Thesis for The Degree of Licentiate of Engineering

Automated Exploration and Explanation of Software Boundaries

Sabinakhon Akbarova

Department of Computer Science and Engineering
Chalmers University of Technology | University of Gothenburg
Gothenburg, Sweden, 2026

Automated Exploration and Explanation of Software Boundaries

Sabinakhon Akbarova

Department of Computer Science and Engineering
Division of Interaction Design and Software Engineering
Chalmers University of Technology | University of Gothenburg
SE-412 96 Göteborg,
Sweden

Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2026.

To the one who believed in me first.

Abstract

Boundary Value Analysis (BVA) is a software testing technique that targets inputs at the transitions between different program behaviors, where faults are most likely to occur. Automating this process, known as Boundary Value Exploration (BVE), requires not only finding input pairs that lie on opposite sides of these transitions, but finding enough of them, across different behavioral regions, to give testers a complete picture of where a program's behavior changes and why. This thesis advances BVE toward a search that is broader, more general across input types, and more interpretable to the testers.

To achieve this, this thesis focuses on unit-level functions and introduces a search-based framework that systematically seeks boundary candidates that are not only sharp but also spread across a wide range of behavioral regions, so that the search covers many different kinds of transitions rather than converging on the most extreme ones. Building on this foundation, a deeper limitation is addressed: existing approaches require search operators to be hand-crafted for each input type, confining BVE mostly to numeric domains. An agentic, LLM-driven framework that autonomously generates its own exploration strategies removes this bottleneck, extending BVE to functions with non-numeric inputs. Finally, since discovering boundaries is only part of the challenge, a mixed methods study with software professionals investigates whether LLMs can generate natural-language explanations for discovered boundaries, and what properties such explanations need to be used in practice.

The results show that a broader and more varied set of boundary behaviors can be discovered through diversity-aware search, and that adaptive, LLM-driven exploration generalizes successfully to input types that other existing automated black-box BVE approaches cannot handle. Software professionals found LLM-generated explanations useful overall, though the study also reveals that trust in such explanations is fragile and that correctness and consistency are prerequisites for adoption rather than desirable extras.

Together, these contributions suggest that the path toward practical BVE lies not in optimizing a single dimension of the search, but in balancing quality with diversity, and automation with human understanding. The findings point to diversity as an explicit and configurable testing goal, to LLMs as a practical mechanism for scaling BVE across input domains, and to explanation as a prerequisite for adoption rather than an optional enhancement.

Keywords

Boundary Value Exploration, Quality-Diversity Optimization, Large Language Models, Automated Software Testing

Acknowledgment

I am not sure how to start this, except to say thank you.

To my supervisors, Robert Feldt and Felix Dobslaw, thank you for your guidance, your patience, and for teaching me how to think more clearly, question more deeply, and care about what good research truly means.

To Francisco Gomes de Oliveira Neto, thank you for your energy and for the way you approach ideas and people. You are an inspiration in how you share knowledge and engage with others.

To my examiner, Christian Berger, thank you for helping me stay on track and for asking the kind of questions that make me think more carefully about where this work is going.

To my friends, Ranim and Mazen, thank you for listening and for all our small therapy sessions. Ranim, thank you for being the best officemate I could ever ask for. To Mathilde, Minerva, Marti, and all my WASP friends, there are too many of you to name, thank you for the laughter, the karaoke nights, and for making this time so much brighter. To everyone at the IDSE division, thank you for your kindness and for always being ready to help.

To my partner, Aria, thank you for your endless support, for every "you can do it", and for bringing me chocolate when I am working late evenings. You have a way of making things feel lighter, even on the difficult days, and that means more than I can say.

And to my mom, you are far in distance but always closest to my heart. Your love, your strength, and your belief in me have carried me further than I can ever explain.

List of Publications

Appended publications

This thesis is based on the following publications:

[**Paper A**] **Akbarova, S.**, Dobslaw, F., de Oliveira Neto, F. G., Feldt, R., *SETBVE: Quality-Diversity Driven Exploration of Software Boundary Behaviors*
ACM Transactions on Software Engineering and Methodology (TOSEM) 2026.

[**Paper B**] **Akbarova, S.**, Dobslaw, F., Feldt, R., *Adaptive Strategy Generation for Boundary Value Exploration Beyond Numeric Inputs*
Submitted to IEEE/ACM International Conference on Automated Software Engineering (ASE).

[**Paper C**] **Akbarova, S.**, Dobslaw, F., Feldt, R., *Understanding on the Edge: LLM-generated Boundary Test Explanations*
In 2026 ACM/IEEE International Conference on Automation of Software Test (AST) (2026, April).

Research Contribution

The contributions of each author to the appended papers are summarized in Table 1 using the CRediT (Contribution Roles Taxonomy) model[1]. Authors are abbreviated as follows: SA = Sabinakhon Akbarova, FD = Felix Dobslaw, FG = Francisco Gomes de Oliveira Neto, RF = Robert Feldt. The order of abbreviations in each cell follows the author order of the respective paper.

Table 1: Author contributions to the appended papers using the CRediT taxonomy.

Role	**Paper A**	**Paper B**	**Paper C**
Conceptualization	**SA**, FD, FG, RF	**SA**, FD, RF	**SA**, FD, RF
Methodology	**SA**, FD, FG, RF	**SA**, FD, RF	**SA**, FD, RF
Software	**SA**	**SA**	**SA**, FD
Validation	**SA**	**SA**	**SA**, FD
Formal analysis	**SA**, FD, FG, RF	**SA**, FD, RF	**SA**, FD, RF
Investigation	**SA**, FD, FG, RF	**SA**, FD, RF	**SA**, FD, RF
Data curation	**SA**	**SA**	**SA**, FD
Writing – original draft	**SA**	**SA**	**SA**, FD
Writing – review & editing	FD, FG, RF	FD, RF	FD, RF
Visualization	**SA**, FG, RF	**SA**	**SA**

[1] `https://credit.niso.org`

Ethical Considerations

Paper A. This study consists entirely of computer experiments with no human participants involved. All functions used in the evaluation are publicly available or implemented by the authors, and all experiments are self-contained and reproducible. No ethical concerns are identified for this study.

Paper B. This study consists of computer experiments with no human participants. It uses commercial LLM APIs as part of the experimental framework. The functions under test are treated as black boxes and contain no sensitive, personal, or proprietary data. All inputs and outputs are synthetically generated or drawn from public implementations. While the use of external LLM services introduces a theoretical risk of data exposure through API calls, this risk is mitigated by the nature of the data: no real user data, confidential code, or sensitive information is transmitted. The energy consumption associated with repeated LLM API calls carries environmental implications worth acknowledging, particularly given the scale of the evaluation. The monetary and computational cost of LLM usage is reported transparently in the paper.

Paper C. This study involves human participants and was exempt from formal ethics review, as no sensitive personal data were collected. Participation was entirely voluntary, and participants could withdraw at any time without consequence. The survey was anonymous, and any contact details provided for interview scheduling were deleted after use. Interview sessions were audio-recorded with explicit informed consent, and all data shared with the research community are fully anonymized. LLM-generated explanations were produced using a commercial API, which carries a minor energy footprint worth noting.

AI use in writing. All three papers were originally written by the authors. AI writing assistants (ChatGPT and Claude.ai) were used to improve phrasing in selected passages.

List of Abbreviations

Abbreviation	Meaning
BC	Boundary Candidate
BD	Behavioural Descriptor
BU	Bituniform sampling
BMI	Body Mass Index
BVA	Boundary Value Analysis
BVE	Boundary Value Exploration
CTS	Compatible Type Sampling
ECP	Equivalence Class Partitioning
EE	Error-Error
FUT	Function Under Test
LLM	Large Language Model
PD	Program Derivative
RAC	Relative Archive Coverage
RPD	Relative Program Derivative
QD	Quality-Diversity
SBST	Search-Based Software Testing
VE	Valid-Error
VV	Valid-Valid

Contents

Abstract iii

Acknowledgment v

List of Publications vii

Research Contribution ix

Ethical Considerations xi

List of Abbreviations xiii

1 Introduction 1
- 1.0.1 Research Purpose . . . 2
- 1.1 Background and Related Work . . . 3
  - 1.1.1 Boundary Value Analysis and Exploration . . . 3
  - 1.1.2 Quality-Diversity Optimization . . . 4
  - 1.1.3 LLMs in Software Testing . . . 5
- 1.2 Research Approach . . . 5
- 1.3 Exploring and Explaining Boundaries . . . 7
  - 1.3.1 Broadening Boundary Value Exploration . . . 7
  - 1.3.2 Generalizing Boundary Exploration Across Input Domains 11
  - 1.3.3 Explaining Discovered Boundaries . . . 13
- 1.4 Reflections . . . 15
- 1.5 Next Steps . . . 19
- 1.6 Conclusion . . . 21

2 Paper A 23
- 2.1 Introduction . . . 24
- 2.2 Background . . . 26
  - 2.2.1 Characterizing Software Boundaries . . . 26
  - 2.2.2 AutoBVA . . . 28
  - 2.2.3 Components of Quality-Diversity Optimization . . . 29
- 2.3 Related Work . . . 29
  - 2.3.1 Boundary Value Analysis . . . 29
    - 2.3.1.1 Automation of Boundary Value Analysis . . . 30

2.3.2 Quality-Diversity Optimization . . . 31
2.3.2.1 Quality-Diversity Optimization in Testing . . . 31
2.4 SETBVE Approach . . . 32
2.4.1 Sampler . . . 33
2.4.1.1 Definition of Behavior Space $\mathcal{B}$ . . . 34
2.4.2 Explorer . . . 37
2.4.3 Tracer . . . 38
2.4.3.1 Cells Prioritization . . . 39
2.4.3.2 Boundary Tracing . . . 39
2.5 Empirical Evaluation . . . 42
2.5.1 Quality and Diversity Metrics . . . 44
2.5.1.1 Relative Program Derivative (RPD) . . . 44
2.5.1.2 Relative Archive Coverage (RAC) . . . 45
2.5.2 Description of FUTs . . . 46
2.5.3 Experimental Setup . . . 47
2.6 Results and analysis . . . 49
2.6.1 RQ1 - Quality and Diversity of Boundary Candidates . 49
2.6.1.1 Bytecount . . . 49
2.6.1.2 Circle . . . 50
2.6.1.3 BMI . . . 51
2.6.1.4 Date . . . 52
2.6.1.5 FUTs from Julia Base . . . 54
2.6.1.6 Aggregate Patterns Across FUTs . . . 55
2.6.2 RQ2 - Coverage of Behavioral Space . . . 58
2.6.3 RQ3 - Tracing Identified Boundaries . . . 61
2.6.4 RQ4 - Exploring Framework's Adaptability . . . 64
2.7 Discussion . . . 67
2.7.1 Limitations and Validity Threats . . . 69
2.7.2 Future work . . . 69
2.8 Conclusion . . . 71

3 Paper B **73**
3.1 Introduction . . . 74
3.2 Background and Related Work . . . 75
3.2.1 Boundary Discovery . . . 75
3.2.2 Quality-Diversity Optimization . . . 76
3.2.3 Software Testing with LLMs . . . 77
3.3 Framework . . . 77
3.3.1 Coordinator . . . 78
3.3.2 Ideator and Strategy Generator . . . 79
3.3.3 Explorer . . . 81
3.4 Evaluation . . . 82
3.4.1 Research Questions . . . 83
3.4.2 Experimental Setup . . . 83
3.5 Results . . . 85
3.5.1 RQ1: Numeric Boundary Discovery . . . 85
3.5.2 RQ2: Non-numeric Boundaries . . . 86

3.5.3 RQ3: Strategy Effectiveness by Input Type 89
3.5.4 RQ4: Ablation Study 91
3.6 Discussion 92
3.6.1 Lessons Learnt and Future Directions 94
3.6.2 Validity Threats 94
3.7 Conclusion 95

4 Paper C **97**
4.1 Introduction 100
4.2 Background and Related Work 102
4.3 Method 102
4.3.1 Study Design and Rationale 102
4.3.2 Context and Materials 103
4.3.3 Participants and Recruitment 104
4.3.4 Survey 104
4.3.5 Interview Procedure 105
4.3.6 Data Analysis 106
4.4 Results 106
4.4.1 RQ1: LLMs and Software Boundary Explanations . . . 106
4.4.1.1 Clarity 108
4.4.1.2 Correctness 108
4.4.1.3 Completeness 109
4.4.1.4 Perceived Usefulness 109
4.4.1.5 Summary 110
4.4.2 RQ2: Good Boundary Explanations 111
4.4.2.1 Theme 1: Boundary Explanation Design . . . 111
4.4.2.2 Theme 2: Use-cases and Tooling 112
4.4.2.3 Theme 3: Trust and Reliability in Auto-generated Explanations 113
4.5 Discussion 114
4.5.1 Requirements for LLM-based Boundary Explanation Tools 116
4.5.1.1 Preliminary Test of Requirements via Prompting 117
4.6 Conclusion 120

Bibliography **121**

Chapter 1

Introduction

Software testing plays a central role in ensuring software quality by verifying that a program behaves correctly across its range of possible inputs. As software systems grow in complexity and scale, the consequences of undetected edge-case failures become increasingly costly, whether in reliability, security, or user trust. In practice, however, testing every possible input is infeasible: even a function that accepts two 32-bit integers has over four billion possible input combinations per argument, making exhaustive testing computationally out of reach for any realistic system. The challenge is therefore to reduce the number of test cases to a manageable set while still selecting those that are most likely to reveal problems.

One of the most well-established strategies for this is Boundary Value Analysis (BVA), which targets inputs near the transitions between different behavioral regions of a program, since faults tend to concentrate at these edges [1], [2], [3]. Boundaries are typically understood as pairs of inputs that lie close together yet belong to adjacent sub-domains with distinct behavior [4]. BVA has been shown empirically to outperform both random testing and equivalence partitioning in fault detection [5]. Despite its effectiveness, BVA has long remained a largely manual and creative process, which is particularly challenging for software with large input spaces, complex input types, or heavily interdependent inputs, where manual analysis becomes impractical [6]. It is only recently that research has begun to explore systematic approaches to its automation [6], [7], [8].

Dobslaw et al. [6] introduced Boundary Value Exploration (BVE) to address this limitation. BVE is a systematic approach to selecting inputs that detect and identify boundary candidates, complementing BVA by helping testers find and explore boundaries that are difficult to identify through manual analysis alone. By quantifying how strongly a program's behavior changes between nearby inputs, BVE can surface boundary candidates directly from program executions, making it applicable even when specifications are incomplete, vague, or unavailable. An early application illustrated that even a well-tested, widely-used library can have boundary behavior that deviates from its own specification, suggesting that systematic exploration can uncover

subtle problems that manual analysis or conventional testing may miss [6]. Building on this foundation, Dobslaw et al. [8] developed AutoBVA, the first fully automated BVE framework, which searches for high-quality boundary candidates as measured by the program derivative and demonstrated clear effectiveness on numeric inputs. However, AutoBVA optimizes for a single quality criterion, which can leave parts of the behavioral landscape unexplored. It also requires search algorithms hand-crafted for specific input types, limiting its applicability.

1.0.1 Research Purpose

The overarching purpose of this thesis is to advance BVE into a more capable and practically useful technique for testers, with a focus on functions under test (FUTs). The work is scoped to black-box testing at the unit level and does not address system-level testing, white-box approaches that exploit source code structure, or fault detection effectiveness. Within this scope, the thesis improves both the boundary search process and the way its results are presented. On the search side, we pursue two directions: *broadening* the boundary search to discover a richer and more complete picture of a program's behavioral landscape, and making BVE *more generally applicable* by removing the need to manually engineer search components for each new input type. On the presentation side, we investigate how boundaries can be *explained* to support testers in understanding and acting on what has been found. To address these directions, we formulate the following research questions:

- **RQ1.** How can automated boundary value exploration be improved to find more boundary candidates across a wider range of program behaviors?
- **RQ2.** How can automated boundary value exploration be made applicable across different input types without manually engineered search components?
- **RQ3.** How do software professionals perceive automatically generated natural-language explanations of software boundaries?

This thesis addresses these research questions through three papers. Paper A answers RQ1 by introducing diversity into the boundary search process. Rather than converging on a small set of the most pronounced boundary candidates, it seeks a broad collection of candidates that together cover a wider range of behavioral transitions in the program. Paper B answers RQ2 by proposing an approach that autonomously generates its own exploration strategies, removing the dependence on type-specific, hand-crafted search components and extending BVE to a wider range of input types. Paper C answers RQ3 by investigating whether automatically generated natural-language explanations alongside boundary candidates are perceived as useful by software professionals, and what properties such explanations should have.

1.1 Background and Related Work

This section introduces the key concepts and related work that underpin the three studies presented in this thesis. It begins with an overview of BVA and BVE, covering the historical development of boundary testing, how boundary quality is quantified, and how boundary candidates are classified. This is followed by an introduction to Quality-Diversity optimization and its core components, which provide the algorithmic foundation for incorporating diversity into the boundary search process. Finally, Large Language Models (LLMs) are discussed in the context of software testing, as they play two roles in this thesis: driving adaptive exploration strategies and generating natural-language explanations for discovered boundaries.

1.1.1 Boundary Value Analysis and Exploration

The input domain of a program can be organized into regions of similar behavior through Equivalence Class Partitioning (ECP), which groups inputs that produce the same output into a single class. BVA complements this by focusing on the critical thresholds where program behavior shifts between classes. White and Cohen [1] formalized this through a domain-testing strategy targeting boundaries between mutually exclusive subdomains, and subsequent work has extended BVA to integrate with search-based techniques [9], [10]. A recurring challenge across these approaches is that they assume partitions and boundaries are derivable from a specification, yet most non-trivial specifications are incomplete or expressed only informally. This is particularly problematic for software with large input spaces or inputs of complex structured types, where manual boundary identification is impractical [6].

Early automation efforts relied on access to program internals. White-box approaches instrument source code or apply symbolic execution to extract boundary conditions [11], [12], while others use ECP or machine learning to locate boundary regions [13], [14]. Although effective in controlled settings, these approaches are limited when source code or explicit specifications are unavailable, motivating the shift toward black-box Boundary Value Exploration (BVE).

BVE employs distance-based metrics to systematically explore the input space and quantify behavioral changes across program executions. The central concept enabling this quantification is the *program derivative* (PD), introduced by Feldt et al. [15]. Given two inputs i_1 and i_2 with corresponding outputs o_1 and o_2, the PD is defined as:

$$PD(i_1, i_2) = \frac{d_o(o_1, o_2)}{d_i(i_1, i_2)} \tag{1.1}$$

where d_i and d_o are distance measures on the input and output spaces respectively. An input pair is considered a *boundary candidate* when its PD exceeds zero, meaning the inputs are close yet produce different outputs. A high PD indicates high *boundariness*, where small input changes cause large behavioral shifts. To further characterize boundary candidates, Dobslaw et al.

[8] introduced *validity groups*, which classify each candidate into VV, where both outputs are valid; VE, where one is valid and the other is an error; and EE, where both are errors. These groups depend only on observable execution results, enabling consistent characterization of boundaries across diverse programs. AutoBVA [8] operationalized these concepts into the first fully automated black-box BVE framework, combining compatible type sampling with bituniform sampling to explore the input space and ranking candidates using the program derivative. Its application to Julia's standard library notably uncovered unexpected behavior that led to a documented fix in the language's codebase [8].

1.1.2 Quality-Diversity Optimization

Rather than converging on a single optimal solution, Quality-Diversity (QD) optimization algorithms seek a broad collection of high-performing solutions that together cover a wide range of behaviors [16]. This distinguishes QD from both single-objective optimization, which targets one best solution, and multimodal optimization, which targets multiple optima. QD instead aims to populate an entire behavioral space with diverse, high-quality candidates. This breadth makes QD particularly valuable in domains where understanding the variety of possible behaviors matters as much as finding the best one, making it a natural fit for BVE, where we seek boundary candidates that are not only of high quality as measured by the program derivative, but also diverse across the behavioral space.

The intellectual roots of QD trace back to novelty search [17], which demonstrated that rewarding behavioral novelty rather than objective fitness can lead to more effective exploration. MAP-Elites [18] built on this by combining exploration with quality: it maintains a grid of elite solutions organized by behavioral descriptors, retaining the highest-performing solution in each cell. Subsequent work has extended this foundation, introducing surprise as an additional exploration criterion [19] and bringing QD to multi-objective and Bayesian optimization settings [20], [21]. MAP-Elites underpins the framework used in this thesis, as its archive-based structure provides a natural mechanism for organizing boundary candidates by behavioral characteristics while retaining the highest-quality solution in each region.

A widely adopted modular framework for QD algorithms was proposed by Cully et al. [22] and later implemented in the Python library pyribs [23]. The framework revolves around three core components: behavioral descriptors, the archive, and emitters. *Behavioral descriptors* define the measurable characteristics that distinguish one solution's behavior from another. They can capture general properties such as output length or number of exceptions, but also program-specific semantics such as a classification label or the number of database queries executed. The *archive* is a structured collection of solutions organized by their behavioral descriptors rather than their quality alone. Each cell in the archive corresponds to a distinct region of the behavioral space and retains the highest-quality solution found for that region. A new solution enters the archive if it occupies a previously empty cell or outperforms the

existing occupant in quality. Finally, *emitters* are responsible for generating new candidate solutions, operating with different strategies that prioritize quality, novelty, or curiosity. Unlike traditional genetic operators, emitters can encapsulate entire sub-optimization processes, offering a flexible mechanism for exploring and refining the behavioral space. In the context of BVE, this translates naturally into the notion of exploring the input space in search of undiscovered boundary behaviors.

1.1.3 LLMs in Software Testing

LLMs have recently been applied to a wide range of software engineering tasks, with automated testing emerging as one of the most active areas [24]. Their use in testing broadly falls into two directions: generating test artifacts and producing natural-language text to accompany those artifacts.

Their ability to leverage programming patterns and semantics makes LLMs well-suited for generating test inputs [25], [26], and recent surveys confirm their broad effectiveness across test case generation, debugging, and program repair [24]. However, most existing approaches rely on single-shot generation or fixed prompting schemes without incorporating execution feedback [27], [28], which can lead to redundant or invalid test cases [29]. While some work explores iterative or feedback-driven testing [30], these approaches typically operate within predefined generation patterns and do not adapt how the input space is explored over time. Agent-based systems have emerged as a more adaptive paradigm, with multi-agent architectures proposed for Android GUI testing [31] and earlier work establishing the foundations of autonomous testing agents [32].

While LLMs have been extensively explored as generators of test artifacts, their use for producing accompanying natural-language text has received less attention. Developers frequently discard auto-generated tests that lack transparent justification [33], making natural-language accompaniment a prerequisite for adoption rather than an optional enhancement. Related work shows that LLMs can generate accurate and context-aware rationales for source code [34], and natural-language summaries of test cases [35]. Beyond documentation, clarity and context in explanations have been shown to reduce false-positive fatigue in static analysis and code review [36]. Evaluating such text requires criteria grounded in both Human-Computer Interaction and Natural Language Generation research. Qualities such as clarity, correctness, completeness, and usefulness have been identified as central to effective explanation [37], [38], [39], and form the basis for the empirical evaluation conducted in this thesis.

1.2 Research Approach

This thesis presents empirical studies conducted to advance automated boundary exploration toward making it more useful for software testers. A summary of the data collection and analysis elements that shaped the research approach is given in Table 1.1.

Table 1.1: Overview of the research methodology across the appended papers.

Paper	Research Type	Data Type	Collection Method	Analysis Method
Paper A	Framework development with empirical evaluation	Quantitative (with qualitative complement)	Experiments on 30 FUTs	Descriptive statistics; qualitative analysis of boundary candidates
Paper B	Framework development with empirical evaluation	Quantitative (with qualitative complement)	Experiments on 20 FUTs	Descriptive statistics; qualitative analysis of boundary types
Paper C	Mixed methods	Quantitative and qualitative	Survey (27 participants); interviews (6 participants)	Descriptive statistics; thematic analysis

The two search-based studies each propose and evaluate a new automated BVE framework through experiments on unit-level functions under test, following Wohlin et al.'s guidelines for software engineering experiments [40]. The development of each framework was iterative in nature, sharing characteristics with design science research, though without formally adopting that methodology. Unit-level functions provide a well-scoped setting for systematic comparison of search strategies, and evaluation is conducted in a black-box manner, reflecting the practical motivation to identify boundaries solely from observable input–output behavior. Paper A evaluates multiple configurations of a diversity-aware search framework on 30 integer-based functions, comparing them against the state-of-the-art and a random baseline across different time budgets and repeated runs. Paper B builds on this by replacing hand-crafted, type-specific search components with an agentic LLM-driven approach, evaluating it on 20 functions spanning integer, string, array, and mixed input types. In both studies, we measure how well the frameworks find boundary candidates in terms of quality and diversity, and complement the quantitative results with qualitative inspection of the discovered candidates and the types of boundaries. Across both studies, the goal is not only to find high-quality boundary candidates, but to do so more broadly by covering more of a program's behavioral landscape in the first study, and reaching input types that existing automated black-box approaches cannot handle in the second.

Paper C takes a different direction. Rather than building a tool, we investigate whether LLM-generated natural-language explanations can help software professionals understand automatically discovered boundaries. A sequential mixed methods design was chosen because quantitative ratings across participants and boundary pairs needed to be complemented by qualitative depth to understand the reasoning behind those judgments, which neither method alone would have provided. We ran a survey with 27 software professionals from both research and industry, asking them to rate explanations for 20 boundary

pairs across four functions on clarity, correctness, completeness, and perceived usefulness, followed by semi-structured interviews with six participants to explore their experiences in more depth. Survey responses are analyzed using descriptive statistics, and interview and open-ended responses through thematic analysis. This study connects back to the search-based work by using boundary candidates the frameworks produce, and extends the focus of the thesis from finding boundaries to making them meaningful in practice.

As a result of these studies, two BVE frameworks are contributed alongside an empirical account of how software professionals perceive LLM-generated boundary explanations. The findings of each are detailed in the following section.

1.3 Exploring and Explaining Boundaries

In what follows, incorporating Quality-Diversity optimization into the boundary search is shown to lead to a richer and more complete picture of a program's behavioral landscape (Section 1.3.1). Adaptive, LLM-driven strategy generation is then demonstrated to successfully replace hand-crafted search components, discovering meaningful boundary behaviors across non-numeric input types, including error-triggering conditions (Section 1.3.2). Finally, software professionals are found to perceive LLM-generated boundary explanations as useful overall, though their responses also reveal concrete requirements that future explanation tools will need to meet (Section 1.3.3).

1.3.1 Broadening Boundary Value Exploration

To broaden boundary value exploration beyond what quality-focused approaches find, we introduced SETBVE, a modular framework that incorporates Quality-Diversity optimization into the search. As illustrated in Figure 1.1, the framework operates in three stages. During setup, the tester provides a black-box function under test (FUT) and selects behavioral descriptors (i.e., measurable characteristics of the inputs and/or outputs that define the axes of a multi-dimensional archive grid). Each cell in the archive corresponds to a distinct region of the behavioral space and retains only the highest-quality candidate found for that region, ensuring that the search covers a variety of behavioral transitions rather than clustering around a single high-scoring archive area.

The search itself is carried out by three modular components. The ***S**ampler* generates random input pairs, queries the FUT, and populates the archive by storing candidates in the appropriate cells based on their behavioral descriptors and program derivative. The ***E**xplorer* then selects existing candidates from the archive, applies a mutation that brings the two inputs of a pair closer together, and submits the resulting candidates back to the archive, either filling new cells or improving previously filled ones. Together, the Sampler and Explorer drive exploration, steering the search toward underrepresented regions of the behavioral space. The ***T**racer* shifts the focus to exploitation: rather than seeking new archive cells, it performs a local search around high-quality candidates already in the archive, generating additional boundary

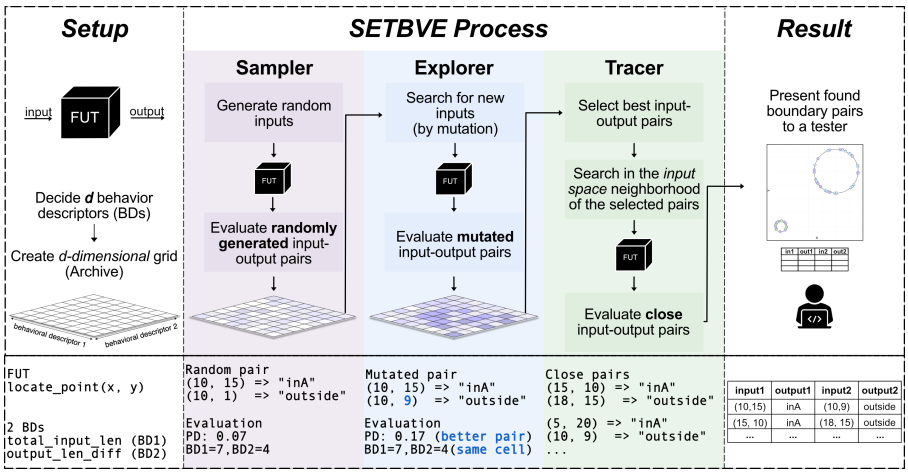

Figure 1.1: Overview of the SETBVE framework, showing the three stages: setup, search process, and result. The Sampler, Explorer, and Tracer interact with the archive to discover diverse boundary candidates. The bottom part illustrates the process using the `locate_point(x, y)` function.

pairs in their immediate neighborhood to build a more detailed picture of each boundary's shape and structure. The result is a diverse set of boundary candidates presented to the tester for test selection or further inspection.

To see how the three components work together in practice, consider the `locate_point(x, y)` example from Figure 1.1. This function takes 2D coordinates as input and returns whether the point lies in region "inA" or "outside". The Sampler first generates a random input pair, for instance (10, 15) and (10, 1), which produce the outputs "inA" and "outside" respectively. This pair is a boundary candidate: the two inputs are close yet land on opposite sides of a regional boundary. Its behavioral descriptors are computed, placing it in a specific archive cell, and its program derivative determines whether it is stored. The Explorer then selects this candidate and mutates it, bringing the inputs closer together: (10, 15) stays unchanged while (10, 1) is shifted to (10, 9), yielding a higher program derivative since the inputs are now closer while still producing different outputs. This improved candidate replaces the original in the same archive cell. Finally, the Tracer takes this high-quality candidate and searches its immediate neighborhood in the input space, generating additional pairs such as (15, 10) and (18, 15), and (5, 20) and (10, 9), both straddling the boundary between "inA" and "outside". These candidates are not stored in the archive but added to the final output list, giving the tester a denser and more complete picture of where the boundary lies.

The effect of this approach becomes tangible when looking at actual discovered boundary candidates. Figure 1.2 shows boundary candidates found by AutoBVA (a state-of-the-art BVE framework) and SETBVE for a Body Mass Index (BMI) function that takes height (cm) and weight (kg) as inputs and returns one of five categories (Underweight, Normal, Overweight, Obese, and

Severely Obese) or throws an exception for negative inputs. Each dot on the plot represents a single input, that is, a (height, weight) combination, colored by the BMI category it produces. Boundary candidates are pairs of inputs, so each boundary candidate appears as two nearby dots of different colors. The color difference signals the behavioral transition between them.

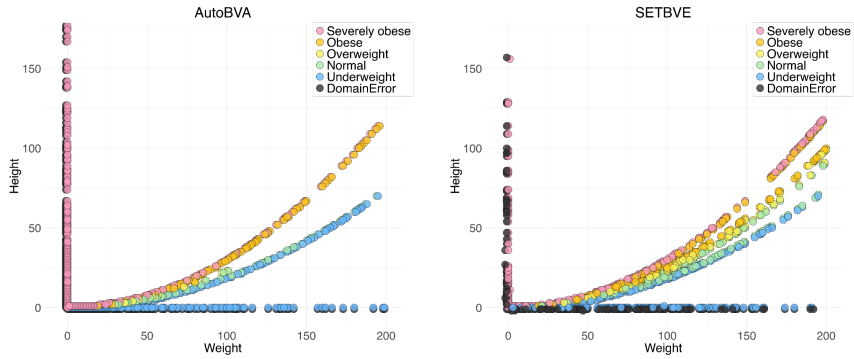

Figure 1.2: Comparison of boundary candidates found by AutoBVA and SETBVE for the BMI function. Each dot is a single (height, weight) input colored by BMI category, with boundary candidates appearing as nearby pairs of differently colored dots.

AutoBVA discovers boundaries across the input space, but its focus on maximizing the program derivative means it converges on the input pairs where behavioral transitions are sharpest, leaving some regions of each boundary unexplored. For this FUT, SETBVE does not find different boundaries, **it finds more of the same boundaries.** By keeping the best candidate in each region of the behavioral space rather than converging on the few input pairs with the highest program derivative, the archive ensures that the search continues to fill underrepresented areas, covering each boundary curve far more completely. A telling example is the Normal-Overweight boundary (green and yellow dot pairs): AutoBVA finds only a handful of candidates along this curve, leaving most of it unexplored, while SETBVE traces it continuously across a wider range of height and weight combinations. Regions of the boundary landscape with lower program derivative values risk going undetected by quality-focused approaches, even when meaningful behavioral transitions exist there.

Figure 1.3 visualizes the quality-diversity tradeoff across four FUTs of increasing complexity: `bytecount` (a single-argument function converting byte counts to human-readable strings), `circle` (a two-argument function determining whether a point lies inside a circle), `bmi` (a two-argument function returning BMI categories from height and weight), and `date` (a three-argument function with interdependent calendar constraints). Each point in the plot represents the average performance of one search configuration across repeated runs. The horizontal axis shows Relative Archive Coverage (RAC), which measures how much of the behavioral space has been explored, and the vertical axis shows Relative Program Derivative (RPD), which measures the quality of the bound-

ary candidates found. A configuration in the top-right corner would be ideal: high quality and high diversity. The two rows correspond to a 30-second and a 600-second time budget. In each plot, the red and orange colors represent the two baselines, AutoBVA and random sampling, while the remaining colors are different SETBVE configurations.

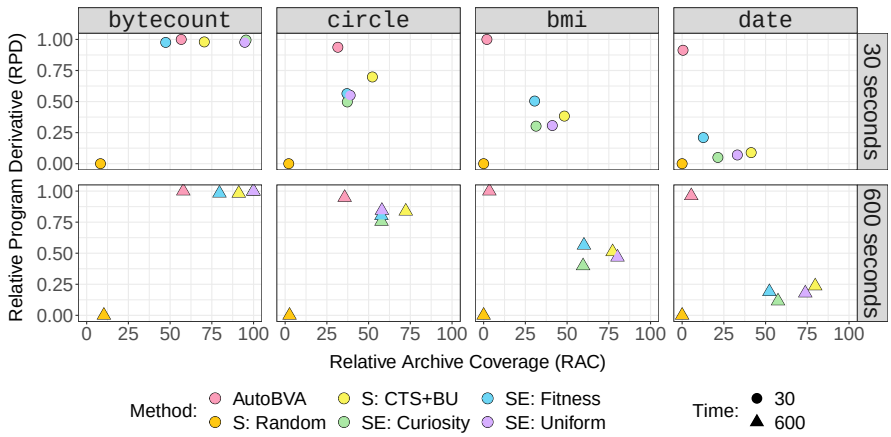

Figure 1.3: Quality (RPD) and diversity (RAC) of boundary candidates discovered by AutoBVA, random sampling, and different SETBVE configurations across four FUTs, under 30-second and 600-second time budgets.

Two patterns stand out clearly. The first is the quality-diversity tradeoff: AutoBVA consistently sits at the top of each plot, achieving high RPD, but far to the left, indicating low coverage. SETBVE configurations spread further to the right, achieving higher diversity, but at somewhat lower quality. This is the cost of explicitly pursuing coverage: the search distributes its effort across the behavioral space rather than repeatedly refining the same high-scoring region.

The second pattern relates to time. Comparing the 30-second and 600-second rows, SETBVE configurations shift noticeably to the right with the extended budget, gaining diversity as the search continues to fill previously undiscovered archive cells. AutoBVA, by contrast, shows almost no horizontal movement: it converges quickly and finds little additional behavioral coverage with more time. This suggests that quality-focused search reaches a ceiling early, while diversity-aware search continues to improve as long as the budget allows, a property that becomes increasingly valuable as function complexity grows, as illustrated by the progressively larger spread of configurations from `bytecount` to `date`.

Broadening Boundary Discovery (RQ1): Incorporating a Quality-Diversity approach into automated BVE leads to discovering more boundary candidates across a wider range of defined behavioral regions. This comes with a tradeoff: configurations that achieve higher diversity tend to sacrifice some boundary quality. The diversity gains accumulate over time, with QD-based configurations continuing to improve under extended time budgets.

1.3.2 Generalizing Boundary Exploration Across Input Domains

SETBVE demonstrated that incorporating diversity into the boundary search leads to a richer picture of a program's behavioral landscape. However, it still relies on operators (the procedures that sample, mutate, and refine input pairs) that must be manually defined for each input type, keeping automated BVE confined mainly to numeric domains. For a function that takes integers, defining how to mutate an input pair is straightforward: bring the two values closer together and apply a small numeric shift. For a function that takes strings the same question has no obvious answer. What constitutes a "small shift" for a string? Answering this question requires both knowledge of the data type and understanding of the specific function's semantics, knowledge that must be supplied by the tester for each new input type, a bottleneck that ultimately limits the reach of automated BVE.

To address this, we introduced ABEX, an agentic, LLM-driven framework that builds on SETBVE's archive structure and behavioral descriptors while replacing hand-crafted operators with autonomously generated exploration strategies. Rather than specifying how to explore the input space in advance, **the framework generates its own strategies** during the search, guided by execution feedback and the current state of the archive. The main tradeoff of this approach is cost: because each exploration step involves LLM calls, each run consumes on the order of millions of tokens, amounting to a few dollars per run at current API pricing.

As illustrated in Figure 1.4, the framework follows a three-layer architecture: a strategic layer that directs the overall search, a tactical layer that produces exploration strategies, and an operational layer that executes them. The *Coordinator* agent implements the strategic layer, monitoring search progress and deciding when to continue with the current strategy, switch to a previously generated one, or request an entirely new one. Given a search gap identified by the Coordinator, the *Ideator* and *Strategy Generator* collaborate at the tactical layer to produce a concrete, step-by-step natural-language instruction for generating new input pairs — the exploration strategy. The *Explorer* then implements the operational layer, following this strategy to generate candidate input pairs, evaluating them against the FUT, and updating the QD archive accordingly.

A key insight emerging from both the framework design and the empirical evaluation is that not all strategies serve the same purpose. Generation strategies, which produce new input pairs from scratch, are the primary driver

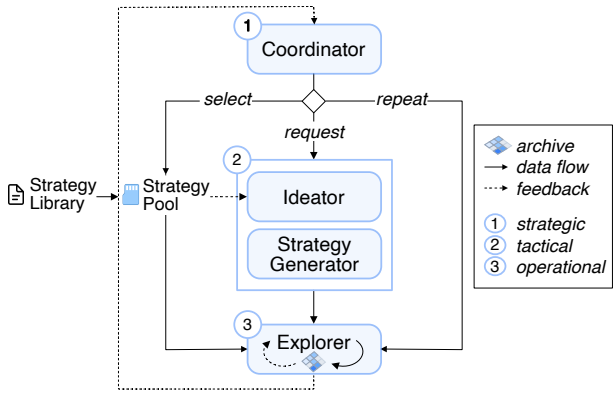

Figure 1.4: High-level overview of the ABEX agentic framework and its components.

of coverage, steering the search toward unexplored regions of the behavioral space. Mutation strategies, which select existing archive candidates and apply targeted modifications, are the primary driver of quality, refining the program derivative of candidates already in the archive. This division of labor mirrors the exploration-exploitation tradeoff familiar from evolutionary search, but here it arises naturally from the different roles of the strategy types rather than from a predetermined schedule.

The framework was evaluated on 20 FUTs, listed in Table 1.2, spanning four input categories: integer, string, array, and mixed inputs. For numeric inputs, where comparison with existing automated BVE is possible, adaptive strategy generation proves competitive, producing higher-quality boundary candidates than both a search-based and a single-prompt LLM baseline on most tested functions.

The more significant contribution lies in what happens beyond numeric inputs, where automated black-box BVE had not previously been demonstrated. For string-based functions, the framework discovers meaningful behavioral transitions aligned with the domain logic of each function, including VV transitions between output categories, as well as VE transitions where one input produces a valid output and the other triggers an error. For array-based functions, it uncovers structural boundaries including empty versus non-empty transitions and length-change boundaries, again spanning both VV and VE validity groups. For mixed-input functions combining arrays, integers, and booleans, meaningful boundaries are discovered even in the most complex configurations, spanning both VV transitions between different valid outputs and VE transitions where one input triggers an error. Across all non-numeric input types, these boundaries are discovered without any input-type-specific engineering: the same framework, the same archive, the same strategy generation mechanism, applied uniformly across domains.

Table 1.2: Functions under test used in the ABEX evaluation.

Name	Description	#Args	Data type
bmi [8], [41]	Classify BMI category	2	Integer
bytecount [8], [41]	Format bytes with SI prefix	1	Integer
calDate [12]	Convert to Julian day number	3	Integer
circle [41]	Point inside/outside circle	2	Integer
complexCheck* [12], [42]	Nested conditional branches	3	Integer
date [8], [41]	Format as ISO date string	3	Integer
englishScore [43]	Determine pass/fail status	2	Integer
findMiddle [12]	Return median of three	3	Integer
nextDate [12]	Compute next calendar date	3	Integer
tritype [12], [44]	Classify triangle type	3	Integer
passwordStrength	Assign strength level	1	String
printTokens	Tokens classification	1	String
replace	Match pattern and substitute	3	String
stringPalindrome [45]	Check palindrome or not	1	String
validateEmail [46]	Email format validity	1	String
insertionSort	Insertion sort algorithm	1	Array[int]
normalize [41]	Min-max array scale to [0, 1]	1	Array[int]
maxLexString [47]	Max lexicographical string	1	Array[str]
binarySearch	Search for target value	2	Mixed (arr[int], int)
tcas [12]	Collision avoidance system	12	Mixed (int, bool)

* The FUT has two specifications (docstrings): a short version consistent with other FUTs and a more detailed one.

Applicability Across Input Domains (RQ2): An agentic, LLM-driven system can replace hand-crafted, type-specific operators, making automated BVE more generally applicable across diverse input types. Applied to functions with integer, string, array, and mixed inputs, it discovers meaningful boundary candidates, including VV and VE transitions, without any per-type engineering.

1.3.3 Explaining Discovered Boundaries

Discovering boundary candidates is only part of the challenge. A boundary that cannot be understood or justified is unlikely to be trusted or acted upon, much like auto-generated test cases, which developers often discard when they lack transparent justification [33].

To investigate whether LLMs can close this gap, we conducted a mixed-methods study involving participants from both industry and academia, who evaluated LLM-generated explanations on clarity, correctness, completeness, and perceived usefulness. **The overall reception was positive**: across all criteria and functions, the majority of ratings fell on the agreeing side of the scale, and most participants indicated they would consider using tools that provide such explanations. This suggests that LLM-generated explanations can add genuine value to automated boundary testing workflows, helping testers understand why a particular input pair represents a meaningful behavioral transition.

Yet the results also revealed important nuances. Correctness proved the most fragile dimension: a single hallucination noticeably reduced ratings not only

for correctness but also for perceived usefulness, as participants unsurprisingly found little value in explanations they could not trust. Participants were also critical of explanations that lacked sufficient context, relied on undefined technical terminology, or failed to situate the boundary within a broader picture of the function's behavior.

The interview analysis deepened these findings through three interconnected themes, illustrated in Figure 1.5. The first concerns **boundary explanation design**: participants were clear that a good explanation goes beyond stating facts. It should include additional examples of nearby boundary pairs to show how behavior changes, present information in a predictable structure, and calibrate its level of detail to the reader's expertise. The second theme addresses **use cases and tooling**: participants envisioned explanations surfacing naturally within debugging workflows, static analysis tools, and documentation, rather than as standalone outputs. For junior engineers or those unfamiliar with a system, explanations were also seen as a valuable learning resource. The third theme concerns **trust and reliability**: trust was not a fixed quality but one that fluctuated with accuracy. Participants were open to using auto-generated explanations, especially when they filled gaps in domain knowledge, but they cautioned against over-reliance, emphasizing that explanations should support rather than replace engineers' own reasoning.

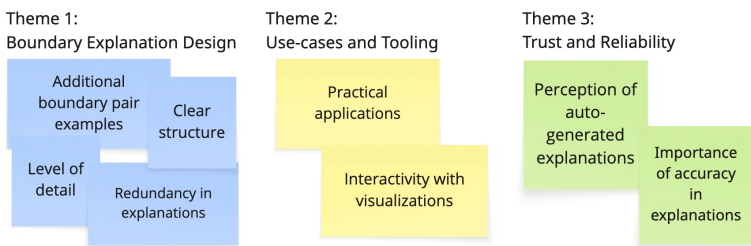

Figure 1.5: Themes and codes identified through thematic analysis of the interview data on boundary explanations.

Taken together, the survey and interview findings crystallize into a concrete set of design requirements[1] for future boundary explanation tools, derived from both the survey feedback and the thematic analysis of the interviews. The requirements go beyond surface-level formatting advice and describe the kind of relationship between a tester and an explanation tool needed for boundaries to be genuinely useful in practice.

At the core is the need for adaptive verbosity: explanations should adjust their depth to the reader's expertise, offering concise summaries for experienced testers while providing expandable detail, definitions, and links to authoritative sources for those less familiar with the domain. Closely related is the requirement for explicit structure: presenting valid and invalid behaviors in a consistent, predictable sequence reduces cognitive load and makes it easier to compare explanations across boundary pairs. Participants were also clear

[1] A more detailed description of these requirements is provided in Paper C (Section 4.5.1)

that explanations should not stand in isolation. They called for multiple examples around each boundary, showing how behavior changes incrementally on either side of the transition, and for contextual justification that makes the domain assumptions behind each boundary explicit, thereby reducing the risk of hallucinations going unnoticed.

Two further requirements reflect a more ambitious vision of how explanation tools could be integrated into testing workflows. First, participants expressed a desire for interactive dialogue, where testers can query or challenge specific claims in real time rather than passively reading static text. Second, they emphasized the importance of tooling integration, so that explanations surface in context during debugging, static analysis, or test validation, rather than requiring a separate step. Taken together, these requirements do not just define what a good explanation contains; they describe a direction for making automated boundary testing genuinely understandable and adoptable in practice.

Perceiving Generated Boundary Explanations (RQ3): LLM-generated natural-language explanations of software boundaries are perceived as useful. However, correctness is the most critical dimension: hallucinations erode trust and reduce perceived usefulness. The findings translate into a requirement checklist for future boundary explanation tools, emphasizing adaptive verbosity, explicit structure, contextual justification, and interactive dialogue.

1.4 Reflections

The previous sections have presented the studies and their results individually, each answering a specific research question. This section steps back from the individual contributions and reflects on what they reveal together. Rather than summarizing findings, the goal here is to examine the tensions, limitations, and open questions that cut across all three studies: what the results suggest about how boundary quality should be understood, the different challenges that arise when deploying LLMs as generators versus explainers, the difficulty of measuring progress in a field without established ground truth, and what the relationship between the frameworks developed here looks like when viewed side by side. The section closes with a reflection on the role of the tester in relation to these tools, a question that none of the studies fully resolves but that points toward what comes next.

Quality and diversity as complementary goals. A recurring question across the studies is what constitutes a good set of boundary candidates. Quality in BVE is operationalized through the program derivative, which measures how sharply program behavior changes between two nearby inputs. This is a principled and tractable definition, but it carries an implicit assumption: that the sharpest boundaries are the most valuable ones. The results challenge this assumption. Many high-PD candidates turn out to be structurally similar to one another, providing redundant information about the same behavioral

transition while leaving other regions of the input space unexplored. A boundary candidate that scores slightly lower on the program derivative may reveal a distinct behavioral region that a quality-focused search would never reach. Quality is therefore better understood as encompassing both the strength of a behavioral transition and the diversity of the region it represents, with the two working together rather than against each other. Operationalizing diversity, however, requires choices. It is defined through behavioral descriptors that partition the search space into cells, each representing a distinct behavioral region, and the choice of descriptors shapes which distinctions the search treats as meaningful. The descriptors used in this thesis are deliberately general, which supports applicability across different functions, but the right balance between quality and diversity, and what diversity should mean in a given context, ultimately depends on the testing goals and the complexity of the function under test.

Our metrics are proxies, not ground truth. A structural challenge is the lack of ground truth for what a complete or representative set of boundaries looks like for a given function. This is not a failure of individual study design but a fundamental property of the field: automated BVE is recent enough that no established benchmark exists with known boundaries against which discovered candidates can be measured. We do not know, for a given function, how many distinct behavioral boundaries exist, how much of any single boundary has been traced, or whether the candidates we discover are the ones that matter most for testing. The metrics used across the studies are carefully motivated, but they all measure relative improvement rather than absolute coverage.

Defining what counts as a boundary for the purpose of ground truth is non-trivial. Boundaries are not discrete, enumerable objects like lines of code or branches in a control flow graph. They are continuous regions in the input space where behavior shifts, and their significance depends on the distance functions used to measure behavioral change and the testing goals of the person using the tool. A boundary that is highly significant from one perspective, say, a transition between a valid output and an error, may be less informative than a subtler VV transition in a different region, depending on what the tester is trying to understand. This context-dependence makes it difficult to define what complete or representative coverage would even mean.

The field is therefore in a position similar to where mutation testing or combinatorial testing were in their early stages: methodologically promising, but lacking the evaluation infrastructure that would allow rigorous comparison across tools and techniques. Progress will likely require the community to invest in constructing benchmark suites of functions with manually verified and annotated boundary sets, alongside metrics that connect discovered boundaries to outcomes testers actually care about, such as fault detection or specification coverage. Until such infrastructure exists, the metrics we use will remain useful approximations, and the claims we can make about the effectiveness of automated BVE tools will remain appropriately modest.

LLMs as generators and explainers: two different challenges. Both ABEX and the explanations study use LLMs, but in fundamentally different roles that come with very different reliability profiles. In ABEX, the LLM acts as a strategy generator: it proposes exploration directions, which are then executed and evaluated. Errors in strategy generation are naturally filtered by execution feedback: a bad strategy simply fails to produce useful boundary candidates and is deprioritized by the Coordinator. The system is self-correcting by design, and the non-determinism of LLMs is actually a feature rather than a liability: producing varied and creative strategies across runs is precisely what drives exploration of new behavioral regions. Even when the Coordinator filters out ineffective strategies, the cost of a failed attempt is bounded, a few wasted iterations rather than a breakdown of the search.

In the explanations study, the picture is different. The LLM acts as a rationale producer, generating text that a human must evaluate and trust directly. Here there is no automatic filter. A hallucination reaches the tester unmediated, and as the results show, even a single factual error is enough to erode confidence not just in that explanation but in the tool as a whole. Critically, the same non-determinism that makes LLMs valuable as creative strategy generators becomes a liability when they serve as explainers. Participants in the study noted that inconsistent explanations across repeated runs would further reduce trust. A tester who receives different rationales for the same boundary pair on different occasions cannot build a stable understanding of why that boundary matters. Consistency, which is at odds with the stochastic nature of LLMs, turns out to be a prerequisite for explanation to be trusted.

This asymmetry reveals something deeper about how LLMs should be deployed in automated testing. When LLMs operate within a feedback loop that filters their output by execution results, their creativity and variability are assets. When they produce artifacts consumed directly by humans with no mediating filter, those same properties become risks. The implication is not that LLMs are unsuitable for explanation, but that deploying them in that role requires additional safeguards: lower sampling temperatures, verification steps, or mechanisms that flag when the same boundary receives markedly different explanations across runs. The two roles call for fundamentally different engineering choices, and conflating them risks undermining trust in what could otherwise be a powerful combination.

Complementarity over competition. A natural reading of the two search-based studies is that each one improves upon the previous: SETBVE adds diversity to AutoBVA, and ABEX removes the operator engineering bottleneck. But a more nuanced reading reveals that SETBVE and ABEX are not in competition. They make different tradeoffs that suit different settings. SETBVE covers more of the behavioral space under a fixed budget and is computationally cheap; ABEX discovers higher-quality candidates per cell and generalizes to a wider range of input types without requiring manually engineered search operators, but at a higher cost per iteration. Importantly, ABEX already builds on SETBVE's archive structure, using the same behavioral descriptors. The two frameworks therefore share a common foundation, and the

question is less about which to use and more about how to use them together most effectively.

One way to think about this is in terms of budget allocation: a combined system could dedicate part of the search budget to SETBVE's efficient, search-based exploration and part to ABEX's adaptive, LLM-driven strategy generation, dynamically shifting the balance depending on the input type and how the search is progressing. For numeric inputs where SETBVE is fast and effective, the LLM overhead of ABEX may not be justified early in the search, but could add value once the search stagnates. For inputs where SETBVE has no applicable operators, ABEX would take the lead from the start. This kind of adaptive, budget-aware combination is not yet realized in either framework, but the shared archive structure means the technical foundation for it already exists. Reflecting on the two studies together, what emerges is a picture not of two competing tools but of two complementary perspectives on the same underlying problem — one optimizing for breadth and efficiency, the other for adaptability and semantic richness.

The human in the loop remains underspecified. All three studies assume that testers are the ultimate consumers of boundary candidates, but none fully addresses what the human-tool interaction should look like in practice. Both SETBVE and ABEX produce ranked lists of boundary candidates for the tester to inspect. The explanations study pairs these candidates with natural-language rationales. Yet in all cases, the tester is presented with results after the fact, with limited ability to guide or redirect the search while it is happening.

The interview data from the explanations study hints at what a richer interaction might look like: testers who want to challenge assumptions, or request additional examples around a boundary they find interesting. But the desire for interaction goes beyond explanation. A tester who looks at a ranked list of hundreds of boundary candidates and finds them overwhelming is receiving information without a way to prioritize or navigate it. A tester who notices that a particular region of the input space seems underexplored has no mechanism to direct the search there. The tools, as they currently stand, are not designed to receive that kind of feedback.

What this reveals is that the current studies treat the tester primarily as a consumer of outputs rather than a participant in the search. This is a reasonable first step in a young field, where establishing that automated BVE can work at all is the primary goal. But it points to an important open question about what a genuinely useful boundary exploration tool should ultimately look like: one where the boundary between exploration and explanation is less sharp, and where the tester's understanding of what has been found can actively shape what the tool looks for next.

Threats to Validity

Construct validity. Boundary quality is measured through the program derivative, which depends on the choice of distance functions and may not cap-

ture all forms of behavioral distinction equally well. The behavioral descriptors used to organize the archive define what the search treats as diverse, meaning coverage metrics reflect diversity with respect to those specific descriptors rather than in any absolute sense. In the explanations study, explanation quality is assessed through subjective Likert ratings, which are shaped by participants' prior expertise and interpretation of the criteria.

Internal validity. Both search-based studies involve stochastic components, and results are averaged across repeated runs to account for variation. ABEX additionally introduces LLM non-determinism, which is harder to characterize and control than algorithmic randomness alone. In the explanations study, some participants may have rated explanations based on their perception of the boundary pairs themselves rather than the explanations, introducing a potential confound.

External validity. All three studies operate at the unit level, evaluating between 20 and 30 functions under test. While the functions vary in structure and complexity, they are small and well-defined relative to functions in real-world software systems. Generalization to larger systems or richer input types cannot be assumed without further investigation.

Conclusion validity. The relatively small number of functions evaluated, combined with LLM non-determinism and the limited participant sample in the explanations study, constrains the statistical confidence of the conclusions. Results should be interpreted as indicative of trends rather than definitive claims about the effectiveness of the approaches across all settings.

1.5 Next Steps

The reflections presented earlier point to several open questions to address in the continuation of this research.

From functions to systems. All three studies operate at the unit level, targeting individual functions under test. While this is a natural starting point that enables controlled comparison and clear measurement, real software systems are rarely tested one function at a time. Extending BVE to the system under test level, where multiple functions interact and the input space is shaped by external interfaces and system state, is a compelling next step. The black-box nature of the approach means that this extension should be feasible without fundamental changes to the underlying framework, but the practical challenges of scaling to larger systems, handling inter-component dependencies, and managing the increased search space warrant dedicated empirical investigation. Another challenge at the system level is presentation: as the input space grows and the number of discovered boundary candidates increases, new ways of summarizing and visualizing results will likely be needed.

Building a boundary benchmark. The lack of a ground truth for boundary discovery, discussed in the reflections, points to a more foundational need: the community requires a benchmark suite of functions with manually verified and annotated boundaries. Such a benchmark would not only enable rigorous comparison across tools and techniques but also make it possible to measure how much of a boundary has actually been found, rather than only how much has been found relative to other methods. Constructing this benchmark is non-trivial precisely because of the context-dependence of boundaries discussed earlier, but a curated set of functions with well-understood behavioral structure and documented transitions would be a valuable contribution in itself. This is a concrete research artifact that the BVE community can build on, and one worth pursuing as a direct contribution to the field.

Rethinking boundary types. A question that sits beneath all three studies is what kinds of boundaries actually exist, and whether current tools find all of them. The validity group classification into VV, VE, and EE captures whether outputs are valid or erroneous, but it does not capture the structural complexity of boundaries. In a function with multiple output classes, a boundary candidate can straddle two adjacent partitions, but a richer and potentially more informative kind of boundary is one that lies at the convergence of three or more partitions, a point in the input space from which small changes in different directions lead to qualitatively different behaviors. Such "multi-partition boundaries" are likely to be particularly fault-revealing, yet current search strategies are not explicitly designed to seek them out. Understanding the taxonomy of boundary types that can exist, and designing exploration strategies that target each type deliberately, is a promising direction.

Toward interactive exploration. A final direction builds on the observation that current tools treat the tester as a consumer of results rather than a participant in the search. The reflections stop at identifying this gap; the next step is to close it. One concrete mechanism would be to allow testers to interactively adjust the behavioral descriptors that define the archive during the search, effectively redirecting exploration toward regions they find more interesting or relevant. This is a richer form of interaction than simply inspecting a ranked list of results: a tester who notices that the archive is densely populated in one region but sparse in another could shift the search to incentivize exploration elsewhere. Another mechanism would be to use the feedback collected during boundary explanation, for instance a tester flagging an explanation as incorrect or requesting a different example, to actively inform what the search prioritizes next. This vision of a genuinely bidirectional tool, where discovery and understanding reinforce each other, is a longer-term goal, but the components are beginning to be in place.

The broader picture. Taken together, these directions reflect a broader ambition: to make boundary value exploration a practical and integrated part of how software is tested, not just an academic technique applied to isolated

functions. Software is increasingly developed, deployed, and maintained with automation at its core, and the role of the tester is shifting from manually writing test cases to guiding, validating, and understanding the output of automated systems. The studies in this thesis are a step toward BVE tools that fit naturally into that shift, ones that can be pointed at a function or a system, run without manual operator engineering, and return results that a tester can understand and build on. As agentic systems become more capable of autonomous exploration and LLMs become more reliable as reasoning partners, the boundary between what a tool does and what a tester does will continue to evolve. The goal is not to remove the tester from the process, but to give them better tools for the questions that matter most: where does this program's behavior change, why does it change there, and is the software doing what it should?

1.6 Conclusion

This thesis has advanced automated boundary value exploration along three directions: making the search broader through quality-diversity optimization, extending it to a wider range of input types through adaptive, LLM-driven strategy generation, and investigating whether natural-language explanation can make discovered boundaries more understandable to the testers. Together, the contributions move BVE toward a process that is more general, more diverse, and more connected to the people who use its results. What remains is to close the gap between the tools developed here and the practical realities of software testing: establishing ground truth through benchmarks, scaling from individual functions to larger systems, and building the kind of bidirectional human-tool interaction that makes boundary exploration not just automated, but practical and adoptable in practice.