof strategies, interpreting solver feedback, and maintaining proofs as code evolves affect both beginners and experienced practitioners, directly addressing RQ1.

While recent advances in automation and AI-assisted verification try to reduce manual effort, the literature indicates that these approaches do not fully eradicate the underlying challenges. Instead, they emphasize the need for improved interaction models, enhanced explainability, and more robust user support mechanisms. Although VA languages are widely applied in safety-critical domains, their adoption is often driven by external requirements rather than perceived productivity benefits.

Overall, the review reveals a gap in current research, as there is limited systematic studying of the challenges and practical strategies developers use to overcome them. This gap motivates the next phases of this work, which aim to study practitioner behavior through the analysis of video content, and to derive the resulting insights into concrete improvements in the context of VA languages, addressing RQ2 and RQ3.

Chapter 3

Methodology

3.1 Introduction

One of the goals of this study is to find the practical techniques and methods used by programmers when using VA languages. Many developers share their experiences through online educational content, including demonstrations and discussions that are not always written in formal documentation. This helps us see how developers deal with the limitations of these languages.

This chapter describes the complete process followed to answer RQ2, which questions what strategies and practices do practitioners employ to overcome challenges when working with VA frameworks. In Section 3.2, we describe how we collected data, which includes the creation of the search string and the retrieval of video metadata. In Section 3.3, we detail the criteria applied to select relevant videos from the initial dataset. In Section 3.4, we present the preprocessing pipelines applied to prepare the corpus for analysis. In Section 3.5, we describe how we filtered excerpts to extract relevant ones from the raw transcripts. Finally, in Section 3.6, we describe the topic modeling stage, including the approach selection, model configuration, and the process of generating, refining, and assigning topics.

3.2 Data collection

While prior work has used text-based platforms to study developer challenges with VA languages [30], these sources only show us the outcome of the problem-solving process and we do not know how the developer actually got to that solution.

Video content, particularly live coding sessions and interactive verification walkthroughs, can provide further information, since we can see developers interacting with VA tools in real time, making it possible to observe their reasoning process and actions, something that text-based platforms cannot do. As Oliveira, Mendes, and Carreira [30] acknowledge, their analysis of online discussions was able to identify what challenges developers face, but did not extend to the strategies used to overcome them, constrained by their choice of data source.

Although video content is not as structured as written text, that limitation is acknowledged and dealt with in later stages of this work, through a specialized text segmentation and semantic filtering pipeline. For our goal of understanding how developers interact with verification systems, video content is the most authentic source of evidence available. Therefore, data was retrieved directly from the **YouTube**[1] platform to construct a representative dataset composed of both video information and conversational transcripts for subsequent analysis.

The data was collected using a rigorous process to ensure reproducibility and comprehensiveness of the dataset, as described in the following steps.

3.2.1 Search String Formulation

The search string used on YouTube was developed through an iterative process. The goal of the query was to retrieve videos in which users interacted actively with verification tools to optimize the relevance and coverage of verification-aware programming tools and associated challenges.

The final search string is composed of two groups of terms combined using a logical AND operator:

```
("Agda" OR "Bandera" OR "Rocq" OR "Dafny" OR "Eiffel" OR "Escher
C Verifier" OR "Frama-C" OR "Idris" OR "Java PathFinder" OR
"JML" OR "KeY" OR "Lean 4" OR "mbeddr" OR "P" OR "Spark Ada" OR
"Spec#" OR "TLA+" OR "VCC" OR "VeriFast" OR "Verus" OR "Whiley"
OR "Why3") AND ("live coding" OR "interactive session" OR "proof
demo" OR "verification walkthrough" OR "verification tutorial" OR
"assertion failure" OR "precondition failure" OR "postcondition
violation" OR "solver timeout" OR "verification failure" OR
"resource exhaustion" OR "common errors" OR "debugging" OR
"verification errors" OR "auxiliary lemma" OR "proof strategy" OR
"refactoring proofs" OR "proof decomposition" OR "ghost variable"
OR "loop invariant" OR "induction proof" OR "opaque definitions"
OR "fixing verification error" OR "smt solver explained")
```

The first group of terms corresponds to a set of VA languages and verification tools, including languages and systems commonly used in the formal verification community. The languages were obtained once again from the **Formal Methods Europe** committee, as discussed in Chapter 2 for the literature review methodology.

The list was complemented with additional tools selected according to their relevance within the verification community and literature, and the availability of material on YouTube. We added *Verus*, due to the growing interest in the language within the academia and industry, showed in recent publications and its adoption by *Amazon Web Services*. Amazon Web Services (AWS) has used Verus for the formal verification of Rust systems and developed tools like AutoVerus[41] to improve proof automation. The ITPs Rocq and Lean were also added to the search string despite not being exclusively classified as verification-aware programming languages. Besides their

[1] https://youtube.com

presence in online content, the inclusion of these ITPs is justified by the similar set of main activities between them and VA languages, which the literature treats as continuous rather than distinct practices [39, 42]. Specifically, when automated solvers fail to verify complex proof obligations in VA languages, practitioners frequently turn to ITP reasoning, introducing lemmas and structuring proofs interactively [39, 42]. Given this, we considered that the techniques observed in ITP can be applied to the VA language context of this study. Table 3.1 formalizes the inclusion criteria applied when deciding which verification-aware languages were considered eligible for this study.

Table 3.1: Language Inclusion Criteria considered in this study.

ID	Description
LIC1	The language is referenced by the Formal Methods Europe industry committee as an auto-active language.
LIC2	The tool shares its core verification activities with VA languages, even when not exclusively classified as an auto-active language.
LIC3	Sufficient material referencing the language exists on YouTube to support the construction of a representative video corpus.

The second group of terms covers contexts where developers interact with verification-aware tools and demonstrate verification tasks. These terms include expressions associated with interactive scenarios like *live coding*, *interactive session*, *proof demo*, and *verification walkthrough*, which are likely to retrieve videos demonstrating the verification process in real time.

The query also includes a group of terms related to errors and debugging situations, such as *assertion failure*, *precondition failure*, *solver timeout*, *verification failure*, and *verification errors*. When this happens, practitioners usually have to fix or refine the proof, which once again shows the practical application of verification techniques.

Finally, the query includes general concepts related to developers' experiences with VA languages, and also a set of terms related to common verification techniques known in the literature. Terms such as *auxiliary lemmas*, *loop invariants*, *ghost variables*, *induction proofs*, and *proof decomposition*, are commonly used to guide formal verification in VA programming languages. The goal of looking into these terms is to observe their adoption and application in real-time settings.

During the formulation of the query, particular care was taken to exclude excessively general terms. Initial manual searches showed that keywords such as *tutorial* or *demo* generated excessive noise, consisting of introductory materials that lacked the practical approach required for our analysis. Whenever possible, these terms were therefore refined with more specific descriptors or removed.

In addition, videos without captions were excluded from the results based on the observation that the audio of such content consisted of music or suffered from poor quality, which prevented reliable transcription and explained the absence of automatic transcriptions. To avoid unnecessarily limiting the dataset, no linguistic restrictions were applied, instead, non-English transcripts were translated using the **Deep Translator** Python package to ensure an inclusive analysis.

Overall, the final search string returned a representative set of results for each search, allowing the construction of a dataset that reflects the diversity of relevant videos available on YouTube.

3.2.2 Video Data Retrieval

Following the definition of the search string, video data was obtained from YouTube using *YouTube Data API v3*[2]. Since the YouTube Data API v3 no longer provides dislike counts, that information was obtained via the *Return YouTube Dislike API*[3]. By storing these metrics, the dataset provides more insights into how each video was received by the audience. Table 3.2 contains a description for each column of the dataset, corresponding to each video metadata.

Table 3.2: Description of columns of the videos dataset

Column Name	Description
URL	Direct link to the video on YouTube, facilitating access to the original content
Title	Title of the video as provided by the uploader
Publication date	How long was the video published on the platform
Views	Total number of times the video has been watched
Likes	Total number of positive reactions (likes) expressed by viewers
Dislikes	Total number of negative reactions (dislikes) expressed by viewers
Comment Count	Total number of user comments associated with the video
Channel	Name of the YouTube channel that published the video
Duration	Total length of the video, measured in hours, minutes and seconds
Original Language	Detected language of the video's audio and captions

Since this study looks at video content, it was necessary to have access to textual content of what is being said in the videos. We retrieved the transcripts available on YouTube using the *YouTube Transcript API for Python*, which allowed direct access to existing closed captions.

3.3 Selection of Relevant Videos

We adopted a screening process based on the inclusion and exclusion criteria presented in Table 3.3 and Table 3.4, carried out by two reviewers, followed by a third one to resolve disagreements.

Table 3.3: Inclusion Criteria considered in this study.

ID	Description
IC1	Consists of live coding sessions.
IC2	Focuses on practical programming and problem solving.
IC3	Relates to verification-aware programming.

[2] https://developers.google.com/youtube/v3
[3] https://returnyoutubedislike.com/

Table 3.4: Exclusion Criteria considered in this study.

ID	Description
EC1	Consists of lecture-style (slides or theoretical) explanations and no live coding.
EC2	Does not include practical demonstrations of verification coding.
EC3	Unrelated to formal verification.
EC4	The audio consists of music and/or does not contain spoken speech.

Inter-rater reliability was assessed using Cohen's kappa. The resulting coefficient was $\kappa = 0.881$, which corresponds to an almost perfect agreement according to the interpretation proposed by Jacob Cohen [8].

The initial retrieval process gathered a total of 2759 videos. After removing duplicates, we kept 2109 unique videos. The two independent reviewers agreed on the exclusion of $1,911$ videos and the inclusion of 166. Disagreements regarding 32 videos were resolved by the third reviewer, resulting in a final dataset of 198 included videos.

Figure 3.1 presents the distribution of the 1911 excluded videos by the exclusion criterion, revealing that the predominant reason for exclusion was the absence of practical verification coding demonstrations (EC2), which accounted for 1645 removals. Figure 3.2 shows how the final video dataset spreads out across verification-aware languages. The *Others* category includes languages that were not originally specified in the search string, such as Haskell, Isabelle, and Rust, but still appeared in the results and were considered relevant during the screening phase. The *General* category groups videos that show techniques that can be used in different verification environments.

Figure 3.1: Distribution of excluded videos by exclusion criterion.

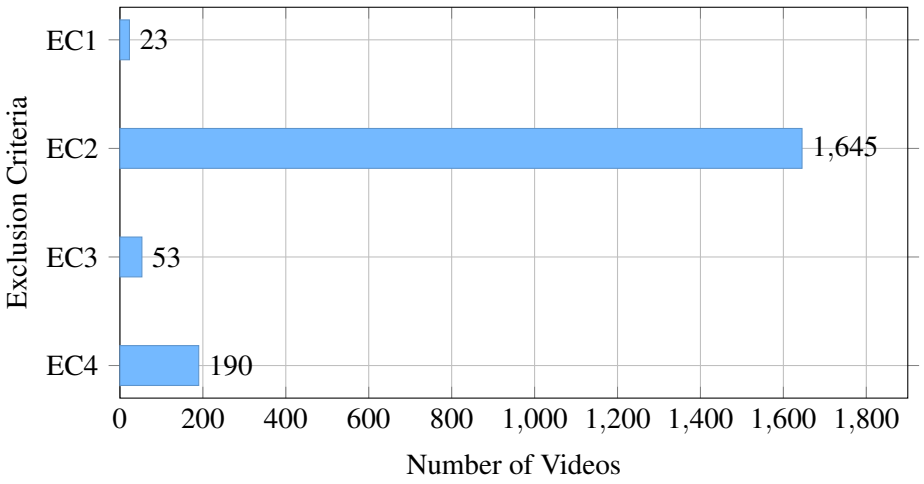
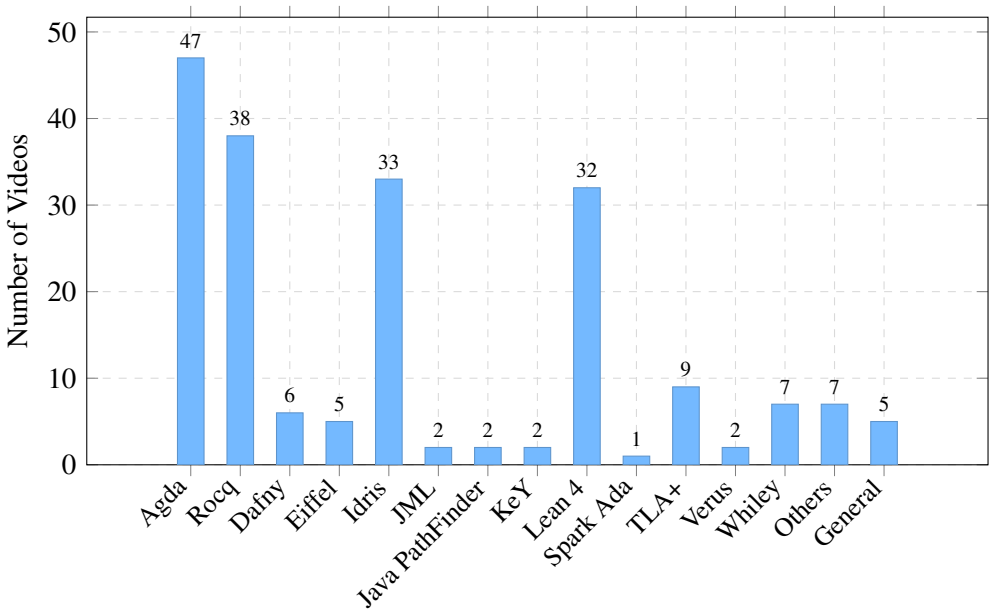


Figure 3.2: Distribution of the final video dataset by verification-aware language.

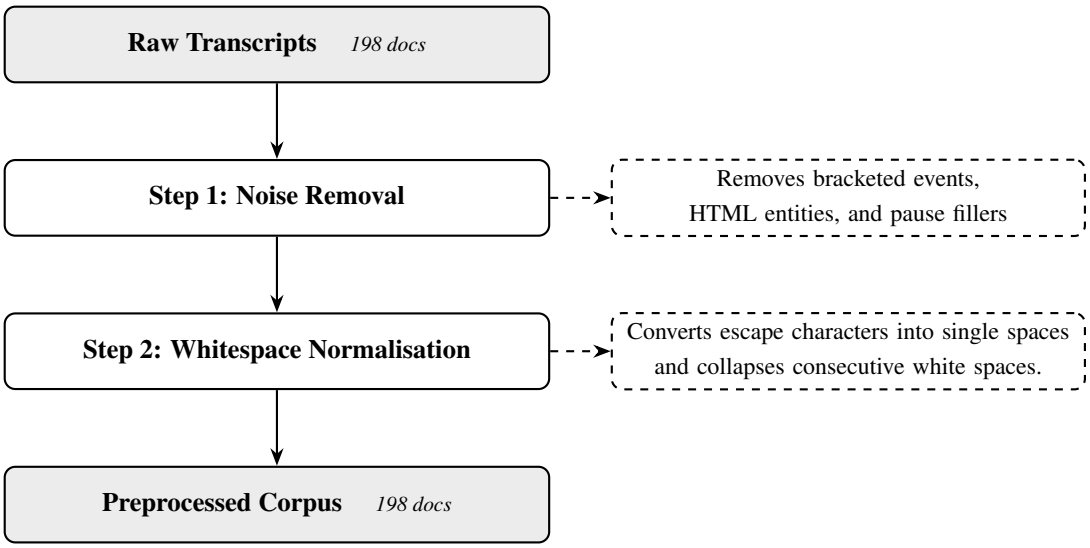


3.4 Data Preprocessing

Before performing topic modeling, the transcripts obtained from YouTube went through a sequence of preprocessing steps to normalize the data and improve the quality of the input. The preprocessing pipeline was designed according to the characteristics of the raw transcript data. Although YouTube Automatic Speech Recognition (ASR) transcripts contain noise artefacts that introduce tokens with no semantic relevance to the verification workflow, they were not submitted to an intensive preprocessing, typical of frequency-based statistical models, such as Latent Dirichlet Allocation (LDA). Since the subsequent phases of this work rely on Sentence-BERT embeddings and an LLM-based topic modeling framework, both of which operate on natural language input, the pipeline was constrained to operations that produce clean text without compromising its structure. Therefore, heavy normalization techniques, like tokenization and lemmatization, were excluded, as they affect the input quality for models trained on full sentences [15]. Sentence-BERT is designed to generate semantically meaningful sentence embeddings from textual input [33], therefore, preserving the integrity of the natural language signal is not a limitation of this pipeline, but a requirement for the performance of the next stages.

The complete preprocessing pipeline is illustrated in Figure 3.3. Each step is described below and its details are presented in Table 3.5.

Figure 3.3: Overview of the data preprocessing pipeline applied to the transcripts.



3.4.1 Transcript Noise Removal

The noise includes non-verbal event annotations, such as `[Music]`, `[Applause]`, and `[Laughter]`, as well as round-bracket variants such as `(inaudible)` and `(crosstalk)`. Speaker-change markers expressed as chevron sequences (»>) and residual HTML entities (`&`, `<`, `>`, ` `) may also be present in transcripts retrieved via the YouTube Transcript API. Additionally, speech markers for pauses, such as “uh”, “um”, “hmm”, are removed using targeted regular expressions [9].

Although an LLM would be capable of reading past such annotations, their presence introduces tokens that do not contribute to technically relevant excerpts and would make it harder to find them, by compromising the quality of the sentence embeddings computed during the subsequent stage (Section 3.5). Embedding a chunk that contains such noise would reduce cosine similarity against the verification query set and risk the rejection of relevant content. The cosine similarity is a metric that measures the semantic similarity between two vector representations by computing the cosine of the angle between them. It depends on the orientation of the vectors and it provides an effective measure of semantic proximity between embeddings.

To quantify the noise removed and verify that it was operated correctly, an inspection was done before and after the step. Noise tokens were found in 92.9% of the transcripts, with a mean of 135.1 tokens per transcript. Even though these tokens were common, the slight decrease in character count is due to the fact that the removed artefacts consist mostly of short tokens, rather than long textual expressions. After removal, the mean length of a document decreased in 2.4% of the characters, and 3.2% of the words. Along with this, a manual inspection was performed on random sample of transcripts to verify that no technically relevant content was removed by the noise removal step, increasing confidence in the reliability of the results. The corpus was divided into three groups based on video duration:

- Short duration videos: < 20 min, $n = 67$
- Medium duration videos: 20–60 min, $n = 75$
- Long duration videos: > 60 min, $n = 56$

Approximately 10% of the transcripts within each group were sampled, selecting seven documents from the first group, eight from the second and six from the third. This approach resulted in a validation sample of 21 documents and their analysis confirmed that no relevant content was lost during the noise removal process.

3.4.2 Whitespace Normalization

Following noise removal, all text formatting is standardized by replacing newlines, tabs, and carriage returns with a single space. Any consecutive spaces are collapsed into one, and any leading or trailing white spaces are removed.

This step is necessary because some YouTube transcripts contain line breaks designed for screen display timing rather than grammatical sentences. If left unmodified, these breaks could split a single sentence or technical phrase into two separate text chunks. This fragmentation would affect the quality of the text embeddings during the text segmentation step of the semantic filtering stage in Section 3.5, where the segmentation logic uses punctuation at the end of a sentence and whitespace as the way to split the text. Collapsing extra whitespace ensures that future text division is guided by actual structure of the language and not the way the subtitles are formatted.

We also found that 5.6% of the transcripts contained mid-sentence line breaks, with a mean of 3.9 per affected transcript and a range of 0 to 195. After removal, the mean length of the documents went from 33,110.6, to 32,975.4 characters. It is worth noting, this operation did not affect mean word count per document, confirming that the removal did not include content. The stable values of the word count mean confirm that the character reduction reflected only the removal of the non-semantic formatting elements without any loss of textual content.

Table 3.5: Corpus Statistics at Each Preprocessing Step.

Step	Mean chars	Min chars	Max chars	Mean words
Raw transcripts	33,908.5	756	211,670	6,640.6
After noise removal	33,110.6	756	180,476	6,425.9
After whitespace normalization	32,975.4	756	180,410	6,425.9

No minimum document length threshold was applied in the preprocessing phase. Documents that are too short to contain technically relevant content will be handled appropriately by the semantic similarity filtering stage in Section 3.5, which operates at the chunk level. Those documents will produce zero retained chunks and be excluded from topic modeling.

3.5 Relevant Excerpt Extraction via Semantic Similarity Filtering

A single video from the dataset may alternate between conceptual introductions, live coding demonstrations, audience interactions, and even off-topic discussions, meaning that feeding an entire transcript into a topic model would dilute the signal from technically relevant segments and introduce noise into the analysis. This challenge comes from the heterogeneity of the resulting videos, where some consist predominantly of live coding sessions in which practitioners have a continued interaction with a verification tool, and others are conference talks or lecture recordings. These conference-style videos were deliberately retained in the dataset to ensure a comprehensive coverage of the practitioner ecosystem, despite the fact that practical live coding or troubleshooting occupies only a brief fraction of their total runtime.

To illustrate this problem, we consider a representative conference talk from the dataset with a duration of approximately 45 minutes. From a total of 61 extracted chunks, only 14 contained technically relevant content related to verification strategies, while the remaining 47 consisted of non-technical speech. Without filtering, these 47 non-technical chunks would be treated as equally informative signals in the topic modeling process, reducing the impact of the 14 relevant segments.

To address this, a semantic filtering stage was included before performing topic modeling, with the goal of retaining only the transcript excerpts that are semantically proximate to the kinds of technical activity under study.

3.5.1 Model Selection

Filtering was performed using **Sentence-BERT**, a modification of the BERT architecture that produces fixed-size sentence embeddings optimized for semantic similarity tasks by means of siamese and triplet networks [33]. The *all-mpnet-base-v2* model was selected, which is based on the MPNet architecture [35]. Among the models available in the Sentence Transformers library, *all-mpnet-base-v2* achieves the highest average performance on the SBERT semantic textual similarity benchmarks [33], making it the most suitable choice. Its architecture captures subtle semantic relationships that are particularly important in a specialized domain like formal verification, where a snippet may share no words with the query and still be semantically close to it.

3.5.2 Query Set Design

Semantic filtering with Sentence-BERT computes the cosine similarity between embeddings of transcript chunks and a set of reference queries that define what constitutes active verification practice in the context of this study. The query set was constructed to follow a structured and domain-guided query expansion process [6], grounded in the findings of the systematic literature review presented in Chapter 2.

For each challenge or practitioner behavior documented in the literature, a corresponding query category was defined. The categories were derived directly from the three clusters identified in Section 2.5, which include the usability and learning curve barriers, the proof structuring

and debugging difficulties, the specification and annotation practices, and the proof maintenance challenges. This mapping ensures that the query set provides systematic coverage of the developer behaviors relevant to RQ2, rather than relying on unregulated manual creation. Each subsection of Section 2.5 was reviewed individually, and the specific developer behaviors or recurring difficulties discussed within it were extracted as themes and their mapping can be seen in Table A.2.

Within each derived category, multiple natural language phrases were formulated following the principle of synonymy handling in query expansion [6]. A single query phrase cannot capture the full range of expressions through which a verification activity manifests in spoken discourse, so each category is represented by multiple semantically equivalent but lexically varied phrases.

This distinction is crucial in the context of live coding transcripts, where developers rarely invoke theoretical constructs explicitly. As referred by Carpineto and Romano [6], the vocabulary mismatch between a searcher's terms and the terms used in relevant documents is one of the primary sources of retrieval failure. In spoken informal discourse, a practitioner experiencing proof brittleness, for instance, may say "the proof was working before but now it fails" than to use the technical term, and the query set was designed to capture both registers.

The resulting categories and their associated queries are presented in Table A.1.

To verify that the query set correctly discriminated between practical and non-practical content, a validation procedure was conducted on the same 21-document stratified sample used in Section 3.4. After computing similarity scores for all chunks extracted from these documents, resulting in a total of 1,164 chunks, the resulting score distribution was divided into three groups based:

- High similarity score: score ≥ 0.50 (top tercile)
- Medium similarity score: $0.40 \leq$ score < 0.50 (middle tercile, spanning the region around the expected decision boundary where classification errors are most likely to occur)
- Low similarity score: score < 0.40 (bottom tercile)

From the low and high groups, 10% of chunks were sampled (n=39 each), and from the medium group, given its higher relevance to threshold determination, 15% were sampled (n=58). This procedure resulted in a validation sample of 136 chunks. Each chunk was manually inspected and classified by one team member as corresponding to a moment of active verification practice or to off-topic content.

An analysis of the groups confirmed its effective filtering and high precision. The low group successfully isolated non-relevant data, and the medium and high groups demonstrated high technical density. Manual inspection verified that medium and high-scored chunks focused consistently on active verification practices, while the low group caught most irrelevant content.

3.5.3 Text Segmentation and Similarity Scoring

Transcripts generated by ASR systems frequently lack consistent punctuation or contain punctuation inserted by automatic restoration models, systems that trained to predict punctuation boundaries in unpunctuated text, which might make sentence-boundary segmentation less reliable than

in written text [36]. The segmentation hierarchy described below was designed with this limitation in mind, through a fallback mechanism described at the end of the section.

To transform the speech data into a representation suitable for semantic analysis, the text must first be segmented into manageable units. The segmentation process operates hierarchically to preserve the integrity of technical expressions, where text is primarily split at sentence boundaries, identified by terminal punctuation followed by whitespace. Transcript chunks must be long enough to provide the encoder with adequate syntactic and semantic context, since very short inputs tend to generate less stable embeddings [33]. However, excessively long chunks may dilute semantic focus by mixing relevant and irrelevant content within a single embedding, reducing retrieval accuracy. The choice of chunk size and similarity threshold involves a careful balance. Chunks that are too long or thresholds that are too permissive risk diluting the topic modeling input with non-relevant content, while chunks that are too short or thresholds that are too strict risk discarding technically relevant material expressed through paraphrase or informal speech register.

An initial chunk size of 800 characters was adopted as a starting point, consistent with the literature and text segmentation practices in retrieval systems [2, 3, 32]. However, a preliminary analysis revealed that chunks of this size demonstrated a reduced discriminative power, due to excessive contextual breadth. At 800 characters, multiple topics were mixed within the same text segment, diluting the model's focus and causing nearly all chunks, whether relevant or not, to receive average similarity scores between 0.40 and 0.50, degrading the separation between relevant and irrelevant content and making it impossible to define an adequate threshold.

Reducing the chunk size to 600 characters increased this separation. At this size, chunks typically capture one to three complete phrases, which is enough to preserve the syntactic context required for stable Sentence-BERT embeddings [33], while being short enough that a single chunk is unlikely to include multiple unrelated topics. This length was therefore adopted for the full corpus. No further reduction was tested, as values below 600 characters approach the length of individual sentences and would risk splitting complex technical sentences across chunk boundaries, which Reimers and Gurevych [33] identify as a source of degraded embedding quality.

To prevent boundary effects introduced by fixed-size segmentation, a contextual overlap of 120 characters is carried over from the end of each chunk to the beginning of the next, aligned at a word boundary. This value was chosen proportionally to the chunk size, falling within the 10–20%, range recommended by common practices in text segmentation pipelines [2, 3]. This overlap value promotes a balance between preserving context and avoiding redundancy. A smaller window risked fragmenting technical expressions across chunk boundaries, whereas a larger overlap would introduce excessive repetition.

When a segment of text lacks standard punctuation or exceeds the character limit, a fallback mechanism is applied. The algorithm attempts to split the text at natural pauses, such as commas, or at conversational anchor, such as *"so"*, *"okay so"*, *"now"*, and *"let's"*. If the text block remains larger than the limit, it is subdivided at the word level. This fallback ordering reflects a preference for splits that align with natural units of spoken reasoning over arbitrary character-based cuts, which is particularly relevant for ASR transcripts where punctuation is unreliable.

Following segmentation, semantic filtering is performed by computing the cosine similarity between the embeddings of the transcript chunks and the manually constructed reference query set. A threshold establishes a lower bound on the cosine similarity, ensuring that only segments with a high semantic affinity to the verification queries are retained and videos for which no chunk meets the threshold are excluded from the corpus entirely.

To determine the optimal similarity threshold, an empirical procedure was conducted over the full corpus of 198 videos. The cosine similarity scores produced by the *all-mpnet-base-v2* model distribute meaningfully across the [0, 1] range, where semantically similar sentence pairs consistently score above 0.50, while dissimilar pairs fall below that boundary. This interpretation of the score space provided a principled starting point for threshold selection.

As detailed in Table 3.6, an initial threshold of 0.50 was adopted as a natural starting point, as it corresponds directly to the boundary between related and unrelated sentence pairs [33]. However, this setting proved overly restrictive, reducing the corpus to only 125 videos with an average of 4.0 relevant chunks per document. Manual analysis confirmed that this setting was systematically rejecting technically valid content expressed through paraphrase or indirect phrasing. This observation motivated the exploration of lower threshold values.

Progressively lower values within 0.40 and 0.50 were then tested in steps of 0.01. For each value, the number of videos retaining at least one chunk above the threshold, the mean similarity score among the retained chunks, and the mean number of retained chunks per video were recorded. The results are presented in Table 3.6.

Table 3.6: Effect of the similarity threshold on corpus composition.

Threshold	Videos retained	Mean similarity (retained)	Mean chunks / video	Total chunks retained
0.40	180	0.463	14.0	2519
0.41	177	0.471	12.5	2204
0.42	172	0.479	11.2	1931
0.43	169	0.487	9.9	1671
0.44	166	0.496	8.6	1425
0.45	163	0.504	7.5	1229
0.46	159	0.514	6.5	1032
0.47	**154**	**0.523**	**5.6**	**859**
0.48	142	0.534	5.0	711
0.49	131	0.544	4.5	590
0.50	125	0.552	4.0	506

It can be seen in Table 3.6 that between 0.40 and 0.47, each 0.01 increment removes between 3 and 5 videos from the corpus, while the mean similarity of retained chunks increases steadily by approximately 0.008–0.010 per step. From 0.47 onward, this pattern breaks and the step from 0.47 to 0.48 alone removes 12 videos, and the step from 0.48 to 0.49 removes a further 11, more than double the rate observed at lower thresholds. This indicates that a substantial subset of videos contains its most relevant excerpts in the range 0.47–0.49. Raising the threshold past 0.47 exchanges a small gain in mean similarity for a disproportionate loss of entire videos. Based on this analysis,

0.47 was selected as the threshold, as the last value before this discontinuity. The mean similarity of retained chunks at this threshold (0.523) lies comfortably above the 0.50 boundary for semantically related sentence pairs, providing a safety margin against false positives that a lower threshold would not, while also preserving the representativeness of the corpus across the verification-aware languages. Finally, the retained videos provide a median of 4 and a mean of 5.6 chunks each, a density considered sufficient to characterize the verification strategies discussed in each video for the subsequent topic modeling stage. The 44 videos excluded at this threshold were manually inspected. These corresponded predominantly to conference talks in which, no transcript segment described a concrete interaction with a language or tool. Their exclusion is therefore consistent with the goal of this stage.

Finally, for each of the 154 retained videos, the chunks scoring at or above 0.47 are concatenated in their original transcript order, forming a single filtered document per video. This step ensures that the resulting representation preserves the context of the source while being filtered for semantic relevance.

3.6 Topic Modeling

This section describes the topic modeling stage applied to the preprocessed corpus of VA video transcripts, detailing each step of the chosen framework and the configuration decisions specific to this study. The goal of this stage is to identify recurring strategies that practitioners employ when working with verification-aware languages, directly addressing RQ2.

3.6.1 Approach Selection

Traditional statistical topic models such as LDA have been widely employed in prior software engineering research, including in the work by Oliveira, Mendes, and Carreira [30], on which this study builds. LDA operates within the bag-of-words paradigm, relying on term frequency and co-occurrence statistics [17], and mainly captures the topics being discussed rather than deeper insights beyond the discussed subjects [31]. However, the objective of this work goes further than identifying the general themes present in the videos, focusing on the discovery of concrete verification techniques and strategies that are being used by programmers in online multimedia content. As a result, it became necessary to shift toward a different approach.

Given the specialized nature of the domain and the need to derive insights that go beyond the explicitly discussed content, this study adopts TopicGPT [31], a prompt-based topic modeling framework that uses LLMs to generate and assign topics in natural language, producing thematic labels and descriptions with greater semantic depth and flexibility than conventional bag-of-words approaches. Pham et al. [31] report that TopicGPT outperforms several topic modeling methods, including LDA in alignment with human-annotated ground truth, achieving a harmonic mean purity of 0.74 compared to 0.64 for the strongest statistical baseline.

An additional reason for adopting TopicGPT is its support for seed topics, by which an initial set of thematically grounded topics can be provided to guide the generation process. In the literature review, we idenfied techniques when discussing challenges, therefore, by encoding these as seed topics, the framework is guided while still allowing the discovery of new themes.

Our implementation follows the official TopicGPT repository[4], using the *topicgpt_python* package and adapting the sample prompts provided to the specific domain requirements.

3.6.2 Model Selection

The modeling pipeline executes over the fully preprocessed and filtered dataset of 154 documents that resulted from Section 3.5 by prompting an LLM. We selected **OpenAI GPT-4o-mini**[5] model, since it provides an appropriate balance between output quality and Application Programming Interface (API) cost, which did not exceed a budget of $15, enabling the processing of the full corpus across multiple pipeline stages without infeasible computational costs, given that the topic generation process involves repeated LLM calls and experiments need to be run multiple times for evaluation and tuning. The same model was used across all pipeline stages.

3.6.3 Topic Generation

The first stage of the pipeline, *generate_topic_lvl1*, creates the initial set of high-level topics. The framework analyzes a representative sample of documents from the corpus one document at a time. For each document, the LLM receives both the document and the current version of the topic hierarchy. It then decides whether the document fits into an existing topic or whether it introduces a new topic that should be added to the hierarchy. As this process is repeated for every document, the topic hierarchy is gradually expanded. This incremental approach allows topics to emerge naturally from the data, without requiring the number of topics to be defined beforehand, in contrast with LDA that requires the number of topics to be specified prior to inference.

The generation prompt was designed to guide the model toward the extraction of actionable verification techniques, explicitly defining documents as transcripts of tutorials or live coding sessions about formal verification programming, and the model's goal as identifying generalizable techniques that developers use to overcome verification challenges. After inspecting an initial set of generated topics, a guiding principle was added to ensure that topics describe how the programmer operates the language and the solver, rather than what the desired outcome is. This distinction was necessary because early generations produced labels that were too abstract to constitute actionable techniques. Additionally, three illustrative examples were included to cover the main cases the model would encounter, which consist of adding a new technique, returning an existing one, and handling a domain-specific scenario. The full prompt is provided in Appendix B.1.

[4] https://github.com/chtmp223/topicGPT
[5] https://developers.openai.com/api/docs/models/gpt-4o-mini

With these rules, the model is then instructed to perform topic generation in two steps. Firstly, it determines which topics are present in the document, secondly, either output existing matching topics or add new ones to the hierarchy.

The generation stage produced topics across all documents, forming the initial topic hierarchy from which the refinement stage described in Section 3.6.4 builds.

3.6.4 Topic Refinement

TopicGPT includes an automated refinement stage *refine_topics*, intended to combine semantically equivalent topics by prompting the LLM to merge entries from the generated hierarchy. However, in practice, the automatic refinement step exhibited critical flaws that invalidated the reliability of the resulting topic list. Entries that described the same verification tactic under slightly different labels or phrasings were frequently retained as distinct topics. On the other hand, in attempting to consolidate the topic set, this step occasionally discarded topics that were both relevant and semantically distinct, removing entries that captured important verification strategies.

We decided to proceed replacing *refine_topics* with a manual refinement, following three phases with the same logic but applied by humans.

The refinement began by addressing the lower-frequency topics that were generated in the previous stage. Topics were ranked by their total number of attributions and progressively accumulated until cumulative coverage reached 80% of all attributions. This selection retained the most recurrent topics while eliminating the ones that appeared too infrequently to constitute reliable patterns.

Despite the redundancy constraints applied during the generation step, some topics remained semantically overlapping. Through an iterative merge process, pairs of topics were evaluated for semantic overlap using cosine similarity between their embeddings, and candidate pairs above a specific similarity threshold were assessed independently by two authors. The first author determined whether each pair should be merged and, where applicable, whether the resulting topic should be assigned to an existing label or represented by a new one. The second author reviewed these decisions, flagging disagreements, which were subsequently resolved through discussion.

The remaining topics were then evaluated against a set of criteria to determine whether each constituted an actual verification strategy:

- The topic is a technique.
- The topic describes a deliberate programmer action rather than a concept or broad thematic area.
- The topic is specific enough to be considered a technique.
- The topic is generalizable across verification languages.

Topics that failed to satisfy all four criteria were removed.

3.6.5 Topic Assignment

To assign the final topics to the videos, *assign_topics* matches each document against the refined hierarchy of topics produced in Section 3.6.4. For each document, the model receives the full filtered transcript alongside the complete topic list and is instructed to identify all topics present, returning for each assignment a topic label, a reasoning justification, and a supporting quote extracted directly from the transcript. The prompt explicitly constrains the model to the predefined hierarchy, prohibiting the creation of new topics, ensuring that all assignments are grounded in both the refined taxonomy and the original source material. The full assignment prompt is provided in Appendix B.2.

The pipeline produced the final topic taxonomy that are analyzed in Chapter 4.

3.7 Conclusion

This chapter described the methodological pipeline used to construct the empirical basis for answering RQ2. Starting from YouTube video retrieval, the process combined manual relevance screening, transcript preprocessing, semantic filtering, and topic modeling to obtain a focused corpus of practitioner interactions with verification-aware tools.

The resulting workflow establishes how the raw video data was transformed into a refined taxonomy of verification techniques and video-level topic assignments. These outputs provide the foundation for the results analyzed in Chapter 4, where the identified techniques are examined in different contexts.

Chapter 4

Results

4.1 Introduction

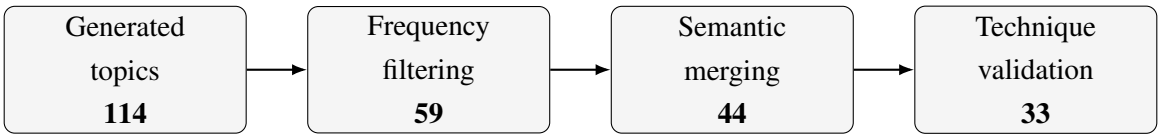
This chapter presents the results obtained from the topic modeling pipeline described in Section 3.6, applied to the processed video transcripts. Its goal is to answer RQ2 by characterizing the verification strategies and problem-solving patterns that practitioners employ when working with verification-aware languages, as captured from the 154 videos retained after semantic similarity filtering.

The chapter is organized as follows. Section 4.2 presents the final taxonomy of verification techniques and the distribution of topic assignments across the corpus. Section 4.3 analyses in detail the seven thematic categories created, relating the observed techniques to recurrent verification challenges reported in the literature. Section 4.4 examines how techniques are combined within the same videos, using pairwise association, frequent combinations, and lift, while Section 4.5 compares the distribution of techniques across verification paradigms. The chapter concludes by summarizing how these analyses contribute to answering RQ2.

4.2 Overview of the Identified Taxonomy

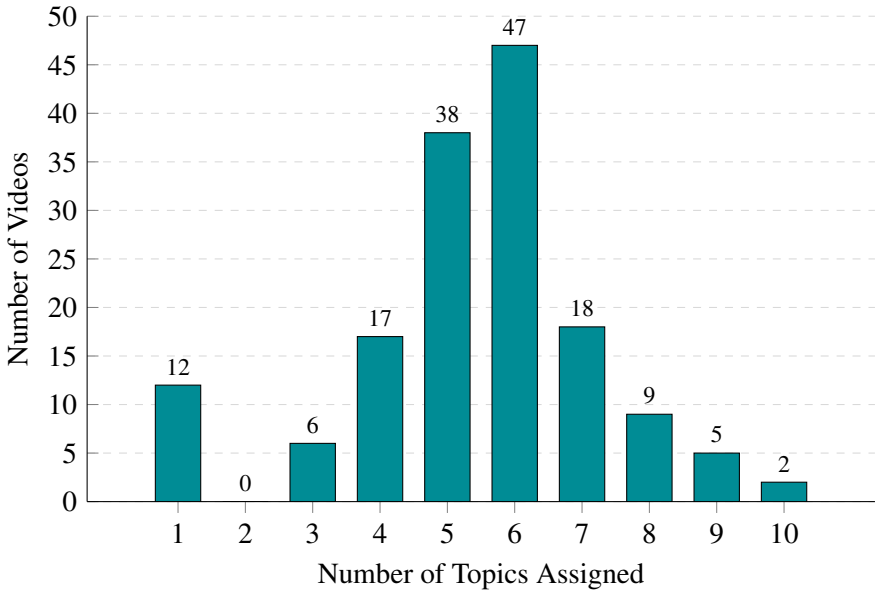
The generation stage of the topic modeling produced a total of 114 top-level topics across all documents, forming the initial topic hierarchy from which the refinement stage described in Section 3.6.4 built. The selection performed in that next stage retained the most recurrent topics while eliminating the ones that appeared too infrequently to constitute reliable patterns, reducing the topic set from 114 to 59 topics. After three iterations of refinement, with a regressive threshold, in which no further merges were identified, the topic list was reduced from 59 to 44 topics. Topics that failed to satisfy all four criteria were then removed, reducing the final topic list from 44 to 33 topics. Figure 4.1 shows the manual refinement process, summarizing how the initial set of 114 generated topics was reduced, resulting in the final taxonomy of 33 verification techniques.

Figure 4.1: Overview of the topic refinement process.



The final assignment table shows a total of 833 topic occurrences across the retained topics, which means that the 33 topics were identified 833 times in the whole dataset. Of these topics, all 33 received at least one assignment. On average, each video was assigned 5.41 topics, with a range of 1 to 10 and a median of 6, indicating that most verification sessions involve multiple techniques in combination rather than a single isolated strategy. The full list of topics and counts can be observed in Table B.1.

Figure 4.2: Distribution of the number of topics assigned per video transcript ($N = 154$).



To facilitate interpretation and comparison with the literature, the 33 topics were grouped into seven thematic categories, based on inspection of their descriptions and the nature of the programmer action they represent. The categories organization was conducted by two independent reviewers, where each reviewer performed the classification independently without access to the other's decisions. The reviewers analyzed the identified topics and assigned them to the most appropriate categories. After the individual assessment, disagreements were identified and resolved through discussion until a consensus was reached. Table 4.1 summarizes the categories that resulted from the double-blind review, listing their corresponding topics, the total number of attributions, and the percentage of the overall attribution count.

Table 4.1: Thematic categories of identified verification techniques, with constituent topics and attribution counts.

Category	Description	Topics	Count	% of Total
Proof Strategy and Decomposition	Techniques for splitting complex proofs into goals, subgoals, cases, and evidence chains to simplify verification.	Goal Decomposition, Inductive Proof Techniques, Proof Goal Management, Evidence Chain Validation, Subgoal Management, Proof Case Splitting, Function Composition	280	33.6%
Invariant and Specification Refinement	Refining invariants, preconditions, postconditions, and function definitions to better guide the verifier.	Invariant Handling, Function Definition Refinement, Precondition and Postcondition Specification, Postcondition Refinement	159	19.1%
Lemma and Hypothesis Engineering	Introducing and manipulating auxiliary lemmas, hypotheses, witnesses, and constraints to bridge gaps the solver cannot infer automatically.	Hypothesis Introduction, Witness Construction, Lemma Engineering, Implication Introduction, Hypothesis Manipulation, Constraint Derivation, Proof Binding	152	18.2%
Solver Automation and Hinting	Managing automated tactics, proof state, and hint databases to guide proof search.	Automated Proof Techniques, Model Checking Debugging, Proof State Management, Hint Database Management	101	12.1%
Termination and Recursion Handling	Managing recursive calls, termination metrics, pattern matching, and inversion principles to establish provable progress.	Recursive Call Handling, Pattern Matching Utilization, Termination Metric Specification, Inversion Principle Application	62	7.4%
Proof Rewriting and Algebraic Manipulation	Rewriting and simplifying proof obligations through goal manipulation, notation changes, substitution, congruence, and transitivity.	Goal Manipulation, Notation Simplification, Congruence Rule Application, Transitivity Utilization, Substitution Function Application	58	7.0%
Specialized Verification Paradigms	Domain-specific techniques such as temporal logic, model checking, and monadic verification.	Temporal Logic Specification, Monadic Verification	21	2.5%
Total		**33 topics**	**833**	**100%**

The distribution exhibits a strong asymmetry across the categories. *Proof Strategy and Decomposition* stands out as the primary category, accounting for around a third of the attributions, followed by *Invariant and Specification Refinement* and *Lemma and Hypothesis Engineering*, which are close to each other. Together, these three categories constitute 70.9% of all attributions. This concentration indicates that the most prevalent techniques center on proof structuring, and specifically, determining how to decompose goals to identify intermediate results to establish, and refining specifications to guide the solver. The remaining four categories represent important but comparatively less frequent types of strategies, with *Specialized Verification Paradigms* representing only 2.5% of attributions, reflecting the domain-specific nature of tools.

4.3 Qualitative Analysis of the Topics

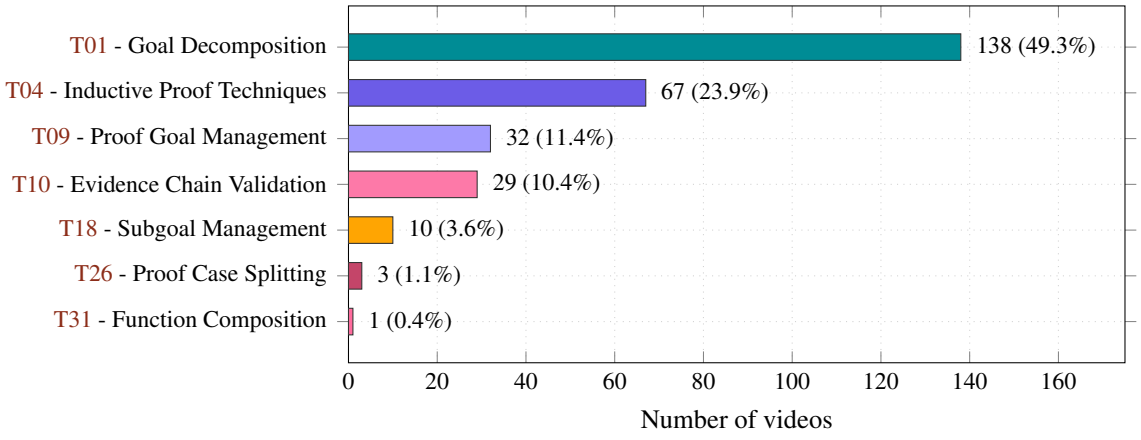
This section analyses each of the seven thematic categories in depth. For each category, we characterize the corresponding verification activities, discuss why these techniques appear necessary based on evidence from the transcripts, and relate the observed patterns back to the challenges documented in the literature in 2.5.

4.3.1 Proof Strategy and Decomposition

This category corresponds to 280 attributions across seven topics and its dominance confirms the finding of Mugnier et al. [28] that proof strategy development, the act of deciding how to structure a proof before attempting to discharge its obligations, is the most significant cognitive burden in verification-aware programming.

These seven topics range from strategic choices and organizational practice to specific structural decisions. Figure 4.3 shows the attribution breakdown within this category.

Figure 4.3: Distribution of topic attributions within the *Proof Strategy and Decomposition* category. Percentages refer to the category total ($N = 280$ attributions).



T01 – Goal Decomposition

Goal Decomposition is the most frequent technique in this category and also in the entire taxonomy, appearing in 138 of 154 videos. This prevalence reflects a structural characteristic of VA programming. Regardless of the tool or domain, practitioners cannot simply state a high-level property and expect the verifier to prove it automatically. Because solvers require granular guidance, developers are forced to identify and verify a chain of sub-goals rather than proving a high-level property in a single step.

The transcripts reveal two operationally distinct modes of goal decomposition. The first is reactive decomposition, triggered when the verifier signals an obligation it cannot verify. The practitioner inspects the error, identifies the missing intermediate fact, and adds it explicitly as a new sub-goal:

> *"[...] to identify the unproven subformula, we can trace the truth statuses evaluated by the proof."*[1]

The second mode is proactive decomposition, applied at the outset of a verification task. Here, the practitioner reasons about the logical structure of the property before attempting any verification, explicitly building the decomposition strategy:

> *"[...] there are two things we have to do, so there is a convenient little tactic* `split` *which says I would like to do these separately please."*[2]

> *"[...] we have to prove an if and only if statement [...] we have to prove the* `X` *implies* `Y` *and* `Y` *implies* `X`*."*[3]

The high frequency of this topic is also a negative signal for tool design. The fact that almost every session in the corpus requires explicit goal decomposition suggests that current VA tools do not provide sufficient automated decomposition, placing the full burden of structuring on the developer. This directly reflects the proof-strategy barrier described by Oliveira, Mendes, and Carreira [30] and the observation of Brugger, Denis, and Müller [5] that proof structuring remains highly manual and expertise-intensive.

T04 – Inductive Proof Techniques

Inductive reasoning is the second most present technique in this category, appearing in almost half of the corpus. The technique is applied whenever the program or property involves recursive data structures and it consists of selecting an induction principle, establishing a base case, and then reasoning under the inductive hypothesis. In the transcripts, practitioners are frequently observed selecting an induction variable and managing the hypothesis explicitly:

> *"[...] we can prove this lemma by structural induction on the tree [...] there is a base case and an inductive step."*[4]

[1] https://www.youtube.com/watch?v=8e-q9Jf1h_w
[2] https://www.youtube.com/watch?v=WIK1i_7lvHA
[3] https://www.youtube.com/watch?v=POHVMMG7pqE
[4] https://www.youtube.com/watch?v=p6D0RSo3TAY

> *"[...] if I want to prove that some predicate `P` is true for all natural numbers I can do it [...] if I can show that it's true for a zero and [...] for `n` plus one."*[5]

A recurring sub-pattern is the need to backtrack when the chosen induction variable turns out to be wrong, which reveals that selecting the right variable is itself a non-trivial part of the technique. This behavior is consistent with the observation of Mugnier et al. [28] that users tend to adopt non-ideal proof strategies because the tool offers no guidance during the strategy selection phase.

T09 – Proof Goal Management

Proof Goal Management captures the organization perspective of interactive proof construction. The practice of explicitly tracking which obligations remain open, in what order they should be addressed, and how far along the proof tree the developer currently is. This is necessary because a proof can quickly accumulate multiple open obligations to verify, and losing track of them can lead to inconsistent proof states.

In ITP environments, goal management is mediated by the proof state panel and developers make use of them to navigate between open goals and to determine whether a tactic application was locally successful before proceeding:

> *"[...] I have got two sub goals [...] I only get to see the full-proof context for that first sub-goal as I am working on it."*[6]

> *"[...] this is a way as a human you can focus your attention a little bit, especially if you've got a really big proof going on [...]"*[7]

Specifically, in Dafny sessions, the equivalent activity involves using the Integrated Development Environment (IDE)'s inline diagnostics to determine which assertions are accepted and which remain undischarged:

> *"[...] we can use assertions to check what Dafny knows about the function or about the method's behavior."*[8]

The prominence of this topic is consistent with the finding of Grebing, Klamroth, and Ulbrich [14], that the lack of a view linking the proof state and verification output is a significant usability barrier in current tools, forcing developers to maintain a mental model of the proof state independently of the tool.

[5] https://www.youtube.com/watch?v=VTZtuYjuLAM
[6] https://www.youtube.com/watch?v=hmcxlqs4_cU
[7] https://www.youtube.com/watch?v=hmcxlqs4_cU
[8] https://www.youtube.com/watch?v=P2durYFsJSA

T10 – Evidence Chain Validation

Evidence Chain Validation captures the practice of explicitly constructing and checking a step-by-step chain of justified equalities or implications, where each step appeals to a previously established fact or a definitional equality.

> *"[...] the goal is now exactly the three assumptions put together."*[9]

> *"[...] to get from* `a * b * c` *to* `a * b * a * c`*, that is really just invoking the left shelf property [...] in these curly brackets I am writing left because I am invoking the left shelf axiom."*[10]

Evidence chains serve a dual purpose. They guide the automated reasoner through a precise sequence of steps, while acting as human-readable documentation of the proof strategy. This aspect is particularly valuable in collaborative settings or when revisiting a proof after a code change. This aligns with Mugnier et al. [27], who frame assertion hints as a way to expose intermediate reasoning steps that solvers fail to reconstruct automatically.

T18 – Subgoal Management

Subgoal Management involves syntactically isolating the proof of each sub-goal, typically using braces or bullet points, to prevent tactics from accidentally affecting the wrong obligation:

> *"[...] when we have two sub goals [...] we close the proofs of these two sub goals into braces to not be confused and do not get mixed up [...]"*[11]

> *"[...] we are now only shown the first sub goal since we have placed our cursor into the first pair of braces [...]"*[12]

> *"[...] we have two sub goals to fill [...] we have to fill in the first argument here and we have to fill in the second argument [...]"*[13]

This topic was kept separate from T09 during manual refinement to distinguish between high-level strategy and low-level execution. *Goal Management* captures how developers plan and structure their main verification targets. In contrast, *Subgoal Management* represents the reactive process of dealing with the smaller and intermediate proof obligations that emerge during verification. Separating these two concepts allows us to isolate the exact source of the developer's cognitive burden, showing that much of the effort is spent managing secondary subgoals..

The scoping discipline enforced by this technique mirrors good software engineering practice, that is by making scope explicit, the developer reduces the risk of unintended side effects from

[9] https://www.youtube.com/watch?v=0QZI_m8WZ0Q
[10] https://www.youtube.com/watch?v=eXJEPRzZF50
[11] https://www.youtube.com/watch?v=VHpwZKw_qGg
[12] https://www.youtube.com/watch?v=VHpwZKw_qGg
[13] https://www.youtube.com/watch?v=8JIhJJyO5VM

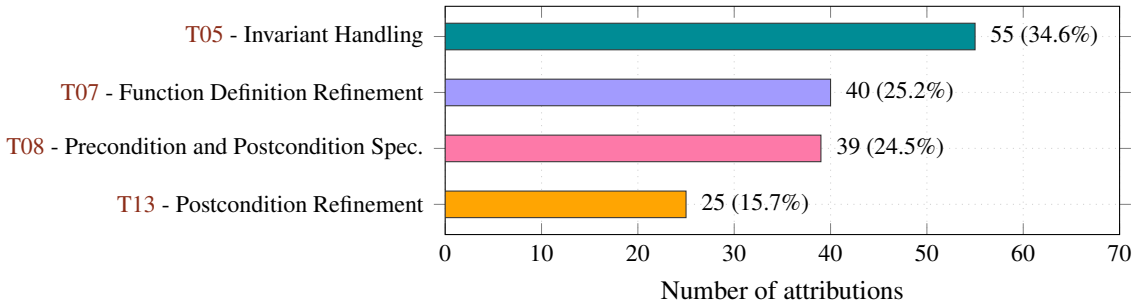
tactic application and makes the proof structure more maintainable. It also relates to the scalability concern reported by Mugnier et al. [28], where larger proof contexts make obligations harder to discharge unless developers actively constrain and organize the proof state.

The remaining low-frequency topics T26 and T31 are also structural, but appear too rarely to warrant separate discussion. T26 covers explicit case splitting when a proof depends on a disjunction or a datatype distinction, while T31 captures the modular composition of already verified functions or intermediate results. Their low frequency suggests that these practices are either subsumed into broader goal decomposition work or appear only in more specialized proof situations. Nevertheless, both connect to the proof-strategy selection problem discussed by Mugnier et al. [28] and to the maintainability concerns raised by Brugger, Denis, and Müller [5], since choosing the wrong decomposition or composing poorly specified components can increase proof repair effort.

4.3.2 Invariant and Specification Refinement

This category accumulates 159 attributions (19.1%) across the topics T05, T07, T08 and T13. Together, these topics capture the iterative process through which practitioners tighten the formal description of program behavior until the verifier can confirm correctness. The contrast between T08 (initial specification) and T13 (subsequent refinement) is particularly analytically meaningful, since it reveals that specification is rarely a one-shot activity but a feedback loop driven by verifier rejections. Figure 4.4 shows the attribution breakdown within this category.

Figure 4.4: Distribution of topic attributions within the *Invariant & Specification Refinement* category. Percentages refer to the category total ($N = 159$ attributions).



T05 – Invariant Handling

Loop invariants are the most frequently discussed specification artefact in this category, appearing in more than a third of the data. This is expected given that loops are the most common constructs in imperative programs and that, unlike type checkers, automated verifiers cannot infer loop invariants on their own. The developer must explicitly define, for each possible loop, a predicate that the solver can verify holds before the loop begins, is maintained by each iteration, and is strong enough to imply the desired postcondition at the end.

The transcripts confirm that practitioners encounter invariants as an iterative strengthening process rather than a one-time activity:

> *"[. . .] if we run this it should say post condition not satisfied [. . .] because we have not put it in the loop invariant it does not know that, but if we added it to a loop invariant like this then we would know this [. . .] we should be able to verify that."*[14]

> *"[. . .] now we do not get the post-conditional not satisfied error anymore because our loop invariant is enough to allow Z3 to prove that [. . .]"*[15]

A particularly instructive moment appears in a transcript about a model checker detecting a violated invariant-a situation qualitatively different from the ITP and auto-active contexts, yet requiring the same fundamental response:

> *"[. . .] the model checker should spot it, in fact it says that invariant is violated."*[16]

The common thread across all tool types is that an invariant violation is the trigger for a strengthening iteration. The developer adds another condition to the invariant, re-runs verification, and checks whether the new formulation is both inductive and strong enough. This cycle can repeat many times before convergence, and the burden of finding the right strengthening is entirely on the developer, a direct connection with the "steep learning curve" identified by Oliveira, Mendes, and Carreira [30] and Juhošová, Zaidman, and Cockx [18].

T07 – Function Definition Refinement

Function Definition Refinement captures the need to revise, not the implementation logic, but the mathematical definition of a function, its formal specification, so that the verifier can reason about it effectively. This requirement arises from the abstraction challenge in VA programming, where it is desirable for modularity, but automated solvers need to expand definitions to verify properties about them.

Transcripts show programmers navigating this tension in three ways. The first is definitional restructuring, where programmers rewrite a function's recursive clauses so that the pattern of recursion matches the expected induction principle:

> *"The takeaway that I would like you to get from today's lecture is that definition by recursion and proof by induction are the same thing [. . .] when we defined addition, plus of successor of M and N is defined in terms of plus of M and N [. . .] that's called structural induction."*[17]

> *"We would unfold the definition of plus and say that if the first argument is the successor of some other number then the result is the successor of that number plus two [. . .] it's a recursive definition."*[18]

[14] https://www.youtube.com/watch?v=-ULEDYUxwtY
[15] https://www.youtube.com/watch?v=7WtWA0TTBqg
[16] https://www.youtube.com/watch?v=D_sh1nnX3zY
[17] https://www.youtube.com/watch?v=9yplm_dsQHE
[18] https://www.youtube.com/watch?v=jl1npW7b_2E

The second is opaqueness management, where programmers add or remove opaque or ghost annotations to control whether the solver can see the definition body:

> *"[. . .] if you ever see a goal that has this identifier in it, then you unfold that identifier and then just keep working [. . .] telling it to unfold will now allow it to get the goal where it can actually match against this lemma and apply it [. . .]"*[19]

The third is *type-driven refinement*, where programmers adjust the return type of a function to carry additional proof-relevant information, as in the introduction of dependent types or refinement types:

> *"[. . .] when you make your result type instead of just being some type it's a proposition that depends on the value you're folding [. . .]"*[20]

In all three variants above, the change is made not because the function is wrong but because its current formulation is opaque to the verifier, an important distinction that novice users often fail to make. This is an example of the semantic gap identified by Wang, Scazzariello, and Chiesa [38], in which they refer that mathematically equivalent definitions can differ substantially in how easy they are for a verifier to unfold, match, and reason about.

T08 – Precondition and Postcondition Specification
Defining explicit method contracts is one of the most fundamental activities in VA programming, and its high frequency reflects the fact that virtually every non-trivial verification task requires at least one *requires* or *ensures* clause. The transcripts reveal that practitioners often discover what preconditions are necessary only after the verifier rejects a caller:

> *"[. . .] there is an assertion that states that the argument to the method is not equal to literal c, but then there is no precondition that would mandate that the argument cannot be equal to literal c."*[21]

> *"[. . .] preconditions or postconditions are things that you specify about your program [. . .] the code is correct if we assume the precondition [. . .] the code executes then the postcondition is verified."*[22]

The second quote represents when programmers identify a runtime assumption implicit in the code and must translate it into a formal precondition that the verifier can use, which is straightforward in simple cases but becomes non-trivial when the required precondition involves complex relational properties over heap-allocated structures. This connects with the learning-curve barriers described by Oliveira, Mendes, and Carreira [30] and Juhošová, Zaidman, and Cockx [18], because users must learn to express assumptions that are implicit in ordinary programming as explicit logical contracts.

[19] https://www.youtube.com/watch?v=sNjHzNkTMTM
[20] https://www.youtube.com/watch?v=--TIMdy2jow
[21] https://www.youtube.com/watch?v=8UMAACjLejU
[22] https://www.youtube.com/watch?v=18QI1TNM9os

T13 – Postcondition Refinement

Beyond the initial specification of contracts, practitioners were consistently observed strengthening postconditions in response to verifier feedback, adding conditions that capture properties the verifier needs downstream but that the developer had not initially anticipated:

> *"This means the postcondition could not be proven to hold."*[23]

> *"[. . .] we need to make sure that the list really is acyclic."*[24]

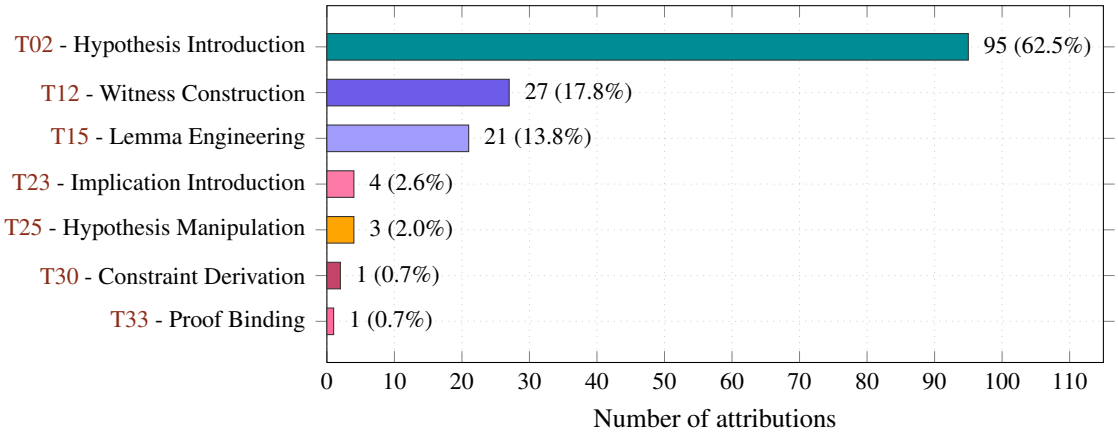
> *"[. . .] we can ensure that the value of the result according to the standard definition is the value of the sequence [. . .]"*[25]

The distinction between T08 and T13 is important, because T08 refers to the initial act of specifying what a function is supposed to do, while T13 captures the iterative act of strengthening a contract in response to a verifier rejection. T08 is driven by design intent and T13 is driven by proof failure. This feedback-driven refinement echoes the continuous verification loop discussed by Silva, Mendes, and Ferreira [34] and Wang, Scazzariello, and Chiesa [38], where failed proof attempts become information for revising the formal artefact.

4.3.3 Lemma and Hypothesis Engineering

This category groups techniques that build the auxiliary mathematical infrastructure needed to bridge the gap between what the verifier needs and what it can derive on its own. These techniques introduce new facts into the proof context, so that the solver has enough material to complete the proof. Figure 4.5 shows the distribution across the seven topics in this category, dominated by T02 which accounts for 62.5% of all attributions.

Figure 4.5: Distribution of topic attributions within the *Lemma and Hypothesis Engineering* category. Percentages refer to the category total ($N = 152$ attributions).



[23] https://www.youtube.com/watch?v=8e-q9Jf1h_w
[24] https://www.youtube.com/watch?v=kbJO-U9Wp-s
[25] https://www.youtube.com/watch?v=P2durYFsJSA

T02 - Hypothesis Introduction

With the most assignments within this category and with the second most assignments in the entire taxonomy, *Hypothesis Introduction* consists of explicitly bringing an assumption, previously established fact, or intermediate result into the proof context so that the solver can use it in subsequent steps. This technique is prevalent because automated solvers operate within a strictly bounded context. They are blind to prior or external facts unless those properties are explicitly brought into their immediate scope:

> *"[...] the tactic corresponding to assume is called intro for introduction [...] we start this proof by introducing the assumption* `a`*, which changes our goal statement."*[26]
>
> *"[...] we now have the assumption* `a`*,* `tiny a`*, of which is a proof of the proposition* `a` *in the hypothesis [...]"*[27]
>
> *"[...] we are being shown only the first subgoal, which is to prove* `a`*, knowing that we have a proof of* `a` *in our toolbox [...] we will just apply our hypothesis."*[28]

This pattern is another manifestation of the incomplete-automation challenge described by Mugnier et al. [27]: the relevant fact exists, but it must be explicitly introduced into the local proof context before automation can use it.

T12 - Witness Construction

Witness Construction is used when the goal requires proving an existential statement, that some value satisfying a property exists. Rather than letting the solver search for a witness, the practitioner provides one explicitly:

> *"[...] we can see that 10 is even only if there exists a number* `r` *such that 10 equals* `r + r`*. Clearly* `r` *should be five. So let's use five and the proof is over."*[29]
>
> *"Notice that* `use 1` *substitutes* `R` *as 1. But we can't prove this. It's false. What we just saw is the introduction of an exist statement."*[30]

This pattern connects to the semantic gap identified by Wang, Scazzariello, and Chiesa [38]: facts that are obvious in an informal mathematical proof, such as the value that witnesses an existential claim, must be made explicit before the verifier can check them. It also illustrates the incomplete-automation problem discussed by Mugnier et al. [27], since the solver can verify the witness once supplied but often cannot infer the intended existential instantiation on its own.

[26] https://www.youtube.com/watch?v=1PVN6lRjlv8
[27] https://www.youtube.com/watch?v=1PVN6lRjlv8
[28] https://www.youtube.com/watch?v=1PVN6lRjlv8
[29] https://www.youtube.com/watch?v=0QZI_m8WZ0Q
[30] https://www.youtube.com/watch?v=0QZI_m8WZ0Q

T15 - Lemma Engineering

Lemma Engineering involves extracting a property into a separately named and verified lemma that can then be referenced in the main proof. This happens when the same property is needed in multiple places, or when the inline proof attempt fails and the developer diagnoses that a missing intermediate theorem is the root cause. This topic maps directly to the missing-lemma problem studied by Silva, Mendes, and Ferreira [34], and to the broader need for intermediate proof artefacts highlighted by Mugnier et al. [27]:

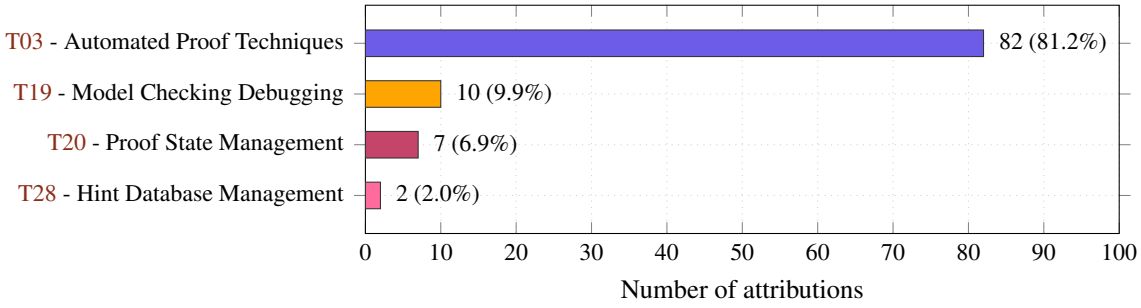
> *"[. . .] let's create a new lemma and what we want to make is a statement about refack with an access and an arbitrary accumulator."*[31]
>
> *"[. . .] we're going to want to figure out what the loop actually does to R and then encode that in invariants in the loop."*[32]

The remaining topics T23, T25, T30, and T33 are low-level moves that appear in highly interactive ITP sessions, reinforcing the observation by Grebing, Klamroth, and Ulbrich [14] that developers need fine-grained visibility and control over proof state evolution. T23 involves introducing the antecedent of an implication as an assumption. T25 involves adjusting or specialising an existing hypothesis to match the current goal. T30 derives additional constraints from the types or definitions in scope. and T33 establishes explicit bindings between separate parts of a proof.

4.3.4 Solver Automation and Hinting

This category groups techniques through which practitioners interact with the automation layer of the verifier, that is configuring and guiding automated tactics rather than performing all reasoning manually. The category is strongly dominated by T03, which reflects the fact that automation is present in all tools in the corpus and must be actively managed in every session. Figure 4.6 shows the distribution across the three topics in this category.

Figure 4.6: Distribution of topic attributions within the *Solver Automation and Hinting* category. Percentages refer to the category total ($N = 101$ attributions).



[31] https://www.youtube.com/watch?v=93ihwerN93M
[32] https://www.youtube.com/watch?v=P2durYFsJSA

T03 - Automated Proof Techniques *Automated Proof Techniques* refers to the invocation and configuration of automated tactics and solvers. Practitioners select specific tactics suited to the goal at hand, restrict the set of lemmas available to the tactic, and combine automated steps with manual ones:

> *"I will now execute the 'Full Auto Pilot' macro which verifies each formula of the proof obligation in a different branch."*[33]
>
> *"[...] the tactic mode in lean is started with the keyword begin and terminated with the keyword end."*[34]

A common pattern across transcripts is the alternation between a manual step that transforms the goal into a standard form and an automated tactic that closes it. This reveals that automation works best not as a standalone strategy but as the final move in a sequence that the developer has carefully set up.

T19 - Model Checking Debugging *Model Checking Debugging* consists of analysing counterexample traces produced by a model checker to diagnose whether a specification error or a genuine model violation has been detected. Unlike a failed proof in an ITP, a model checking counterexample is an execution trace that must be replayed and interpreted step by step:

> *"[...] the model checker should spot it in fact, it says that invariant is violated and, what is more useful, it gives us a trace [...] the shortest sequence of states leading to a state where the invariant does not hold."*[35]
>
> *"[...] not only does it tell us that deadlock exists, it gives us a trace we could follow and explore what it is doing [...]"*[36]
>
> *"[...] what we actually want is this trace here that shows us what went on. Fortunately, TLC generates this trace [...]"*[37]

In traditional deductive verification, developers face the steep hurdle of manually guiding the solver and checking granular proof obligations. Model checking mitigates this by automating the verification process itself. Consequently, as observed in the transcripts, the developer's effort is redirected from fighting the solver to correctly formulating the property itself, making specification writing the primary activity.

T20 - Proof State Management *Proof State Management* involves explicitly controlling the visibility and organisation of the proof context. Practitioners hide exhausted hypotheses, focus the prover on a single subgoal, and use structured proof blocks to prevent context pollution:

[33] https://www.youtube.com/watch?v=IV-dEnpCLkI
[34] https://www.youtube.com/watch?v=1PVN6lRjlv8
[35] https://www.youtube.com/watch?v=D_sh1nnX3zY
[36] https://www.youtube.com/watch?v=H6PjGdd6vGg
[37] https://www.youtube.com/watch?v=wjsI0lTSjIo

> *"[. . .] as soon as I put the cursor inside that begin and block there's this display over on the right half of the screen where the computer is telling me what it thinks is going on."*[38]

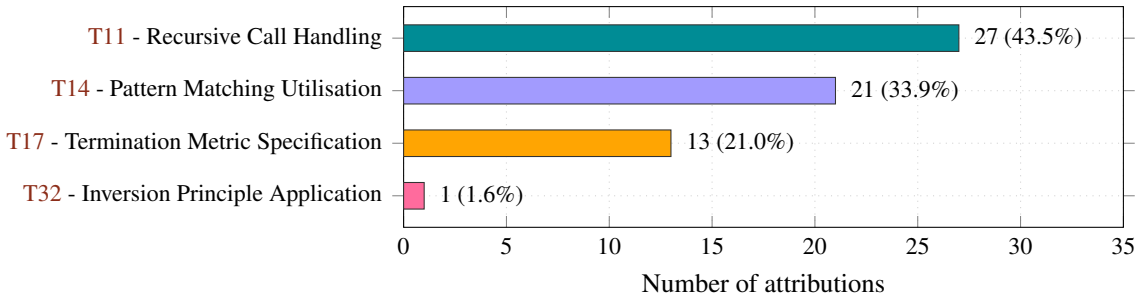
The remaining low-frequency topic T28 captures hint database management, such as adding specific lemmas to automation contexts so that later tactic calls can use them without explicit invocation.

The automation management techniques in this category directly address the challenge documented in the literature that automated verifiers act as opaque systems whose behavior is difficult to predict and control [5].

4.3.5 Termination and Recursion Handling

This category groups techniques that address the specific challenges introduced by recursive functions and loops, where programmers have to convince the verifier that the computation terminates. These techniques arise only when the program under verification contains recursion or iteration, which explains their lower frequency compared to the universally applicable techniques of the preceding categories. Figure 4.7 shows the distribution across the four topics in this category.

Figure 4.7: Distribution of topic attributions within the *Termination and Recursion Handling* category. Percentages refer to the category total ($N = 62$ attributions).



T11 - Recursive Call Handling *Recursive Call Handling* involves making the structural properties of recursive calls explicit so that the verifier can accept the function definition and reason about it. This includes specifying how the argument decreases at each call site or reorganising the recursive structure to match the shape of the induction principle being used:

> *"[. . .] the thing that decreases with every call is the parameter n."*[39]

> *"[. . .] we want to define this function that I'm calling count by threes recursively."*[40]

[38] https://www.youtube.com/watch?v=WIK1i_7lvHA
[39] https://www.youtube.com/watch?v=kbJO-U9Wp-s
[40] https://www.youtube.com/watch?v=zaz7TQItWDk

T14 - Pattern Matching Utilisation *Pattern Matching Utilisation* leverages the case analysis inherent in pattern matching to structure proofs that follow the shape of the data. When a function is defined by pattern matching on a constructor, the proof of its properties can often be structured by matching on the same constructor, which automatically creates one proof obligation per case:

> *"[...] so I'm going to do pattern match so the keyword is match and the parameter you want to match."*[41]

> *"[...] when we pattern match on one of these vectors we refine its type in particular its length index based on which constructor we actually matched."*[42]

> *"[...] we are pattern matching on the first argument."*[43]

T17 - Termination Metric Specification *Termination Metric Specification* consists of providing an explicit decreasing measure - a function of the program state that strictly decreases at each recursive call or loop iteration - so the verifier can confirm that the computation always terminates. In some tools this is expressed via a `decreases` clause and in others the termination order may be inferred or must be guided:

> *"[...] we need to help the tool otherwise this verification is flawed. So that's the reason why we need to introduce a decreases as part of the specification."*[44]

> *"[...] we associate with it a function which is called the variant function or a bound function or a termination metric."*[45]

> *"[...] isabelle found a termination order."*[46]

The remaining low-frequency topic T32 concerns inversion principle application, where the practitioner extracts structural information from an inductive hypothesis by reasoning backwards from how that hypothesis could have been constructed. Its rarity suggests that this is a specialized ITP move rather than a general recursion-handling strategy.

The techniques in this category demonstrate that recursion introduces unique challenges, such as proving termination and ensuring structural correctness. Consequently, practitioners must use targeted and manual strategies to guide the verifier, highlighting a distinct layer of interaction that is absent in non-recursive programs [30].

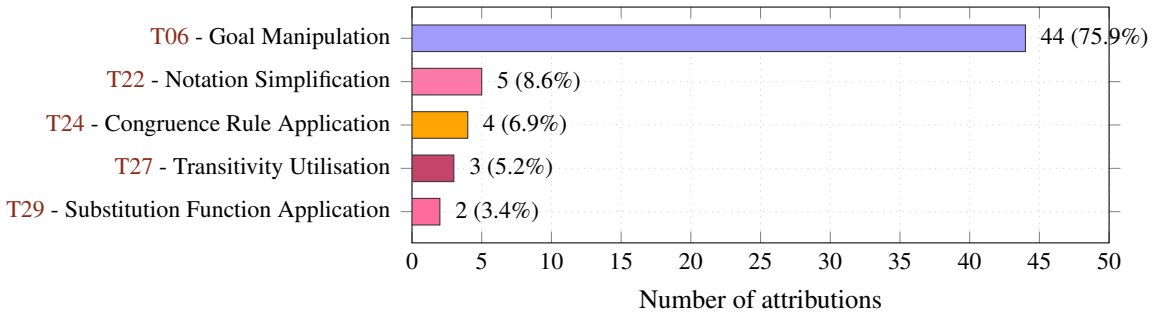
4.3.6 Proof Rewriting and Algebraic Manipulation

This category groups techniques that transform proof goals into logically equivalent but syntactically different forms. The underlying motivation is that the solver's ability to reach a goal often

[41] https://www.youtube.com/watch?v=fOKfk9Bup_o
[42] https://www.youtube.com/watch?v=z_kaJ4CQztw
[43] https://www.youtube.com/watch?v=SWNDAH_k710
[44] https://www.youtube.com/watch?v=2JGdhKU7eIs
[45] https://www.youtube.com/watch?v=kbJO-U9Wp-s
[46] https://www.youtube.com/watch?v=93ihwerN93M

depends on its syntactic shape a goal that is logically simple may be expressed in a form the solver cannot recognize, and rewriting brings it into a form the automation can handle. Figure 4.8 shows the distribution across the five topics in this category, dominated by T06.

Figure 4.8: Distribution of topic attributions within the *Proof Rewriting and Algebraic Manipulation* category. Percentages refer to the category total ($N = 58$ attributions).



T06 - Goal Manipulation Manipulating goals refers to rewriting using equations, unfolding definitions, and applying simplification steps to transform the current goal. The technique is often used as a preparatory move before invoking an automated tactic:

> *"[. . .] our goal statement has changed from B to A."*[47]
>
> *"Now, I will apply rule 'orRight' to improve readability."*[48]

T22 - Notation Simplification Simplifying notation involves introducing abbreviations or adjusting notation to make proof goals easier to read and manipulate, particularly when goals involve complex type coercions or nested expressions:

> *"[. . .] we can make this a little bit nicer if we added some notations for set operations like intersections, unions, and all that [. . .]"*[49]
>
> *"[. . .] I already have defined some notations [. . .] you can define arbitrary notations [. . .] these are kind of things that I would like to have here as well but with some slight modifications."*[50]

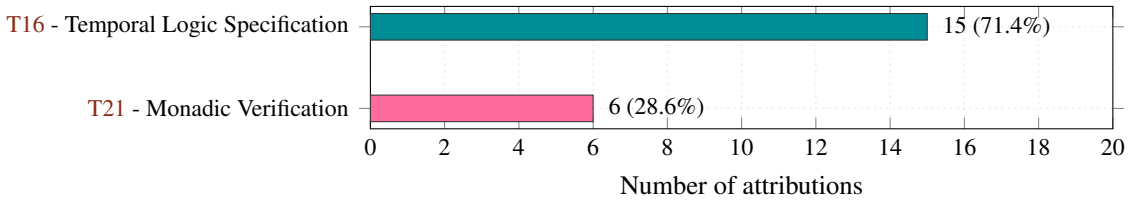
The remaining low-frequency topics T24, T27, and T29 are more low-level moves that appear as supporting steps within larger proofs. T24 applies congruence reasoning, using the fact that equal arguments remain equal under the same function. T27 chains equalities or order relations through transitivity. T29 substitutes a term by an equal term to reshape the goal. Their low counts suggest that these techniques are rarely the central strategy of a session, but they are important local transformations that help bring goals into the syntactic form expected by the solver.

[47] https://www.youtube.com/watch?v=vIPdHRZ6q8o
[48] https://www.youtube.com/watch?v=IV-dEnpCLkI
[49] https://www.youtube.com/watch?v=nOzN6xXj6fc
[50] https://www.youtube.com/watch?v=nOzN6xXj6fc

4.3.7 Specialized Verification Paradigms

This is the smallest category, with 21 attributions, and it groups techniques that are specific to particular verification paradigms rather than transferable across all VA tools. Their lower frequency reflects the narrower set of tools and tasks they apply to, not a lower level of importance within those contexts. Figure 4.9 shows the distribution across the two topics in this category.

Figure 4.9: Distribution of topic attributions within the *Specialized Verification Paradigms* category. Percentages refer to the category total ($N = 21$ attributions).



T16 - Temporal Logic Specification *Temporal Logic Specification* involves expressing safety and liveness properties using temporal logic, a technique central to model checking tools. This approach requires the developer to reason about continuous sequences of system states rather than individual, isolated function executions.

> *"[...] temporal logic which is actually the backbone of TLA plus."*[51]

T21 - Monadic Verification *Monadic Verification* shows up in sessions where the code under verification uses monadic structures to manage effects, and the practitioner must use monad-specific reasoning principles to establish correctness properties:

> *"[...] we've just constructively proven that the mono that we've defined will always uphold these laws."*[52]
>
> *"[...] the simplicity comes when you have to deal with bindings."*

[53]

While highly focused, this category reflects the specialized nature of the tools and domains it addresses. Within certain environments such as TLA+ and Lean, these techniques are just as crucial as goal decomposition is to general VA programming [16].

4.4 Co-occurrence Analysis of Verification Techniques

The qualitative analysis presented in Section 4.3 characterised each of the 33 identified techniques in isolation. However, the section already established that the average video is assigned in average

[51] https://www.youtube.com/watch?v=D_sh1nnX3zY
[52] https://www.youtube.com/watch?v=P82dqVrS8ik
[53] https://www.youtube.com/watch?v=7u-jx1UyRmc

5.41 topics, indicating that it is rare when practitioner rely on a single technique to resolve a verification obstacle. To complement the previous analysis, this section examines how techniques co-occur within the dataset, with the goal of discovering combinations that function as composite problem-solving patterns rather than independent strategies.

For every pair of topics, we computed the number of videos in which both topics were jointly assigned, the Jaccard similarity coefficient [37] between their video-level assignment sets, and the lift [4], defined as the ratio between the observed co-occurrence frequency and the frequency expected under the assumption that the two topics are assigned independently. A lift value above 1 indicates that two techniques co-occur more often than chance would predict, whereas a lift below 1 indicates that they tend to be applied in mutually exclusive contexts. In addition, frequent itemset mining was applied to the topic-assignment sets to identify the most common combinations of two and three techniques across the corpus.

4.4.1 Pairwise Associations Between Techniques

Figure 4.10 presents the full pairwise Jaccard similarity matrix between the techniques. The matrix reveals a block concentrated in the upper-left region, corresponding to the topics T01 to T05, all belonging to the *Proof Strategy and Decomposition* and *Invariant and Specification Refinement* categories. With most pairs with a substantially above the baseline pairwise similarity observed in the rest of the matrix, this cluster confirms that the most frequent techniques in the taxonomy are rarely individually prevalent, but are also habitually applied together within the same verification session. It also reinforces the result reported in Section 4.2, that each video is assigned a mean of 5.41 topics, and suggests that this multiplicity is not randomly distributed across the taxonomy.

Beyond this main cluster, a smaller but visually distinct block is present in the matrix, connecting T16 and T19. The block is more localised than the main one, suggesting that these techniques form tightly coupled, paradigm-specific pairings rather than being generally combinable with the rest of the taxonomy. This pattern is consistent with the discussion in Section 4.3, where T16 and T19 were both attributed to the *Specialized Verification Paradigms* category and tend to appear exclusively in model-checking-oriented sessions.

Outside the main cluster, the matrix is dominated by low similarity values, below 0.15, indicating that most of the techniques are applied independently of one another across the corpus. This can be seen as a positive signal for the validity of the taxonomy, since it might suggest that the topic refinement process succeeded in producing techniques that are conceptually distinct rather than near duplicates of one another, since strongly redundant categories would be expected to exhibit inflated pairwise overlap.

An exception to this sparsity is the pair T16 and T19, which displays an elevated similarity of approximately 0.4, despite both techniques having comparatively not so high attribution counts. This is consistent with the discussion in 4.3.7, where it was mentioned that both techniques are tied to the same subset of model-checking-oriented videos.

A smaller number of weak or moderate similarities around 0.15 to 0.30 can also be observed in two areas of the matrix. The first is an intermediate zone still within the upper-left quadrant,

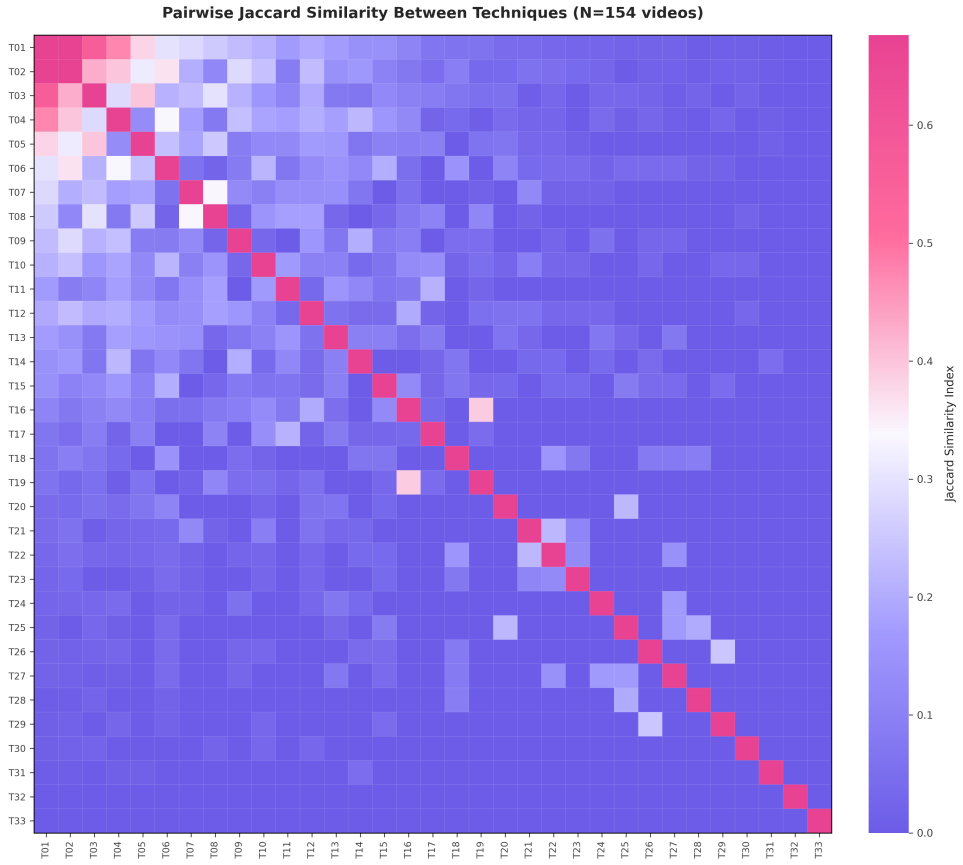


Figure 4.10: Pairwise Jaccard similarity between the 33 identified techniques (N = 154 videos).

extending further than the core cluster into techniques T07, T08, T09, T10, T11 and T12. These techniques exhibit moderate cooccurrence with one another and with members of the core cluster. This is consistent with the qualitative findings in Sections 4.3.2 and 4.3.3. These techniques are associated with the proof construction workflow, but are more triggered rather than being present in virtually every session as T01 through T04 are.

The second region of moderate similarity is present in the lower-right portion of the matrix. Given that these techniques have a lower attribution count, those values should be interpreted with caution. With such small denominators, the Jaccard index is highly sensitive to individual videos, and a single shared attribution can produce a visually salient cell without necessarily reflecting a generalizable relationship.

4.4.2 Recurrent Technique Combinations

The analysis in the previous section normalizes the overlap between two techniques by the size of their combined assignment sets, making it useful for detecting pairs that are unusually tightly connected even when they are not frequent in absolute terms. By contrast, the present analysis focuses on absolute support. It questions which combinations occur in the largest number of videos, regardless of whether the techniques are individually common, and also extends the analysis beyond

pairs to include triples. To quantify which combinations of techniques are most representative of practitioner behavior, Figure 4.11 reports the twenty most frequent technique pairs and the fifteen most frequent technique triples, ranked by the number of videos in which the combination appears.

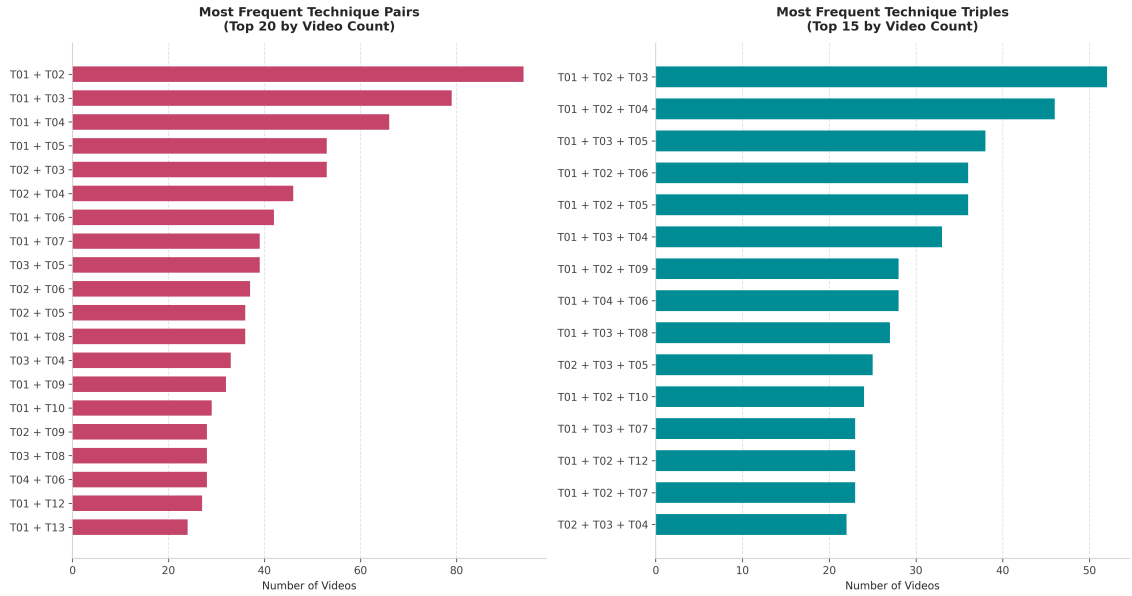


Figure 4.11: Most frequent technique pairs and triples applied across the dataset.

The combination of T01 and T02 is the most recurrent pairing in the corpus, co-occurring in 94 videos. Followed by T01 and T03 in 79 videos, and T01 and T04 in 66. The dominance of T01 across nearly all top-ranked pairs and triples is expected given its individual prevalence, but the consistently high co-occurrence values indicate that goal decomposition rarely is an individual action.

At the level of triples, the most frequent combination is T01, T02 and T03 with 52 videos, followed by T01, T02 and T04 with 46. These results suggest a three-step pattern, where a goal is decomposed, a hypothesis or assumption relevant to the sub-goal is introduced into the local context, and the resulting obligation is either discharged automatically or by an inductive argument. Rather than showing that the proof-strategy bottleneck is solved, this composite pattern indicates how practitioners respond to the challenge in practice. They rely on recurring sequences of proof decomposition and context enrichment to make verification obligations more manageable. The observed combinations connect the techniques identified in this study to the challenges reported by Mugnier et al. [28] and Oliveira, Mendes, and Carreira [30], without implying that those challenges disappear entirely.

4.4.3 Relative Lift Analysis

While the previous ranking favours common techniques, lift highlights combinations that are disproportionately associated with each other relative to their individual frequency, even if rare in

absolute terms. Figure 4.12 shows the twenty technique pairs with the highest lift values observed in the corpus.

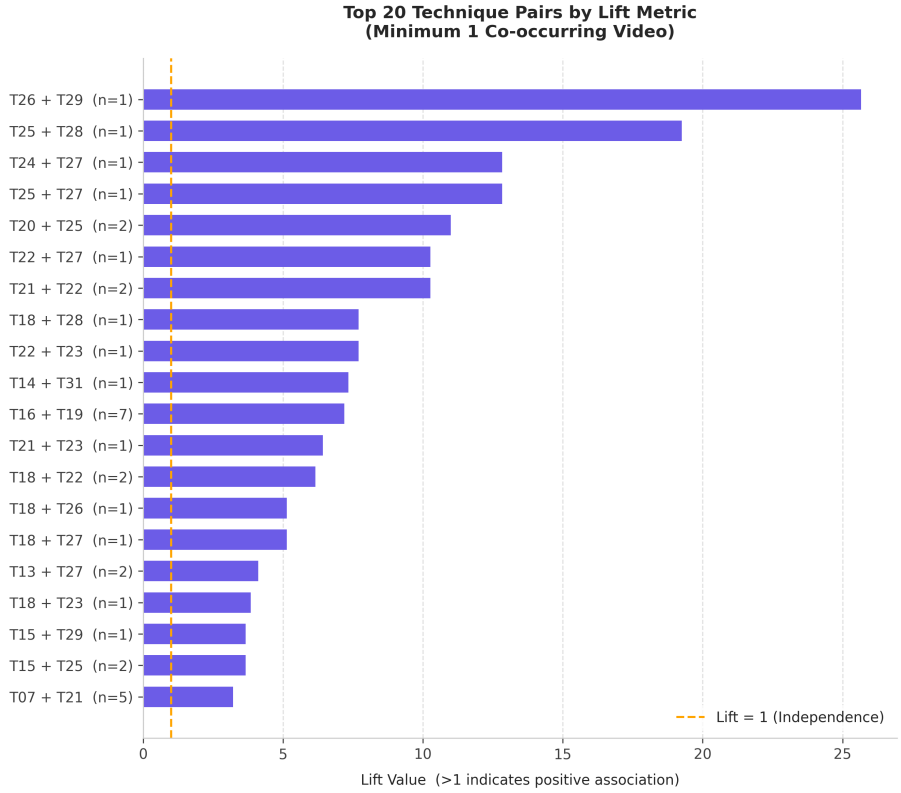


Figure 4.12: Top 20 technique pairs by lift, with the number of co-occurring videos (n) reported. The dashed line marks lift = 1.

Most of the highest-lift pairs are supported by only one or two co-occurring videos, which limits the extent to which they can be interpreted as established practitioner patterns rather than incidental occurrences specific to a single session. The exception is the pairing of T16 and T19, which combines a comparatively high lift of 7.19 with a higher support of seven co-occurring videos, out of only ten and fifteen videos in which T19 and T16 appear individually, respectively. This indicates that model-checking debugging is almost never performed independently of specifying the temporal property under inspection, already suggested by the heatmap in Figure 4.10.

In summary, the co-occurrence analysis demonstrates that the identified verification techniques are not applied as independent actions. The data reveals a clear structure, where a predominant core of general techniques that are frequently used together, with a few tightly bound, domain-specific pairs. This co-occurrence mapping provides a more granular answer to RQ2 than the category-level distribution presented in Section 4.2.

4.5 Comparison of Verification Paradigms

As discussed in Section 3.2.1, the two ITPs Rocq and Lean 4 were included in the YouTube search string along with the verification-aware languages. This section investigates whether that assumption holds, by comparing the distribution of identified techniques between the ITP subset, summing 53 videos, and the verifiable programming languages.

4.5.1 Category Distribution Across Paradigms

Figure 4.13 compares the relative distribution of attributions across the seven thematic categories defined in Section 4.2, normalized within each paradigm to compare directly, independently of the difference in sample size between the two groups. Videos were classified as belonging to the ITP or the verification-aware language group according, once again, to the tool support obtained from the Formal Methods Europe association.

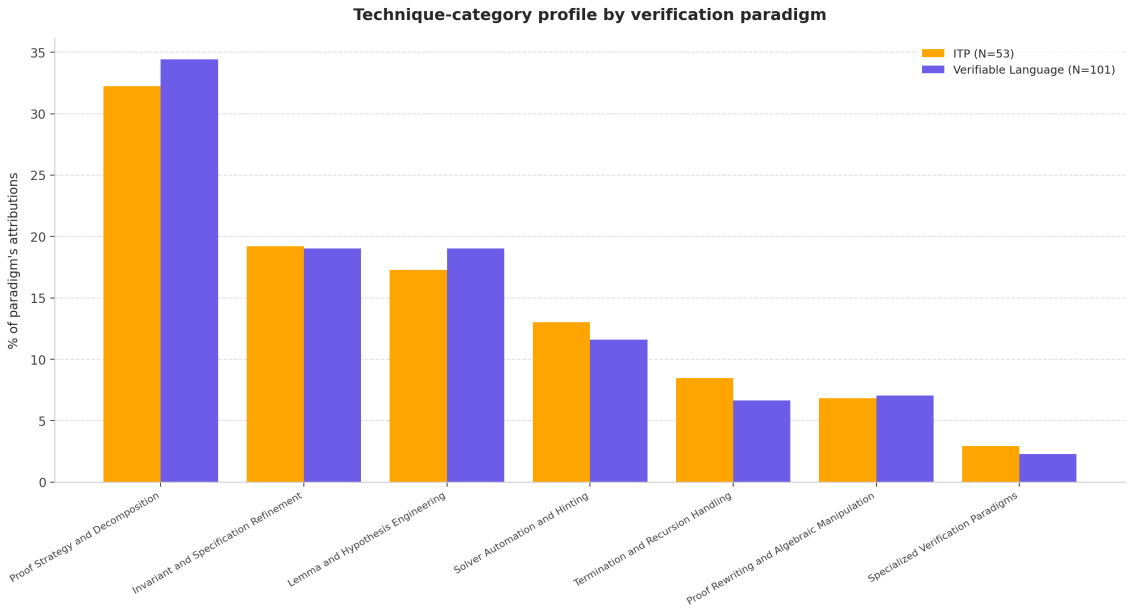


Figure 4.13: Technique-category profile by verification paradigm

Both paradigms demonstrate similar patterns across the different categories and the relative ranking of all seven categories is preserved across paradigms. This similarity provides direct empirical support for the methodological decision to treat ITP and verifiable languages interaction as transferable practices. Practitioners channel their effort to the same broad categories of activity, in comparable proportions.

4.5.2 Top Techniques by Paradigm

Figure 4.14 compares the ten most frequently assigned techniques within each paradigm.

The top five techniques are the same and almost identically ranked in both paradigms, reinforcing the category-level convergence observed above. The two paradigms diverge, however, in their

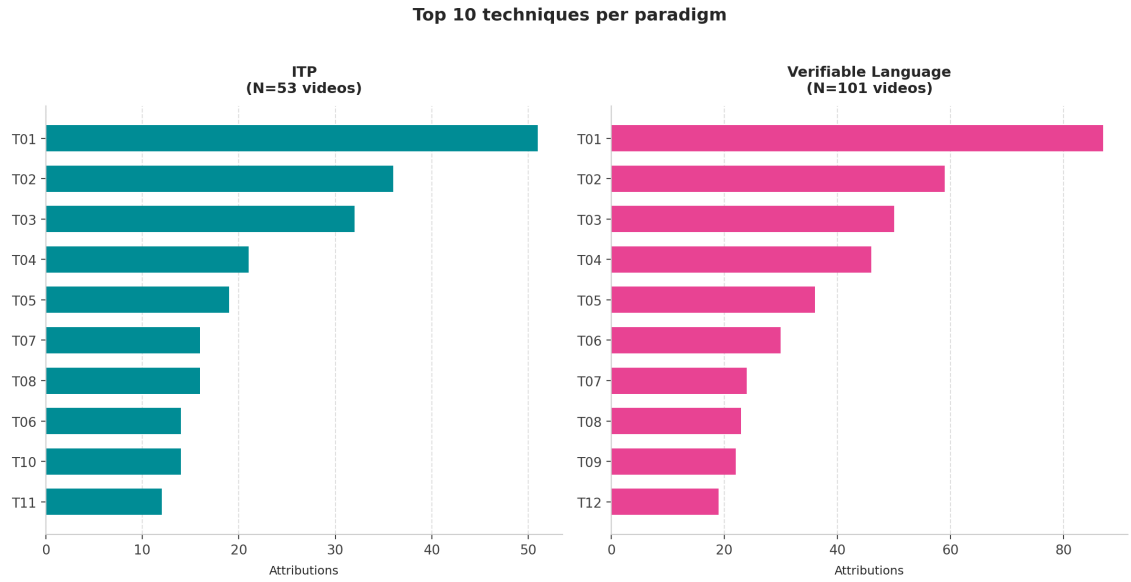


Figure 4.14: Top 10 techniques per verification paradigm, by attribution count.

second half of the ranking. The ITP subset includes T10 and T11 among its top ten techniques, neither of which appears in the equivalent verifiable languages ranking, explained by the fact that are both connected to the interaction style of Rocq and Lean. Also, the verifiable languages subset includes T09 and T12 among its top ten, neither of which appears in the ITP ranking.

Overall, the comparison indicates that the choice between an ITP and a verifiable languages does not significantly affect the main strategies applied by practitioners, which remain highly consistent regardless of the paradigm. Consequently, these verification patterns are overall generalizable across both ecosystems.

4.6 Conclusion

This chapter shows that verification-aware programming is best understood as a coordinated process, where practitioners combine several techniques to make verification tasks manageable. Rather than treating each technique as an isolated action, the analysis highlights how they form recurring patterns of interaction with the verifier.

These findings provide the basis for answering RQ2 and for connecting the observed practices to the challenges identified in the literature. Chapter 5 builds on this interpretation by validating and refining the results presented here.

Chapter 5

Evaluation and Validation

5.1 Introduction

This chapter describes the design of the survey developed to evaluate and validate the outcomes of Chapter 4. The survey aims to validate the taxonomy of verification strategies identified through the topic modeling pipeline applied to YouTube video transcripts, providing empirical grounding for the outcomes that address RQ2. Furthermore, it gathers practitioner perspectives on tooling limitations and learning resources, contributing evidence towards RQ3.

This chapter begins with Section 5.2, which presents the validation methodology. Section 5.3 then describes the target audience for participation. It is followed by Section 5.4, which describes the design and organization of the survey used in the current phase of the study and details the purpose of each section. Section 5.5 then discusses the strategy used to recruit participants. Finally, Section 5.6 analyses the collected responses and interprets the results, and Section 5.7 concludes the chapter by synthesizing its main contributions.

5.2 Validation Methodology

The validation stage was designed as an empirical analysis of the taxonomy of verification strategies presented in Chapter 4, derived from the topic modeling pipeline. The purpose of the validation is to determine whether those strategies are recognizable and meaningful to developers with experience in verification-aware programming. To achieve this, an online survey was adopted as the validation instrument. Surveys are widely adopted as validation instruments in software engineering research [26], enabling the collection of data from practitioners with relevant domain expertise. They can also be easily distributed across global communities, allowing participants from different locations and professional contexts to contribute to the study.

To reduce nonresponse rates and ensure data quality, multiple design strategies were deliberately integrated to balance the need for comprehensive data collection with the risk of participant fatigue. Attention checks were strategically placed to detect and exclude low-effort or random

responses prior to analysis. Second, to minimize mental burden and prevent dropouts, the survey length was carefully controlled. The specific measures adopted to address these concerns are detailed in the following section.

5.3 Target Population

The survey was targeted to individuals with practical experience in verification-aware programming languages, regardless of their professional role. This encompassed a wide range of profiles, from the industry to academia. No restrictions were imposed with respect to geographical location, however, given that the survey was distributed in English, a level of English fluency was required to ensure that participants could understand the questions.

The only eligibility criterion was that participants must have worked with any VA language. Individuals with no prior experience with VA languages were considered out of scope for this study.

5.4 Survey Design

The survey was designed and distributed using Qualtrics[1]. We chose this platform because it offers advanced features for building customized survey flows. The survey was structured into six sections, with the organization and details of each question detailed below.

5.4.1 Section *Consent*

The survey starts with the *Consent* section. Before proceeding to any substantive questions, we presented participants with an information text describing the purpose of the study, the voluntary nature of their participation, the anonymity of their responses, and the intended use of the collected data.

The section consists of two questions, both mandatory, as detailed in Table 5.1. Question C01 asked participants to confirm that they had read and understood the information provided, while C02 asked whether they wished to proceed with the survey, under the specified conditions. Both questions were binary, accepting only a *Yes* or *No* response.

[1] https://www.qualtrics.com/

Table 5.1: Details of Questions in Section *Consent*

ID	Question	Response Format	Response Options
C01	**I have read and understand the information above.**	Single Choice	– Yes – No
C02	**I want to participate in this research and continue with the survey.**	Single Choice	– Yes – No

Qualtrics branching logic was configured so that a *No* response to either question did not allow to continue with the survey, preventing the participant from advancing to following sections. This design made sure that only individuals who had been informed and gave their consent contributed to the study. After responding *Yes* to both questions, the participants were directed to the *Developer Background* section, where the data collection began.

5.4.2 Section *Developer Background*

The *Developer Background* section was designed to characterize the experience profile of eligible participants and to contextualize their next responses throughout the survey. All five questions in this section were mandatory, as detailed in Table 5.2.

The first question, DB01, collects participants' level of experience with Verification-Aware programming languages, but also screened them for eligibility, presenting options ranging from less than one year to over ten years of experience. Participants who submitted *I don't have any experience* were immediately disqualified, and the survey was terminated, since the following sections required familiarity with verification. Only those who had prior experience got access to the remainder of the survey.

DB02 asked participants to indicate their current professional role, and because practitioners frequently have multiple positions simultaneously, such as being both a software engineer and a researcher, the question allowed for multiple selections.

DB03 asked which specific VA languages participants had worked with, using the curated list previously established during the data collection stage in Section 3.2. This allowed for an assessment of the participants' range of experience with various formal tools.

DB04 and DB05 gathered information about the sources through which participants had learned VA programming and the contexts in which they apply it, respectively. Together, these questions help describe the participants' profile and to interpret their answers in the following sections.

Table 5.2: Details of Questions in Section *Developer Background*

ID	Question	Response Format	Response Options
DB01	**How many years of experience with Verification-Aware (VA) programming languages do you have?**	Single Choice (Dropdown)	– I don't have any experience – < 1 year – 1 year – 2 years – 3 years – 4 years – 5 years – 6 years – 7 years – 8 years – 9 years – +10 years
DB02	**What is your current role?** ***Please select all that apply.***	Multiple Choice	– Student – Researcher – Software Engineer – Other
DB03	**Which of the following VA languages have you worked with?** ***Please select all that apply.***	Multiple Choice	– Agda – Bandera – Rocq – Dafny – eCv – Eiffel – Frama-C – Idris – JPF – JML – KeY – Lean 4 – mbeddr – P – Spec# – Spark Ada – TLA+ – VCC – VeriFast – Verus – Whiley – Why3 – Other
DB04	**Where did you primarily learn how to program with VA languages?** ***Please select all that apply.***	Multiple Choice	– Official Documentation/Reference Manuals – Online Discussions – Online Video Tutorials – Mandatory University Course – Optional University Course – Work Context – Other

Continued on next page

Table 5.2: Details of Questions in Section *Developer Background* (continued)

ID	Question	Response Format	Response Options
DB05	**In which context do you mainly use VA languages? *Please select all that apply.***	Multiple Choice	– Academic – Research – Industrial – Personal Projects – Teaching – Other

Having satisfied the initial screening in DB01 and provided full responses to questions DB02 through DB05, the participants proceeded to the *Techniques Evaluation* section.

5.4.3 Section *Techniques Evaluation*

The *Techniques Evaluation* section constituted the core of the survey, directly addressing the validation of the taxonomy of verification strategies identified in Chapter 4. The section comprised three questions, all mandatory when applicable, as detailed in Table 5.3.

Question TE01 asked participants to evaluate each verification technique based on its perceived difficulty and frequency of use in their daily practice. For each technique, participants selected a difficulty rating from a five-point Likert scale [19] ranging from *Very Easy* to *Very Difficult*, with an additional option of *I never used this technique* for cases where the participant had no prior experience. The frequency of use was captured through a separate six-point scale ranging from *Never* to *Very Frequently*. Prior to filling the table, participants were instructed to proceed technique by technique, providing both ratings for each item before moving on to the next, to reduce the completion time.

Question TE02 asked participants to reflect on whether their real-life verification workflow included any technique or problem-solving pattern not covered by the listed items. The question logic was configured so that participants who answered *Yes* or *I'm not sure* were presented with TE03, a follow-up question in which they were required to specify any such omitted techniques or patterns. Participants who selected *No* skipped TE03 entirely and were directed to the next section. This last question allowed the detection of potential gaps in the taxonomy.

Table 5.3: Details of Questions in Section *Techniques Evaluation*

ID	Question	Response Format	Response Options
TE01	**For each of the following verification techniques, indicate how hard it is to master and how often you use it.** *Asked for each technique. Participants answer both ratings before moving to the next technique.*	Single Choice Matrix + Dropdown	– Difficulty: Very Easy; Easy; Neutral; Difficult; Very Difficult; I never used this technique – Frequency: Never; Very Rarely; Rarely; Occasionally; Frequently; Very Frequently
TE02	**Is there any technique or problem-solving pattern that you frequently use in your real-life verification workflow that was not listed above?**	Single Choice	– Yes – No – I'm not sure
TE03	**Please specify any technique or problem-solving pattern that you frequently use in your real-life verification workflow that was not listed above.**	Open-Ended	

* To reduce survey fatigue, Qualtrics counter-balanced randomization was used to divide the full set of techniques into random buckets. Each participant was randomly assigned to one bucket and evaluated 15 of the total set. Since each technique was rated by approximately half of the sample, responses were standardized within each bucket using z-scores prior to cross-technique comparisons, ensuring that differences in group composition did not confound the results.

After completing TE01, TE02 and, where applicable, TE03 participants were directed to the *Technique Application Scenarios* section.

5.4.4 Technique Application Scenarios

The *Technique Application Scenarios* section contextualizes the identified techniques within concrete verification contexts. This section presented four realistic scenarios drawn from the challenges documented throughout this study. The selection followed a deliberate methodological strategy and did not attempt to cover the full extent of challenges identified in either the systematic literature review of Chapter 2 or reported difficulties analyzed by Oliveira, Mendes, and Carreira [30]. Instead, the four scenarios were constructed around challenges that appear consistently across both sources of evidence. This intersection strengthened the validity of the scenarios,

since each one reflects an obstacle corroborated by both academic literature and empirical evidence from practitioner communities. It also kept the section more concise, helping maintain an acceptable estimated survey completion time of 15 minutes and reducing the risk of participants abandon it before completion. Including scenarios for every challenge identified across all sources would have substantially increased the length of this section and could have reduced response quality in the later stages of the survey. The four selected scenarios therefore represent a robust and validated subset of challenges, balancing with the usability of the instrument.

The formulation of all scenarios was an iterative process. Initial versions were written and discussed in collaboration with two other reviewers, to make sure that each of them was not ambiguous and represented the corresponding challenge according to reality. Multiple rounds of review led to refinements to the wording and level of detail. The last iteration phase included feedback from the piloting phase, where pilot participants pointed to ambiguous cases or scenarios that could be interpreted as already pointing to a specific technique. Every issue was addressed before the survey was distributed.

For each scenario, participants were asked to select all techniques from the taxonomy that they would apply to overcome the described obstacle. As described in 5.4.3, Qualtrics randomization was used to assign each participant a subset of 15 techniques out of the full set of 33, ensuring that no participant was exposed to the complete taxonomy in a single session, which would have made the survey excessively long. The response options presented for each of the four scenarios consisted precisely of this same individually assigned subset, maintaining full consistency between the two sections and ensuring that participants were never asked to choose from techniques they had not previously evaluated. All four questions were mandatory and both the order in which they were presented, and the order of the technique options within each scenario were randomized across participants, mitigating biases at both levels.

Table 5.4: Details of Questions in Section *Technique Application Scenarios*

ID	Question
S01	You are verifying a sorting algorithm with a proof by induction. In the inductive step, you need to prove that inserting a new element into an already sorted list of size preserves the sorted property. Although your current approach could eventually work, you recognize it becomes non-ideal, as tracing the element's exact final position through conditional swaps generates complex subgoals. *Which technique(s) would you use to overcome this?*
S02	You are verifying an implementation and, initially, proving basic properties required only a few annotations. However, as you introduce stronger guarantees involving multiple method interactions and object states, the verification context explodes. Each new property demands several additional invariants and lemmas, causing verification times to spike and minor code changes to break previously successful proofs. *Which technique(s) would you use to overcome this?*
S03	You are verifying a method, but the verifier reports that the proof cannot be completed. The tool only indicates that verification failed, without showing which assumptions were available, which proof obligation could not be established, or how the verifier reached its conclusion. As a result, you cannot determine why the proof failed. *Which technique(s) would you use to overcome this?*
S04	Your verifier fails, throwing a generic "Verification failed: Postcondition might not hold" error that points vaguely to the end of a block. With no counterexamples or detailed logs, you cannot tell if you have a real bug or if the SMT solver is just hitting a limitation. *Which technique(s) would you use to overcome this?*

S01 - Suboptimal Strategies

This scenario addresses practitioners' tendency to adopt proof strategies that, while technically valid, introduce unnecessary complexity. The question concretizes this challenge in the context of a sorting algorithm, where the practitioner must prove that inserting an element into a sorted list preserves sortedness. The strategy of tracing the element's exact position via conditional swaps is explicitly flagged as non-ideal, prompting participants to reflect on which techniques they would use to restructure or simplify the proof.

S02 - Proof Complexity Snowballing

This scenario captures the growing complexity of verification manageability. The question is designed to understand which techniques participants associate with managing proof expansion and maintaining long-term verification stability.

S03 - Lack of Tool Explainability and Interaction

This scenario targets one of the most structurally problematic aspects of current verification tools, which is their opacity. The scenario is therefore designed to assess which techniques participants rely on to reconstruct the proof state and recover from opaque failures in the absence of adequate tool feedback.

S04 - Ambiguity of Uninformative Verifier Error Messages

While S03 concerns the absence of any structural feedback from the verifier, this scenario focuses on a subtler but equally disorienting situation, which is the presence of a generic error message that provides a label for the failure without offering diagnostic content. The scenario is designed to understand which diagnostic and hinting techniques participants use to disambiguate the source of such failures and proceed with verification.

After completing all technique application scenarios, participants proceeded to the *Demographics* section.

5.4.5 Demographics

The *Demographics* section collected basic sociodemographic information about each participant. All four questions were mandatory, as detailed in Table 5.5.

Questions D01 and D02 collected participants' age range and gender, respectively, with D02 including a *Prefer not to say* option to accommodate those who did not wish to share that information. Question D03 asked participants to identify their racial background from a set of predefined categories, also offering a *Prefer not to say* option. Finally, D04 recorded the highest level of formal education completed, ranging from some high school or less to graduate or professional degree.

Table 5.5: Details of Questions in Section *Demographics*

ID	Question	Response Format	Response Options
D01	**How old are you?**	Single Choice	– Under 18 years old – 18 - 24 years old – 25 - 34 years old – 35 - 44 years old – 45 - 54 years old – 55 - 64 years old – 65+ years old
D02	**What is your gender?**	Single Choice	– Female – Male – Non-binary/Third gender – Other – Prefer not to say
D03	**What is your race?**	Single Choice	– White or Caucasian – Black or African American – American Indian/Native American or Alaska Native – Asian – Native Hawaiian or Other Pacific Islander – Other – Prefer not to say
D04	**What is the highest level of education you have completed?**	Single Choice	– Some high school or less – High school diploma or GED – Some college but no degree – Associates or technical degree – Bachelor's degree – Graduate or professional degree (MA, MS, MBA, PhD, JD, MD, DDS, etc.) – Prefer not to say

5.4.6 Participant Feedback

The survey ends with the *Participant Feedback* section, which offers participants an open-ended opportunity to share additional thoughts or comments. Unlike all preceding questions, F01 was

optional. Throughout the period during which the survey was open, the responses to this question were monitored in order to identify and address any issues with the survey instrument that participants might report.

Table 5.6: Details of Questions in Section *Participant Feedback*

ID	Question	Response Format
F01	**Do you have any other feedback for us? Feel free to share your thoughts about anything, from our survey to formal methods and strategies.**	Open-Ended

The complete survey, spanning the six sections described above, was estimated to require approximately 15 minutes to complete, a duration considered appropriate given the voluntary and uncompensated nature of the participation.

5.5 Participant Recruitment Strategy

The recruitment of participants followed a multi-channel strategy to reach a diverse sample of practitioners with experience in Verification-Aware programming languages, including both academic and industrial contexts.

First, we contacted directly practitioners identified within the domain, resulting in 15 email contacts from researchers and professionals in both academic and industrial settings. Additionally, 10 practitioners were approached via LinkedIn[2], targeting professionals from organizations like Amazon Web Services, a research group at the University of York, and other fellow academics.

Besides direct reaching, the survey was disseminated more extensively through public channels. A post was published on LinkedIn and shared by the members of the research group within which this work was conducted, extending its reach to a wider network of practitioners. One contact also shared the survey within the Amazon Web Services community, and the survey was additionally disseminated through the Scottish Programming Languages Institute[3]. The survey was also shared in relevant online communities, including multiple related Zulip[4] servers, mailing lists, and other forums frequented by the formal methods community, maximizing visibility among developers.

In both cases, we solicited our contacts to further distribute the survey to relevant members of their professional and student networks, adopting a snowball sampling approach to extend the reach of the recruitment effort.

[2] https://www.linkedin.com/
[3] https://spli.scot/
[4] https://zulip.com/

Finally, the survey was also shared with 26 students enrolled in the Formal Methods for Critical Systems course[5] at Faculty of Engineering of the University of Porto in the 2025/2026 instance, serving as a means to connect with students who are presumably less experienced.

5.6 Data Analysis

This section describes the process adopted to analyze the survey responses, including the qualitative analysis of open-ended responses and the quantitative analysis of the technique ratings, while also evaluating the survey results through this methodology.

5.6.1 Qualitative Analysis

The responses to open-ended questions, specifically to TE03 and the open-ended fields included in the scenarios, were analyzed and coded to determine if they corresponded to any of the already identified topics in taxonomy or if they required new codes.

The responses were blind-coded by two reviewers, in an iterative process, using a codebook obtained directly from the list of verification techniques. In the first round, both coders assigned codes to each response, matching it to an existing technique or flagging it as a potential new code if it could not be mapped to any item in the taxonomy. The two coders then compared their assignments and discussed any disagreements until reaching consensus. This first round brought out one candidate emergent code, after which the codebook was updated. A second independent coding round was then carried out using the updated codebook to confirm that no other new codes were brought out and that the coding had stabilized. Disagreements throughout both rounds were resolved through discussion.

In the scenario questions exclusively, the open-ended responses were merged with the corresponding close-ended selections, so that it recognized the same way as the others. This was done to mitigate the fact that each participant was only presented a list with 15 techniques, and this ensured that the techniques mentioned in open-ended fields were included in the scenario analysis.

5.6.2 Quantitative Analysis

Perceived difficulty was modeled using a Cumulative Link Mixed Model (CLMM) Ordinal Mixed with a default logit link and flexible threshold. This approach was selected, since the response is ordinal instead of interval-scaled, and a conventional linear model, such as Analysis of Variance (ANOVA) on mean scores, would assume equal spacing between adjacent response categories, an assumption that Likert-type data does not support. Also, because each participant rated multiple techniques, individual ratings cannot be treated as independent observations. The model includes a random intercept per participant, taking into account this repeated measures structure, as well as for between participant differences in general rating tendencies.

[5] https://sigarra.up.pt/feup/en/UCURR_GERAL.FICHA_UC_VIEW?pv_ocorrencia_id=562471

A CLMM, like the one used for difficulty, was planned for the frequency ratings, however, the test to verify the proportional-odds assumption could not be calculated due to the small number of observations available for each combination of technique and response category. Since this assumption could not be verified, frequency ratings are only presented with descriptive analysis.

5.6.3 Participant Profile

The survey received a total of 54 valid responses after applying the eligibility screening. We excluded participants who did not pass the attention checks and reported no prior experience with formal verification.

This section characterizes the sample in terms of demographics and developer background. In Table 5.7, we present the demographic analysis of the participants, and in Table 5.8, we present their background.

Table 5.7: Demographic characteristics of participants ($N = 54$).

Dimension	Category	*n*	*%*
Age	18–24 years old	12	22.2
	25–34 years old	20	37.0
	35–44 years old	8	14.8
	45–54 years old	10	18.5
	55–64 years old	4	7.4
Gender	Male	44	81.5
	Female	4	7.4
	Non-binary / Third gender	2	3.7
	Prefer not to say	4	7.4
Race	White or Caucasian	41	75.9
	Asian	2	3.7
	Black or African American	1	1.9
	American Indian or Native American	1	1.9
	Prefer not to say	10	18.5
Education	Graduate or professional degree	41	75.9
	Bachelor's degree	10	18.5
	Some college but no degree	2	3.7
	Prefer not to say	1	1.9

Considering the participants' responses, we observe that most of them are male (81.5%) and highly educated, with 94.4% holding at least a university degree. The sample covers multiple career stages, concentrated in the 25–34 age range. Women represent 7.4% of the participants, and non-binary or third-gender individuals, 3.7%. Regarding the race, 75.9% of participants identified

as white and 18.5% preferred not to specify. All of the remaining demographic data is detailed in the table.

Table 5.8: Participant background characteristics ($N = 54$). Multiple-selection questions are marked with [a]; percentages for those dimensions may exceed 100%.

Dimension	Category	*n*	%
Experience	$<$ 1 year	3	5.6
	1 year	3	5.6
	2 years	5	9.3
	3 years	6	11.1
	4 years	5	9.3
	5 years	4	7.4
	6 years	6	11.1
	7 years	2	3.7
	8 years	3	5.6
	9 years	2	3.7
	+10 years	15	27.8
Role	Researcher	36	66.7
	Student	22	40.7
	Software Engineer	21	38.9
	CTO	1	1.9
Languages Used	Rocq	27	50.0
	Lean 4	22	40.7
	Dafny	20	37.0
	Agda	16	29.6
	Why3	16	29.6
	TLA+	10	18.5
	Idris	10	18.5
	Frama-C	10	18.5
	Spark Ada	8	14.8
	Verus	7	13.0
	JML	4	7.4
	VeriFast	3	5.6
	Eiffel	3	5.6
	KeY	2	3.7
	VCC	1	1.9
	Other	8	14.8

Continued on next page

Table 5.8: Participant background characteristics ($N = 54$) (continued)

Dimension	Category	*n*	*%*
Learning Sources	Official Documentation / Reference Manuals	39	72.2
	Online Discussions	20	37.0
	Optional University Course	19	35.2
	Work Context	19	35.2
	Mandatory University Course	11	20.4
	Online Video Tutorials	6	11.1
	Books	4	7.4
	Other	8	14.8
Usage Context	Research	38	70.4
	Academic	36	66.7
	Personal Projects	27	50.0
	Teaching	17	31.5
	Industrial	15	27.8

The participants are, in general, highly experienced. 79.6% of respondents have at least three years of experience, and 27.8% exceed ten years. 11.1% have less than two years of experience, which might mitigate the risk of technique evaluations being biased by unfamiliarity. It can also be observed that the sample includes mostly researchers and students, while software engineers constitute a smaller fraction of the sample. 26 out of the 54 respondents have multiple roles, capturing the dual profile typical in formal methods.

In terms of verification tools, Rocq, Lean, Dafny, Agda and Why3 predominate as the most used. Some respondents also reported experience with unlisted languages, including VerCors, F*, LiquidHaskell, Isabelle/HOL, Istari, Alloy, and Spin/PROMELA, all mentioned only once, and Creusot, mentioned twice. This diversity strengthens the validation by ensuring evaluations reflect the taxonomy's generalizable design rather than single-tool specificities.

We also conclude that most participants learn about Verification-Aware languages as a deliberate choice within their education or career path. Official documentation is the most frequently reported source of learning, followed by online discussions, optional university courses, and work contexts. At the same time, the presence of mandatory and optional university courses suggests that knowledge about Verification-Aware languages is appearing earlier in developers' careers, before or alongside professional practice. Notably, online video tutorials were selected by only 11.1%, which is relevant because the taxonomy in Chapter 4 was derived entirely from video transcripts. By exposing the participants to these techniques, the study allows the audience to evaluate practices they might not typically encounter in their preferred learning channels. Consequently, the survey can validate and formalize this hidden practical knowledge.

5.6.4 Technique Frequency

This section analyses how often practitioners report using each of the 33 identified verification techniques in their daily practice. Because each participant was randomly assigned a subset of 15 techniques, the number of respondents per technique ranges between 21 and 34. Figure D.1 summarizes the median and mode for each technique and D.3 the percentage of "never used" responses for each technique

The following analysis discusses the results by category, considering the categories defined in Chapter 4.

Proof Strategy and Decomposition

Figure 5.1 shows the frequency distribution for the techniques in this category. T01 stands out as the single most frequently used technique in the entire taxonomy. 70.6% of respondents report using it *Very Frequently* and 14.7%, *Frequently*, resulting in both a median and a mode of *Very Frequently*.

This result strengthens the finding from Chapter 4, where T01 was the technique with the highest attribution count in the dataset, 138 out of 154 videos. The consistency between both sources provides strong convergent validity, showing that Goal Decomposition is both highly visible in recorded sessions and essential to practitioners' daily workflows.

T04 and T09 also showed high usage frequencies. For T04, the median was *Frequently* and the mode was *Very Frequently*. In contrast, T09 presented a median and mode of *Very Frequently* with a bimodal distribution, where 59.3% of respondents reported using it *Very Frequently*, while 22.2% answered *Never*. This polarization suggests that this is a core technique for a subset of practitioners, while remaining non-essential or unfamiliar to others. The distribution shows that T09 is used intensively in some verification workflows but is not equally prominent across the whole sample.

T26 presents a more uniform distribution, with a median of *Frequently* and responses spread across all levels. T18 and T10 show more moderate frequencies, with medians of *Frequently* and *Rarely* respectively, and T31 is the least used technique in this category, with 50% of respondents responding with *Never*.

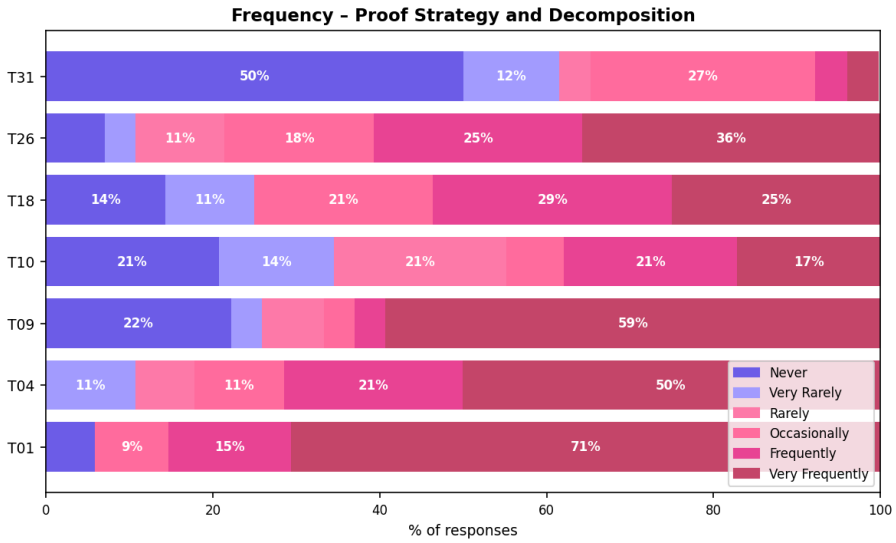
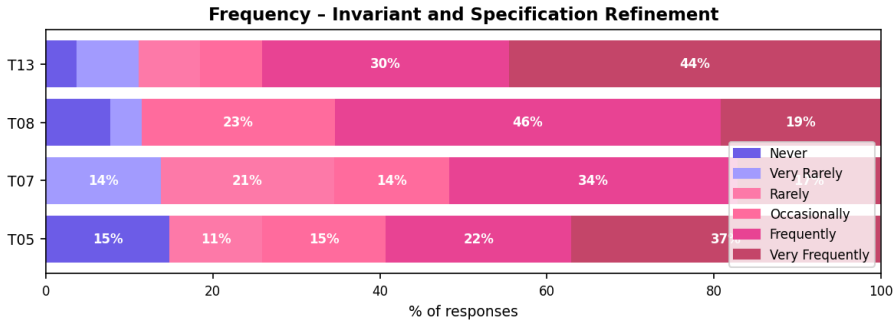


Figure 5.1: Frequency distribution for the Proof Strategy and Decomposition category.

Invariant and Specification Refinement

Figure 5.2 shows the frequency distribution for this category. T13 and T08 are the most frequently used techniques, with medians of *Frequently* and modes of *Very Frequently* and *Frequently*, respectively. T05 has a median of *Frequently*, but its distribution is more dispersed, as 14.8% of respondents report *Never* using it and 37.0% using it *Very Frequently*. T07 has a median and mode of *Frequently*, with 34.5% of respondents placing it there, and only 14% reporting they never use it.

Figure 5.2: Frequency distribution for the Invariant and Specification Refinement category.

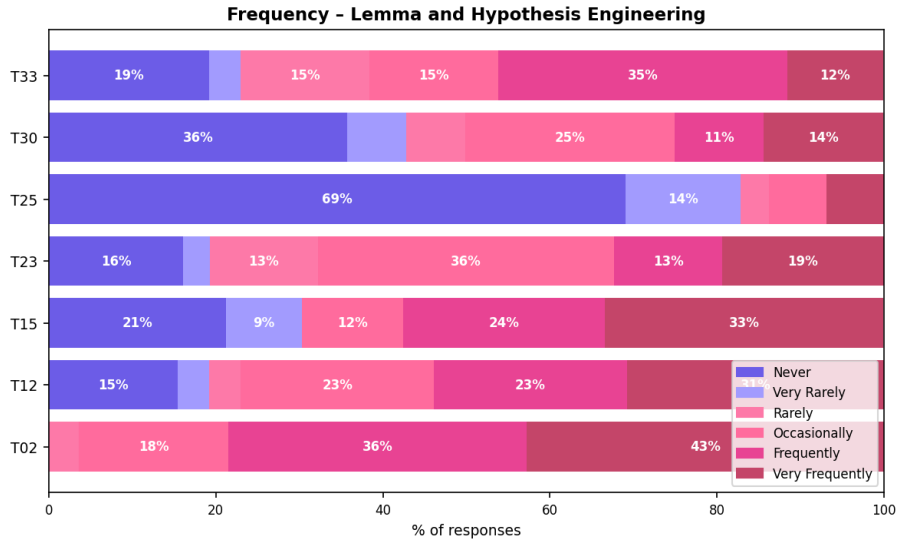


Lemma and Hypothesis Engineering

Figure 5.3 presents the results for this category. According to the results, T02 is the second most frequently used technique in the entire taxonomy after T01. In our survey, no respondent reports *Never* using it or using it *Very Rarely*. On the contrary, 79% report using it *Frequently* or *Very Frequently*, with a mode of *Very Frequently*. This result confirms that explicitly introducing assumptions into the local proof context is almost a universal practice.

T15 and T12 show median frequencies of *Frequently*, with modes of *Very Frequently*, indicating that these are regularly used tools in most practitioners' repertoires. By contrast, the lower-frequency techniques in this category, such as T25, T30, T23, and T33, show medians of *Never* or *Occasionally*, confirming that they are specialised low-level moves used only in more specific proof situations. T25, in particular, has the second highest *Never* rate in the entire taxonomy, consistent with the very low attribution count of 3 videos, stated in Chapter 4.

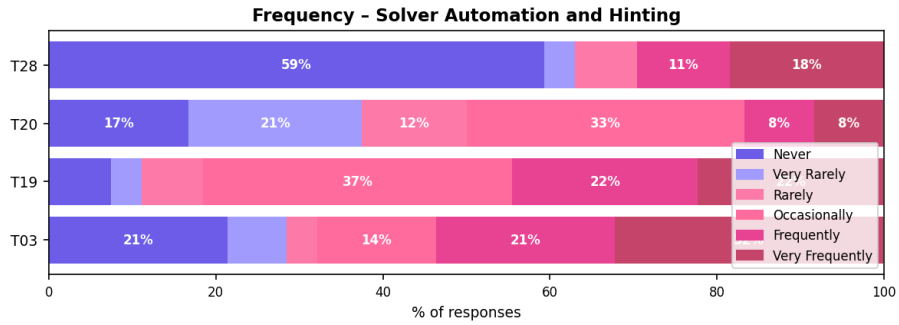
Figure 5.3: Frequency distribution for the Lemma and Hypothesis Engineering category.



Solver Automation and Hinting

Figure 5.4 shows the frequency distribution for the three techniques in this category. T03 has a median and mode of *Frequently* and *Very Frequently*, respectively, but its distribution is bimodal, since 32.1% of respondents report *Very Frequently* and 21.4% report *Never*. This variation suggests that automation is not experienced the same way in different verification workflows. For some participants, it is clearly a routine part of proof development, while for others it is less common within their usual verification process. T19 has a median of *Occasionally*, consistent with the specialised nature of model-checking tasks. T28 (Hint Database Management) has the second highest *Never* rate (59.3%) in the taxonomy, confirming that explicit management of hint databases is restricted to a small subset of practitioners, most likely those working with Rocq or Lean tactics libraries.

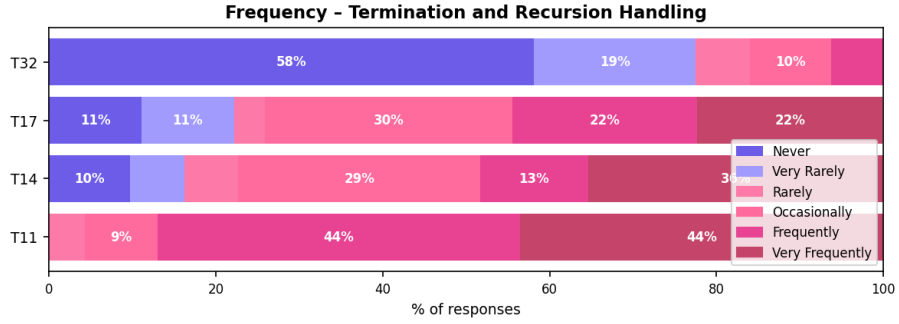
Figure 5.4: Frequency distribution for the Solver Automation and Hinting category.



Termination and Recursion Handling

Figure 5.5 presents the frequency results for this category. T11 is the most frequently used technique in the group, with a 44% *Frequently* and 44% *Very Frequently* usage, and a *Never* rate of 0%, making it the only technique in the entire taxonomy that no respondent reports never using. This matches the high number of recursive programs in the data, as verifying these functions always requires developers to explicitly show the decreasing structure of the calls. T14 shows relatively high frequency and T17 has a median of *Occasionally*, reflecting that explicit termination metrics are required only when the verifier cannot infer termination automatically. T32 has a *Never* rate of 58.1%, the highest in its category, confirming that it is an unknown or very specialized tactic.

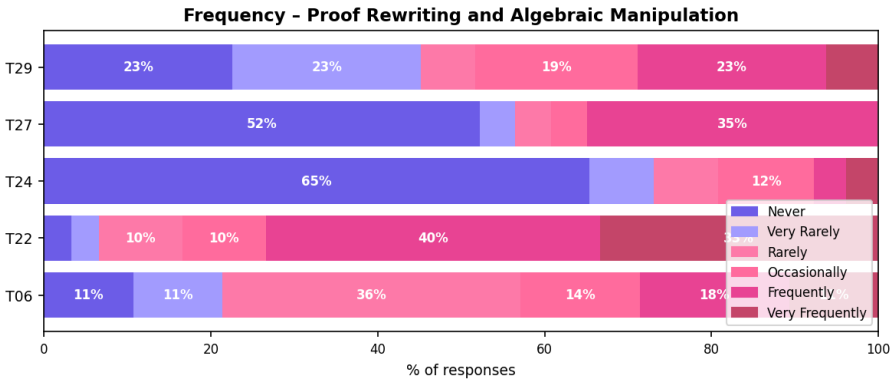
Figure 5.5: Frequency distribution for the Termination and Recursion Handling category.



Proof Rewriting and Algebraic Manipulation

Figure 5.6 shows the distribution for this category. T22 is the most frequently used technique in the group, with a median and mode of *Frequently*, selected by 40% of respondents. T06 stands out for the divergence between its video-corpus attribution count, which places it sixth, and its survey frequency, where the median is *Rarely* and only 29% of respondents report using it *Frequently* or above. An explanation for this can be the fact that goal manipulation steps are explicit during live-coding sessions, where developers describe every step, but is an implicit background activity in real-life programming. T24, T27, and T29 all have *Never* rates above 50%, confirming them as unknown or highly specific techniques.

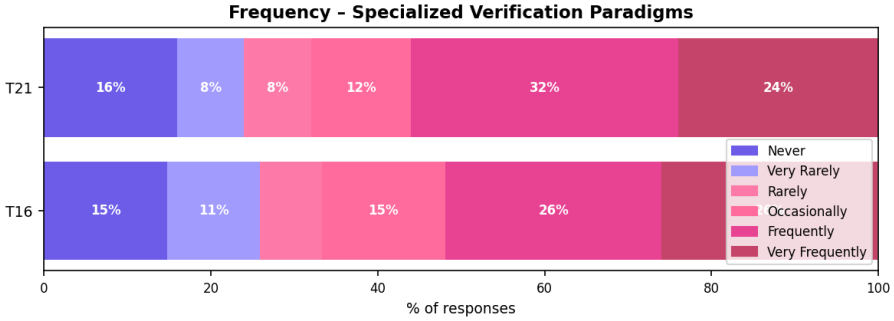
Figure 5.6: Frequency distribution for the Proof Rewriting and Algebraic Manipulation category.



Specialized Verification Paradigms

Figure 5.7 shows the frequency results for this category. T21 has a median of *Frequently* and a mode of *Frequently*, with 56% of respondents reporting using it at that level or above, which is higher than its attribution count in the video corpus might suggest. This may indicate that practitioners who work with monadic structures use the technique regularly whenever their codebase involves effect management, even if the absolute number of such practitioners in the sample is small. T16 has a median of *Frequently* with a stable distribution, reflecting its role as a crucial activity.

Figure 5.7: Frequency distribution for the Specialized Verification Paradigms category.



Overall patterns

Across all categories, two groups of techniques emerge with markedly different frequency profiles. The first group, constituted by T01, T02, T04, T05, T08, T09, T11, T13, T15, T22, and T26, consistently receives medians and modes of *Frequently* or *Very Frequently* and low *Never* rates. These techniques correspond to the core of the verification workflow as identified in Chapter 4. Those are the ones that are used regularly in most sessions and are recognized as routine practices by most of the respondents.

The second group, which includes T24, T25, T27, T28, T31, and T32, has *Never* rates above 50% and medians of *Never* in the survey. These are the same techniques that received the fewest

attributions in the video corpus and that a majority of respondents also marked as *I never used this technique* in the complexity question. Their low frequency in both sources confirms that they are highly specialized strategies tied to specific proof contexts.

The divergence in T06 points to a methodological observation, as video analysis captures what practitioners do and show, while the survey captures what they recognize and name as deliberate techniques. These two views complement each other and their differences actually provide valuable insights.

5.6.5 Technique Complexity

This section analyses the perceived difficulty of mastering each of the 33 techniques in the taxonomy. Difficulty was measured on a five-point ordinal scale ranging from *Very Easy* to *Very Difficult*, with an additional option of *I never used this technique*, which was excluded from the complexity analysis. Because each participant was randomly assigned a subset of 15 techniques, the effective sample size per technique, after excluding the *never used* responses, ranges from 9 to 32 respondents. Figure D.2 summarizes the median and mode of each.

Each participant evaluated more than one technique, which means that multiple sets of ratings came from the same person. To address this, the model introduces an adjustment per participant, which captures the fact that some participants may generally rate techniques as easier or harder than others. The proportional-odds assumption was satisfied (nominal-effects LR test, $p = 0.27$), and the model converged without warnings.

The model produces an expected difficulty score on a scale from 1 to 5 for each technique and enables pairwise comparisons with a Tukey adjustment. This adjustment groups techniques that are not statistically distinguishable from one another under a shared letter, providing a clear visual hierarchy of difficulty. The global test results demonstrate that the choice of technique has a highly significant effect on how hard practitioners perceive a task to be ($\chi^2(32) = 152.6$, $p < 0.001$), meaning that the differences in difficulty observed across the taxonomy are not random. Table D.1 reports the full CLMM results sorted from easiest to hardest, which serves as the primary foundation for the discussion below.

The model uses estimated marginal means to predict the probability of a technique falling into specific difficulty categories. This analysis identified T05, T01 and T15 at the easiest of the taxonomy, with a predicted probability of 77% to 83% of being rated as either *Easy* or *Very Easy*. At the hardest extreme, techniques T24, T32 and T04 were identified as the most challenging, showing a predicted probability of approximately 48% to 57% of being rated as *Difficult* or *Very Difficult*. Besides these two distinct poles, the majority of the techniques in the taxonomy cluster around a *Neutral* rating. Because their confidence intervals overlap under the Tukey adjustment, these intermediate techniques cannot be statistically separated from one another, indicating that practitioners share a relatively uniform perception of their moderate difficulty.

Proof Strategy and Decomposition

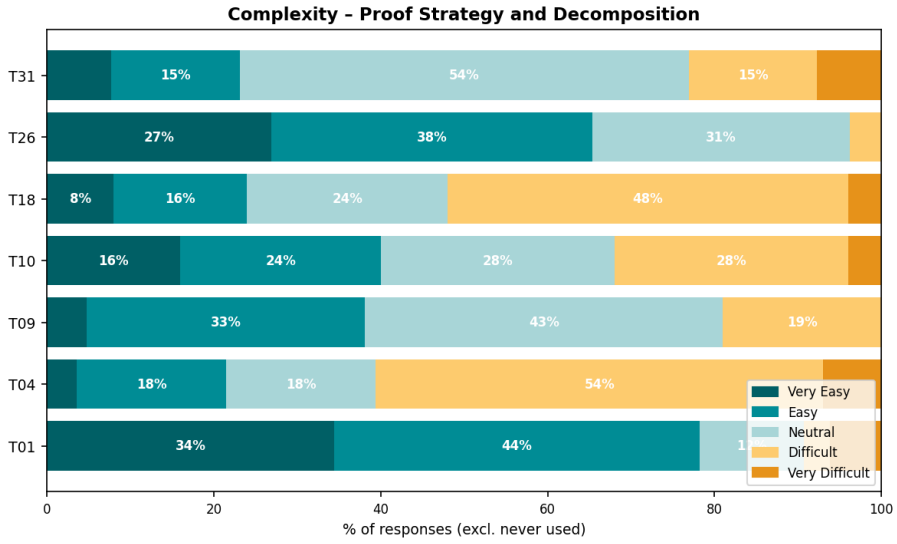
Figure 5.8 shows the complexity distribution for this category. T01 and T05 occupy the two easiest positions in the entire taxonomy according to the CLMM, with expected scores of 1.95 and 1.85 respectively, both belonging to group **a** or **ab**, meaning they are significantly easier than the majority of all other techniques. 78.2% of respondents rate T01 as *Very Easy* or *Easy*, and for T05 the same rate reaches 91.7%, making it the most accessible technique in the taxonomy. This result is consistent with the very high frequency of both techniques. Techniques that practitioners apply constantly tend to be the ones they have most internalized, reducing perceived cognitive effort.

T26 is also rated as relatively easy, with 65.4% of responses in the *Very Easy* or *Easy* categories. T09 and T10 stand in the middle of the taxonomy, both belonging to group **abcdef**, with approximately equal proportions of *Easy*/*Neutral* and *Neutral*/*Difficult* ratings.

T18 stands out as the most difficult technique in this category, with an expected score of 3.22 (group **ef**) and 52% of respondents rating it *Difficult* or *Very Difficult*. Despite being a structural practice, practitioners find it demanding, likely because it requires maintaining a mental model of the evolving proof context while applying tactics that may interact with multiple obligations.

T04 is the hardest technique in this category and the second hardest in the entire taxonomy, with an expected score of 3.36 and belonging exclusively to group **f**, meaning it is significantly harder than all techniques except T32 and T24. 53.6% of respondents rate it as *Difficult*, and 7.1% as *Very Difficult*, while no respondent reported that never used it.

Figure 5.8: Complexity distribution for the Proof Strategy and Decomposition category, excluding *I never used this technique* responses.

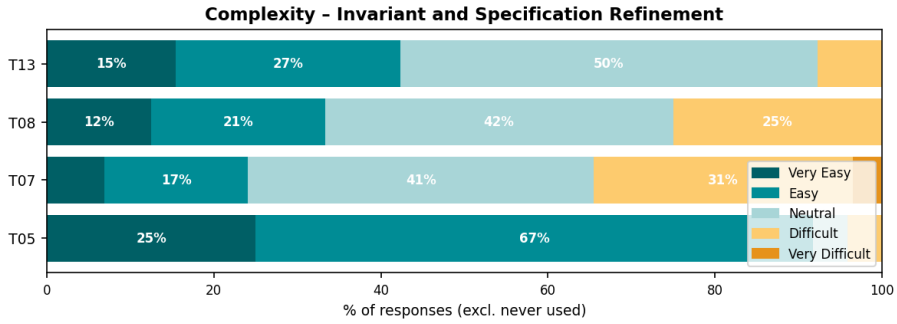


Invariant and Specification Refinement

Figure 5.9 shows the results for this category. As already seen, T05 is the easiest technique in the entire taxonomy. Despite requiring the developer to formulate a predicate that is simultaneously inductive and strong enough to imply the desired postcondition, practitioners rate it as accessible, suggesting that, once learned, the invariant strengthening loop becomes a routine skill.

T13 and T08 are rated as *Neutral* by the largest portion of respondents, with expected scores of 2.63 and 2.78 and *never used* rates below 8%. T07 is the most difficult technique in the category, with an expected score of 3.12, belonging to group **ef** and 31% of respondents rating it as *Difficult* or *Very Difficult*.

Figure 5.9: Complexity distribution for the Invariant and Specification Refinement category, excluding *I never used this technique* responses.



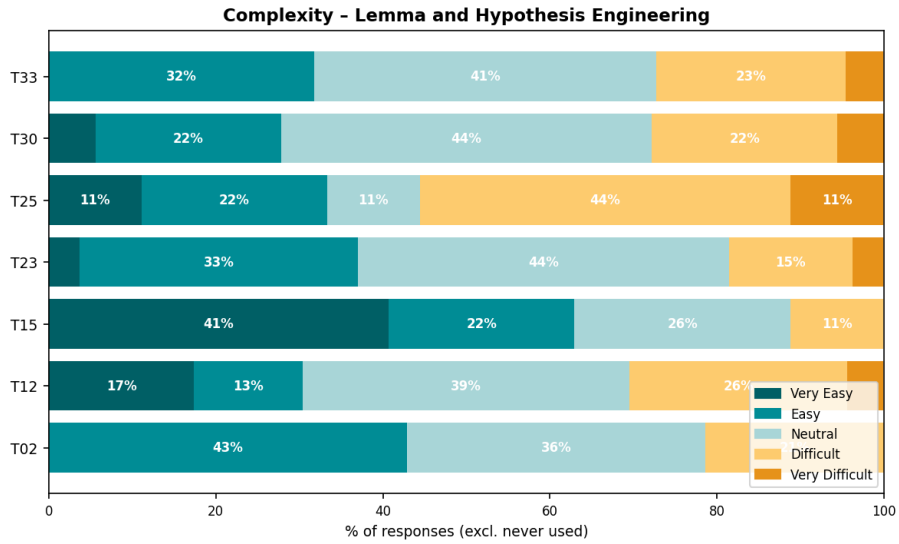
Lemma and Hypothesis Engineering

Figure 5.10 shows the complexity results for this category. T15 is the third easiest technique in the entire taxonomy, with expected score 2.00, placed in group **abc**, with 63% of respondents rating it *Very Easy* or *Easy* and a mode of *Very Easy*.

T02 is rated *Easy* by the plurality (43%), with a *never used* rate of 0% and an expected score of 2.78, placed in group **bcdef**. The modest difficulty relative to the technique's reflects its mechanical nature of introducing an assumption requiring knowing it exists and is relevant, but the act of introduction itself is syntactically straightforward in most tools.

The lower frequency techniques in this category, T25, T23, T30 and T33, present a more challenging picture, with expected scores ranging from 2.79 to 3.32. T25 is the most difficult among them and also has the highest *never used* rate in the taxonomy. This combination confirms that T25 is a highly specialized or unknown tactic that practitioners rarely apply, and when they do, they find it demanding to do it correctly.

Figure 5.10: Complexity distribution for the Lemma and Hypothesis Engineering category, excluding *I never used this technique* responses.



Solver Automation and Hinting

Figure 5.11 shows the results for this category. T03 is one of the hardest techniques in the taxonomy, with an expected score of 3.20, placed in group **ef** and 46% of respondents rating it as *Difficult*. T20 is perceived as similarly challenging, with 35% of respondents rating it *Difficult* or *Very Difficult*.

T28 has a more dispersed distribution across all difficulty levels and a *never used* rate of 55.6%, reducing its effective sample to 12 respondents. Among those who do use it, ratings are spread in the full scale, suggesting that its difficulty depends heavily on familiarity with the specific context involved rather than on any intrinsic property of the technique itself.

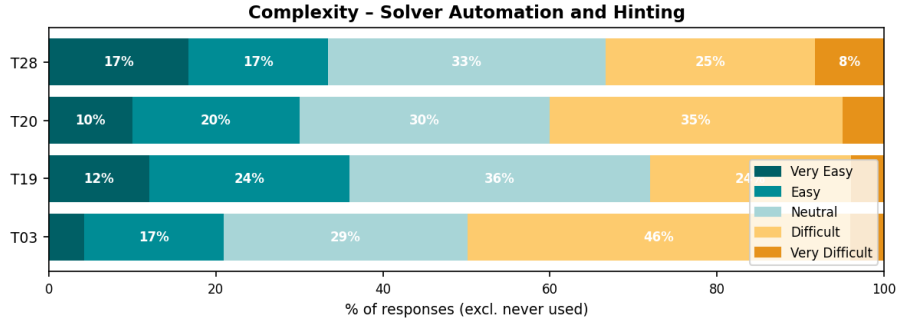


Figure 5.11: Complexity distribution for the Solver Automation and Hinting category, excluding *I never used this technique* responses.

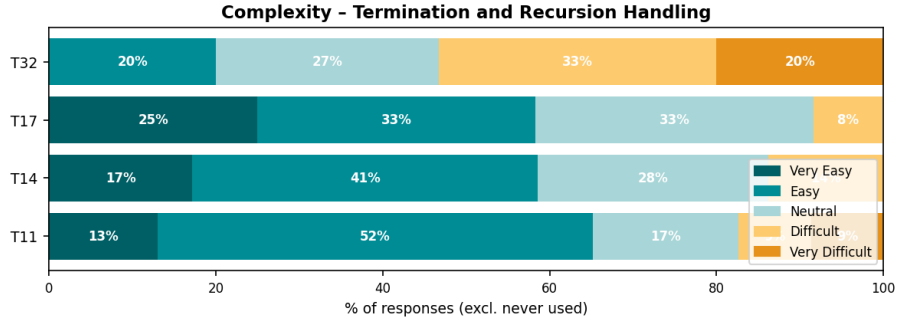
Termination and Recursion Handling

Figure 5.12 presents the results for this category. T11 is the easiest technique in this category and one of the easiest in the taxonomy after T05 and T01, with 65% of respondents rating it *Very Easy* or *Easy* and a *never used* rate of 0%. This accessibility, combined with its universal usage observed in the frequency analysis, confirms that making the decreasing structure of recursive calls explicit is a learnable skill, even if the underlying reasoning can be subtle.

T14 and T17 are both rated as predominantly *Easy*, with expected scores 2.39 and 2.19, respectively, reflecting that, once a practitioner knows the technique and when to apply it, its execution is mechanically straightforward.

T32 is the hardest technique in this category and the third hardest in the entire taxonomy, with an expected score of 3.44 in group **ef**, a combined *Difficult* or *Very Difficult* rate of 53.3%, and a *never used* rate of 51.6%. Its position at the difficult end of the scale is consistent with its nature as a specialized move that requires reasoning backwards through the inductive definition of a type. Unlike more procedural termination techniques, it depends on recognizing which structural information can be recovered from a constructor or hypothesis, making it harder to apply routinely and more dependent on specific proof situations.

Figure 5.12: Complexity distribution for the Termination and Recursion Handling category, excluding *I never used this technique* responses.



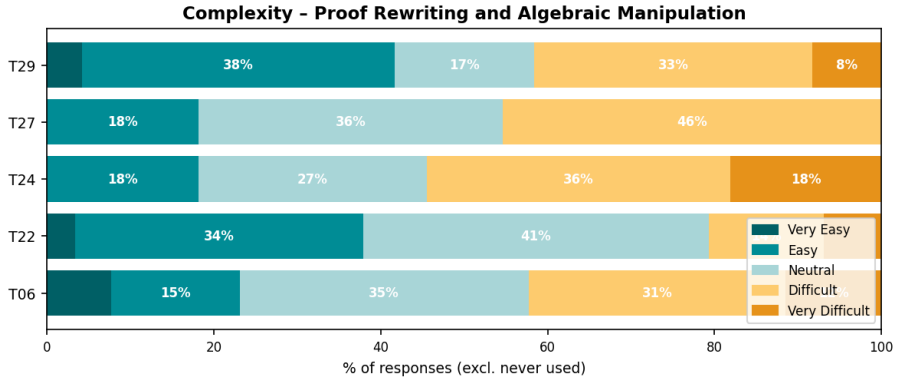
Proof Rewriting and Algebraic Manipulation

Figure 5.13 shows the results for this category. T06 is among the hardest techniques in the taxonomy despite its relatively moderate *never used* rate (7.1%), with an expected score of 3.26 (group **ef**) and 42% of respondents rating it *Difficult* or *Very Difficult*. This result connects with the divergence between the video corpus and the frequency survey discussed in Section 5.6.4. Goal manipulation is both widely encountered in practice and perceived as cognitively demanding, which might be due to choosing which rewriting step to apply, and in what order, is not always evident from the goal's syntactic form alone.

T22 is the easiest technique in this category, with an expected score 2.81 in group **cdef**, with 35% of respondents rating it *Easy* and a *never used* rate of 3.3%. T24 is the hardest technique in the entire taxonomy, with an expected score of 3.55 (group **ef**) and 54.4% of respondents rating

it *Difficult* or *Very Difficult*. However, its *never used* rate of 57.7% means the effective sample is only 11 respondents, and T24 is not significantly different from T32, T04, T25, and T03 after Tukey adjustment, limiting the strength of this ranking at the tail. T27 follows a similar pattern, with a *never used* rate of 52.2% and a combined *Difficult* or *Very Difficult* rate of 46% among those who do use it. These results confirm that rare algebraic techniques are not just unfamiliar, they are also genuinely difficult to use.

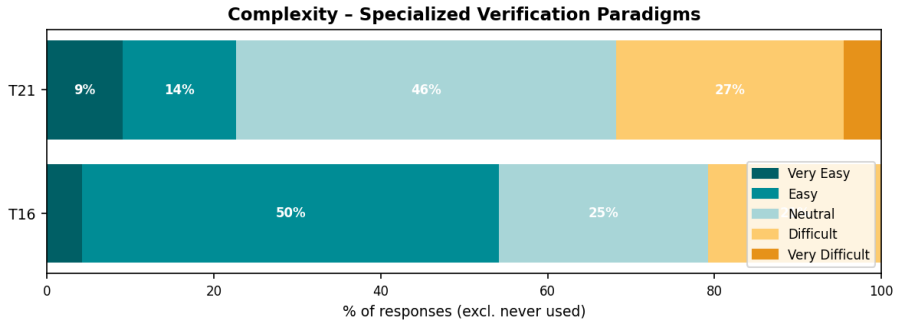
Figure 5.13: Complexity distribution for the Proof Rewriting and Algebraic Manipulation category, excluding *I never used this technique* responses.



Specialized Verification Paradigms

Figure 5.14 shows the results for this category. T16 is the most accessible technique in the group, with an expected score of 2.56 in group **abcdef** and 50% of respondents rating it *Easy*. Practitioners who work with model checkers use temporal logic as a means for expressing properties and, for them, it constitutes a core skill rather than an advanced one. T19 sits at *Neutral* (expected score 2.85, group **cdef**), with responses distributed evenly across the positive and negative halves of the scale. T21 is the most challenging technique in the category, with an expected score 3.08 in group **def**, with 27% of respondents rating it *Difficult* or *Very Difficult*. This difficulty likely comes from the extra effort needed to reason about monadic composition and its proofs, which require knowing concepts that are not usually taught in standard verification learning.

Figure 5.14: Complexity distribution for the Specialized Verification Paradigms category, excluding *I never used this technique* responses.



Overall patterns

The CLMM results reveal a clear gradient across the taxonomy. The easiest pole is anchored by T05, T01, T15, T26, and T17, all with expected scores below 2.2. These techniques share a common trait, as they are either part of the most frequently used (T05, T01) or represent conceptually bounded moves where the practitioner knows precisely what to provide to the verifier (T15, T26, T17).

The hardest pole is constituted by T04, T32 and T24, with expected scores above 3.35. All three require a form of non-trivial reasoning that is not directly guided by verifier feedback.

T04 is simultaneously the second most attributed technique in the video corpus (67 videos), used by all respondents at a *Frequently* or *Very Frequently* rate by the majority, and the second hardest technique to master. T03 presents a similar pattern, as it's widely used but consistently perceived as difficult. These two techniques represent the clearest instances of essential complexity in the taxonomy.

A final structural observation concerning the *never used* rates shown in Figure D.3 is that techniques T25, T24, T28, T27, T32, and T31 form a distinct cluster. They all combine low frequency, a position at the tail of the distribution, and scores near the harder end of the complexity scale. This high rate of *never used* responses indicates that most participants have not encountered contexts that require these techniques. However, this pattern might not simply reflect a lack of relevance, it can suggest a combination of essential complexity and unfamiliarity within the community. These algebraic and specialized operations are often powerful tools that are highly effective when applied, but their steep learning curve means that many practitioners have not mastered or even encountered them in standard workflows. Consequently, while their difficulty ratings are consistently high among the few people who use them, these scores are based on small sample sizes and should be interpreted with caution.

5.6.6 Techniques Not Covered

Questions TE02 and TE03 invited participants to reflect on whether their real-life verification workflow included any technique not present in the taxonomy, and to describe it in an open-ended field if so. Of the 54 respondents, 22 provided at least one response in TE03. The four scenario fields also included an open-ended option, used by a further subset of participants to describe techniques they would apply but that were not among their assigned options.

The majority of responses mapped to codes already present in the taxonomy, confirming that the taxonomy provides broad coverage of the techniques practitioners report using. The only technique that required a new code was *Ghost State*, mentioned by two respondents to describe variables or data structures that exist exclusively for verification purposes and have no effect on runtime behavior. The resulting coding is presented in Table D.2.

In the videos, practitioners may have described Ghost State using varied terminology, or associated it with other strategies, rather than naming it as a distinct technique. When using ghost state, instead of referring to it by that term, practitioners may have described the operation in ways

such as introducing an auxiliary variable to help the verifier or adding a field to help express an invariant. In that case, the model may have associated those occurrences into a more general topic, such as Invariant Handling or Hypothesis Introduction, instead of coding a dedicated one for it. This demonstrates the value of complementing the results with a structured survey.

A subset of 3 responses described practices that do not constitute verification techniques in themselves, but are still worth mentioning for being a relevant strategy that shows how practitioners overcome challenges. Those participants responded that asking for help online or using an LLM to assist with proof construction or debugging. These responses were not coded as techniques, since they describe a source of assistance instead of a concrete action applied to the proof, but their presence suggests that LLM-assisted reasoning is already part of some verification workflow, which is discussed further as future work in Chapter 6.

The open-ended responses in TE03 do not affect the frequency and complexity results, which were collected exclusively through the closed-ended technique table. Their effect is limited to the scenario analysis, where techniques mentioned in an open-ended field were merged with that participant's closed-ended selections, so that a technique described in free text was credited in the same way as an explicit selection, compensating for the fact that participants could only choose from their individually assigned subset of 15.

5.6.7 Technique Application Scenarios

The *Technique Application Scenarios* section presented participants with four realistic verification challenges drawn from the intersection of the systematic literature review in Chapter 2 and the empirical findings of Oliveira, Mendes, and Carreira [30]. For each scenario, participants selected all techniques from their individually assigned subset of 15 that they would apply to overcome the described challenge. Because each respondent only saw 15 of the 33 techniques, selection rates are calculated as the number of times a technique was selected within the number of times that technique was shown to respondents who had not marked it as *I never used this technique* in the previous section. This denominator reflects the effective sample of respondents for whom the technique was both visible and applicable.

S01 - Suboptimal Proof Strategy

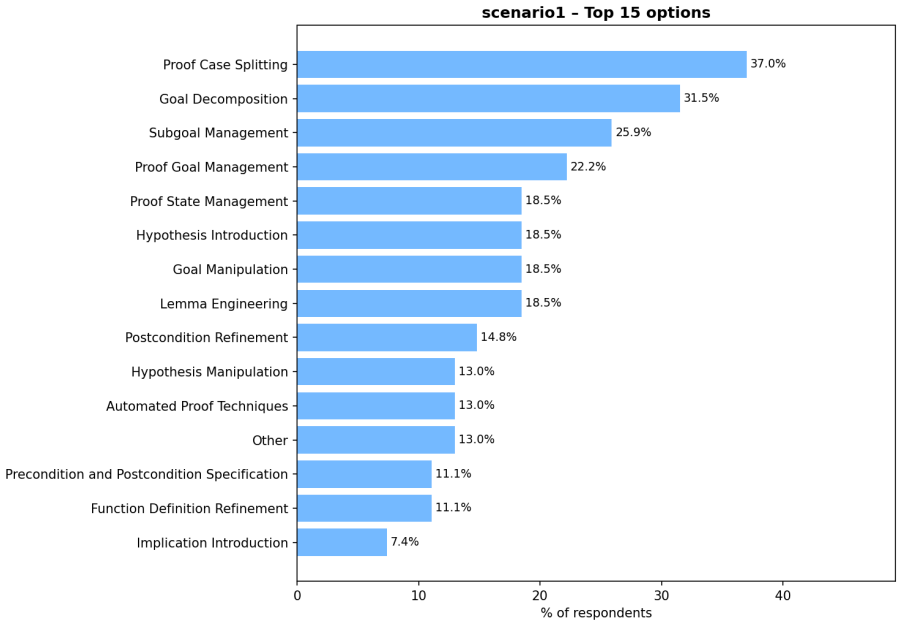
Figure 5.15 presents the top 15 selected techniques for S01. The dominance of Proof Case Splitting is informative, because instead of abandoning the inductive approach entirely, practitioners choose to decompose the inductive step into cases, distinguishing where the new element is inserted relative to the current head, so that each case presents the solver with a structurally simpler goal.

The selection of Goal Decomposition (31.5%) and Subgoal Management (25.9%) along with Proof Case Splitting confirms this interpretation. Practitioners do not apply a single technique but combine structural decomposition at the strategic level with Goal Decomposition, case analysis on the data with Proof Case Splitting, and syntactic scoping of the resulting obligations with Subgoal

Management. This three-technique pattern mirrors the most frequent triple in the co-occurrence analysis of Chapter 4 (T01 + T02 + T03), highlighting the observation that goal structuring is rarely a single action.

The presence of Lemma Engineering (18.5%) suggests that a subset of practitioners opts for a different strategy altogether, by extracting the auxiliary property that the insertion step preserves sortedness into a separately proved lemma, which the main inductive argument can then invoke without exposing the positional reasoning to the verifier's context.

Figure 5.15: Top 15 techniques selected by respondents for Scenario 1.



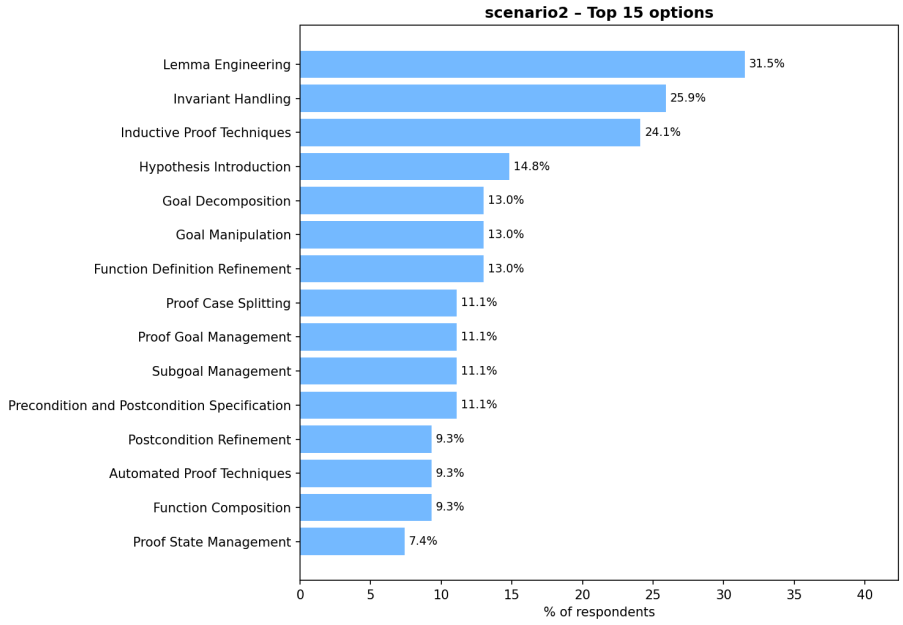
S02 - Proof Complexity Snowballing

Figure 5.16 presents the top 15 selected techniques for S02. Lemma Engineering's position as the top strategy applied in this scenario reflects the intuition that the root cause of context explosion is a missing layer of abstraction. Properties that the verifier is being asked to re-derive at every call site should instead be encapsulated as proved lemmas that can be invoked individually. This directly addresses the scalability concern of Mugnier et al. [28], who observe that proof contexts become unmanageable when intermediate facts are not extracted and the solver must reconstruct them from first principles on each invocation.

Furthermore, the selection of Invariant Handling (25.9%) and Inductive Proof Techniques (24.1%) along with Lemma Engineering suggests a second strategy operating at a lower level. Tightening the invariants that anchor the proof so that the verifier can exploit them without requiring additional hints is consistent with the iterative invariant strengthening pattern identified in Section 4.3 and with the observation of Brugger, Denis, and Müller [5] that proof stability requires correct invariants as well as invariants that are strong enough to propagate sufficient information across method boundaries.

Function Definition Refinement (13.0%) indicates that a subset of practitioners would address snowballing complexity by revising the definitions of functions involved in the proof, making them more transparent to the solver. This is consistent with the opacity management strategy of T07 discussed in Section 4.3. If a function definition is opaque, the verifier cannot exploit its structure when discharging obligations, forcing the user to provide more and more hints as the proof grows.

Figure 5.16: Top 15 techniques selected by respondents for Scenario 2.

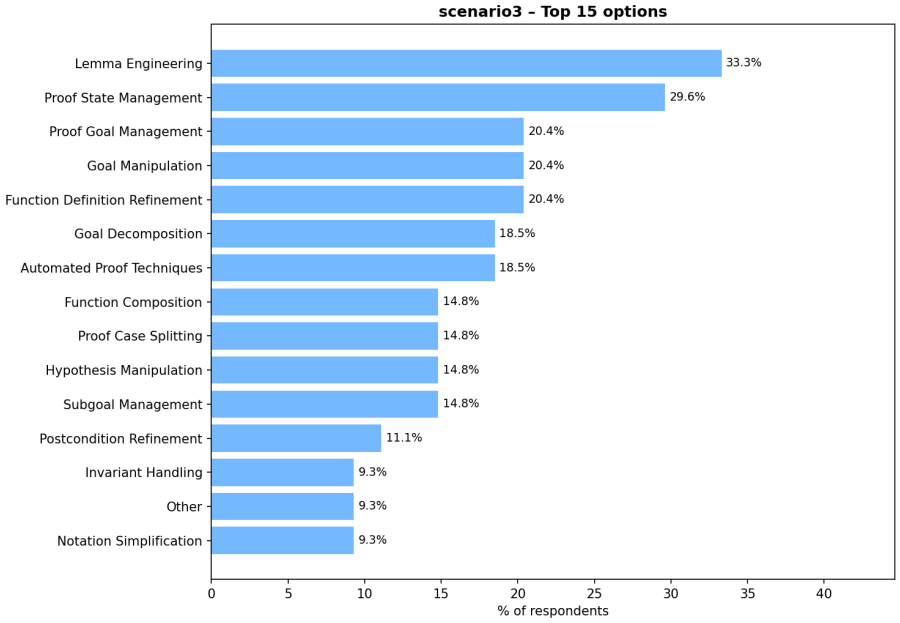


S03 - Opaque Verifier Failure

Figure 5.17 presents the top 15 selected techniques for S03. The strong presence of Lemma Engineering over all other techniques is notable because the scenario describes a failure with no diagnostic information. When the verifier provides no counterexample and no indication of which obligation failed, practitioners proactively reduce the verification task by extracting candidate intermediate facts into separately proved lemmas.

Proof State Management (29.6%) is the highest-ranked technique for this scenario relative to its overall frequency profile, where it sits in the mid-range. This elevation confirms that Proof State Management is specifically a diagnostic and recovery technique, applied when normal verification fails and the developer must reconstruct the implicit state the solver is working with. The strong co-selection of Goal Manipulation (20.4%) alongside Proof State Management confirms the pattern from Section 4.3. When the proof state is opaque, practitioners transform the goal into a syntactic form that reveals more structure.

Figure 5.17: Top 15 techniques selected by respondents for Scenario 3.



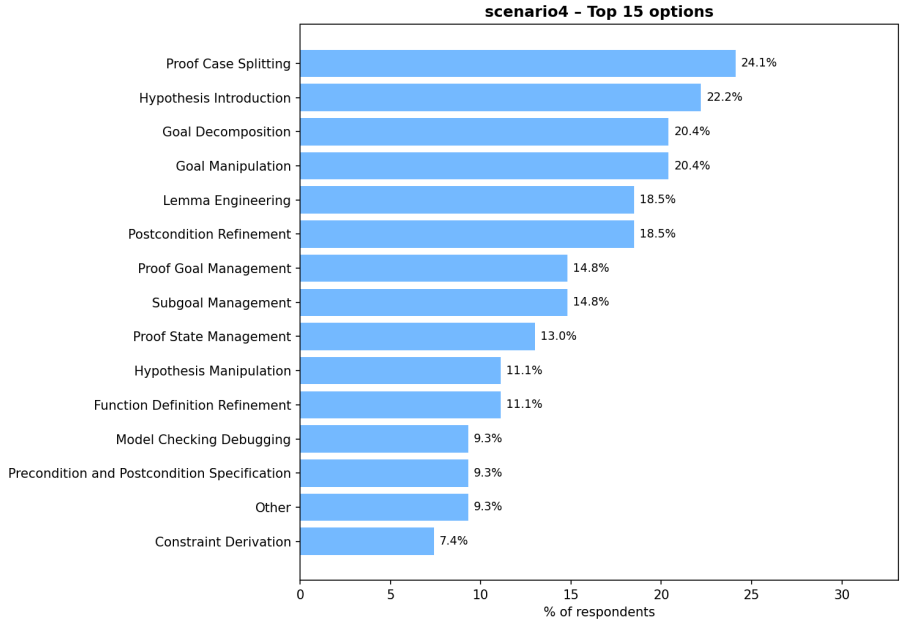
S04 - Uninformative Error Message

This scenario is structurally related to S03 but targets a more subtle case, where the verifier does provide a label for the failure but offers no diagnostic content beyond it. The selection pattern changes accordingly to the situation and Proof Case Splitting replaces Lemma Engineering at the top, indicating that practitioners read the partial information in the error message as a signal to decompose the failing obligation rather than to extract intermediate lemmas.

The high presence of Hypothesis Introduction (22.2%) suggests a complementary diagnostic strategy of enriching the local proof context with explicit intermediate facts to test whether the solver can discharge the postcondition once the missing assumptions are provided. If adding a hypothesis causes the proof to succeed, the practitioner has identified the gap, and if it still fails, the postcondition itself may be incorrectly stated or in fact violated.

Postcondition Refinement (18.5%) appears here at a higher rate than in any other scenario, which is consistent with the specific nature of the error message. A “postcondition might not hold” message is a direct signal to examine whether the postcondition is correctly stated, perhaps too weak to be provable from the method’s body or incorrectly capturing the intended property.

Figure 5.18: Top 15 techniques selected by respondents for Scenario 4.



Cross-Scenario Patterns

By analysing the four scenarios together, we can notice a core set of techniques used. T01, T15, T06 and T09 appear in the top selections of all four scenarios, with selection rates ranging between 13% and 33%. This stability confirms that these four techniques might constitute a general diagnostic set of tools. Regardless of the specific challenge type, practitioners reach first for proof decomposition, context management, goal rewriting and modular lemma extraction. This is consistent with the co-occurrence analysis of Chapter 4, where the same techniques form the dominant cluster of jointly applied strategies.

Three techniques show pronounced differences in selection rate across scenarios, indicating that they are specifically applied in particular challenge types. T20 rises from 7.4% in S02 to 29.6% in S03. This confirms that Proof State Management is primarily a diagnostic technique for opaque failures, not a routine part of the verification workflow. Its frequency profile, where 17% of respondents report using it *Very Frequently* and 21% *Never*, is consistent with this situational role.

Also, T13 grows from 9.3% in S02 to 18.5% in both S03 and S04, indicating that it is activated specifically when the verifier's feedback implicates the specification rather than the proof strategy. In S04, where the error message explicitly mentions the postcondition, this rise is expected and its appearance in S03, where no feedback is provided, suggests that practitioners treat a completely opaque failure as a signal to question the specification before questioning the proof.

Finally, T04 is the second leading technique in S02 (24.1%) but drops to below 6% in S03 and S04. This is consistent with S02's description of a growing proof context with recursive properties.

Induction is the natural tool for managing recursive structure, but it is not a diagnostic technique for opaque failures where the first challenge is identifying what is failing and not how to prove it.

5.7 Conclusion

This chapter validated the taxonomy of techniques identified in Chapter 4 through a survey of 54 practitioners, confirming that most of the assigned techniques match the most that are used in everyday practice. The results also revealed patterns of frequency and perceived difficulty across techniques, including a set of widely used techniques that are still hard to master.

These findings provide the basis for answering RQ3 and are discussed further, together with the study's limitations and future directions, in Chapter 6.

Chapter 6

Conclusion

The dissertation investigated the strategies practitioners apply to try to overcome challenges they face while working with VA programming languages.

Our work had three main goals, motivated by the gap between what verification can offer and how limited its adoption is. We began by documenting the existent challenges through a systematic literature review, followed by investigating practitioner knowledge from an unexplored multimedia sources, and finally, validating the results through a survey.

The literature confirmed that a good part of the barriers to the use of formal verification are mostly practical, so building on this, we designed a topic modeling pipeline over a dataset of 154 carefully chosen videos in which practitioners interact directly with VA tools, to gather the techniques they put to practice, when trying to overcome a challenge.

The analysis and refinement of the resulting topics allowed the creation of a taxonomy of 33 techniques, that were evaluated by a total of 54 survey participants. In general the survey confirmed that most of the assigned techniques in the corpus match the ones that are most used in everyday practice. This evaluation was done through the rating of frequency of use and perceived complexity of the techniques, and their application in four realistic scenarios, by beginner and advanced practitioners.

Overall, this work contributes with a replicable method for extracting developer knowledge and transform it into a set of concrete verification techniques whose frequency and difficulty profiles point to directions worth exploring. It can serve as a foundation for future investigations targeting specific challenges and as reference to guide the development of VA languages and their supporting environments.

6.1 Answers to the Research Questions

This section revisits each research question defined in Chapter 1, in order to summarize an answer to them and where they are supported.

6.1.1 RQ1: What challenges do practitioners face when using verification-aware languages in different contexts?

The challenges faced by practitioners are mainly described in Chapter 2, through the systematic literature review. The literature shows that the main barriers to the use of VA languages are strongly connected to usability, learning curve, tool maturity, solver feedback, and integration into existing development workflows. The main challenge types identified and highlighted are related with proof construction and debugging, specification quality, tool immaturity and integration, steep learning curve, and scalability. These challenges affect beginner practitioners, who struggle with unfamiliar specification and proof concepts, and also experienced users, who still face opaque verifier failures and maintenance difficulties as systems grow in complexity. Chapter 4 complements this answer by showing that the most common techniques observed in practice are concentrated in the areas directly associated with these challenges, in the areas of proof structuring, specification refinement, and lemma construction.

6.1.2 RQ2: What strategies and practices do practitioners employ to overcome these challenges when working with VA frameworks?

The strategies and practices used by developers to overcome the challenges are mostly addressed in Chapter 4, through the topic modeling analysis of the 154 verification video transcripts. The chapter describes the final taxonomy of 33 verification techniques, grouped into seven thematic categories, and shows that practitioners do not usually rely on individual actions. It can be seen that they apply multiple practice combinations to make their tasks more manageable. The most recurrent strategies are related to proof structuring, specification refinement, and lemma and hypothesis engineering, which together represent 70.9% of all attributions. By analysing the survey responses in Chapter 5, in general, the results validate the techniques found, especially Goal Decomposition, Hypothesis Introduction, Invariant Handling, Precondition and Postcondition Specification, Lemma Engineering, Proof Goal Management, and Proof Case Splitting that are very frequently used in practice. The scenario analysis also shows that practitioners adapt these strategies according to the type of challenge. They use decomposition and case splitting for suboptimal proof strategies, lemma engineering and invariant strengthening for proof complexity snowballing, proof state management and goal manipulation for opaque verifier failures, and postcondition refinement when verifier feedback points to a specification problem. Nevertheless, the techniques mentioned before appear as the top selections in all scenarios, indicating that practitioners tend to always reach to this set.

6.1.3 RQ3: What insights can be derived to improve the usability and adoption of VA programming languages and their supporting tools?

This research question is addressed by the combination of the corpus analysis and the survey results in Chapter 5. The results from this work present techniques that are more implicit behaviors that practitioners already perform, than innovative and unknown strategies. The contribution of

this study to RQ3 is a mapping of the techniques to specific challenges scenarios, creating a structured base for practical solutions. This also reveals where current tools are insufficient and where practitioner efforts are being spent due to those gaps.

The scenario analysis shows that certain techniques are put to use specifically in response to certain failures. For example, Goal Decomposition and Proof Case Splitting are more popular for suboptimal proof strategies, while Proof State Management and Goal Manipulation take over the selections when the verifier failure is opaque. The fact that practitioners are combining these specific techniques suggests that they are performing a cycle of decomposing and checking, and verification tools could try to automate and support this process.

Many current techniques, such as Proof State Management, Goal Manipulation and Model Checking Debugging, are acting as just a compensation to make up for inadequate tool feedback, increasing the necessary effort for a proof to be validated. Improving the diagnostic feedback provided by verifiers would reduce the need for these workarounds and decrease cognitive barriers, for example by distinguishing failing preconditions from failing postconditions and producing more structured error messages.

In addition, some of the techniques, like Inductive Proof Techniques and Automated Proof Techniques, that are some of the most needed, proved to be also some of the hardest to apply. This gap shows exactly where educational efforts should be invested to have the greatest impact on practitioner productivity.

Finally, the data shows another clear pattern. Some techniques show a combination of high "never used" rates with high perceived difficulty ratings. Instead of indicating that these techniques are irrelevant, this combination of evaluations can suggest that they face an excessively steep learning curve. This barrier could have the potential to be lowered through better documentation.

6.2 Threats to Validity

This section discusses the main factors that may have affected the validity of the results presented in this study.

6.2.1 Internal Validity

The final set of techniques was obtained from an LLM based topic modeling pipeline. Even though the prompts were iteratively refined and the resulting topics were carefully merged and refined, it is still a process that depends on the model's interpretation of what constitutes a technique. A different model or prompting set could generate different topics with different level of detail.

Human-based coding and refinement also introduces a risk to validity, since the interpretation can vary between reviewers. We tried to mitigate this, by independently review most of the analysis by two researchers, with disagreements resolved through discussion with a third, when needed.

Another threat relates to the composition of the search string used to build the video dataset. A part of the query terms was explicitly oriented towards error and debugging situations, reflecting

our interest in observing how practitioners resolve concrete obstacles. However, this choice may have biased the resulting corpus towards videos centered on error-handling, potentially under-representing techniques applied during proactive specification and proof design, where no error is triggered. As a result, the identified taxonomy may weigh reactive, error-driven techniques more heavily than techniques used in error-free verification workflows.

6.2.2 External Validity

The survey respondents sample is constituted of mostly highly educated, white and male participants, who in majority have the roles of researchers and students. It received a total of 132 responses, of which only 54 met the criteria, 4 were excluded due to lack of experience and the other 74 were abandoned before completion. Since the final sample is mainly constituted of more experienced practitioners, it is possible that less experienced or even less engaged participants tended to abandon the survey before reaching the more complex sections.

Even though the survey was designed to mitigate participant fatigue, the rate of dropouts suggests that the measures taken were still not sufficient to prevent it. Most dropouts happened in specific sections, with 49.2% occurring before the completion of the techniques section and 26.0% before the completion of the scenarios section. Those being the most demanding parts of the survey was likely what contributed for participants quitting.

6.3 Future Work

The findings and limitations of this study open directions for future work, which will be discussed in the current section.

- **Extending data sources**: Future work could extend the video methodology to additional platforms and formats, such as talks outside YouTube and discussion threads on Zulip servers used by formal methods communities.
- **Industry Validation**: The survey sample obtained in this study concentrates academic and research contexts, with a smaller number of respondents working in industry settings. A next step could be a follow-up targeting industrial practitioners, from organizations known to apply verification languages in software development. This validation could also take a new format, through structured interviews, to get a more detailed view on how techniques are adapted, combined, or rejected under failure.
- **Newly Identified Techniques**: The participants reported a new technique, the introduction of the Ghost State that was not validated because they were only discovered through the survey. Also, multiple responses mentioned other external strategies, such as asking for help online and using an LLM for help, and unlike the Ghost State technique, these responses could not be coded as techniques. However, given the growth of LLM-assisted reasoning in formal verification discussed in Chapter 2, this points to a direction worth investigating.

References

[1] Wolfgang Ahrendt et al. “The key tool”. In: *Software Systems Modeling* 4 (Feb. 2005), pp. 32–54. DOI: 10.1007/s10270-004-0058-x.

[2] Mahmoud Amiri and Thomas Bocklitz. *Chunk Twice, Embed Once: A Systematic Study of Segmentation and Representation Trade-offs in Chemistry-Aware Retrieval-Augmented Generation*. 2025. arXiv: 2506.17277 [cs.IR]. URL: https://arxiv.org/abs/2506.17277.

[3] Sofia Bennani and Charles Moslonka. *A Systematic Analysis of Chunking Strategies for Reliable Question Answering*. 2026. arXiv: 2601.14123 [cs.CL]. URL: https://arxiv.org/abs/2601.14123.

[4] Sergey Brin, Rajeev Motwani, and Craig Silverstein. “Beyond market baskets: generalizing association rules to correlations”. In: *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’97. Tucson, Arizona, USA: Association for Computing Machinery, 1997, pp. 265–276. ISBN: 0897919114. DOI: 10.1145/253260.253327. URL: https://doi.org/10.1145/253260.253327.

[5] Lea Salome Brugger, Xavier Denis, and Peter Müller. “Toward Practical Deductive Verification: Insights from a Qualitative Survey in Industry and Academia”. In: (2025). URL: https://arxiv.org/abs/2510.20514.

[6] Claudio Carpineto and Giovanni Romano. “A Survey of Automatic Query Expansion in Information Retrieval”. In: *ACM Comput. Surv.* 44.1 (Jan. 2012). ISSN: 0360-0300. DOI: 10.1145/2071389.2071390. URL: https://doi.org/10.1145/2071389.2071390.

[7] Ernie Cohen et al. “VCC: A Practical System for Verifying Concurrent C”. In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 23–42. ISBN: 978-3-642-03359-9.

[8] Jacob Cohen. “A Coefficient of Agreement for Nominal Scales”. In: *Educational and Psychological Measurement* 20.1 (1960), pp. 37–46. DOI: 10.1177/001316446002000104. eprint: https://doi.org/10.1177/001316446002000104. URL: https://doi.org/10.1177/001316446002000104.

[9] Christopher Cooper. “The Pre-processing of YouTube Transcripts for Corpus-based Spoken Language Analysis.” In: Oct. 2022.

[10] A. Ferrari et al. “Systematic Evaluation and Usability Analysis of Formal Methods Tools for Railway Signaling System Design”. In: *IEEE Transactions on Software Engineering* (2022). DOI: 10.1109/TSE.2021.3124677.

[11] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 — Where Programs Meet Provers". In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128. ISBN: 978-3-642-37036-6.

[12] Mario Gleirscher, Jaco van de Pol, and Jim Woodcock. "A manifesto for applicable formal methods". In: *Software and Systems Modeling* (2023). DOI: 10.1007/s10270-023-01124-2. URL: https://link.springer.com/article/10.1007/s10270-023-01124-2.

[13] G. Gokulnath and J. J. Kizhakkethottam. "A Study on Formal Verification of Smart Contracts in Distributed Ledger Technology". In: *2023 IEEE International Conference on Recent Advances in Systems Science and Engineering (RASSE)* (2023). DOI: 10.1109/RASSE60029.2023.10363616.

[14] Sarah Grebing, Jonas Klamroth, and Mattias Ulbrich. "Seamless Interactive Program Verification". In: Mar. 2020, pp. 68–86. ISBN: 978-3-030-41599-0. DOI: 10.1007/978-3-030-41600-3_6.

[15] Maarten Grootendorst. *BERTopic: Neural topic modeling with a class-based TF-IDF procedure*. 2022. arXiv: 2203.05794 [cs.CL]. URL: https://arxiv.org/abs/2203.05794.

[16] I. Haddou-Oumouloud et al. "Toward Secure and Reliable IoT Systems: A Comprehensive Review of Formal Methods Applications". In: *IEEE Access* (2024). DOI: 10.1109/ACCESS.2024.3501587.

[17] Hamed Jelodar et al. *Latent Dirichlet Allocation (LDA) and Topic modeling: models, applications, a survey*. 2018. arXiv: 1711.04305 [cs.IR]. URL: https://arxiv.org/abs/1711.04305.

[18] S. Juhošová, A. Zaidman, and J. Cockx. "Pinpointing the Learning Obstacles of an Interactive Theorem Prover". In: *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)* (2025). DOI: 10.1109/ICPC66645.2025.00024.

[19] Malcolm Koo and Shih-Wei Yang. "Likert-Type Scale". In: *Encyclopedia* 5 (Feb. 2025), p. 18. DOI: 10.3390/encyclopedia5010018.

[20] Herb Krasner. *The Cost of Poor Software Quality in the US: A 2022 Report*. Tech. rep. Consortium for Information & Software Quality (CISQ), 2022. URL: https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf.

[21] Alessandro Lattuada et al. "Verus: Verifying Rust Programs using Linear Ghost Types". In: *Proceedings of the ACM on Programming Languages (PACMPL)* 7.OOPSLA1 (2023). DOI: 10.1145/3586037. URL: https://dl.acm.org/doi/abs/10.1145/3586037.

[22] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. ISBN: 978-3-642-17511-4.

[23] Mario Gleirscher and Diego Marmsoler. "Formal methods in dependable systems engineering: a survey of professionals from Europe and North America". In: *Empirical Software Engineering* 25 (2020), pp. 4473–4546. DOI: 10.1007/s10664-020-09836-5. URL: https://link.springer.com/article/10.1007/s10664-020-09836-5.

[24] Atif Mashkoor, Michael Leuschel, and Alexander Egyed. *Validation Obligations: A Novel Approach to Check Compliance between Requirements and their Formal Specification*. 2021. arXiv: 2102.06037 [cs.SE]. URL: https://arxiv.org/abs/2102.06037.

[25] N. Mehdipour et al. "Formal methods to comply with rules of the road in autonomous driving: State of the art and grand challenges". In: *Automatica* 152 (2023). DOI: 10.1016/j.automatica.2022.110692. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85150893712&doi=10.1016%2Fj.automatica.2022.110692&partnerID=40&md5=f73f5e3f6f014f73bf827ab06a14b0a3.

[26] Jorge Melegati, Kieran Conboy, and Daniel Graziotin. "Qualitative Surveys in Software Engineering Research: Definition, Critical Review, and Guidelines". In: *IEEE Transactions on Software Engineering* PP (Dec. 2024), pp. 1–16. DOI: 10.1109/TSE.2024.3474173.

[27] Eric Mugnier et al. "Laurel: Unblocking Automated Verification with Large Language Models". In: *Proc. ACM Program. Lang. 9, OOPSLA1, Article 134 (April 2025), 27 pages* (2025). DOI: 10.1145/3720499.

[28] Eric Mugnier et al. "On the Impact of Formal Verification on Software Development". In: *Proc. ACM Program. Lang. 9, OOPSLA2, Article 403 (October 2025), 27 pages* (2025). DOI: 10.1145/3763181.

[29] Ulf Norell. "Dependently Typed Programming in Agda". In: Jan. 2009, pp. 1–2. ISBN: 978-3-642-04651-3. DOI: 10.1007/978-3-642-04652-0_5.

[30] Francisco Oliveira, Alexandra Mendes, and Carolina Carreira. "What Challenges Do Developers Face When Using Verification-Aware Programming Languages?" In: *arXiv preprint arXiv:2506.23696v2* (2025). URL: https://arxiv.org/abs/2506.23696v2.

[31] Chau Minh Pham et al. *TopicGPT: A Prompt-based Topic Modeling Framework*. 2024. arXiv: 2311.01449 [cs.CL]. URL: https://arxiv.org/abs/2311.01449.

[32] Renyi Qu, Ruixuan Tu, and Forrest Bao. *Is Semantic Chunking Worth the Computational Cost?* 2024. arXiv: 2410.13070 [cs.CL]. URL: https://arxiv.org/abs/2410.13070.

[33] Nils Reimers and Iryna Gurevych. "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks". In: *CoRR* abs/1908.10084 (2019). arXiv: 1908.10084. URL: http://arxiv.org/abs/1908.10084.

[34] Á. Silva, A. Mendes, and J. F. Ferreira. "Leveraging Large Language Models to Boost Dafny's Developers Productivity". In: *2024 IEEE/ACM 12th International Conference on Formal Methods in Software Engineering (FormaliSE)* (2024).

[35] Kaitao Song et al. *MPNet: Masked and Permuted Pre-training for Language Understanding*. 2020. arXiv: 2004.09297 [cs.CL]. URL: https://arxiv.org/abs/2004.09297.

[36] Ottokar Tilk and Tanel Alumäe. "Bidirectional Recurrent Neural Network with Attention Mechanism for Punctuation Restoration". In: Sept. 2016, pp. 3047–3051. DOI: 10.21437/Interspeech.2016-1517.

[37] Gonzalo Travieso and Luciando da F. Costa. *The Jaccard Similarity Mean*. 2024. arXiv: 2311.12959 [physics.data-an]. URL: https://arxiv.org/abs/2311.12959.

[38] C. Wang, M. Scazzariello, and M. Chiesa. "From Scientific Texts to Verifiable Code: Automating the Process with Transformers". In: *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)* (2025). DOI: 10.1109/LLM4Code66737.2025.00032.

[39] Zhiqing Wang et al. *Agentic Verification of Software Systems*. 2025. arXiv: 2511.17330. URL: https://arxiv.org/abs/2511.17330.

[40] JCP Woodcock et al. "Formal Methods: Practice and Experience". In: *ACM Computing Surveys* 41 (Oct. 2009). DOI: 10.1145/1592434.1592436.

[41] Chenyuan Yang et al. "AutoVerus: Automated Proof Generation for Rust Code". In: *Proceedings of the ACM on Programming Languages* 9.OOPSLA2 (Oct. 2025), pp. 3454–3482. ISSN: 2475-1421. DOI: 10.1145/3763174. URL: http://dx.doi.org/10.1145/3763174.

[42] Huan Zhao et al. *Lemma Discovery in Agentic Program Verification*. 2026. arXiv: 2603.22114 [cs.SE]. URL: https://arxiv.org/abs/2603.22114.

Appendix A

Semantic Similarity Filtering Details

Table A.1: Query set used for semantic similarity filtering.

Category	**Query phrases**
Proof Strategy Development	*"how to prove this property"* *"proof strategy for verification"* *"break a proof into smaller steps"* *"prove complex properties step by step"* *"splitting cases in the proof"* *"prove a property of a recursive function by induction"* *"pattern matching on the constructor"*
Verification Debugging	*"verification fails here"* *"why the solver cannot prove this"* *"fix this verification error"* *"what to do when the verifier fails"* *"debug a verification failure"* *"understand solver error messages"* *"identify why the verifier fails"* *"isolate the failing proof obligation"* *"analyze verification conditions"* *"debug SMT solver failures"* *"the SMT solver can discharge this goal"*

continued on next page

Table A.1 – *continued from previous page*

Category	**Query phrases**
Lemma and Invariant Management	*"introduce a lemma"* *"helper lemmas for proofs"* *"discover missing lemmas"* *"prove this lemma first"* *"introducing a helper lemma"* *"define loop invariants"* *"strengthen an invariant"* *"fix an incorrect invariant"* *"refine invariants to help the solver"* *"techniques for proving invariants"* *"the loop invariant holds at each iteration"*
Proof Decomposition and Scalability	*"fix this proof obligation"* *"function correctness proof"* *"verifying this function"* *"verify termination of a recursive function"* *"write a termination measure for this function"* *"unfolding the definition"* *"simplify verification conditions"*
Specification Writing and Annotation	*"guide the verifier with annotations"* *"guide the solver with assertions"* *"add proof hints"* *"add helper assertions"* *"make properties explicit for the verifier"* *"help the solver prove this property"* *"specifying preconditions and postconditions in Dafny"*
Proof Maintenance and Refactoring	*"refactor code for verification"* *"structure functions for easier proofs"*
Automation Control	*"control proof automation"* *"reduce solver search space"* *"avoid solver timeouts"* *"help automated theorem provers"*

continued on next page

Table A.1 – *continued from previous page*

Category	**Query phrases**
Model-Based Verification	*"specify system behaviour with states and transitions"* *"define the set of valid system states"* *"find a counterexample that violates a property"* *"check whether a safety property holds"* *"check whether a liveness property holds"* *"refine a model to eliminate spurious counterexamples"* *"model checking counterexample"* *"temporal property verification"*

Table A.2: Correspondence between query categories and the findings of Chapter 2.

Query Category	**Source in Chapter 2**
Proof Strategy Development	2.5.1.1
Verification Debugging	2.5.1.1, 2.5.3.2
Lemma and Invariant Management	2.5.1.1, 2.5.3.1, 2.5.3.2
Proof Decomposition and Scalability	2.5.1.1, 2.5.1.3
Specification Writing and Annotation	2.5.3.2
Proof Maintenance and Refactoring	2.5.1.3
Automation Control	2.5.1.1, 2.5.3.2
Model-Based Verification	2.5.3.2

Appendix B

Topic Modeling Details

B.1 Topic Generation Prompt

The following prompt was used in the `generate_topic_lvl1` stage of the TopicGPT pipeline. The placeholders `{Topics}` and `{Document}` are populated dynamically at runtime with the current topic hierarchy and the document being processed, respectively.

Listing B.1: Prompt used in the generation of topics in the TopicGPT pipeline

```
1 You will receive a document and a set of top-level topics from a topic
    hierarchy. The document contains transcripts from tutorials or live
    coding sessions about formal verification programming. Your task is
    to identify generalizable topics within the document that can act as
     top-level topics in the hierarchy. If any relevant topics are
    missing from the provided set, please add them. Otherwise, output
    the existing top-level topics as identified in the document.
2 The topics must be related to techniques or strategies used by
    developers to overcome verification challenges. These techniques
    should represent reusable methods that programmers apply when
    dealing with difficulties in the verification process.
3
4 Each document may contain multiple techniques, and you should output all
     relevant ones.
5
6 Ensure that topics are high-level and non-redundant. Merge closely
    related concepts into a single topic instead of creating multiple
    similar ones (e.g., induction techniques, proof by induction, and
    structural induction should be grouped under a single topic).
7
8 Identify reusable techniques and strategies that developers apply to
    overcome verification bottlenecks. A topic must not describe WHAT
    the goal is (e.g., 'Ensure Correctness' or 'Error Handling'), but
```

```
    HOW the programmer operates the language and the solver to achieve
    it (e.g., 'Invariant Strengthening', 'Ghost State Injection', or '
    Witness Construction').
9
10 ## Topic Guidelines
11 1. **Mechanics-First:** Labels must reflect the technical maneuver
    applied (e.g., using a 'decreases' clause to prove termination).
12 2. **Action-Oriented:** Topics should represent an intentional action by
     the programmer to resolve a verification mismatch or provide a hint
     to the solver.
13 3. **Generalizable:** Labels should be applicable across different
    formal verification tools (Dafny, Verus, SPARK, etc.) but remain
    technically precise.
14 4. **Non-Redundant:** Merge related concepts (e.g., 'Induction' and '
    Structural Induction') into the most descriptive technical term.
15
16 ### Negative Constraints (DO NOT USE)
17 Avoid high-level software engineering or architectural terms.
18 - DO NOT use: 'Error Handling', 'Assertion Management', 'Verification
    Process', 'Program Correctness', 'Logic Abstraction', 'Unit Testing
    ', 'Function Reuse', or 'SMT Solver Integration'.
19 - If a user is fixing a proof, identify the formal tactic used (e.g., '
    Postcondition Refinement' or 'Opaque Function Triggering').
20
21 [Top-level topics]
22 {Topics}
23
24 [Examples]
25 Example 1: Adding a new technique
26
27 Document:
28 When the verifier fails to prove the property automatically, the
    developer introduces intermediate lemmas to break the proof into
    smaller steps. These helper lemmas guide the prover and make the
    overall verification succeed.
29
30 Your response:
31 [1] Lemma Engineering: Introducing auxiliary lemmas or helper proofs to
    guide automated verification and decompose complex proofs into
    smaller, provable steps.
32
33 Example 2: Returning an existing technique
34
35 Document:
```

```
The developer inspects the counterexample produced by the verifier to understand why the proof fails and then adjusts the invariant accordingly.

Your response:
[1] Counterexample-Guided Debugging: Using counterexamples produced by verification tools to diagnose proof failures and refine specifications or invariants.

Example 3: Handling Recursive Data Structures
Document: "Since the function is recursive, I'll provide a termination metric using the height of the tree so the verifier can see the recursion depth is finite."
Your response:
[1] Termination Metric Specification: Defining explicit rank functions or measures (e.g., height, length) to prove that recursive calls or loops move towards a base case.

[Instructions]
Step 1: Determine topics mentioned in the document.
- The topic labels must be as GENERALIZABLE as possible. They must not be document-specific.
- The topics must reflect a SINGLE topic instead of a combination of topics.
- The new topics must have a level number, a short general label, and a topic description.
- Focus on the interaction between the programmer and the solver.
- Identify how the programmer "hints" or "guides" the prover.
- Avoid high-level software engineering terms. Do NOT output generic topics such as: 'Error Handling', 'Assertion Management', 'Verification Process', 'Program Correctness', or 'Logic Abstraction'. If a technique involves fixing an error, identify the specific mechanism used (e.g., 'Witness Construction' or 'Postcondition Refinement').
Step 2: Perform ONE of the following operations:
1. If there are already duplicates or relevant topics in the hierarchy, output those topics and stop here.
2. If the document contains no topic, return "None".
3. Otherwise, add your topic as a top-level topic. Stop here and output the added topic(s). DO NOT add any additional levels.

[Document]
{Document}
```

```
Please ONLY return the relevant or modified topics at the top level in the hierarchy. Your response should follow this format:
[Topic Level] Topic Label: Topic Description

Your response:
```

B.2 Topic Assignment Prompt

The following prompt was used in the `assign_topics` stage of the TopicGPT pipeline. The placeholders `{Topics}` and `{Document}` are populated dynamically at runtime with the current topic hierarchy and the document being processed, respectively.

Listing B.2: Prompt used in the assignment of topics in the TopicGPT pipeline

```
You will receive a document and a topic hierarchy. The document contains transcripts from tutorials or live coding sessions about formal verification. Assign the document to the most relevant techniques (topics) the hierarchy. Then, output the topic labels, assignment reasoning and supporting quotes from the document.
Carefully inspect the full topic list before making your decision. You should always pick the most suited topics for the given document.
A single document may be assigned to MULTIPLE topics.
DO NOT make up new topics or quotes.

[Topic Hierarchy]
{tree}

[Examples]
Example 1: Assign multiple techniques
Document:
When the proof fails, the developer inspects the counterexample and then adds a helper lemma to break the proof into smaller parts.

Assignment:
[1] Counterexample-Guided Debugging: The developer analyzes counterexamples to understand proof failures ("...inspects the counterexample...")
[1] Lemma Engineering: The developer introduces helper lemmas to decompose the proof ("...adds a helper lemma to break the proof...")

Example 2: Assign a single technique

Document:
```

```
The developer simplifies the goal step by step until it becomes trivial to prove.

Assignment:
[1] Goal Simplification: The developer reduces the complexity of the proof goal ("...simplifies the goal step by step...")

---

[Instructions]
1. Topic labels must be present in the provided topic hierarchy. You MUST NOT make up new topics.
2. The quote must be taken from the document. You MUST NOT make up quotes.
3. If the document is not related to challenges, issues, or obstacles faced by a practitioner, respond with "[0] No topics: no topics found ()".
4. If you respond with "[0] No topics: no topics found ()", you MUST NOT respond with any other topic.

[Document]
{Document}

Double check all topics and that your assignment exists in the list!
Your response should be in the following format:
[Topic Level] Topic Label: Assignment reasoning (Supporting quote)

Your response:
```

Table B.1: Distribution of identified top-level topics and their respective document counts.

ID	Topic	Description	Count
T01	Goal Decomposition	Breaking down complex proof goals into simpler sub-goals to facilitate the verification process.	138
T02	Hypothesis Introduction	Introducing assumptions or hypotheses into the proof to simplify the verification process and guide the prover.	95
T03	Automated Proof Techniques	Leveraging and adapting automated features and tactics within theorem provers to handle specific verification challenges and reduce manual effort.	82
T04	Inductive Proof Techniques	Utilizing induction principles and hypothesis application to guide and simplify proofs for recursive properties.	67

continued on next page

Table B.1 – *continued from previous page*

ID	Topic	Description	Count
T05	Invariant Handling	Utilizing and enhancing loop and class invariants to assist the prover in verifying that properties hold across all iterations.	55
T06	Goal Manipulation	Adjusting and refining proof goals to align with the capabilities of the theorem prover and enhance the likelihood of successful verification.	44
T07	Function Definition Refinement	Improving the definitions of functions to clarify their behavior and ensure that they meet the necessary specifications for verification.	40
T08	Precondition and Postcondition Specification	Defining explicit preconditions and postconditions in the program to guide the verifier in understanding the expected behavior and constraints of functions.	39
T09	Proof Goal Management	Organizing and breaking down proof goals to streamline the verification process and ensure clarity in the proof structure.	32
T10	Evidence Chain Validation	Constructing and validating chains of evidence to support the correctness of computations in verification.	29
T11	Recursive Call Handling	Managing recursive function calls by providing structural insights or metrics to assist in proving termination and correctness.	27
T12	Witness Construction	Creating and utilizing witness functions to demonstrate the validity of propositions and facilitate automated verification.	27
T13	Postcondition Refinement	Introducing specific postconditions in methods to ensure that properties hold after method execution, aiding in the verification process.	25
T14	Pattern Matching Utilization	Leveraging pattern matching techniques to simplify proof obligations by directly matching function arguments against constructors and handling cases systematically.	21
T15	Lemma Engineering	Introducing auxiliary lemmas or helper proofs to guide automated verification and decompose complex proofs into smaller, provable steps.	21
T16	Temporal Logic Specification	Defining properties and behaviors of systems using temporal logic to specify correctness, safety, and liveness conditions for model checking.	15
T17	Termination Metric Specification	Defining explicit rank functions or measures (e.g., size, depth) to prove that recursive calls or loops progress towards a base case.	13

continued on next page

Table B.1 – *continued from previous page*

ID	Topic	Description	Count
T18	Subgoal Management	Managing subgoals during the proof process to break down complex proofs into manageable parts, allowing for focused reasoning and verification.	10
T19	Model Checking Debugging	Utilizing debugging tools and techniques to analyze and refine specifications during the model checking process, including handling exceptions and evaluating formulas.	10
T20	Proof State Management	Organizing and managing the state of proofs to ensure clarity and facilitate the verification process, including hiding or revealing parts of the proof as needed.	7
T21	Monadic Verification	Using monadic structures to manage state and computations in the verification process, ensuring that properties are maintained throughout.	6
T22	Notation Simplification	Streamlining and clarifying notation to improve the readability and manageability of proofs, making it easier for both the human engineer and the proof assistant to process statements.	5
T23	Implication Introduction	Using the tactic of introducing implications to structure proofs and derive conclusions from assumptions.	4
T24	Congruence Rule Application	Using congruence rules to manipulate and rewrite expressions in order to facilitate proof verification.	4
T25	Hypothesis Manipulation	Adjusting and refining hypotheses and assumptions to facilitate proof progression and guide the verification process effectively.	3
T26	Proof Case Splitting	Dividing a proof or program specification into separate, manageable scenarios based on data structures or logical conditions.	3
T27	Transitivity Utilization	Leveraging the transitive property of equality to connect multiple equalities in a proof, facilitating the establishment of desired conclusions.	3
T28	Hint Database Management	Utilizing and managing a database of hints or tactics to assist automated proof search and improve efficiency in verification tasks.	2
T29	Substitution Function Application	Employing substitution techniques to replace variables or terms in proofs to demonstrate properties or equalities.	2
T30	Constraint Derivation	Applying constraints to guide the verification process and ensure that the properties of data types are satisfied.	1
T31	Function Composition	Combining multiple functions in a structured manner to create new functions, which can help in organizing proofs and clarifying relationships between components.	1

continued on next page

Table B.1 – *continued from previous page*

ID	Topic	Description	Count
T32	Inversion Principle Application	Utilizing the inversion principle to reason about the structure of data types and their constructors, aiding in the verification of properties related to those types.	1
T33	Proof Binding	Establishing connections between different parts of a proof or program to ensure that all necessary conditions are met for successful verification.	1

Appendix C

Detailed Survey

Consent

Study Title
Survey on Developer Experience and Workflows in Verification-Aware Languages

Summary
Our study involves a survey that aims to understand how developers interact with verification-aware programming languages in their daily workflows. We want to explore the practical strategies, problem-solving patterns, and usability challenges practitioners face when working with these tools. Your participation could provide valuable insights to help shaping better developer experiences, documentation, and tooling for the community.

Purpose
Our study involves a survey that aims to get information about your

background and experience with verification-aware programming languages. The survey will contribute to identifying and understanding the barriers to the adoption and usability of software verification technologies, helping uncover how development practices and tools can be improved.

Procedures
In this study, you'll answer questions about your experience with verification-aware programming languages and software engineering practices. None of your personal data will be collected during the study. The expected duration of the survey is 15 minutes.

Participant Requirements
Participation in this study is limited to individuals that are fluent in English and have some level of prior experience with verification-aware programming languages.

Risks
The risks and discomfort associated with participation in this study

---

are no greater than those ordinarily encountered in daily life or other online activities.

Compensation & Costs
There is no compensation to complete the form. There may be no personal benefit from your participation in the study, but the knowledge received may be of value to humanity.

Personal data and Confidentiality
No personal data will be collected or processed in the context of this research. The survey is completely anonymous, and no personally identifiable information is captured at any point. In open-ended questions, please do not reveal any personally identifiable information about yourself or others. The data is processed based on your consent. Access is restricted to the researchers conducting the study and password-controlled.

Right to Ask Questions & Contact Information
If you have any questions about this study, you should feel free to

ask them by contacting the Principal Investigator now at up201907201@up.pt. If you have questions later, desire additional information, or wish to withdraw your participation, please contact either of the Principal Investigators by mail, phone, or e-mail in accordance with the contact information listed above.

Voluntary Participation
Your participation in this research is voluntary. You may discontinue participation at any time during the research activity. You may print a copy of this consent form for your records.

Researcher responsible for this study, Principal Investigator
Alexandra Mendes
Assistant Professor at Department of Informatics Engineering, Faculty of Engineering, University of Porto
afmendes@fe.up.pt

Researchers conducting this study (collecting data, etc.)
Sofia Moura
Master's Student
Department of Informatics Engineering, Faculdade de Engenharia da Universidade do Porto, Universidade do Porto
up201907201@up.pt

Carolina Carreira
PhD Student
Computer Science and Engineering Department, Instituto Superior Técnico, Universidade de Lisboa
carolina.carreira@tecnico.ulisboa.pt

I have read and understand the information above.

- ○ Yes
- ○ No

I am over 18 years old, fluent in English, and I want to participate in this research and continue with the survey.

- ○ Yes
- ○ No

How many years of experience with Verification-Aware (VA) programming languages do you have?

Verification-Aware languages are programming languages or tools that support mathematically proving the correctness of software against its specification

Background

What is your current role? *Please select all that apply.*

- ☐ Student
- ☐ Researcher
- ☐ Software Engineer
- ☐ Other

Which of the following VA languages have you worked with? *Please select all that apply.*

- ☐ Agda
- ☐ Bandera
- ☐ Rocq
- ☐ Dafny
- ☐ Escher C Verifier
- ☐ Eiffel
- ☐ Frama-C
- ☐ Idris
- ☐ Java PathFinder
- ☐ JML
- ☐ KeY
- ☐ Lean 4
- ☐ mbeddr
- ☐ P
- ☐ Spec#
- ☐ Spark Ada
- ☐ TLA+
- ☐ VCC
- ☐ VeriFast
- ☐ Verus
- ☐ Whiley
- ☐ Why3
- ☐ Other

Where did you primarly learn how to program with VA languages? *Please select all that apply.*

- [ ] Official Documentation/Reference Manuals
- [ ] Online Discussions
- [ ] Online Video Tutorials
- [ ] Optional University Course
- [ ] Mandatory University Course
- [ ] Work context
- [ ] Other

In which context do you mainly use VA languages? *Please select all that apply.*

- [ ] Academic
- [ ] Research
- [ ] Industrial
- [ ] Personal projects
- [ ] Teaching
- [ ] Other

Bloco 4

For each of the following verification techniques, indicate how hard it is to master (from Very Easy to Very Difficult) and how often you use it (from Never to Very Frequently). *Please answer both ratings for each technique before moving on to the next one.*

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
Proof Case Splitting *Dividing a proof or program specification into separate, manageable scenarios based on data structures or logical conditions.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Inductive Proof Techniques *Utilizing induction principles and hypothesis application to guide and simplify proofs for recursive properties.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Invariant Handling *Utilizing and enhancing loop and class invariants to assist the prover in verifying that properties hold across all iterations.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Lemma Engineering *Introducing auxiliary lemmas or helper proofs to guide automated verification and decompose complex proofs into smaller, provable steps.*	○	○	○	○	○	○	

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Pattern Matching Utilization *Leveraging pattern matching techniques to simplify proof obligations by directly matching function arguments against constructors and handling cases systematically.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Termination Metric Specification *Defining explicit rank functions or measures (e.g., size, depth) to prove that recursive calls or loops progress towards a base case.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Recursive Call Handling *Managing recursive function calls by providing structural insights or metrics to assist in proving termination and correctness.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Function Definition Refinement *Improving the definitions of functions to clarify their behavior and ensure that they meet the necessary specifications for verification.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Precondition and Postcondition Specification *Defining explicit preconditions and postconditions in the program to guide the verifier in understanding the expected behavior and constraints of functions.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Congruence Rule Application *Using congruence rules to manipulate and rewrite expressions in order to facilitate proof verification.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Goal Decomposition *Breaking down complex proof goals into simpler sub-goals to facilitate the verification process.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Goal Manipulation *Adjusting and refining proof goals to align with the capabilities of the theorem prover and enhance the likelihood of successful verification.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Automated Proof Techniques *Leveraging and adapting automated features and tactics within theorem provers to handle specific verification challenges and reduce manual effort.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Hypothesis Introduction *Introducing assumptions or hypotheses into the proof to simplify the verification process and guide the prover.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Implication Introduction *Using the tactic of introducing implications to structure proofs and derive conclusions from assumptions.*	○	○	○	○	○	○	

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Substitution Function Application *Employing substitution techniques to replace variables or terms in proofs to demonstrate properties or equalities.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Transitivity Utilization *Leveraging the transitive property of equality to connect multiple equalities in a proof, facilitating the establishment of desired conclusions.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Proof State Management *Organizing and managing the state of proofs to ensure clarity and facilitate the verification process, including hiding or revealing parts of the proof as needed.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Hypothesis Manipulation *Adjusting and refining hypotheses and assumptions to facilitate proof progression and guide the verification process effectively.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Witness Construction *Creating and utilizing witness functions to demonstrate the validity of propositions and facilitate automated verification.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Proof Goal Management *Organizing and breaking down proof goals to streamline the verification process and ensure clarity in the proof structure.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Subgoal Management *Managing subgoals during the proof process to break down complex proofs into manageable parts, allowing for focused reasoning and verification.*	○	○	○	○	○	○	

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Postcondition Refinement *Introducing specific postconditions in methods to ensure that properties hold after method execution, aiding in the verification process.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Temporal Logic Specification *Defining properties and behaviors of systems using temporal logic to specify correctness, safety, and liveness conditions for model checking.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
Proof Case Attention *This is an attention check question, please select "Easy", and "Frequently", respectively. You can ignore the rest of this sentence.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
Model Checking Debugging *Utilizing debugging tools and techniques to analyze and refine specifications during the model checking process, including handling exceptions and evaluating formulas.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
Function Composition *Combining multiple functions in a structured manner to create new functions, which can help in organizing proofs and clarifying relationships between components.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
Proof Binding *Establishing connections between different parts of a proof or program to ensure that all necessary conditions are met for successful verification.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Inversion Principle Application *Utilizing the inversion principle to reason about the structure of data types and their constructors, aiding in the verification of properties related to those types.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Hint Database Management *Utilizing and managing a database of hints or tactics to assist automated proof search and improve efficiency in verification tasks.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Constraint Derivation *Applying constraints to guide the verification process and ensure that the properties of data types are satisfied.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Evidence Chain Validation *Constructing and validating chains of evidence to support the correctness of computations in verification.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Monadic Verification *Using monadic structures to manage state and computations in the verification process, ensuring that properties are maintained throughout.*	○	○	○	○	○	○	

	How hard is it to use this technique						How often do you use this techniques?
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
» Notation Simplification *Streamlining and clarifying notation to improve the readability and manageability of proofs, making it easier for the proof assistant to process and verify statements.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	
Constraint Attention *This is an attention check question, please select "Neutral", and "Never". You can ignore the rest of this sentence.*	○	○	○	○	○	○	
	Very Easy	Easy	Neutral	Difficult	Very Difficult	I never used this technique	

Where did you primarily learn these techniques? *Please select all that apply.*

- ☐ Official Documentation/Reference Manuals
- ☐ Academic Papers
- ☐ Online Discussions
- ☐ Online Video Tutorials
- ☐ Optional University Course
- ☐ Mandatory University Course
- ☐ Work Context
- ☐ Other

Is there any technique or problem-solving pattern that you frequently use in your real-life verification workflow that was not listed above?

- ○ Yes
- ○ No
- ○ I'm not sure

Please specify any technique or problem-solving pattern that you frequently use in your real-life verification workflow that was not listed above.

Bloco 2

Read the following 4 scenarios carefully and select the technique(s) you would most likely use to address each situation.

You are verifying a method, but the verifier reports that the proof cannot be completed. The tool only indicates that verification failed, without showing which assumptions were available, which proof obligation could not be established, or how the verifier reached its conclusion. As a result, you cannot determine why the proof failed. **Which technique(s) would you use to overcome this?**

- [ ] Proof Case Splitting
  Dividing a proof or program specification into separate, manageable scenarios based on data structures or logical conditions.
- [ ] Congruence Rule Application
  Using congruence rules to manipulate and rewrite expressions in order to facilitate proof verification.
- [ ] Hypothesis Manipulation
  Adjusting and refining hypotheses and assumptions to facilitate proof progression and guide the verification process effectively.
- [ ] Proof Binding
  Establishing connections between different parts of a proof or program to ensure that all necessary conditions are met for successful verification.
- [ ] Inductive Proof Techniques
  Utilizing induction principles and hypothesis application to guide and simplify proofs for recursive properties.
- [ ] Goal Decomposition
  Breaking down complex proof goals into simpler sub-goals to facilitate the verification process.
- [ ] Witness Construction
  Creating and utilizing witness functions to demonstrate the validity of propositions and facilitate automated verification.
- [ ] Inversion Principle Application
  Utilizing the inversion principle to reason about the structure of data types and their constructors, aiding in the verification of properties related to those types.
- [ ] Invariant Handling
  Utilizing and enhancing loop and class invariants to assist the prover in verifying that properties hold across all iterations.
- [ ] Goal Manipulation
  Adjusting and refining proof goals to align with the capabilities of the theorem prover and enhance the likelihood of successful verification.
- [ ] Proof Goal Management
  Organizing and breaking down proof goals to streamline the verification process and ensure clarity in the proof structure.
- [ ] Hint Database Management
  Utilizing and managing a database of hints or tactics to assist automated proof search and improve efficiency in verification tasks.

- [ ] **Lemma Engineering**
  Introducing auxiliary lemmas or helper proofs to guide automated verification and decompose complex proofs into smaller, provable steps.
- [ ] **Automated Proof Techniques**
  Leveraging and adapting automated features and tactics within theorem provers to handle specific verification challenges and reduce manual effort.
- [ ] **Subgoal Management**
  Managing subgoals during the proof process to break down complex proofs into manageable parts, allowing for focused reasoning and verification.
- [ ] **Constraint Derivation**
  Applying constraints to guide the verification process and ensure that the properties of data types are satisfied.
- [ ] **Pattern Matching Utilization**
  Leveraging pattern matching techniques to simplify proof obligations by directly matching function arguments against constructors and handling cases systematically.
- [ ] **Hypothesis Introduction**
  Introducing assumptions or hypotheses into the proof to simplify the verification process and guide the prover.
- [ ] **Postcondition Refinement**
  Introducing specific postconditions in methods to ensure that properties hold after method execution, aiding in the verification process.
- [ ] **Evidence Chain Validation**
  Constructing and validating chains of evidence to support the correctness of computations in verification.
- [ ] **Termination Metric Specification**
  Defining explicit rank functions or measures (e.g., size, depth) to prove that recursive calls or loops progress towards a base case.
- [ ] **Implication Introduction**
  Using the tactic of introducing implications to structure proofs and derive conclusions from assumptions.
- [ ] **Temporal Logic Specification**
  Defining properties and behaviors of systems using temporal logic to specify correctness, safety, and liveness conditions for model checking.
- [ ] **Monadic Verification**
  Using monadic structures to manage state and computations in the verification process, ensuring that properties are maintained throughout.

- [ ] **Recursive Call Handling**
  Managing recursive function calls by providing structural insights or metrics to assist in proving termination and correctness.
- [ ] **Substitution Function Application**
  Employing substitution techniques to replace variables or terms in proofs to demonstrate properties or equalities.
- [ ] **Model Checking Debugging**
  Utilizing debugging tools and techniques to analyze and refine specifications during the model checking process, including handling exceptions and evaluating formulas.
- [ ] **Notation Simplification**
  Streamlining and clarifying notation to improve the readability and manageability of proofs, making it easier for the proof assistant to process and verify statements.
- [ ] **Function Definition Refinement**
  Improving the definitions of functions to clarify their behavior and ensure that they meet the necessary specifications for verification.
- [ ] **Transitivity Utilization**
  Leveraging the transitive property of equality to connect multiple equalities in a proof, facilitating the establishment of desired conclusions.
- [ ] **Function Composition**
  Combining multiple functions in a structured manner to create new functions, which can help in organizing proofs and clarifying relationships between components.
- [ ] Other

  _____
- [ ] **Precondition and Postcondition Specification**
  Defining explicit preconditions and postconditions in the program to guide the verifier in understanding the expected behavior and constraints of functions.
- [ ] **Proof State Management**
  Organizing and managing the state of proofs to ensure clarity and facilitate the verification process, including hiding or revealing parts of the proof as needed.

You are verifying a sorting algorithm with a proof by induction. In the inductive step, you need to prove that inserting a new element into an already sorted list of size preserves the sorted property. Although your current approach could eventually work, you recognize it becomes non-ideal, as tracing the element's exact final position through conditional swaps generates complex subgoals. **Which technique(s) would you use to overcome this?**

- [ ] » Proof Case Splitting
  Dividing a proof or program specification into separate, manageable scenarios based on data structures or logical conditions.
- [ ] » Congruence Rule Application
  Using congruence rules to manipulate and rewrite expressions in order to facilitate proof verification.
- [ ] » Hypothesis Manipulation
  Adjusting and refining hypotheses and assumptions to facilitate proof progression and guide the verification process effectively.
- [ ] » Proof Binding
  Establishing connections between different parts of a proof or program to ensure that all necessary conditions are met for successful verification.

- [ ] » Inductive Proof Techniques
  Utilizing induction principles and hypothesis application to guide and simplify proofs for recursive properties.
- [ ] » Goal Decomposition
  Breaking down complex proof goals into simpler sub-goals to facilitate the verification process.
- [ ] » Witness Construction
  Creating and utilizing witness functions to demonstrate the validity of propositions and facilitate automated verification.
- [ ] » Inversion Principle Application
  Utilizing the inversion principle to reason about the structure of data types and their constructors, aiding in the verification of properties related to those types.
- [ ] » Invariant Handling
  Utilizing and enhancing loop and class invariants to assist the prover in verifying that properties hold across all iterations.
- [ ] » Goal Manipulation
  Adjusting and refining proof goals to align with the capabilities of the theorem prover and enhance the likelihood of successful verification.
- [ ] » Proof Goal Management
  Organizing and breaking down proof goals to streamline the verification process and ensure clarity in the proof structure.
- [ ] » Hint Database Management
  Utilizing and managing a database of hints or tactics to assist automated proof search and improve efficiency in verification tasks.
- [ ] » Lemma Engineering
  Introducing auxiliary lemmas or helper proofs to guide automated verification and decompose complex proofs into smaller, provable steps.
- [ ] » Automated Proof Techniques
  Leveraging and adapting automated features and tactics within theorem provers to handle specific verification challenges and reduce manual effort.
- [ ] » Subgoal Management
  Managing subgoals during the proof process to break down complex proofs into manageable parts, allowing for focused reasoning and verification.
- [ ] » Constraint Derivation
  Applying constraints to guide the verification process and ensure that the properties of data types are satisfied.

- [ ] **Pattern Matching Utilization**
  Leveraging pattern matching techniques to simplify proof obligations by directly matching function arguments against constructors and handling cases systematically.
- [ ] **Hypothesis Introduction**
  Introducing assumptions or hypotheses into the proof to simplify the verification process and guide the prover.
- [ ] **Postcondition Refinement**
  Introducing specific postconditions in methods to ensure that properties hold after method execution, aiding in the verification process.
- [ ] **Evidence Chain Validation**
  Constructing and validating chains of evidence to support the correctness of computations in verification.
- [ ] **Termination Metric Specification**
  Defining explicit rank functions or measures (e.g., size, depth) to prove that recursive calls or loops progress towards a base case.
- [ ] **Implication Introduction**
  Using the tactic of introducing implications to structure proofs and derive conclusions from assumptions.
- [ ] **Temporal Logic Specification**
  Defining properties and behaviors of systems using temporal logic to specify correctness, safety, and liveness conditions for model checking.
- [ ] **Monadic Verification**
  Using monadic structures to manage state and computations in the verification process, ensuring that properties are maintained throughout.
- [ ] **Recursive Call Handling**
  Managing recursive function calls by providing structural insights or metrics to assist in proving termination and correctness.
- [ ] **Substitution Function Application**
  Employing substitution techniques to replace variables or terms in proofs to demonstrate properties or equalities.
- [ ] **Model Checking Debugging**
  Utilizing debugging tools and techniques to analyze and refine specifications during the model checking process, including handling exceptions and evaluating formulas.
- [ ] **Notation Simplification**
  Streamlining and clarifying notation to improve the readability and manageability of proofs, making it easier for the proof assistant to process and verify statements.

- [ ] **Function Definition Refinement**
  Improving the definitions of functions to clarify their behavior and ensure that they meet the necessary specifications for verification.
- [ ] **Transitivity Utilization**
  Leveraging the transitive property of equality to connect multiple equalities in a proof, facilitating the establishment of desired conclusions.
- [ ] **Function Composition**
  Combining multiple functions in a structured manner to create new functions, which can help in organizing proofs and clarifying relationships between components.
- [ ] Other
- [ ] **Precondition and Postcondition Specification**
  Defining explicit preconditions and postconditions in the program to guide the verifier in understanding the expected behavior and constraints of functions.
- [ ] **Proof State Management**
  Organizing and managing the state of proofs to ensure clarity and facilitate the verification process, including hiding or revealing parts of the proof as needed.

You are verifying an implementation and, initially, proving basic properties required only a few annotations. However, as you introduce stronger guarantees involving multiple method interactions and object states, the verification context explodes. Each new property demands several additional invariants and

lemmas, causing verification times to spike and minor code changes to break previously successful proofs. **Which technique(s) would you use to overcome this?**

- [ ] Proof Case Splitting
  Dividing a proof or program specification into separate, manageable scenarios based on data structures or logical conditions.
- [ ] Congruence Rule Application
  Using congruence rules to manipulate and rewrite expressions in order to facilitate proof verification.
- [ ] Hypothesis Manipulation
  Adjusting and refining hypotheses and assumptions to facilitate proof progression and guide the verification process effectively.
- [ ] Proof Binding
  Establishing connections between different parts of a proof or program to ensure that all necessary conditions are met for successful verification.
- [ ] Inductive Proof Techniques
  Utilizing induction principles and hypothesis application to guide and simplify proofs for recursive properties.
- [ ] Goal Decomposition
  Breaking down complex proof goals into simpler sub-goals to facilitate the verification process.
- [ ] Witness Construction
  Creating and utilizing witness functions to demonstrate the validity of propositions and facilitate automated verification.
- [ ] Inversion Principle Application
  Utilizing the inversion principle to reason about the structure of data types and their constructors, aiding in the verification of properties related to those types.
- [ ] Invariant Handling
  Utilizing and enhancing loop and class invariants to assist the prover in verifying that properties hold across all iterations.
- [ ] Goal Manipulation
  Adjusting and refining proof goals to align with the capabilities of the theorem prover and enhance the likelihood of successful verification.
- [ ] Proof Goal Management
  Organizing and breaking down proof goals to streamline the verification process and ensure clarity in the proof structure.
- [ ] Hint Database Management
  Utilizing and managing a database of hints or tactics to assist automated proof search and improve efficiency in verification tasks.

- [ ] Lemma Engineering
  Introducing auxiliary lemmas or helper proofs to guide automated verification and decompose complex proofs into smaller, provable steps.
- [ ] Automated Proof Techniques
  Leveraging and adapting automated features and tactics within theorem provers to handle specific verification challenges and reduce manual effort.
- [ ] Subgoal Management
  Managing subgoals during the proof process to break down complex proofs into manageable parts, allowing for focused reasoning and verification.
- [ ] Constraint Derivation
  Applying constraints to guide the verification process and ensure that the properties of data types are satisfied.
- [ ] Pattern Matching Utilization
  Leveraging pattern matching techniques to simplify proof obligations by directly matching function arguments against constructors and handling cases systematically.
- [ ] Hypothesis Introduction
  Introducing assumptions or hypotheses into the proof to simplify the verification process and guide the prover.
- [ ] Postcondition Refinement
  Introducing specific postconditions in methods to ensure that properties hold after method execution, aiding in the verification process.
- [ ] Evidence Chain Validation
  Constructing and validating chains of evidence to support the correctness of computations in verification.
- [ ] Termination Metric Specification
  Defining explicit rank functions or measures (e.g., size, depth) to prove that recursive calls or loops progress towards a base case.
- [ ] Implication Introduction
  Using the tactic of introducing implications to structure proofs and derive conclusions from assumptions.
- [ ] Temporal Logic Specification
  Defining properties and behaviors of systems using temporal logic to specify correctness, safety, and liveness conditions for model checking.
- [ ] Monadic Verification
  Using monadic structures to manage state and computations in the verification process, ensuring that properties are maintained throughout.

» Recursive Call Handling
Managing recursive function calls by providing structural insights or metrics to assist in proving termination and correctness.

» Substitution Function Application
Employing substitution techniques to replace variables or terms in proofs to demonstrate properties or equalities.

» Model Checking Debugging
Utilizing debugging tools and techniques to analyze and refine specifications during the model checking process, including handling exceptions and evaluating formulas.

» Notation Simplification
Streamlining and clarifying notation to improve the readability and manageability of proofs, making it easier for the proof assistant to process and verify statements.

» Function Definition Refinement
Improving the definitions of functions to clarify their behavior and ensure that they meet the necessary specifications for verification.

» Transitivity Utilization
Leveraging the transitive property of equality to connect multiple equalities in a proof, facilitating the establishment of desired conclusions.

» Function Composition
Combining multiple functions in a structured manner to create new functions, which can help in organizing proofs and clarifying relationships between components.

Other

» Precondition and Postcondition Specification
Defining explicit preconditions and postconditions in the program to guide the verifier in understanding the expected behavior and constraints of functions.

» Proof State Management
Organizing and managing the state of proofs to ensure clarity and facilitate the verification process, including hiding or revealing parts of the proof as needed.

Your verifier fails, throwing a generic "Verification failed: Postcondition might not hold" error that points vaguely to the end of a block. With no counterexamples or detailed logs, you cannot tell if you have a real bug or if the SMT solver is just hitting a limitation. **Which technique(s) would you use to overcome this?**

» Proof Case Splitting
Dividing a proof or program specification into separate, manageable scenarios based on data structures or logical conditions.

» Congruence Rule Application
Using congruence rules to manipulate and rewrite expressions in order to facilitate proof verification.

» Hypothesis Manipulation
Adjusting and refining hypotheses and assumptions to facilitate proof progression and guide the verification process effectively.

» Proof Binding
Establishing connections between different parts of a proof or program to ensure that all necessary conditions are met for successful verification.

» Inductive Proof Techniques
Utilizing induction principles and hypothesis application to guide and simplify proofs for recursive properties.

» Goal Decomposition
Breaking down complex proof goals into simpler sub-goals to facilitate the verification process.

» Witness Construction
Creating and utilizing witness functions to demonstrate the validity of propositions and facilitate automated verification.

» Inversion Principle Application
Utilizing the inversion principle to reason about the structure of data types and their constructors, aiding in the verification of properties related to those types.

- [ ] **Invariant Handling**
  Utilizing and enhancing loop and class invariants to assist the prover in verifying that properties hold across all iterations.
- [ ] **Goal Manipulation**
  Adjusting and refining proof goals to align with the capabilities of the theorem prover and enhance the likelihood of successful verification.
- [ ] **Proof Goal Management**
  Organizing and breaking down proof goals to streamline the verification process and ensure clarity in the proof structure.
- [ ] **Hint Database Management**
  Utilizing and managing a database of hints or tactics to assist automated proof search and improve efficiency in verification tasks.
- [ ] **Lemma Engineering**
  Introducing auxiliary lemmas or helper proofs to guide automated verification and decompose complex proofs into smaller, provable steps.
- [ ] **Automated Proof Techniques**
  Leveraging and adapting automated features and tactics within theorem provers to handle specific verification challenges and reduce manual effort.
- [ ] **Subgoal Management**
  Managing subgoals during the proof process to break down complex proofs into manageable parts, allowing for focused reasoning and verification.
- [ ] **Constraint Derivation**
  Applying constraints to guide the verification process and ensure that the properties of data types are satisfied.
- [ ] **Pattern Matching Utilization**
  Leveraging pattern matching techniques to simplify proof obligations by directly matching function arguments against constructors and handling cases systematically.
- [ ] **Hypothesis Introduction**
  Introducing assumptions or hypotheses into the proof to simplify the verification process and guide the prover.
- [ ] **Postcondition Refinement**
  Introducing specific postconditions in methods to ensure that properties hold after method execution, aiding in the verification process.
- [ ] **Evidence Chain Validation**
  Constructing and validating chains of evidence to support the correctness of computations in verification.

- [ ] **Termination Metric Specification**
  Defining explicit rank functions or measures (e.g., size, depth) to prove that recursive calls or loops progress towards a base case.
- [ ] **Implication Introduction**
  Using the tactic of introducing implications to structure proofs and derive conclusions from assumptions.
- [ ] **Temporal Logic Specification**
  Defining properties and behaviors of systems using temporal logic to specify correctness, safety, and liveness conditions for model checking.
- [ ] **Monadic Verification**
  Using monadic structures to manage state and computations in the verification process, ensuring that properties are maintained throughout.
- [ ] **Recursive Call Handling**
  Managing recursive function calls by providing structural insights or metrics to assist in proving termination and correctness.
- [ ] **Substitution Function Application**
  Employing substitution techniques to replace variables or terms in proofs to demonstrate properties or equalities.
- [ ] **Model Checking Debugging**
  Utilizing debugging tools and techniques to analyze and refine specifications during the model checking process, including handling exceptions and evaluating formulas.
- [ ] **Notation Simplification**
  Streamlining and clarifying notation to improve the readability and manageability of proofs, making it easier for the proof assistant to process and verify statements.
- [ ] **Function Definition Refinement**
  Improving the definitions of functions to clarify their behavior and ensure that they meet the necessary specifications for verification.
- [ ] **Transitivity Utilization**
  Leveraging the transitive property of equality to connect multiple equalities in a proof, facilitating the establishment of desired conclusions.
- [ ] **Function Composition**
  Combining multiple functions in a structured manner to create new functions, which can help in organizing proofs and clarifying relationships between components.

Other

» Precondition and Postcondition Specification

☐ *Defining explicit preconditions and postconditions in the program to guide the verifier in understanding the expected behavior and constraints of functions.*

» Proof State Management

☐ *Organizing and managing the state of proofs to ensure clarity and facilitate the verification process, including hiding or revealing parts of the proof as needed.*

Demographics

How old are you?

- ○ 18 - 24 years old
- ○ 25 - 34 years old
- ○ 35 - 44 years old
- ○ 45 - 54 years old
- ○ 55 - 64 years old
- ○ 65+ years old

What is your gender?

- ○ Female
- ○ Male
- ○ Non-binary/Third gender
- ○ [ ] Other
- ○ Prefer not to say

What is your race? *Select all that apply.*

- ☐ White or Caucasian
- ☐ Black or African American
- ☐ American Indian/Native American or Alaska Native
- ☐ Asian
- ☐ Native Hawaiian or Other Pacific Islander
- ☐ [ ] Other
- ☐ Prefer not to say

What is the highest level of education you have completed?

- ○ Some high school or less
- ○ High school diploma or GED
- ○ Some college but no degree
- ○ Associates or technical degree
- ○ Bachelor's degree
- ○ Graduate or professional degree (MA, MS, MBA, PhD, JD, MD, DDS, etc.)
- ○ Prefer not to say

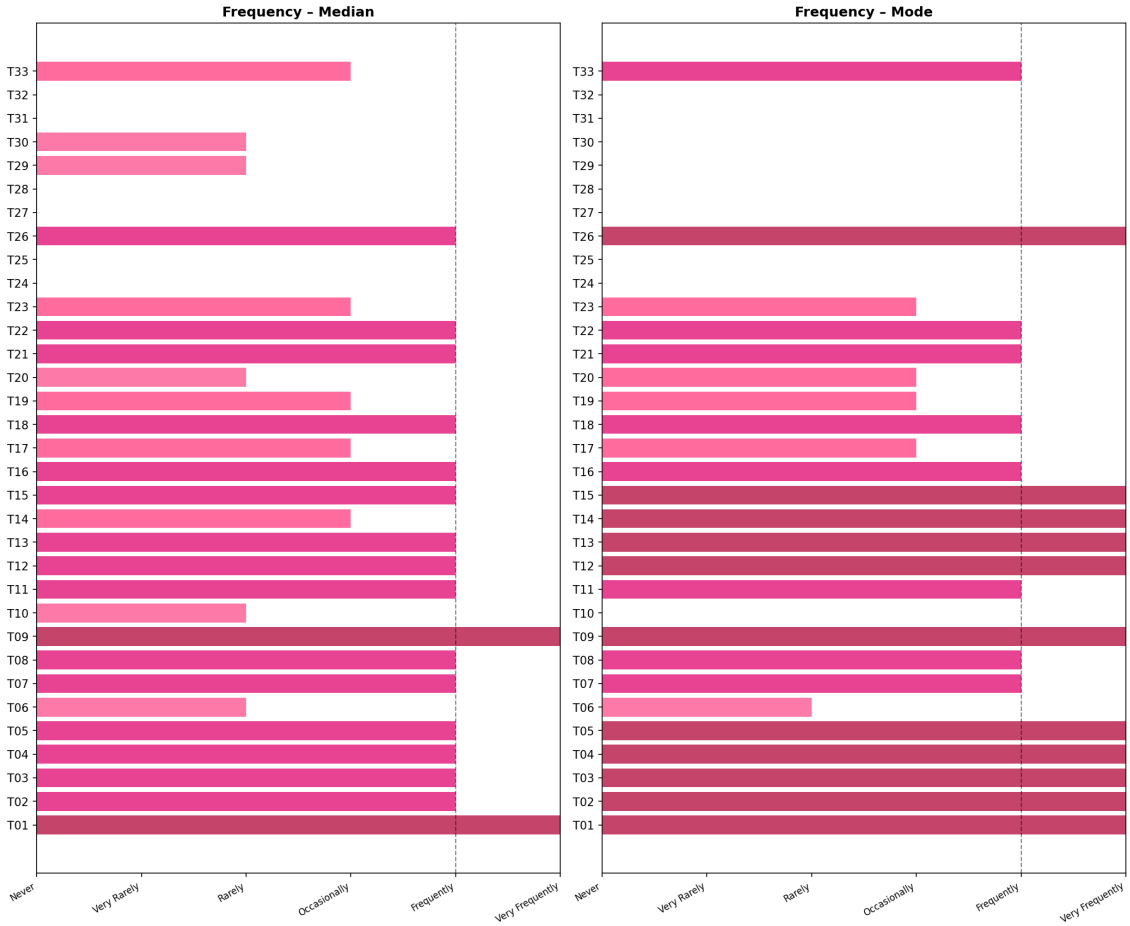
Do you have any other feedback for us? Feel free to share your thoughts about anything, from our survey to formal methods and strategies.

Thank you for participating in our study!

Appendix D

Detailed Survey Analysis

Figure D.1: Median and mode of the frequency ratings for each technique. Techniques for which fewer than half of respondents provided a rating are not shown; their absence reflects a high rate of the *I never used this technique* response in the complexity question, and a corresponding high *Never* rate in frequency.



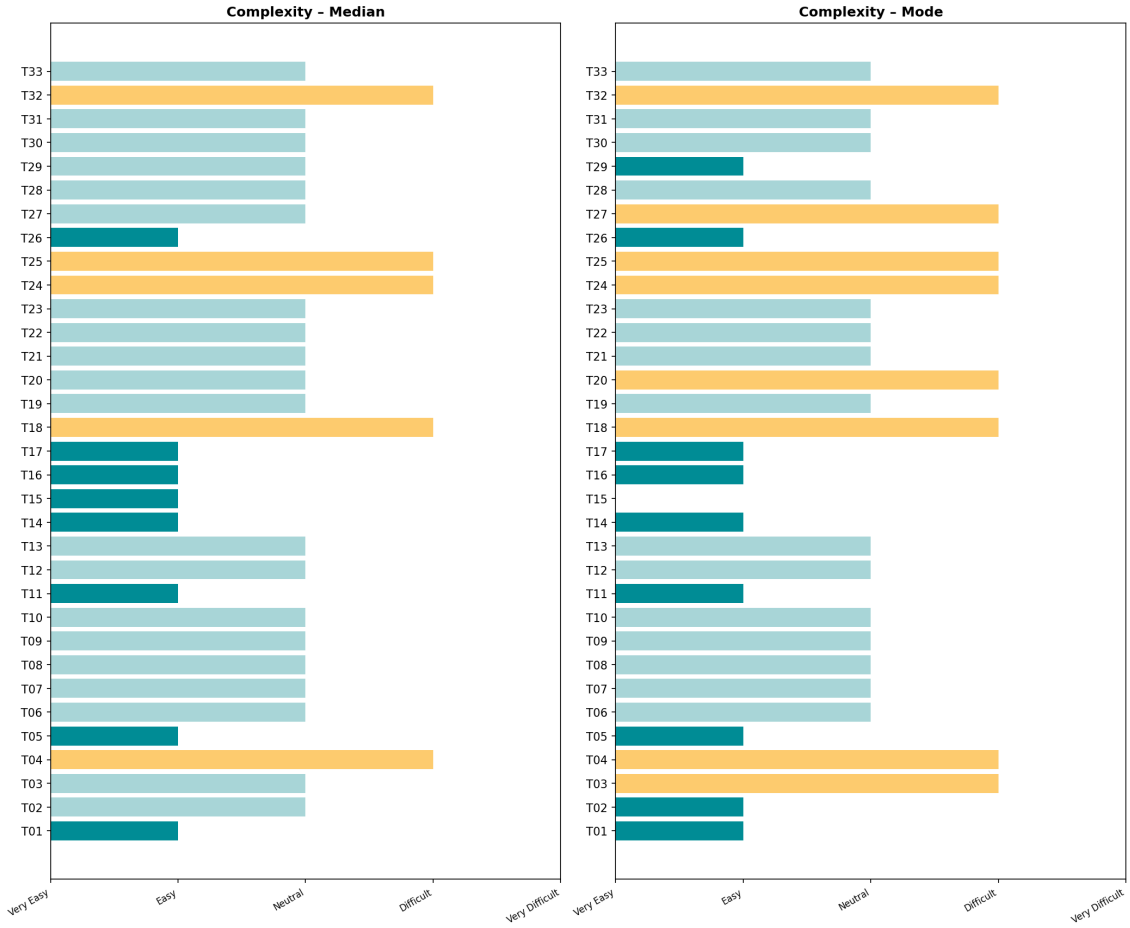


Figure D.2: Median and mode of the complexity ratings for each technique, excluding *I never used this technique* responses.

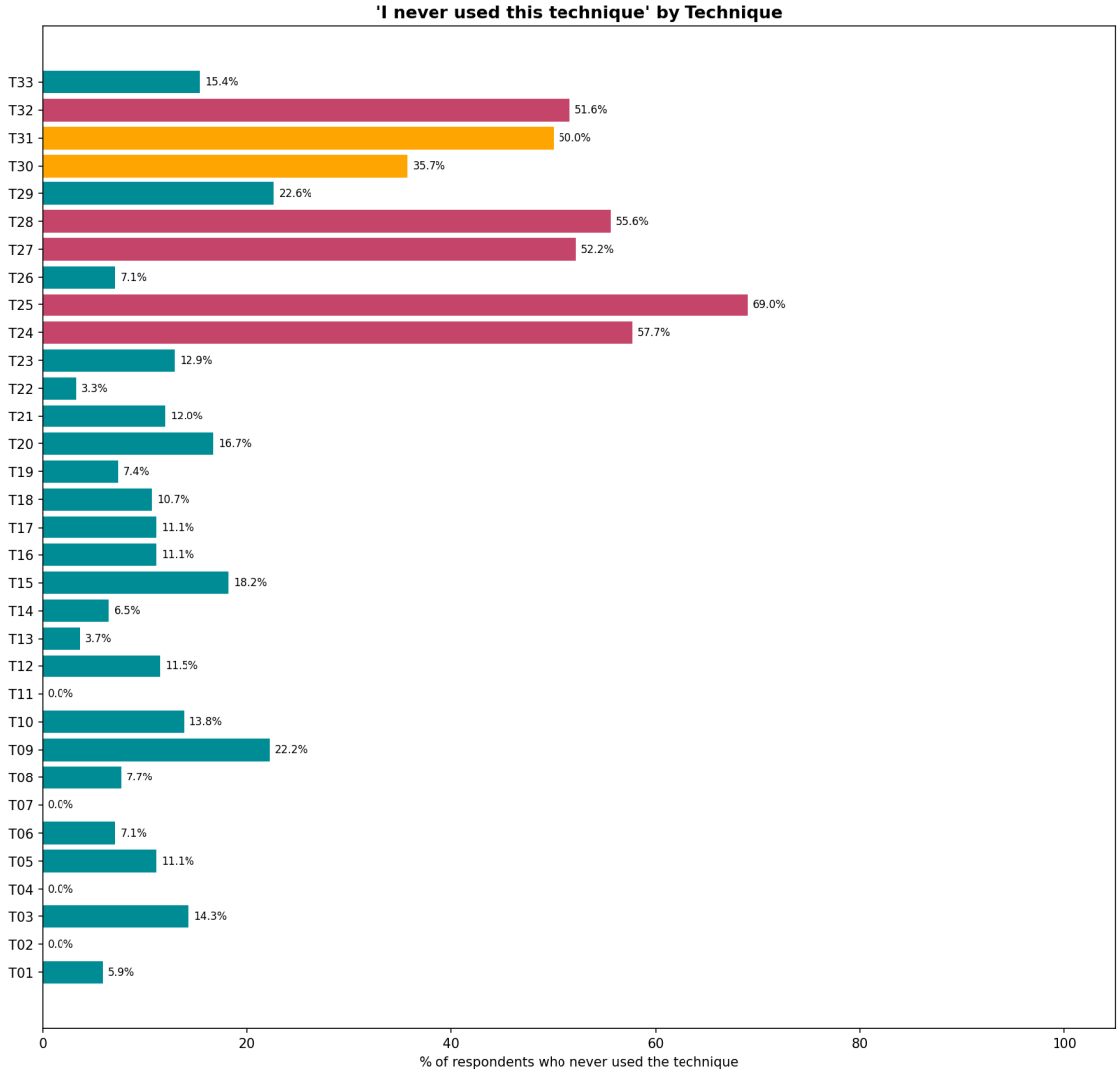


Figure D.3: Percentage of respondents who selected *I never used this technique* for each item. Red bars indicate rates above 50% and orange bars indicate rates between 30% and 50%.

Table D.1: Perceived difficulty by technique, estimated from a CLMM, sorted easiest to hardest.

Technique	Very Easy	Easy	Neutral	Difficult	Very Difficult	Expected Category (Standard Error)	Group
T05	36.1	46.5	13.9	3.2	0.3	1.85 (0.17)	a
T01	30.7	48.1	16.7	4.1	0.4	1.95 (0.16)	ab
T15	28.5	48.5	18.1	4.5	0.4	2.00 (0.18)	abc
T26	21.2	48.1	23.5	6.5	0.6	2.17 (0.18)	abcd
T17	20.7	48.0	24.0	6.7	0.7	2.19 (0.19)	abcd
T11	15.6	45.2	29.1	9.1	0.9	2.35 (0.20)	abcde
T14	14.5	44.2	30.4	9.9	1.0	2.39 (0.17)	abcde
T16	10.6	39.2	35.3	13.4	1.5	2.56 (0.19)	abcdef
T13	9.4	37.1	36.9	15.0	1.7	2.63 (0.19)	abcdef
T09	9.3	37.0	37.0	15.1	1.7	2.63 (0.21)	abcdef
T08	7.0	31.8	39.7	19.2	2.3	2.78 (0.19)	bcdef
T02	7.0	31.6	39.7	19.4	2.3	2.78 (0.18)	bcdef
T23	6.9	31.4	39.8	19.6	2.3	2.79 (0.18)	bcdef
T22	6.6	30.6	40.1	20.3	2.4	2.81 (0.18)	cdef
T19	6.1	29.2	40.5	21.5	2.6	2.85 (0.19)	cdef
T33	6.0	28.8	40.6	21.9	2.7	2.86 (0.20)	bcdef
T10	5.8	28.1	40.8	22.5	2.8	2.88 (0.21)	bcdef
T31	5.3	26.8	41.0	23.8	3.0	2.92 (0.26)	abcdef
T12	5.1	26.1	41.1	24.5	3.2	2.95 (0.21)	cdef
T28	5.0	25.8	41.1	24.8	3.2	2.95 (0.30)	abcdef
T30	4.5	23.9	41.1	26.8	3.6	3.01 (0.22)	cdef
T20	4.2	22.9	41.1	28.0	3.8	3.04 (0.22)	def
T21	3.9	21.6	40.8	29.5	4.1	3.08 (0.20)	def
T07	3.7	20.6	40.6	30.7	4.4	3.12 (0.18)	ef
T27	3.6	20.2	40.4	31.3	4.5	3.13 (0.28)	cdef
T29	3.4	19.4	40.2	32.2	4.8	3.16 (0.21)	def
T03	3.1	18.2	39.6	33.9	5.2	3.20 (0.20)	ef
T18	3.0	17.6	39.3	34.7	5.4	3.22 (0.19)	ef
T06	2.8	16.6	38.6	36.2	5.8	3.26 (0.20)	ef
T25	2.4	14.9	37.4	38.7	6.6	3.32 (0.32)	cdef
T04	2.2	13.8	36.4	40.4	7.2	3.36 (0.18)	f
T32	1.9	12.2	34.5	43.1	8.3	3.44 (0.25)	ef
T24	1.5	9.9	31.2	47.1	10.3	3.55 (0.28)	ef

Table D.2: Open-ended responses to question TE03 and their final coding.

Response	Coding
"generalized rewriting over arbitrary relations; logical relations"	Congruence Rule Application
"Choosing a good definition; Writing the correct helper lemmas to make using your definition bearable"	Function Definition Refinement; Lemma Engineering
"Reifying an object to an object enabling computations towards normal forms (e.g. lia, bit vectors stored as Z numbers as a dependent type) in Rocq"	Automated Proof Techniques
"Using high-powered tactics to blast through uninteresting goals"	Automated Proof Techniques
"Lemma functions (although Lemma Engineering is listed above, this is sufficiently different to be mentioned explicitly), measure functions"	Lemma Engineering; Termination Metric Specification
"Describe states of the function/algorithm using an inductive, prove their equivalence and reason on the inductive rather than the function/algorithm directly"	Inductive Proof Techniques; Function Definition Refinement
"Rephrasing the problem into a theorem prover or SMT-LIB to check if my intuition makes sense on a simpler case"	Model Checking Debugging
"Counterexample extraction [...] Creation of static checks (to be discharged) statically. They are very useful for assessing whether the specifications are strong enough"	Model Checking Debugging; Precondition and Postcondition Specification; Lemma Engineering
"Ghost state"	Ghost State
"Components of verification e.g. postconditions, are compiled and executed within the context of the software product for which verification is desired"	Precondition and Postcondition Specification
"Preconditions and invariants"	Invariant Handling; Precondition and Postcondition Specification
"Testing the specification before proving it by another mean (abstract interpretation or runtime annotation checking)"	Model Checking Debugging
"introducing lemma functions and ghost code to help reasoning or proving programs"	Ghost State; Lemma Engineering
"Writing custom automation tactics"	Automated Proof Techniques
"it's a crucial step to design and iterate on the formal model itself so that it's more amenable for verification"	Function Definition Refinement
"simple rewrite proofs as in calculational programming, or algebra of programming"	Congruence Rule Application; Transitivity Utilization

Continued on next page

Table D.2: Open-ended responses to question TE03 and their final coding (continued).

Response	**Coding**
"Fording, that is, adding explicit equations to data parameters in inductive definitions' constructors [...] Using constructive simplices as constructive evidence"	Inductive Proof Techniques; Witness Construction
"Holes"	Proof State Management
"I didn't see 'equational reasoning' listed. i.e. proving equations via a chain of equalities step-by-step. I use this very frequently in Agda"	Congruence Rule Application; Transitivity Utilization
"Proof by generalization [...] Normalization by evaluation, compute terms into normal forms and compare them to establish equivalence [...] Well-founded relation and accessibility, to show that a recursion is terminating"	Goal Manipulation; Termination Metric Specification