Université de Lille, Faculté des sciences et technologies

Département de formation doctorale en informatique **École graduée MADIS Lille**

Metamodel-Based Language-Agnostic Approach to Unit Test Generation From Execution Traces

Approche indépendante du langage fondée sur des métamodèles pour la génération de tests unitaires à partir de traces d'exécution

THÈSE

présentée et soutenue publiquement le 5 décembre 2025

pour l'obtention du

Doctorat de l'Université de Lille

spécialité Informatique

par

Gabriel Darbord

Composition du jury

Président :	Philippe MERLE	Directeur de Recherche, Inria Lille, France
Rapporteurs :	Olivier BARAIS	Professeur des Universités, Université de Rennes, France
	Gerson SUNYÉ	Maître de Conférences HDR, Nantes Université, France
Examinateur :	Coen DE ROOVER	Professeur, Vrije Universiteit Brussel, Belgique
Directrice :	Anne ETIEN	Professeure des Universités, Université de Lille, France
Co-Directeur :	Nicolas ANQUETIL	Maître de Conférences HDR, Université de Lille, France
Invité :	Benoit VERHAEGHE	Manager de Projet Scientifique, Berger-Levrault, France

Centre de Recherche en Informatique, Signal et Automatique de Lille — UMR 9189
Inria Lille - Nord Europe

Remerciements

Je souhaite tout d'abord remercier les membres du jury pour le temps qu'ils ont consacré à l'évaluation de ce travail, pour la qualité de leurs remarques et pour les échanges qu'ils ont permis lors de la soutenance.

Je remercie sincèrement mes encadrants, Anne, Nicolas et Benoit, pour leur accompagnement tout au long de ces années. Leur disponibilité, leur soutien constant et la confiance qu'ils m'ont accordée ont été déterminants pour la conduite et l'aboutissement de ce travail. J'ai particulièrement apprécié la qualité des échanges, tant scientifiques qu'humains, qui ont rythmé cette thèse.

Je tiens à remercier mon équipe de recherche et son directeur, Stéphane, pour le cadre de travail stimulant, les échanges et l'attention portée au collectif. Merci à mes collègues pour les discussions, l'entraide et les moments partagés au fil de la thèse. Une pensée particulière pour Valentin, avec qui j'ai commencé et terminé cette aventure doctorale, et dont l'amitié a largement contribué à rendre ce parcours plus agréable.

Enfin, je souhaite remercier ma famille. Merci à ma mère, Catherine, pour son soutien indéfectible, malgré la distance géographique qui nous a séparés au cours de ces années, depuis que nos chemins se sont éloignés de Montpellier, le mien vers Lille et le sien vers la campagne portugaise. Sa présence, même à distance, a été une source constante de réconfort et d'encouragement.

Abstract

Legacy software systems often lack automated unit tests, which makes it difficult to detect regressions during maintenance or migration. This challenge is faced by our industrial partner, Berger-Levrault, whose applications are currently being migrated from monolithic Java systems to a Software-as-a-Service architecture with an Angular front-end. To address this, this thesis investigates the automatic generation of unit tests from program execution traces to capture existing system behavior.

We use a technique called test carving to extract tests from execution traces. A trace is a record of behavior observed during execution that includes input and output values. Each trace can serve as the basis for a unit test by replaying the recorded behavior with the same inputs and verifying that the result matches the original output. To support developer adoption, we generate tests that reconstruct recorded values as code, improving their readability and maintainability.

Our approach is based on abstract representations of code, values, and tests, defined through metamodels. These abstractions make it possible to specialize the approach to different programming languages and test frameworks. We implemented the approach in a concrete tool and evaluated it on systems written in Java and Pharo, covering both open-source and industrial software. The results show that it can generate readable tests that capture realistic system behaviors.

Overall, this work contributes a practical solution for deriving regression tests for legacy systems, while demonstrating how metamodel-based abstractions enable extensible test carving across diverse technical contexts.

Keywords: Test generation, Regression testing, Test carving, Execution traces, Metamodels, Software testing tools, Software maintenance.

Résumé

Les systèmes logiciels hérités manquent souvent de tests unitaires automatisés, ce qui rend difficile la détection des régressions lors de la maintenance ou de la migration. Notre partenaire industriel, Berger-Levrault, est confronté à ce défi, car ses applications sont actuellement en cours de migration depuis des systèmes Java monolithiques vers une architecture « Software-as-a-Service » avec une interface Angular. Pour y remédier, cette thèse étudie la génération automatique de tests unitaires à partir des traces d'exécution du programme afin de capturer le comportement existant du système.

Nous utilisons une technique appelée « test carving » pour extraire des tests à partir de traces d'exécution. Une trace est un enregistrement du comportement observé pendant l'exécution qui comprend les valeurs d'entrée et de sortie. Chaque trace peut servir de base à un test unitaire en rejouant le comportement enregistré avec les mêmes entrées et en vérifiant que le résultat correspond à la sortie d'origine. Afin de faciliter l'adoption par les développeurs, nous générons des tests qui reconstruisent les valeurs enregistrées sous forme de code, améliorant ainsi leur lisibilité et leur maintenabilité.

Ces abstractions permettent de spécialiser l'approche pour différents langages de programmation et cadriciel de test. Nous avons implémenté notre approche dans un outil concret et l'avons évaluée sur des systèmes écrits en Java et Pharo, couvrant à la fois des logiciels open-source et industriels. Les résultats montrent qu'elle permet de générer des tests lisibles qui capturent les comportements réalistes du système.

Dans l'ensemble, ce travail apporte une solution pratique pour dériver des tests de régression pour les systèmes hérités, tout en démontrant comment les abstractions basées sur des métamodèles permettent de créer des tests de manière extensible dans divers contextes techniques.

Mots-clés: Génération de tests, Tests de régression, Extraction de tests, Traces d'exécution, Métamodèles, Outils de test logiciel, Maintenance logicielle.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Context

Unit testing is a cornerstone of modern software development practices. It enables developers to check the correctness of individual units of code, detect regressions when code evolves, and ensure that long-term maintenance does not compromise system integrity [Beck 2000, Beck 2002, Shore 2021, Siddiqui 2021]. A comprehensive test suite provides confidence that software behaves as expected, but creating and maintaining such suites is time-consuming and costly. In practice, many software systems lack automated unit tests due to the significant manual effort required to develop them [Daka 2014].

This problem is exacerbated for legacy systems developed *circa* 2000 and earlier, which were created before unit testing became a mainstream practice. These systems have often been in use for decades, gradually accumulating bug fixes and incremental changes. While such evolution results in systems that are relatively stable in their current form, it also makes them fragile to large-scale changes. When organizations attempt to migrate legacy applications, for example, to new architectures, deployment models, or user interfaces, the absence of automated regression tests becomes a critical risk. Without automated regression tests, developers cannot easily verify that the new version preserves the expected behavior of the old one.

This challenge is faced by our industrial partner, Berger-Levrault, which maintains large-scale monolithic Java applications that are currently being migrated to a Software-as-a-Service (SaaS) architecture with an Angular frontend. The scale and criticality of these systems make manual test creation impractical, yet the migration process demands reliable regression tests to ensure functional equivalence between old and new versions.

Automated test generation provides a promising solution to this problem. Among the available techniques, test carving directly addresses the need to capture existing system behavior. Test carving leverages execution traces, *i.e.*, records of individual method executions that capture the inputs provided (pre-state) and the outputs produced (post-state). These traces can then be replayed as unit tests, which act as regression tests whose oracles are defined by the recorded outputs. Carving can generate tests even when no pre-existing tests or formal specifications are available, making it particularly well suited to legacy contexts. Moreover, because these systems have evolved through decades of bug fixes and stabilization, their observed executions provide a reliable basis for defining regression oracles. Empirical evidence further shows that developers value tests derived from such actual executions over those based on synthetic inputs, as they find them more understandable and meaningful in practice [Deljouyi 2023].

Beyond the source of test data, adoption by developers also depends on how tests are expressed. Automatically generated tests that simply serialize inputs and outputs in raw form are often difficult to interpret, limiting their usefulness in practice. To support adoption, generated tests must reconstruct values as source code and integrate naturally with existing test frameworks. Readability is not a single measurable property, but indicators such as the absence of code and test smells are widely used in practice to assess whether tests are likely to be understandable and maintainable [Fowler 1999, van Deursen 2001]. This combination of regression safety and readability forms the foundation of the approach developed in this thesis.

1.2 Problem Statement

The context described above highlights a concrete industrial need: our partner Berger-Levrault maintains monolithic Java applications with thick clients, where users previously had to install a desktop application communicating with the backend through Remote Procedure Call (RPC). These systems are being migrated to a SaaS model, in which the frontend is replaced by a thin, web-based client written in TypeScript using the Angular[1] framework. The backend remains implemented in Java, but is now exposed through an HTTP API. While the frontend is being rewritten with modern development practices, including systematic testing, the legacy backend is preserved with very few unit tests and no formal specifications. The absence of automated regression tests for this backend poses a major obstacle, as there is no reliable way to ensure that its behavior is preserved during and after the migration.

The legacy applications at Berger-Levrault are several decades old, and as a consequence their specifications are either missing or outdated. Although the

[1] `https://angular.dev`

problem first appeared in the context of a Java application, Berger-Levrault faces the same issue with other systems written in different programming languages. Any solution must therefore not only address the lack of tests in the Java backend, but also generalize across ecosystems.

From the constraints of this industrial context, we derive five requirements that the solution investigated in this thesis must fulfill. First, it must operate without pre-existing unit tests. Second, it must not depend on formal specifications. Third, it must provide a regression oracle: tests must verify that previously observed behavior continues to hold across versions, so that no functionality is lost during migration. Fourth, the generated tests must reflect observed executions rather than synthetic inputs designed solely to optimize objectives such as code coverage. Fifth, the solution must not be locked to a single ecosystem, but instead be applicable across multiple programming languages and test frameworks.

Taken together, these requirements define the central problem that this thesis addresses. The challenge is to construct automated regression tests for such systems, so that unintended changes in behavior across versions can be detected. These tests must operate at the unit level and be obtained without imposing substantial manual effort.

In addition to these requirements, we pursue a specific goal: the generated tests should be readable, expressed in the idioms of the target language and test framework so that developers can readily adopt and maintain them. The objective is to bootstrap a missing regression suite by generating an initial suite of regression tests, enabling developers to establish coverage rapidly. Once this baseline is in place, development can proceed with standard practice, in which software evolution is accompanied by manually written tests.

This thesis addresses these requirements and goal by leveraging program execution traces as behavioral ground truth and transforming them into unit tests that developers can adopt. The next section summarizes the concrete contributions of this work, including our metamodel-driven approach, its implementation in our tool called Modest, and its evaluation on Java and Pharo systems (JUnit 4/5 and SUnit).

1.3 Contributions

The first contribution of this thesis is an approach to test carving, that is, the automatic derivation of unit tests from execution traces. A trace records method calls together with their inputs and outputs; by replaying these calls as tests, we obtain regression oracles that verify whether previously observed behavior still holds. Because the traces are collected from realistic executions of the system, the resulting tests reflect the behaviors that matter in practice.

This carving process is organized around explicit metamodels for code,

tests, and values. A metamodel is an abstraction that defines the elements and relations relevant in a given domain. By structuring the approach in terms of such metamodels, we decouple the core process from language- and framework-specific details, making it possible to specialize the same architecture across different technical contexts. Runtime values observed in traces are reconstructed as source code through the value metamodel, so that the generated tests remain readable and maintainable units that developers can adopt as part of their regular workflow.

The second contribution is the implementation of this approach in a tool called MODEST[2]. At present, MODEST supports two programming languages and multiple test frameworks: Java with JUnit 4 and JUnit 5, and Pharo with SUnit. Its design exposes clear extension points for importing application models, importing execution data, exporting values as source code, and exporting unit tests expressed in the idioms of the target framework. Assertions can be produced using default equality (*e.g.*, `assertEquals` in JUnit) or structural (deep) equality. These features make MODEST adaptable to different technical contexts.

The third contribution is an empirical evaluation of MODEST across languages and contexts. We applied the tool to both open-source projects and Berger-Levrault's proprietary systems. The evaluation measured (i) the proportion of eligible methods for which executable tests could be generated, (ii) the pass rate of the generated tests, and (iii) their readability in terms of the presence or absence of code and test smells. Results showed that the approach is feasible in practice, with most generated tests executing successfully and exhibiting readability properties consistent with established developer expectations.

Together, these contributions provide a practical path to obtain automated regression tests for legacy systems, while keeping the generated tests intelligible and compatible with established development workflows.

1.4 Thesis Outline

The remainder of this thesis is organized as follows.

Chapter 2 reviews the state of the art in automated unit test generation. It surveys the main families of techniques, evaluates their suitability in contexts without tests or specifications, and motivates the choice of test carving. It then defines criteria to compare existing carving approaches with an emphasis on readability, and finally reviews existing metamodels to situate our model-based contribution.

Chapter 3 introduces the metamodels that form the foundation of our approach. It explains the role of metamodels in bridging execution traces and

[2]The name MODEST is a portmanteau of "model" and "test."

test generation, presents our modeling infrastructure including an existing Code metamodel, and details the Unit Test and Value metamodels developed in this work.

Chapter 4 presents our carving approach and its implementation in the tool MODEST. It describes the test generation pipeline, the inputs required at each step, and the extension points for supporting new languages and frameworks. Examples from Java and Pharo illustrate how the architecture enables reuse and specialization.

Chapter 5 reports on the empirical evaluation of MODEST. It describes the study design, the subject systems considered, and the metrics used for assessment. It then presents the results obtained when generating tests for both open-source projects and Berger-Levrault's proprietary systems. The evaluation assesses the feasibility of the approach and the extent to which the generated tests satisfy the requirements set out in the introduction.

Chapter 6 concludes the thesis by summarizing the main contributions and outlining directions for future work.

1.5 List of Publications

International conference

Gabriel Darbord, Nicolas Anquetil, Benoit Verhaeghe, Anne Etien. "A Multi-Language Tool for Generating Unit Tests from Execution Traces". In: *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Institute of Electrical and Electronics Engineers, 2025. DOI: 10.1109/SANER64311.2025.00030. URL: https://hal.science/hal-04841805

International journal

Gabriel Darbord, Benoit Verhaeghe, Anne Etien, Nicolas Anquetil, Anas Shatnawi, Abderrahmane Seriai, Mustapha Derras. "Migrating the Communication Protocol of Client-Server Applications". In: *IEEE Software* (2023). DOI: 10.1109/MS.2023.3263019. URL: https://hal.science/hal-04050310

Workshops

Gabriel Darbord, Fabio Vandewaeter, Anne Etien, Nicolas Anquetil, Benoit Verhaeghe. "Modest-Pharo: Unit Test Generation for Pharo Based on Traces and Metamodels". In: *IWST 2024: International Workshop on Smalltalk Technologies*. July 2024. URL: https://hal.science/hal-04622256

Gabriel Darbord, Anne Etien, Nicolas Anquetil, Benoit Verhaeghe, Mustapha Derras. "A Unit Test Metamodel for Test Generation". In: *Proceedings of the 2023 International Workshop on Smalltalk Technologies*. CEUR Workshop Proceedings, Aug. 2023. URL: https://hal.science/hal-04219649

Under peer review

Anas Shatnawi, Bachar Rima, Zakarea Alshara, Gabriel Darbord, Abdelhak-Djamel Seriai, Christophe Bortolaso. "Telemetry of Legacy Web Applications: An Industrial Case Study". working paper or preprint. Nov. 2023. DOI: 10.36227/techrxiv.24449092. URL: https://hal.science/hal-04344518

Other publication

Domenico Cipriani, Gabriel Darbord. "Pharo Sound Design: Adding Auditory Feedback to a Live Programming Environment". In: *Proceedings of the 32nd Journées d'Informatique Musicale*. GRAME and Inria. Lyon, France, June 2025. URL: https://hal.science/hal-05102350

Award

At the 2024 International Workshop on Smalltalk Technologies (IWST), we received the best paper award for our work describing our test generation approach applied to Pharo [Darbord 2024].

Chapter 2

State of the Art

Contents

This chapter presents the state of the art in automated unit test generation, focusing on techniques relevant to the context of this thesis. The previous chapter introduced the industrial problem that we aim to solve, which is generating unit tests in situations when no tests or specifications exist. These tests must reflect real system behavior, support regression testing, and be readable and maintainable for developers.

In Section 2.1, we review the main families of unit test generation techniques and whether they fit our industrial context. This leads us to identify test carving as the most suitable technique. In Section 2.2, we define criteria to compare existing carving approaches and assess the readability of the tests they generate. Finally, in Section 2.3, we review existing metamodels for representing tests, to situate our model-based approach and highlight gaps that our metamodels (introduced in Chapter 3) are designed to address.

2.1 Unit Test Generation Techniques

Selecting a test generation technique for our industrial setting requires making explicit the boundary conditions under which a technique is applicable. We

therefore begin by stating the requirements that any candidate technique must satisfy in our industrial context. Building on these requirements, this section then reviews major families of unit test generation. We use *technique* to mean a family of methods that share core principles (*e.g.*, random testing). This review focuses on peer-reviewed research and presents a representative selection of techniques and approaches. For each technique, we outline its principles and representative approaches, then assess applicability against our requirements and give a brief verdict.

2.1.1 Technique Selection Requirements

The following requirements are derived from the problem statement in Section 1.2. If a technique does not fulfill all of them, it is unsuitable in our setting regardless of other merits.

No existing tests: The technique must synthesize tests from scratch. Approaches that rely on a non-trivial pre-existing test suite are excluded.

No formal specifications: The technique must operate without contracts, models, or properties authored in advance.

Regression oracle: The technique must provide an oracle that checks for behavioral differences across versions of the same system, rather than correctness against an external specification. This ensures that generated tests preserve previously observed behavior and expose unintended changes.

Reflects observed executions: Generated tests should be grounded in actual program executions (production or representative) rather than synthetic scenarios designed only to maximize coverage.

A wide range of techniques have been proposed in the literature for unit test generation. We focus on a representative selection: specification-based testing, random testing, search-based testing, symbolic and concolic testing, test amplification, LLM-based test generation, and test carving. For each technique, multiple approaches have been developed and implemented in concrete tools. The remainder of this section analyzes each technique in turn, discussing its principles, representative approaches, and overall suitability with respect to the requirements. Our requirements are language-agnostic, but most of the surveyed works focus on Java-based tools. This reflects the prominence of Java as a research and industrial target rather than a restriction of our analysis.

2.1.2 Specification-Based Testing

Specification-based testing derives test cases from formal software models, such as algebraic and model-oriented specifications. Test cases can also be derived from design-by-contract annotations embedded in code, including preconditions, postconditions, and invariants. This technique generates tests that verify whether the implementation conforms to its formal description.

Early work in this area has roots in VDM and Z specification languages, which capture system behavior as mathematical models, such as sets and relations [Jones 1986,Spivey 1989]. In these model-based approaches, test cases are synthesized by traversing the model and converting constraints into executable tests [Zander 2011]. In the Java ecosystem, the Java Modeling Language (JML) is a widely used design-by-contract specification language [Leavens 2004b,Leavens 2004a]. One example is JARTEGE, which combines random testing with JML oracles to validate contracts during execution [Oriat 2005]. Another is JMLUNITNG, which generates TestNG[1] tests from JML annotations using either symbolic execution or static analysis to target boundary conditions [Zimmerman 2010].

A subset of specification-based testing is property-based testing, where developers define executable properties that implementations must satisfy. It automatically generates random test inputs to verify each property and shrinks failing cases to minimal counterexamples, effectively treating properties as test oracles. QUICKCHECK for Haskell pioneered this approach [Claessen 2000]. This approach has since been adopted in several programming ecosystems through tools such as SCALACHECK[2], HYPOTHESIS[3] (Python), and FAST-CHECK[4] (JavaScript).

Although specification-based approaches produce strong conformance oracles and can systematically cover specified behaviors, they require formal specifications that describe the expected behavior. In our industrial setting, no such specifications are available, and writing them would involve significant manual effort and domain expertise. As a result, these techniques are inapplicable in our context.

2.1.3 Random Testing

Random testing involves generating sequences of method calls and data inputs by randomly selecting available APIs and values. It is a black-box approach that relies on trial and error to explore program behavior. This technique can quickly produce many test cases and often detects crashes or assertion

[1] Java testing framework: `https://testng.org/`

[2] `https://scalacheck.org/`

[3] `https://hypothesis.works/`

[4] `https://fast-check.dev/`

violations with minimal upfront effort. Specialized forms include fuzz testing, which mutates inputs to target program robustness, and monkey testing, which randomly exercises graphical user interfaces.

A prominent Java tool is RANDOOP [Pacheco 2007], which performs feedback-directed random test generation by constructing method sequences and inferring assertions from runtime behavior. Another example is JARTEGE [Oriat 2005], previously introduced in the context of specification-based testing, which combines JML-based oracles with random input generation to validate contract conformance. Adaptive Random Testing (ART) improves upon naive random generation by distributing inputs more uniformly across the input domain, which reduces redundancy and increases fault detection rates [Chen 2010].

Random testing does not require existing tests or formal specifications, which satisfies two of our requirements. However, it creates artificial execution scenarios that rarely reflect real-world usage patterns. Some tools, such as RANDOOP, use explicit oracles inferred from observed outputs during generation, asserting that the same outputs are produced on subsequent executions. The system is assumed to be correct, and the resulting outputs are considered as expected behavior. However, it provides no external guarantee that the observed behavior is semantically correct because they are not derived from specifications or production behavior. As a result, the generated assertions may preserve unintended behavior and fail to detect regressions.

Other forms of random testing, such as fuzz testing, operate without an explicit oracle and detect failures through crashes or exceptions. These techniques prioritize fault detection over behavior preservation. Overall, random testing lacks reliable regression oracles and cannot guarantee that generated tests reflect real-world behavior.

2.1.4 Search-Based Testing

Search-based software testing (SBST) formulates test generation as an optimization problem. SBST uses techniques such as genetic algorithms and evolutionary strategies to automatically generate input sequences and method calls that satisfy objectives such as branch coverage, fault detection, and mutation score.

The best-known SBST tool is EVOSUITE [Fraser 2011a], which uses genetic algorithms to generate JUnit test suites for Java programs. It optimizes for metrics such as branch coverage and infers assertions from observed behavior. EVOSUITE has also served as a foundation for numerous extensions. One example is μTEST, which augments EVOSUITE with mutation-based fitness functions to guide test generation toward maximizing the mutation score [Fraser 2012]. Beyond Java, notable SBST tools include AUSTIN [Lakhotia 2010], a search-based testing tool for C programs evaluated on industrial automotive systems,

and SAPIENZ [Mao 2016], which applies multi-objective SBST to Android applications and has been successfully deployed at Facebook.

Search-based testing does not require existing tests or specifications, satisfying two of our requirements. Like random testing, many SBST tools infer oracles by observing the outputs of the system under test during generation. In particular, EVOSUITE captures return values and side effects from current executions and uses them to generate assertions. While similar in principle to RANDOOP [Pacheco 2007], EVOSUITE's oracle generation is often more structured, incorporating heuristics for detecting state changes and covering a broader range of output types. These oracles are regression-style: they assume the current behavior is correct, but they are grounded in synthetic, coverage-driven executions rather than in production or representative runs. As a result, they may preserve incidental behavior and do not guarantee that tests reflect real-world usage. Additionally, SBST techniques generate inputs to satisfy structural coverage objectives, which may lead to unrealistic program paths not representative of actual use. Therefore, search-based testing does not satisfy our requirements regarding regression validation and representativeness of real-world behavior.

2.1.5 Symbolic and Concolic Testing

Symbolic and concolic testing are systematic techniques for automatic test generation. *Symbolic execution* is a static program analysis technique that treats inputs as symbolic variables rather than concrete values. As the symbolic interpreter traverses execution paths, it accumulates *path conditions* that describe the constraints under which each path is taken. Solving these constraints with a satisfiability solver yields concrete inputs that drive execution along unexplored paths.

Tools such as EXE [Cadar 2006] and its successor KLEE [Cadar 2008] apply symbolic execution to programs compiled to LLVM[5] bitcode to enable fine-grained path exploration. Although grounded in the classical model of symbolic execution, these tools rely on concretization strategies and simplified memory models to remain scalable in practice.

Concolic execution is a hybrid technique that combines symbolic reasoning with concrete execution, hence its name as a portmanteau of concrete and symbolic. While executing a program on concrete inputs, a concolic engine simultaneously tracks symbolic expressions and gathers path conditions. This mitigates issues that hinder pure symbolic execution such as path explosion, modeling limitations, and the generation of infeasible paths.

Concolic testing was popularized by DART (Directed Automated Random

[5]LLVM is a compiler infrastructure project providing an intermediate representation (IR) and toolchain for program analysis and transformation, see `https://llvm.org`.

Testing) [Godefroid 2005], which explored program paths by negating previously taken branches and solving the resulting constraints. CUTE [Sen 2005] built on this idea, focusing on unit testing for C programs and improving support for pointers and heap structures. Later, SAGE [Godefroid 2012] scaled concolic execution to x86 binaries and applied it to large industrial codebases. SAGE was particularly successful in uncovering deep bugs in file parsers and is credited with detecting roughly one-third of all file-format vulnerabilities during the development of Windows 7.

Symbolic and concolic testing satisfy two of our requirements: they require neither pre-existing test cases nor formal specifications. They can automatically explore execution paths by solving symbolic constraints and generating new inputs. However, the generated inputs are synthetic and not derived from real usage. The resulting test oracles are implicit, typically limited to detecting runtime errors such as assertion failures or crashes. In addition, these techniques are often constrained by the scalability of constraint solvers, which restricts their applicability to large or dynamically typed systems. Consequently, unless extended with explicit specifications or runtime observations, these techniques do not fulfill our requirement for tests that reflect real-world behavior and detect behavioral regression across versions.

2.1.6 LLM-Based Test Generation

Large Language Model (LLM) based test generation is a recent and rapidly growing technique for synthesizing test code from source code, comments, or other artifacts. These methods use pretrained sequence models to generate test cases based on natural language or code prompts that describe the method under test and its context. Unlike traditional approaches that aim to maximize code coverage through metrics such as branch or statement coverage, LLM-based test generation leverages statistical learning over massive code corpora to infer plausible test structures.

Empirical studies have investigated the test generation capabilities of general-purpose LLMs. Siddiq et al. [Siddiq 2024] evaluated Codex, GPT-3.5-Turbo, and StarCoder for JUnit test generation, comparing their output against EvoSuite [Fraser 2011a] on benchmarks and real-world Java projects. They found that while LLMs produced readable and mostly syntactically correct tests, they sometimes failed to compile, and often lacked semantic correctness and behavioral coverage. In addition, the generated tests frequently exhibited poor quality in terms of test smells. Yang et al. [Yang 2024] evaluated five open-source LLMs from the CodeLlama and DeepSeek-Coder families, as well as GPT-4, and compared them with EvoSuite. Their study considered several dimensions: prompt design strategies, the use of in-context learning, and the ability to detect defects. Their findings confirm that LLMs can produce plausible test code but still underperform traditional tools in terms

of coverage and fault detection. Key limitations include hallucinated code, invalid assertions, poor edge-case handling, and ineffective input construction for triggering defects. They also observed that common prompting techniques, such as in-context learning, do not reliably transfer from other tasks to unit test generation.

To address some of the shortcomings of raw LLM-based test generation, frameworks such as ChatUniTest [Chen 2024] embed language models within structured test generation pipelines. ChatUniTest augments prompting with a feedback-driven loop: tests are validated through execution, repaired as needed, and refined using execution traces and context adaptation. This structure mitigates common failure modes such as syntactic errors and misaligned oracles, improving the overall reliability and utility of the generated tests.

LLM-based test generation satisfies two of our four requirements: it does not require existing test suites or formal specifications. However, the generated tests are synthetic and typically lack runtime grounding or reliable oracles. Raw generations often hallucinate code, overlook behavioral nuances, or fail to exercise meaningful program paths. Their quality varies significantly depending on prompt phrasing, model configuration, and context length, making them sensitive to seemingly minor changes and difficult to systematically tune. Furthermore, generated tests are rarely grounded in actual execution traces, which limits their ability to reflect observed behaviors. Reproducibility also remains a concern: model updates and non-deterministic sampling can yield inconsistent results across time or runs.

Framework-assisted approaches such as ChatUniTest mitigate some of these issues, especially syntax and structure, and can introduce partial runtime grounding through test execution and validation steps. However, these frameworks typically lack integration with production data, and their generated tests remain synthetic, shaped by inference rather than real-world inputs or intended specifications. As a result, LLM-based test generation falls short of our requirements for regression validation and real-world representativeness.

2.1.7 Test Amplification

Test amplification techniques take an existing test suite as input and automatically enhance it, either by generating new test cases, modifying existing ones, or strengthening their oracles. The goal is to improve measurable properties such as code coverage, fault detection ability, or assertion strength, while preserving the original test intent and minimizing developer effort.

Test amplification has evolved through a variety of mechanisms, including input diversification, assertion strengthening, and structural variation of test code [Danglot 2019a]. A notable early contribution is Eclat [Pacheco 2005], which introduced strategies for generating and classifying inputs to expose behaviors associated with contract violations. While not explicitly branded as

test amplification, ECLAT laid groundwork for input diversification strategies central to later amplifiers. Fraser and Zeller proposed generalizing concrete unit tests into parameterized tests to enable broader input coverage from compact suites [Fraser 2011b]. Zhang and Elbaum formalized the term *test amplification* to describe input and oracle mutations that enhance behavioral exploration [Zhang 2012, Zhang 2014]. Later, Zhang et al. introduced *isomorphic regression testing* [Zhang 2016], which avoids redundant tests by identifying semantically equivalent input behaviors, and reusing them across versions to detect regressions more efficiently. Baudry et al. proposed DSPOT [Baudry 2015], a test amplification tool for Java that enhances test suites by transforming inputs and strengthening assertions. It applies input transformations such as literal perturbation and object mutation, and adds assertions by observing program state or inferring invariants. Originally designed to assess computational diversity, DSPOT was later shown to significantly improve both coverage and fault detection while preserving the original test's intent [Danglot 2019b].

Test amplification enhances regression capability by reinforcing or expanding the behavior captured by existing tests. Because it operates on concrete test executions, it reflects real-world behavior already exercised by the original suite. By definition, however, test amplification requires the existence of a non-trivial test suite as input, as it does not synthesize tests from scratch. In our industrial setting, such suites are often missing or insufficient. Therefore, while test amplification aligns well with our requirements on regression validation and behavioral realism, it cannot serve as a standalone approach in contexts where no prior tests exist.

2.1.8 Test Carving

Test carving is a technique for generating unit tests from actual executions of a software system. It captures method calls and associated data during program execution, including the method signature, the state of the receiver object, the arguments passed to the method, and the method's result. These recorded method calls are then replayed in a unit test to check that the observed outcome recurs. To re-establish the state before invoking the method under test, two strategies exist: *state-based carving*, which restores a snapshot of the state and then calls the method, and *action-based carving*, which replays the sequence of calls that originally produced the state. The latter is particularly useful when full serialization of framework objects is impractical, such as in mobile or GUI environments.

The seminal work of Elbaum et al. introduced the notion of *differential unit tests* (DUT) [Elbaum 2006], which they defined as unit tests carved from system executions and replayed on later versions of the system. The term *differential* highlights the goal of detecting differences in behavior across versions of a software unit. Elbaum et al. emphasized that DUT combine the precision of

unit tests with the expressiveness of system tests (*i.e.*, functional end-to-end scenarios), enabling automated regression detection based on realistic use cases. Since then, various techniques building on this idea have been developed under different names, such as replay tests or trace-based unit test generation. We discuss other test carving approaches in the next section (Section 2.2).

Test carving fully meets our industrial requirements. It does not require existing tests or specifications, since it derives test cases directly from live executions. It is well-suited for regression testing, as it ensures that future versions of the software produce the same results for previously observed behaviors. And because it captures real executions, the resulting tests are representative of actual system usage.

Summary

Table 2.1 synthesizes how the reviewed techniques meet our requirements. It highlights that only test carving simultaneously avoids dependence on existing tests or specifications, provides a regression oracle, and reflects observed executions. Based on this analysis, we adopt test carving as the foundation of the approach developed in this thesis.

Table 2.1: Comparison of unit test generation techniques with respect to requirements

Test Generation Technique	Without Existing Tests	Without Specification	Regression Oracle	Reflects Observed Executions
Specification-Based	Yes	No	Yes	No
Random	Yes	Yes	No	No
Search-Based	Yes	Yes	No	No
Symbolic & Concolic	Yes	Yes	No	No
LLM-Based	Yes[†]	Yes[‡]	No	No
Amplification	No	Yes	Yes	No[*]
Carving	Yes	Yes	Yes	Yes

[†] Some techniques use existing tests as context.
[‡] Some techniques use specifications as context.
[*] Unless existing tests already cover real executions.

2.2 Comparison of Test Carving Approaches

Having identified test carving as the most appropriate technique for our context, we now examine the diversity of existing carving approaches. We use *approach* to mean a specific instantiation of a technique proposed in a paper, and *tool* to mean its concrete implementation. We begin by defining the criteria used to compare approaches, and then apply them to assess existing work, including the readability of the generated tests. This analysis highlights common patterns

and limitations that motivate the contributions developed in the following chapters.

2.2.1 Comparison Criteria

Test carving approaches diverge in what they verify, which ecosystems they target, and how they obtain the inputs and state needed to run tests. We use the following criteria to make these differences explicit:

Oracle type: Defines what the test considers correct. In practice, oracles are realized in code through mechanisms such as assertions over return values, checks on object state, interaction verifications, or expectations of specific exceptions. Some approaches rely on smoke tests, where correctness is defined by the absence of failures.

Ecosystem support: Specifies the programming languages and testing frameworks for which the approach can generate tests, such as Java with JUnit[6].

Value construction: Describes how a test obtains its inputs and required state. When comparing approaches, this criterion can take different forms, such as reloading serialized data, generating source code that builds values, or replacing dependencies with test doubles (*i.e.*, stand-ins such as mocks or stubs).

Readability is a distinct concern that we do not reduce to a single criterion. Following Oliveira et al. [Oliveira 2023], we distinguish readability from related notions such as legibility, surface-level aspects like formatting, and understandability, the ease of extracting useful information for a task. In this thesis, readability refers to the structural and semantic characteristics that make generated tests easier to understand, diagnose when failing, and maintain. Although readability can be quantified through metrics [Daka 2015], such measures remain partial proxies and do not fully capture how developers experience generated code. We therefore operationalize readability qualitatively along four axes:

Naming: Identifiers for methods and variables should communicate intent and domain role without relying on comments.

Absence of smells: Smells are recurring structural symptoms in code that signal potential maintenance or comprehension problems, without necessarily being bugs. They are widely recognized as indicators of degraded code quality, both in general software [Fowler 1999] and in tests specifically [van Deursen 2001]. We include the absence of smells as a readability

[6] `https://junit.org`

axis because smells make tests harder to read, understand, and maintain. Generated tests should therefore avoid them. For example, a common code smell in generated code is duplication, where the same fragments are emitted repeatedly instead of being factored out. An example of a test smell is assertion roulette, where many assertions are included in the same test without descriptive messages, making it difficult to diagnose which one failed.

Value reconstruction readability: The mechanism used to obtain inputs and state at runtime should be explicit, local, and idiomatic. Explicit means that the construction of a value is visible in the code without requiring hidden knowledge of specific APIs. Local means that values are created within the test or its helpers, not fetched from external sources such as files or databases. Idiomatic means that the construction follows the conventions of the target language and framework, so that generated code resembles what a developer would naturally write.

Scaffolding: We use the term scaffolding for the auxiliary code and resources that a tool produces around the tests to make them executable. It comes in two forms: (i) invariant harnesses, generic code that is always the same across tests, such as utilities for loading serialized objects or initializing the test environment, and (ii) test-specific helpers, such as factory methods that set up values for a particular test so that its body can remain short and focused, or resource files containing serialized values. The naming and placement of this auxiliary code also affect clarity, since they should make its role in the test immediately apparent.

The remainder of this section applies these criteria to compare existing carving approaches.

2.2.2 Assessment of Approaches

We focus our comparison on a corpus of peer-reviewed approaches that are representative of test carving: DUT [Elbaum 2006], ReCrash [Artzi 2008], DyGen [Thummalapenta 2010], Genthat [Křikava 2018], PANKTI [Tiwari 2022], MicroTestCarver [Deljouyi 2023], ARTISAN [Gambi 2023], RICK [Tiwari 2024], and TestGen [Alshahwan 2024]. These approaches are predominantly state-based, with the exception of ARTISAN which is action-based.

Oracle type

Test carving approaches differ in the oracle they use to detect regressions. A common choice is what we call *default equality*, the testing framework's

built-in equality assertion. In Java/JUnit, for example, this corresponds to `assertEquals(a, b)` which ultimately calls `a.equals(b)`. If the `equals` method is not overridden, the comparison degenerates to reference identity. As a result, default equality can be a weak oracle for complex objects unless a domain-specific implementation or comparator is provided.

DUT [Elbaum 2006] serialize a selected part of the program state after the method under test has executed, such as its return value or certain fields of the receiver object. They then check that this serialized representation is the same across versions. Because this can be overly sensitive to non-breaking changes, such as field reordering, they propose strategies that restrict which parts of the state are compared. TestGen [Alshahwan 2024] follows the same principle, comparing JSON serializations of program state across versions to detect differences. DyGen [Thummalapenta 2010], Genthat [Křikava 2018], PANKTI [Tiwari 2022], MicroTestCarver [Deljouyi 2023] utilize default equality assertions provided by their respective testing frameworks (*e.g.*, `assertEquals` in JUnit). RICK [Tiwari 2024] generates tests that check the returned value and the state of parameters after execution using default equality, and verify interactions with collaborators through mock verifications. Some carving approaches focus on errors rather than verifying correctness. ReCrash [Artzi 2008] generates tests aimed at reproducing exceptions from previous executions. ARTISAN [Gambi 2023] generates smoke tests that verify error-free method execution without using assertions.

Test carving approaches adopt different oracles: serialization equality (DUT, TestGen), default equality assertions (DyGen, Genthat, PANKTI, MicroTestCarver, RICK), exception reproduction (ReCrash), and smoke tests (ARTISAN). Although the limitations of default equality are often recognized, it remains the most common oracle in our corpus.

Ecosystem support

The surveyed tools are generally scoped to a single language or ecosystem. Most target `Java` with `JUnit 4`: DUT, ReCrash, PANKTI, MicroTestCarver, RICK, and ARTISAN. ARTISAN specifically handles `Android`. DyGen generates tests for `C#` on the `.NET` runtime using `NUnit`. Genthat generates tests for the `R` language using `testthat`. TestGen generates tests for `Kotlin` without relying on a testing framework.

Overall, ecosystem support is narrow: most tools are bound to a single language and test framework. This raises the question of how more modular designs could support multiple ecosystems.

Value construction

Approaches differ in the mechanisms they use to reconstruct values observed during execution. Several tools deserialize complete graphs of receivers, parameters, and results recorded during execution. This is the case for PANKTI [Tiwari 2022], RICK [Tiwari 2024], and TestGen [Alshahwan 2024]. RICK additionally replaces external objects with mocks. DUT [Elbaum 2006] and ReCrash [Artzi 2008] also rely on deserialization, but they support strategies that prune the captured state to selected fields or depth-bounded portions of the object graph to reduce capture and replay costs.

DyGen [Thummalapenta 2010] inlines primitives and strings but generalizes other inputs into parameters in parameterized unit tests. These parameters are instantiated during generation by symbolic execution, which synthesizes constructor calls and field initializations up to a bounded depth. This allows the generated tests to cover behaviors beyond those directly observed.

Genthat [Křikava 2018] converts runtime values back into R code literals when possible. For more complex or unsupported structures, it falls back to deserialization using a binary format.

MicroTestCarver [Deljouyi 2023] generates source code to reconstruct observed values rather than reloading them from serialized data. For common types such as primitives, collections, and optionals, it generates dedicated factory methods that instantiate objects directly from their captured fields. When no factory method is available, the tool falls back to more generic strategies, such as using reflection to set fields or relying on simple string-based representations. This approach yields code-based reconstruction that makes object creation explicit in the generated tests.

Finally, ARTISAN [Gambi 2023] adopts action-based replay. It reconstructs state by replaying the sequence of GUI actions that led to the method under test, and substitutes framework objects with test doubles, combining mocks and Robolectric to simulate Android components.

Value construction mechanisms fall into three broad categories: deserialization (DUT, ReCrash, PANKTI, RICK, TestGen), code-based reconstruction (DyGen, Genthat, MicroTestCarver), and action-based replay (ARTISAN). Each category entails different trade-offs in how values are represented in tests, which we discuss when assessing readability in Section 2.2.3.

Synthesis

Table 2.2 synthesizes how the reviewed carving approaches differ in their oracle type, ecosystem support, and their value construction mechanisms.

Table 2.2: Criteria assessment of test carving approaches

Tool & Source	Oracle type	Ecosystem support	Value construction
DUT [Elbaum 2006]	Serialization equality	Java	Deserialization
ReCrash [Artzi 2008]	Exception expected	Java	Deserialization & Primitives
DyGen [Thummalapenta 2010]	Default equality	C# (.NET)	Primitives & Symbolic parameters
Genthat [Křikava 2018]	Default equality	R	Literals & Deserialization
PANKTI [Tiwari 2022]	Default equality	Java	Deserialization
MicroTestCarver [Deljouyi 2023]	Default equality	Java	Code-based
ARTISAN [Gambi 2023]	Smoke	Java (Android)	Action-based & Mocks
rick [Tiwari 2024]	Default equality & Mock interactions	Java	Deserialization & Mocks
TestGen [Alshahwan 2024]	Serialization equality	Kotlin	Deserialization

2.2.3 Assessment of Test Readability

To assess readability, we use the axes defined in Section 2.2.1: naming, absence of smells, value reconstruction readability, and scaffolding. Our assessment relies on information reported in the papers and on examples in their artifacts. When a paper does not address an axis explicitly, we note the lack of evidence and avoid strong claims.

Naming

Most papers do not make naming a central concern. DUT [Elbaum 2006] does not discuss it at all. ReCrash [Artzi 2008], DyGen [Thummalapenta 2010], and Genthat [Křikava 2018] show examples where test names are systematically derived from the invoked method, but without further discussion. PANKTI [Tiwari 2022] and ARTISAN [Gambi 2023] explicitly describe systematic naming templates, again tied to the method under test. RICK [Tiwari 2024] also discusses naming, with test methods encoding both the method under test and the oracle type, and with annotations documenting mocked collaborators. TestGen [Alshahwan 2024] shows systematic templates for test names, though naming is not treated as a major topic. MicroTestCarver [Deljouyi 2023] goes further by explicitly proposing a naming scheme that incorporates method, inputs, and outputs, and by describing conventions for variables and mocks.

Overall, naming is treated narrowly, usually limited to method-derived test identifiers, while names for test classes, fixtures, variables are rarely specified. A notable exception is MicroTestCarver, which proposes explicit conventions spanning method names, variable roles, and mocks. RICK also

surfaces collaborator roles via annotations. This gap suggests that naming at all levels (tests, classes, variables, doubles) remains underexplored, despite its importance for readability and maintenance.

Absence of smells

Several papers identify phenomena that correspond to smells, in particular redundancy. Redundancy refers to the presence of multiple tests that exercise the same behavior, and duplication refers to repeated code fragments within or across tests.

DUT [Elbaum 2006] notes that carving many differential unit tests per method can produce duplication at the suite level, since each test may repeat similar setup unless reduced by filtering. ReCrash [Artzi 2008] generates multiple candidate tests per stack frame, which can include redundant cases, and the authors also note boilerplate duplication in each test. DyGen [Thummalapenta 2010] reports that naïve trace-to-test conversion leads to redundant tests, addressed by a suite reduction phase. Genthat [Křikava 2018] similarly highlights massive redundancy in large-scale extraction, as well as the generation of very large tests, which they mitigate through suite minimization and externalization of data. PANKTI [Tiwari 2022] reports that multiple execution traces of the same method can yield redundant tests and addresses this by collapsing identical traces. MicroTestCarver [Deljouyi 2023] reduces duplication by factoring repeated values into shared fixtures and by structuring tests to avoid large or eager tests, though some redundancy remains when multiple traces yield similar cases. RICK [Tiwari 2024] produces up to three tests per invocation, leading to overlapping setups, and they acknowledge fragility to refactorings such as parameter renaming or call reordering. TestGen [Alshahwan 2024] similarly observes redundancy from numerous observations and applies a selection phase to keep a smaller, representative set of tests; they also discard flaky tests.

By contrast, ARTISAN [Gambi 2023] does not discuss smells or related phenomena directly. The authors note their tests are short on average, which limits verbosity-related issues, but some become larger when the setup involves database state. They mention test suite reduction to avoid redundancy as future work.

Overall, redundancy is the dominant smell-related phenomenon acknowledged in our corpus, and most approaches mitigate it through test suite reduction. Other smells, such as code duplication, verbosity due to very large tests, or fragility to refactorings, appear only sporadically and are seldom addressed. A notable exception is MicroTestCarver, which explicitly aims for understandability and incorporates measures to avoid duplication and eager tests. These observations point to the value of making smell avoidance an explicit design goal when generated tests are intended for developer adoption

and maintenance.

Value reconstruction readability

We assess value reconstruction readability in generated tests along explicitness, locality, and idiomaticity.

When deserialization is used, explicitness depends on the serialized format: encodings such as JSON or XML can often be inspected by developers, although additional metadata may complicate interpretation. Explicitness also presumes some familiarity with the serialization library. Locality depends on where the serialized data is stored: inline strings keep it visible in the test, whereas external files make it less accessible. Idiomaticity is lacking, since developers would normally write code to construct values rather than rely on deserialization. This is the case for DUT [Elbaum 2006], ReCrash [Artzi 2008], PANKTI [Tiwari 2022], RICK [Tiwari 2024], and TestGen [Alshahwan 2024].

By contrast, code-based reconstruction avoids deserialization and produces values directly in source code. This makes construction explicit and local by design, and it can be idiomatic when the generated code follows the conventions of the target language and testing framework. However, idiomaticity deteriorates when reconstruction relies on non-idiomatic mechanisms such as reflection. Additionally, code-based reconstruction is often emitted in a single-assignment style with many fresh variables, which is less natural than developer-written code. The use of test doubles in generated code is idiomatic in unit testing and does not, by itself, harm readability. Inlining primitives and simple literals is explicit, local, and idiomatic by nature. Code-based reconstruction is done by DyGen [Thummalapenta 2010], Genthat [Křikava 2018], and MicroTestCarver [Deljouyi 2023].

Action-based replay, as done by ARTISAN [Gambi 2023], expresses setup as sequences of API calls, which are explicit and local in the test code. Idiomaticity depends on how closely the replayed interactions resemble developer-written setup for the target framework.

Along our axes, deserialization-based approaches are the least explicit, local, and idiomatic. Code-based reconstruction generally fares better provided the generator uses conventional constructors and factories, and avoids reflection. Action-based replay is explicit and local, with idiomaticity depending on how closely the replay mirrors developer practice in the target framework. Taken together, these observations point towards guidelines for reconstructing values in tests: prefer code over deserialization, inline literals, factor shared setup into helpers, and minimize reflection.

Scaffolding

Scaffolding appears in two main forms: invariant harness code shared across all generated tests, and test-specific helpers.

Invariant harnesses are used by several approaches. DUT [Elbaum 2006] introduces context loaders and differencing utilities that are generated once and reused across the suite. ReCrash [Artzi 2008], PANKTI [Tiwari 2022], RICK [Tiwari 2024], and TestGen [Alshahwan 2024] similarly generate deserialization code to reload captured objects in the same way for every test.

Test-specific helpers appear in most tools, in varied forms. PANKTI, Genthat [Křikava 2018], and TestGen generate external resource files that hold serialized values, which are then reloaded by tests. ARTISAN [Gambi 2023] generates Mockito[7] mocks and Robolectric[8] shadow classes, and injects them into the carved tests so that GUI elements and other framework objects behave as they did in the original execution. RICK generates per-test helper methods for mock injection, such as inserting a mocked dependency as an attribute in a receiver or argument. MicroTestCarver [Deljouyi 2023] generates a broader set of helpers, including methods that return prepared objects (such as receivers or arguments), factory methods for common types (*e.g.*, collections, optionals) that take their field values as arguments and return fully initialized instances, and shared fixtures (*e.g.*, `@BeforeEach`) that factor repeated setup so test bodies remain short and focused.

DyGen [Thummalapenta 2010] transforms traces into parameterized unit tests, which generalize recorded executions by replacing concrete values with parameters. The recorded executions are then replayed by invoking these parameterized tests with the captured values, and symbolic execution can supply additional inputs. Since no auxiliary code is generated beyond the tests, DyGen does not produce scaffolding.

Overall, all carving approaches rely on some form of scaffolding. Invariant harnesses add a one-time comprehension cost, as developers must understand shared utilities such as loaders or differencing code, but this cost is amortized across the suite. By contrast, test-specific helpers vary more in their impact: resource files that externalize serialized values obscure what a test is doing, while factory methods and shared fixtures can improve readability by shortening test bodies and reducing duplication. This variety shows the role of scaffolding in shaping how generated tests are read and maintained by developers.

Synthesis

Table 2.3 summarizes whether each reviewed approach explicitly addresses the four readability axes we defined: naming, absence of smells, value reconstruction

[7] `https://site.mockito.org`
[8] `https://robolectric.org`

readability, and scaffolding. A checkmark (✓) indicates that the axis is explicitly discussed in the paper or its artifacts, a cross (×) that it is not mentioned. Only the *absence of smells* axis allows partial mention (∼): some papers discuss related phenomena such as redundancy or duplication without explicitly framing them as test smells.

Table 2.3: Coverage of readability axes

Tool & Source	Naming	Absence of smells	Construction readability	Scaffolding
DUT [Elbaum 2006]	×	∼	×	✓
ReCrash [Artzi 2008]	×	∼	×	✓
DyGen [Thummalapenta 2010]	×	∼	×	×
Genthat [Křikava 2018]	×	∼	✓	✓
PANKTI [Tiwari 2022]	×	∼	✓	✓
MicroTestCarver [Deljouyi 2023]	✓	✓	✓	✓
ARTISAN [Gambi 2023]	×	✓	✓	✓
rick [Tiwari 2024]	×	∼	×	✓
TestGen [Alshahwan 2024]	×	∼	×	✓

The table highlights that readability receives uneven attention across approaches. Naming is almost never treated as a concern, with most tools defaulting to mechanical identifiers. Smells are usually acknowledged only in the form of redundancy, which many tools mitigate through reduction or filtering, while broader smell categories are seldom discussed. Value reconstruction readability is rarely addressed directly, even though the underlying mechanisms (*e.g.*, deserialization versus code-based reconstruction) have a strong impact on how tests appear to developers. Scaffolding is universally present, but approaches differ in whether they describe its impact on readability.

In our corpus, readability often appears as a secondary concern, with most approaches focusing on automation rather than developer-oriented clarity. MicroTestCarver [Deljouyi 2023] stands out by giving stronger emphasis to developer-facing quality, explicitly aiming for understandability through conventions, fixtures, and helpers. In a user study with 20 participants, their generated tests were consistently rated as more understandable than those of EvoSuite [Fraser 2011a], and even slightly above manually written tests on average. This suggests that greater attention to readability can improve the usability and maintainability of generated tests for developers.

2.3 Metamodels for Test Generation

Metamodels provide a structured way to represent the elements and relationships involved in a particular domain. In the context of software testing, they can serve as an abstraction to capture details of test cases, their structure, and their relationship to the system under test. Relatively few research efforts have focused specifically on metamodels dedicated to representing unit tests or supporting automated test generation.

Pires et al. propose an approach that relies on the Knowledge Discovery Metamodel (KDM) as an intermediate representation of software systems and their environments to generate unit tests compatible with xUnit frameworks [Pires 2018]. While this demonstrates the utility of metamodels in connecting software artifacts, their approach focuses primarily on integrating with existing code analysis tools and is tightly coupled to Java environments.

A metamodel proposed by Lévesque adds test elements directly to the programming language metamodel [Lévesque 2009], enabling tests to be incorporated alongside production classes. This differs from the more conventional practice of writing tests separately.

Khorram explores a testing framework for executable domain-specific languages (xDSLs) [Khorram 2022]. It provides metamodel-based facilities for defining and executing test cases, measuring test quality through coverage and mutation analysis, and supporting test amplification and debugging. While it demonstrates the potential of metamodel infrastructures for testing, its scope is restricted to DSLs rather than general-purpose programming languages.

Overall, the few metamodels that have been proposed for test generation are tied to specific ecosystems, either by embedding test elements directly in programming-language models or by relying on intermediate representations limited to a given stack. As noted earlier, most carving tools are similarly restricted to a single language and framework. Metamodels offer a way to overcome this limitation: by abstracting test structure and runtime data independently of syntax, they can support more modular designs that extend across ecosystems. This perspective motivates the metamodels introduced in Chapter 3, which aim to capture tests and values in a language-agnostic manner.

2.4 Conclusion

In this chapter, we reviewed major unit test generation techniques against the requirements introduced in Chapter 1: no existing tests, no formal specifications, the presence of a regression oracle, and reflection of observed executions. We found that specification-based, random-based, search-based, symbolic/-concolic execution-based, LLM-based test generation, and test amplification

do not meet these requirements. In contrast, test carving meets all four: it operates without pre-existing tests or specifications, and provides a regression oracle grounded in observed behavior. We therefore adopt test carving as the foundation for our approach.

When comparing carving approaches, we considered oracle type, ecosystem support, value construction, and readability. Many approaches rely on *default equality* or smoke-style pass conditions, which are fragile for complex objects where reference identity or missing `equals` implementations undermine the intent of the test. Ecosystem support is usually scoped to a single language-framework pair, pointing to an opportunity for more modular, cross-ecosystem designs. For value construction, deserialization is common but produces opaque and non-idiomatic tests; code-based reconstruction is more explicit and closer to what developers would naturally write. Readability also depends on naming, the avoidance of redundancy and other smells, and the use of scaffolding to keep test methods short and understandable. Our review is based on peer-reviewed work, mostly focused on Java due to its prominence in both research and industry. Assessments rely on the evidence reported in the original papers. When details were missing, we adopted a conservative interpretation and did not assume additional capabilities. These observations motivate the contributions that follow: metamodels and generation mechanisms that support structural oracles, idiomatic code-based reconstruction, modular ecosystem support, and developer-oriented readability.

Chapter 3

Metamodels for Test Generation

Contents

Automatic test generation requires an accurate understanding of both the structure and the behavior of the software under test. However, software systems are implemented in various programming languages and frameworks, each with their own idioms, semantics, and tools. This heterogeneity poses a challenge to approaches that aim to be general and reusable across projects and ecosystems. To address this, we rely on metamodels for language- and framework-independent test generation.

In Section 3.1, we discuss the role of metamodels in bridging trace-based execution data and test generation. In Section 3.2, we present the metamodeling infrastructure provided by the Moose platform and the Famix family of models. Finally, in Section 3.3, we describe the two metamodels developed in this thesis: the Unit Test metamodel, which represents the structure of executable tests, and the Value metamodel, which reifies runtime data.

This chapter builds on our earlier work published at the 2023 International Workshop on Smalltalk Technologies [Darbord 2023].

3.1 Role of Metamodels in Test Carving

The process of generating unit tests from observed executions requires reasoning about both dynamic and static information: the runtime values seen during execution and the structural context (*e.g.*, methods, classes) that used or produced them. Modeling lets us capture both kinds of information as first-class citizens.

Metamodels provide uniform, language-agnostic representations for code entities and for the observed values they manipulate. This uniformity en-

ables systematic queries and transformations without committing to a specific programming language or testing framework.

We use metamodels in our trace-to-test pipeline to (1) import execution data into Value models; (2) link values to code entities (*e.g.*, types, fields) in the Code model; (3) derive Unit Test models from the Code and Value models; and (4) generate executable test code from the Unit Test and Value models. The pipeline is extensible (new ecosystems can plug in at steps 2 and 4) and deterministic (the same input models produce the same tests).

To implement this strategy, we rely on existing infrastructure for constructing and manipulating software models. Rather than developing a modeling environment from scratch, we build on tools specifically designed for software analysis and metamodeling.

3.2 Modeling Infrastructure: Moose and Famix

The modeling infrastructure used in this work combines the Moose[1] platform and the Famix family of metamodels. Moose provides a flexible environment for model-driven software analysis, while Famix defines a set of extensible, language-independent abstractions for representing software systems. Together, they form the foundation of our test generation process.

A metamodel defines the structure of a model, much like a class defines the structure of its instances. In software engineering, a metamodel describes the concepts and relationships of a domain in a formalized, machine-processable way. By decoupling from language syntax, this abstraction enables tools to analyze, transform, and generate artifacts in a uniform manner across different programming languages or frameworks. In the context of test generation, metamodels provide a way to represent code elements, runtime values, and test structures as first-class, manipulable entities. By defining the types of entities that can appear in a model, as well as their relationships, metamodels support the construction of tools that are not tied to any particular language. For instance, a test generation tool can rely on the model to identify which methods exist, how they are related, or what their signatures are, whether the source language is Java, Pharo, or another.

We rely on Moose, a software analysis platform that supports the construction, extension, and manipulation of models [Anquetil 2020]. Moose provides a general-purpose framework for defining metamodels and includes built-in mechanisms for importing, querying, and transforming models. Its main metamodeling foundation is the Famix family of models. Famix defines a core set of language-agnostic entities, such as structural units, behavioral units, or relationships, which can be reused and extended to fit the specific characteristics of a given language. This extensibility is made possible by the

[1] `https://modularmoose.org`

use of traits [Schärli 2003], a mechanism that allows behavioral and structural aspects of entities to be defined modularly and reused across multiple models.

Each trait captures a specific aspect of a software concept. For example, the concept of a *class* is described by a trait that combines several behavioral aspects, including the ability to define methods, declare attributes, receive invocations, or participate in inheritance relationships. This trait can be reused in different language-specific models: both the Java model and the Pharo model can define their own class entities by reusing this same trait. As a result, even though Java and Pharo differ in syntax and semantics, their class representations share a common structure and can be analyzed using the same tools.

An important feature of Famix is the ability to extend existing metamodels with new entities and relationships. This makes it possible to capture different concerns, such as source code structure, runtime behavior, or test organization, in separate metamodels while still integrating them into a single coherent model. A key aspect of this extensibility is that relationships are bidirectional: when a new entity is linked to an existing one, Famix automatically adds the corresponding inverse relation. For example, a Unit Test metamodel may define an entity that represents a test class and link it to the corresponding production class in the Code metamodel. This allows us to navigate from tests to the code they exercise and back, and to analyze or generate artifacts that integrate both domains without duplicating or modifying the original Code model.

3.3 Defining Metamodels for Test Generation

The approach developed during this thesis involves the use of metamodels to facilitate the representation and generation of unit tests. Three metamodels are used: the Unit Test metamodel, which represents the structure of unit tests (shown in blue); the Code metamodel (Famix), which represents the codebase (shown in green); and the Value metamodel, which specifies the values used to test the codebase (shown in orange). This section describes the Unit Test and Value metamodels, their entities, and how they interact to produce unit tests. The Code metamodel is an important component of our approach, but it is not described in detail here because it is already established within the Moose platform and has been covered in previous work [Anquetil 2020].

3.3.1 Unit Test Metamodel

The *Unit Test metamodel*, shown in Figure 3.1, is a structured representation of the components that make up unit tests in object-oriented programming. It provides a way to represent, define, and generate unit tests by specifying their

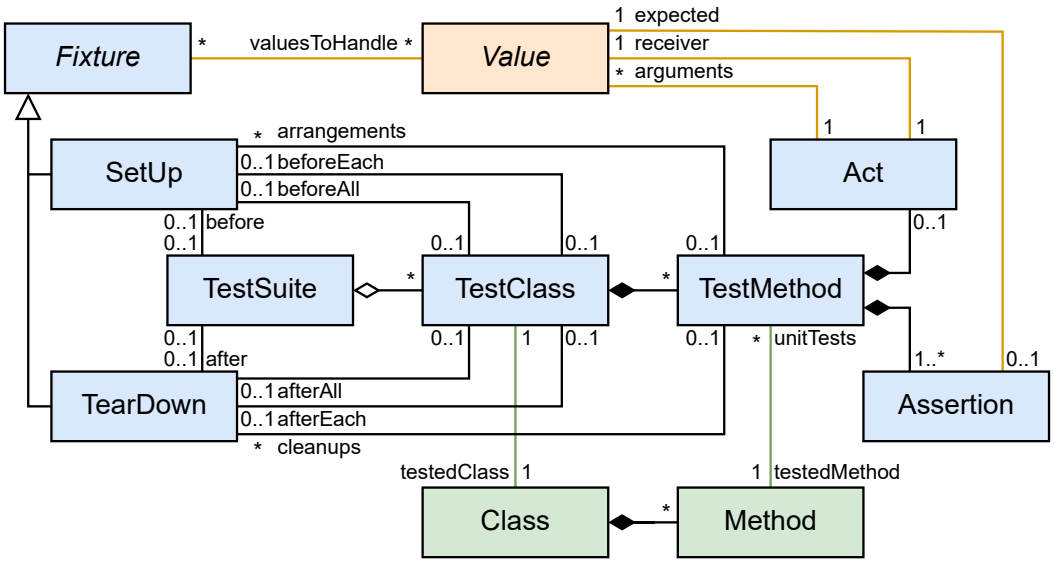

Figure 3.1: Class diagram of the Unit Test metamodel

structure, as well as their inputs, expected outputs, and any necessary setup or teardown fixtures.

Our metamodel is built around the *Arrange Act Assert* (AAA) pattern [Beck 2002], a widely used approach to structuring unit tests. The *Arrange* phase involves setting up the necessary preconditions for the test, usually initializing the receiver, arguments and expected values. The *Act* phase involves executing the method under test, using the values defined in the *Arrange* phase if any. Finally, the *Assert* phase verifies that the results of the *Act* phase are correct, typically by comparing them to expected values.

A `TestSuite` is an entity that groups related `TestClasses` together. Organizing tests into suites provides a logical structure, making it easier to understand what is being tested and why. Suites can also manage test execution, such as running all tests related to a specific feature or component.

The `TestClass` entity represents a test class dedicated to testing a specific production class in the codebase, reflecting the principle that unit tests target a single unit of code. It is thus associated with exactly one `Class` entity, which represents the actual class being tested. Organizing tests for a class into a separate test class allows developers to easily manage and maintain their unit tests. Typically, the name of the test class is derived by appending the suffix "Test" to the name of the class being tested. For example, a class named `User` would typically have its tests in a class named `UserTest`. However, this practice may vary depending on the programming language and project-specific conventions.

A `TestMethod` is a method defined within a `TestClass` that verifies a specific behavior or property of the system under test. It is responsible for

exercising the code and asserting that it behaves as expected under certain conditions. Each test method corresponds to a *unit test*, also referred to as a *test case*, and targets an isolated unit of code with a particular usage pattern. A `TestMethod` is associated with a specific `Method` entity that represents the method it tests in the production code. Conversely, a single `Method` can be associated with multiple `TestMethods` because it can be tested in different contexts, with varying inputs, and under different preconditions.

A `Fixture` ensures that the system under test is executed in a consistent and isolated environment. A fixture is a set of objects and data used to configure the system before running tests and to clean up the environment afterward. Its subclasses, `SetUp` and `TearDown`, can be associated with three different entities: `TestClass`, `TestMethod`, and `TestSuite`. When associated with a `TestClass`, a `SetUp` method is executed before the test methods, establishing a shared context reused across multiple tests. When associated with a `TestMethod`, the `SetUp` prepares the system specifically for that test case. This corresponds to the *Arrange* phase of the AAA pattern and is typically written inline in the test method itself. It includes the initialization of objects, particularly the receiver and arguments, that will later be used in the *Act* phase. When associated with a `TestSuite`, the `SetUp` corresponds to a method executed before all test classes in the suite. The `TearDown` entity mirrors this behavior: it runs after the corresponding tests and cleans up any resources created during the *Arrange* or *Act* phases.

The `Act` entity represents the *Act* phase of the AAA pattern. This phase is responsible for executing the code under test and producing the actual result. In the philosophy of unit testing, each test method should verify one and only one behavior of the system under test. Therefore, a test method should have at most one `Act` to exercise the code under test and produce the actual result for that specific behavior. This ensures that each test method focuses on verifying a single behavior, avoiding the *Eager Test* smell [Meszaros 2007] where multiple methods are exercised in the same test, which hampers the process of identifying and fixing issues revealed by failing tests.

An `Assertion` is a fundamental concept in unit testing. It defines a condition that must be true for the test to pass, confirming that the behavior under test produces the expected outcome. Assertions typically compare expected and actual results, often using return values. They can also verify that a specific exception is thrown, which is important for verifying error-handling behavior and ensuring the system reacts appropriately to unexpected conditions.

Finally, the `Value` entity is used to represent runtime data in unit tests. The relationship between the `Value` and `Assertion` entities represents the expected value of the assertion. When a `Value` is associated with an `Act`, it represents the arguments that will be passed to the method under test. When associated with a `Fixture`, it represents the values that will be set up or torn

down. These relationships bind dynamic data to the corresponding structural elements of the test. The next section introduces the Value metamodel in detail, describing how such values are modeled and how they contribute to the test generation process.

3.3.2 Value Metamodel

The *Value metamodel*, shown in Figure 3.2, provides a representation of runtime data in object-oriented programming languages. It distinguishes between categories of types, such as classes, enumerations, and collections, so that instances of these types can be modeled consistently. Runtime data forms a graph that the metamodel can capture faithfully, including cycles and reference sharing (also known as aliasing, *i.e.*, two or more references pointing to the same object). The metamodel also enables code generation to reconstruct the modeled data, which is detailed in the next chapter, Chapter 4.

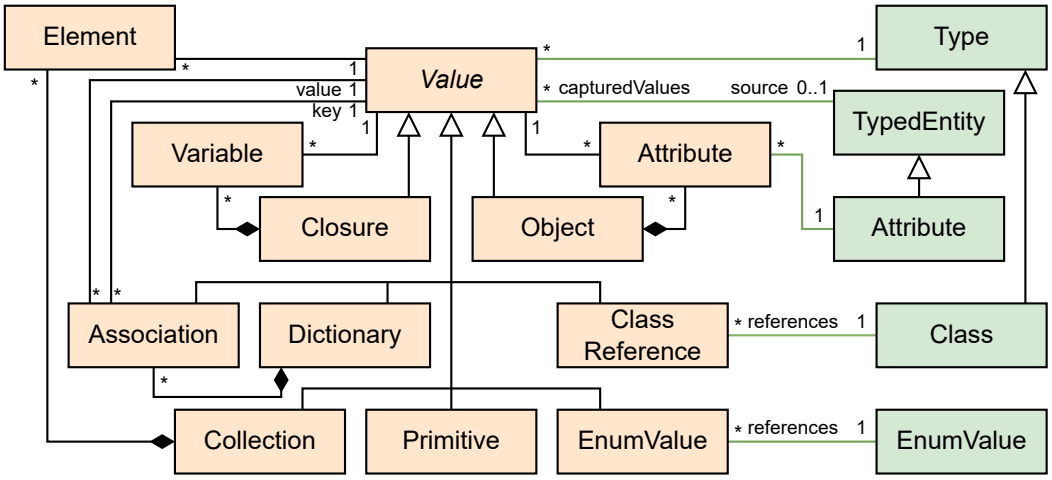

Figure 3.2: Class diagram of the Value metamodel

As with the rest of our approach, the Value metamodel is designed to support multiple programming languages. While its current structure reflects object-oriented execution models, which is our main use case so far, it is not inherently limited to them. The metamodel can be extended to represent values from other paradigms by introducing the necessary abstractions. Its structure is not tied to the syntax or semantics of any particular language. This flexibility is key to supporting test generation for systems implemented in different programming languages.

The metamodel consists of several entities that represent different categories of runtime values. At the root of this metamodel is the `Value` entity, which generalizes all supported value types. It is linked to a `TypedEntity`, which is

a code entity that represents a concrete code element that has a type. Such a code entity can be an attribute, a parameter, or a method, among others. This link between `Value` and `TypedEntity` serves as a bridge between the code elements and the values they can hold or produce at runtime. By linking a value to its typed entity, we can retrieve the context in which the value was produced or used, and understand how it fits into the structure of the codebase.

`Value` is also in a relationship with the `Type` code entity, which is used to represent the type of the value. The reason why `Value` is related to both `TypedEntity` and `Type` is that a value may not always have a typed entity, as is the case when it is in a collection or a dictionary. However, even when a value is associated with a typed entity, its type may differ from the actual runtime type due to polymorphism. Therefore, the relationship between `Value` and `Type` is needed to accurately represent a runtime data model.

The `Object` entity represents an instance of a class. Its attributes are modeled using `Attribute` entities, each of which associates the object with a code-level attribute and the corresponding value. This enables the model to represent object state as a set of key-value pairs structured by the original class definition.

The `Primitive` entity represents a value of primitive type, such as an integer, a string, or a boolean. Rather than modeling it structurally, it stores the actual value in a property. This approach reflects the simplicity of primitive values, which can be expressed immediately in code using literals. This allows them to be handled uniformly, regardless of their specific type.

The `EnumValue` entity represents a member of an enumeration. The member value is represented by a reference to the corresponding enumeration element in the codebase, using the `EnumValue` code entity.

The `Collection` entity represents any kind of collection, such as an array, a list, or a set. Instead of referencing values directly, collections are linked to `Element` entities that associate each collection with its contained values. This allows the same value to appear multiple times in the same collection and preserves ordering semantics when applicable.

The `Dictionary` entity represents a collection of key-value pairs, as found in mapping structures such as Java's `Map` implementations. Each key-value pair is modeled using an `Association` entity, which links the dictionary to both the key and the value. This allows for multiple occurrences of the same value entity as an association key or value within the same dictionary. Associations also follow the ordering semantics of the concrete map implementation when applicable.

The `ClassReference` entity represents a reference to a class itself, rather than to an instance of the class. This situation arises in languages where classes themselves can be manipulated as values, for example when they are passed as arguments, stored in variables, or used in reflective operations.. To

capture this, `ClassReference` is linked to the corresponding `Class` code entity in the code model.

The `Closure` entity represents an executable function value that may capture variables from its lexical scope. It is linked to one or more `Variable` entities, each of which associates the closure with a variable name and the corresponding value. This allows the metamodel to represent the lexical environment of a closure in the same way that attributes represent object state or elements represent collection contents. In addition, closures store their source code as text so that they can be reconstructed.

An example instance of the Value metamodel is shown in Figure 3.3. This example illustrates a `User` object with attributes including a name, a `Session` object, a collection of tags, and a dictionary of preferences.

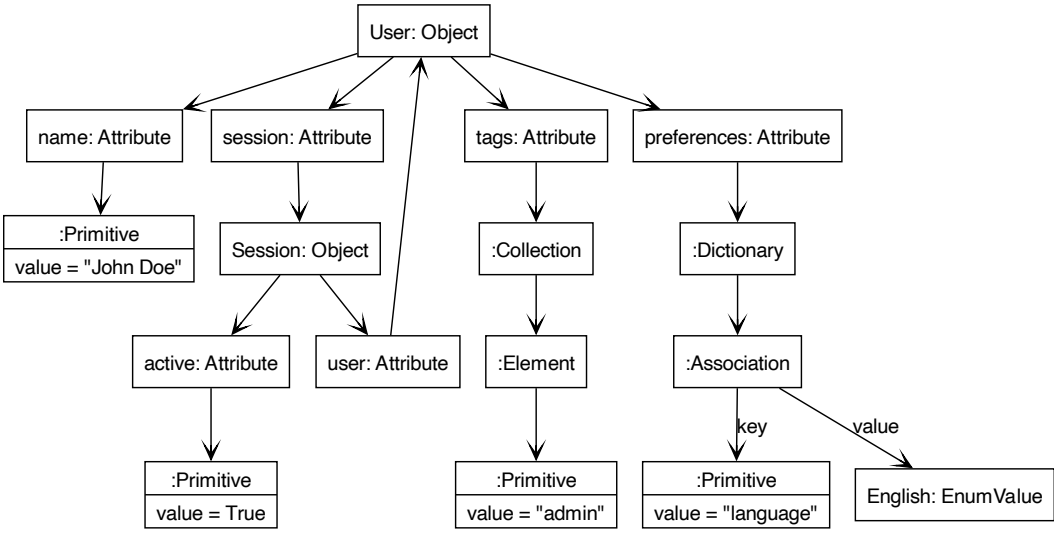

Figure 3.3: Instance diagram of a Value model

3.4 Conclusion

In this chapter, we presented the modeling infrastructure and metamodels that form the basis of our test generation process. We outlined Moose and Famix, and explained how we use their extensibility to design our own metamodels. Specifically, we introduced the Unit Test and Value metamodels, which reify test structure and runtime data respectively. In particular, the Unit Test metamodel follows the Arrange-Act-Assert pattern and enforces a single *Act* per test, reflecting our design focus on clarity and maintainability. The next chapter details our test carving approach, which uses the infrastructure introduced here.

Chapter 4

Language-Agnostic Test Carving Approach

Contents

This chapter presents our test carving approach and its implementation in the tool Modest. We explain the design choices that give the approach its language-agnostic capabilities and describe how it can be adapted to new programming languages and test frameworks. The approach relies on the language-independent metamodels introduced in Chapter 3 to represent code, values, and unit tests. Section 4.1 gives a step-by-step explanation of our test

generation process. Section 4.2 specifies the requirements imposed at each step. Section 4.3 explains how code is generated using our metamodels. Section 4.4 details how to adapt MODEST to new languages and test frameworks by extending the modeling, import, and export extension points, with examples from Java and Pharo. We also share lessons learned throughout the process, highlighting known issues and the design measures that help prevent them. We conclude with a brief summary of the approach's implications for reuse and extensibility.

This chapter builds on our work published at the 2024 International Workshop on Smalltalk Technologies [Darbord 2024] and at the 2025 edition of the International Conference on Software Analysis, Evolution and Reengineering (SANER) [Darbord 2025].

4.1 Test Generation Process

This section describes our five-step test generation process. We first provide an overview, and then we go into the details of each step.

4.1.1 Process Overview

Test carving uses execution traces to automatically generate unit tests by extracting relevant parts of the execution history. This technique replays methods with the same arguments and receiver, and ensures that the output matches what was recorded in the trace. This assumes that the current version of the software system under test is correct, allowing execution traces to serve as an oracle for expected behavior.

During test generation, MODEST uses a defensive strategy to ensure that only compilable tests will be produced. Traces that trigger an error at any stage of the process are discarded, rather than being used to generate potentially incorrect or uncompilable code. This helps avoid introducing tests with semantic faults, such as runtime exceptions during setup or failing assertions due to misconstructed objects.

Our process relies on five steps and three interconnected metamodels to generate unit tests, as shown in Figure 4.1. The Code metamodel represents the codebase under test, and elements related to it are shown in green in the figures. The Value metamodel specifies the values extracted from the traces used to test the codebase, with related elements shown in orange. The Unit Test metamodel captures the structure of unit tests, with related elements shown in blue. These metamodels abstract the key aspects of our approach, allowing us to adapt MODEST to a wide range of contexts.

To connect these metamodels with the practical reality of applying test carving on a system, we also distinguish between different sets of methods

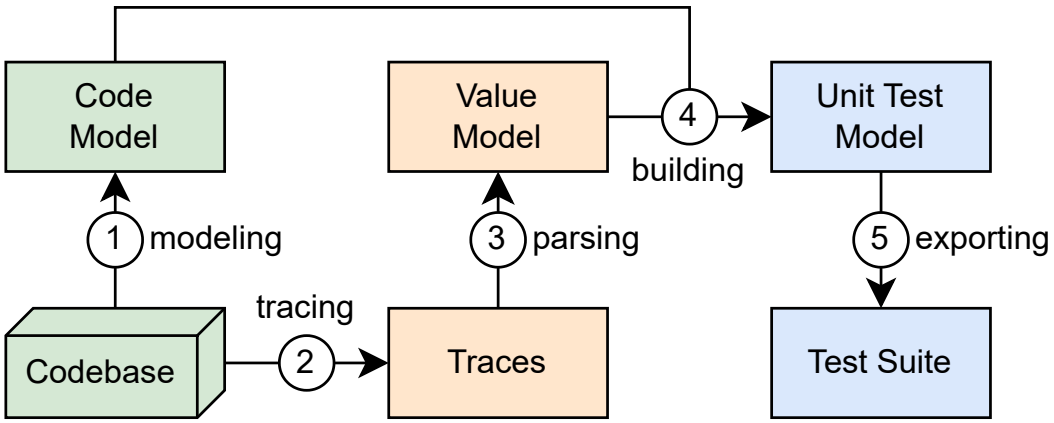

Figure 4.1: The five steps of our approach. Elements representing the code under test are shown in green (left column), elements representing runtime information in orange (middle column), and elements representing the generated tests in blue (right column).

that determine what can ultimately be tested. We distinguish three sets of methods in the target application:

- **Modeled methods** are those represented in the Code model.
- **Instrumented methods** are those targeted by the tracing infrastructure and thus capable of producing execution traces.
- **Covered methods** are those that are actually executed in a given scenario.

Only methods that are simultaneously modeled, instrumented, and covered are *eligible* for test generation, as depicted in Figure 4.2. We now detail the five steps of our approach and note how they relate to the sets defined above.

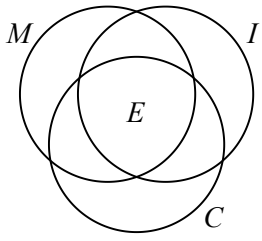

Figure 4.2: Venn diagram of the three sets of methods: modeled (M), instrumented (I), and covered (C). Their intersection (E) denotes methods eligible for test generation.

4.1.2 Step 1: Obtain a Model of the Application

Using the capabilities of the Moose platform, we create a model of the application for which we want to generate tests. This Code model captures the structural aspects of the application, such as its classes and methods, and their relationships (invocations, accesses, references, inheritance). This step determines the *modeled* methods and needs to be done only once for any given version of the software. The following steps can be repeated as many times as desired. Moose uses static analysis, but the Code model could also be built with other means, such as dynamic analysis.

4.1.3 Step 2: Produce Traces of the Application

Data about the execution of the software system is recorded as execution traces. Each trace corresponds to a method execution and must contain the following information: method unique identifier, arguments, return value, and the receiver object. Any of those can be omitted if there is no ground for it, for example procedures do not return any value, and functions do not have a receiver. In Java, a method unique identifier consists of the fully qualified class name together with the method name and its parameter types (method signature). Each recorded method execution provides the basis for a generated test. Thus, a method can have multiple tests generated that differ in the value of the receiver, arguments, or the return value. This step determines the *instrumented* methods, through the configuration of the tracing infrastructure, and the *covered* methods, through the execution of scenarios.

A good starting point for tracing systems is to focus on public methods in public classes, excluding constructors and accessors. This choice is motivated by both technical and practical considerations. First, it avoids the complexity of accessing non-public members, which would require reflection or package-level access in Java. Second, it aligns with the expectations of our industrial partner, who is interested in generating tests specifically for public APIs. Lastly, public methods are externally visible components of a system and implement the contract of their available API, making them appropriate targets for test generation. This practice is consistent with other test generation tools such as EvoSuite, which restrict themselves to public interfaces for similar reasons [Fraser 2013].

Producing a trace may fail because certain objects cannot be serialized. We refer to this error as *unserializable value*. Examples include threads, file handles, sockets, and other operating system resources that lack meaningful or portable serialized representations. Such objects often hold runtime-specific or native state that cannot be captured in a platform-independent format. When they occur, the corresponding execution cannot yield a usable test. Either serialization raises an error and the execution is not recorded, or the object

ends up incomplete in the trace and must be skipped during test generation.

Another possible source of failure is that the code may rely on a state that is external to the object. We refer to this error as *external state*. For example, static fields are not part of the captured execution context, so their value is neither serialized in the trace nor restored during test execution. As a result, the outcome of the method depends on the state of the class at the time the test is run. This leads to flakiness: the test may pass or fail depending on the ordering of test execution, or on the environment, even though the method behaved deterministically during trace collection. The same can occur with methods that depend on external resources, such as files or databases.

4.1.4 Step 3: Import and Parse Trace Data

The raw execution traces, stored as textual artifacts, are imported into MODEST for parsing. The tool first discards duplicate traces by checking textual equality, ensuring that only distinct executions are processed into tests. The Value metamodel, as described in Section 3.3.2, is then instantiated. An importer interprets the serialized representations of method arguments, return value, and receiver, and traverses the captured data to create the corresponding `Value` entities. These are associated with their `Code` counterparts: the method unique identifier determines the method under test, arguments are bound to parameters by index, the return value is bound to the method, and the receiver is bound to its actual runtime type. Attributes of object values are bound to their code definitions using the identifiers contained in the trace. This step identifies the traces that correspond to a *modeled* method and thus become *eligible* for test generation.

An error may occur when a value's runtime type belongs to a category that the Value metamodel does not yet represent. We refer to this issue as *unsupported type*. The Value metamodel classifies runtime data into specific categories, such as primitives, lists, and objects. However, some constructs do not naturally fit into any existing category. For example, closures represent a unique semantic concept that was not included in the first version of the metamodel. When traces contain such values, parsing fails because MODEST lacks the implementation required to represent and handle them. Supporting these cases requires extending the metamodel to encompass additional types as they arise in practice.

4.1.5 Step 4: Build a Unit Test Model

This step is the core of the approach and is agnostic of the language and the test framework as it is based on metamodels. During this step, the Unit Test metamodel, as described in Section 3.3.1, is instantiated. This is when the tests are modeled. Given the trace of a method execution, MODEST creates

a `TestMethod` for the executed method. A `TestClass` is created or reused to hold the `TestMethod`. The `SetUp`, `Act` and `Assertion` entities are created and associated with the `TestMethod`. The `Value` entities of the arguments are associated with the `SetUp` and `Act`. Similarly, the `Value` entity of the receiver is also associated with the `SetUp` and `Act`. Finally, the `Value` entity of the return value is associated with the `SetUp` and `Assertion`.

4.1.6 Step 5: Export the Unit Test Model Into Concrete Tests

This step transforms the unit test model into executable test code. This requires knowledge of the target language and framework.

Within the test method, we follow the AAA pattern to organize the test logic clearly. First, the arrange phase recreates the receiver, arguments, and return value, and stores them in separate variables. The return value is stored in a variable explicitly named "expected". The exporter maps the `Value` entities to their variable names. Then, the act phase exercises the method under test using the variables for the arguments and receiver. The return value is stored in a variable named "actual". Finally, the assert phase checks that the actual and expected values are equal. The way to achieve this can vary depending on the test framework or assertion library. More details about the export architecture and the generated code is given in Section 4.3.

When exporting values, some *language-specific constructs* may raise challenges when trying to transform them back into source code. A key example is closures, such as Java lambdas or Pharo blocks, discussed in more details in Section 4.3.2.6. Another example is non-static inner classes in Java. These classes implicitly capture a reference to their enclosing instance via a synthetic field that only appears in the compiled bytecode, not the source code. Reconstructing these objects requires specialized syntax, such as `new Outer().new Inner()`, and is only feasible when both the inner and outer classes, as well as their constructors, are publicly accessible. Otherwise, generating valid reconstruction code may be impossible, or too intricate and not yet implemented.

To promote readability and shorter tests, we propose to generate *helper methods* to recreate value graphs. Each helper method has no parameters and is specialized for a specific object, collection, or dictionary. Helper methods follow a naming pattern based on the role of the value they recreate and the associated test method. To maximize clarity, names use underscores rather than camel case: this preserves the original casing of variable and test names, and makes each substring visually distinct. The name consists of:

- the string "given";

- if for an argument, the name of the associated parameter followed by "argument"; if for a receiver, the string "receiver"; if for the expected value, the string "expected";
- the string "for";
- the name of the test method.

For example, `given_lastName_argument_for_testSaveUser` is a helper that reconstructs the argument `lastName` for the test method `testSaveUser`. We also propose to export these methods into a dedicated class, one for each test class. This organization keeps the test methods clean and focused on the test logic, while the helper class handles the complexity of object creation and initialization.

We have described the five steps of our test generation process. We now detail the requirements for this process to work.

4.2 Approach Requirements and Challenges

This section presents the requirements and challenges associated with each step of our approach. Steps 1, 2, 3, and 5 impose requirements on the information or mechanisms available to MODEST. Step 4, which builds the Unit Test model, does not introduce additional requirements since it operates solely at the level of the metamodel and is independent of language- or framework-specific constraints.

4.2.1 Requirements and Challenges of Step 1: Obtain a Model of the Application

When modeling the application, MODEST requires **information about user-defined types**. Specifically, we require their structural definition: namespace, modifiers (especially visibility), inheritance, attributes and method unique identifier. Most of this information is needed during test export (step 5). For example, the namespace of the tested class is used to contain the test class. The method unique identifier is used to associate a trace to the method that produced it.

The completeness of the code model is a difficult balance to strike. When a trace contains values of types that are not present in the code model, an error occurs that we call *missing type*. This happens, for example, with classes from native or third-party libraries that were not included in the code model. Including entire platforms, such as the Java Standard Library, would result in impractically large models with high memory requirements. We also cannot know in advance which types will be needed, as this depends on the traces

produced in step 2. As a partial remedy, MODEST handles common data structures, such as lists and maps, via hardcoded logic.

A similar but distinct error is *incomplete type*. It arises when a type's definition in the code model is incomplete. A type is considered incomplete if either it, or any of its superclasses, appears in the code model only as a stub, *i.e.*, a placeholder that signals the type's existence but lacks its full declaration. Such stubs may contain minimal information inferred from usage, but often omit critical details such as declared fields, constructors, and setter methods. A typical example is a class defined in a tested system that extends a class from a third-party library. Without complete knowledge of fields and methods, MODEST cannot reliably generate reconstruction code.

4.2.2 Requirements and Challenges of Step 2: Produce Traces of the Application

The second step in our process is to collect the runtime execution data necessary to generate tests that faithfully replay recorded method executions. The actual mechanism is irrelevant to MODEST, as long as the required data is accurately extracted. For example, bytecode instrumentation is a convenient approach that allows instructions to be injected at runtime, enabling the output of execution data without requiring any changes to the source code. Alternatively, instrumentation can be hard-coded directly into the target methods, although this is a more intrusive approach.

While the mechanics of data collection can vary widely between languages, the essential requirement remains the same: for each invocation of a target method, MODEST requires its **method unique identifier** and its **receiver, arguments, and result**. The input to MODEST is a collection of execution traces, each corresponding to a method invocation. An example trace is shown in Listing 4.1. When capturing runtime values, the traces are typically exported in a serialized format. Any serialization format can be used, as long as a suitable importer can deserialize it and instantiate the Value model. The serialized values must contain two key data points to recreate structured objects:

- the runtime type, which may differ from the static type when one is available, such as the declared type of a parameter, attribute, or method return;
- field identifiers that are mapped to their value.

Let us illustrate a possible serialization format that is supported by MODEST. JSON with metadata is a format produced using configurable serialization features of the Jackson library for Java[1]. This format is designed to handle

[1] `https://github.com/FasterXML/jackson`

Listing 4.1: Example execution trace given as input to MODEST (with receiver truncated)

```
{
  "class": "com.example.User",
  "method": "hasActiveSession()",
  "receiver": ...,
  "arguments": "[]",
  "result": "true"
}
```

Listing 4.2: Example of Java objects with a circular dependency serialized with Jackson

```
{
  "@type": "com.example.User",
  "@id": 1,
  "name": "John Doe",
  "session": {
    "@type": "com.example.Session",
    "@id": 2,
    "active": true,
    "user": 1
  }
}
```

cyclic references and class identification, ensuring that complex data structures can be properly serialized and reconstructed. An example of two serialized Java objects with a circular dependency is shown in Listing 4.2. The *@id* field is used to uniquely identify objects, and the *@type* field specifies their runtime type. The example shows an instance of the *User* class with the id 1, and a *Session* instance with the id 2. The session references the user in its *user* field by its id. In a statically typed language, if a field is declared to hold an object but the trace contains only an integer value, this indicates that the integer is serving as a reference identifier rather than a direct value.

While the concrete format may vary across languages and serializers, the general expectation is that MODEST receives a list of execution traces. Each trace may be complete or partial, and the tool includes fallbacks when certain elements are missing. In case a trace is missing any information among the receiver, arguments or result, MODEST is still able to generate a test. Only the method unique identifier is absolutely required. For example, functions do not have a receiver, and procedures do not return any value. If the method does not return a value (*e.g.*, methods with `void` declared type in Java), or

if the result is missing from the trace, MODEST generates a test without an assertion. Such tests are still useful as smoke tests, which focus on verifying basic functionality and stability. If the receiver is missing, MODEST generates code to create a default instance of the tested class. The constructor with the fewest parameters is chosen, and default values are used for the parameters.

In case the serialized data does not specify runtime type information for non-primitive values, MODEST will try to use the declared type of the value to reconstruct the object. For collections, it is only possible to infer the type of their elements if the collection is parameterized (*e.g.*, `List<User>`).

A complex issue that can arise is what we refer to as *value graph duplication*. The same object may be serialized multiple times in a trace, resulting in separate objects even though they should refer to the same instance. Figure 4.3 illustrates this problem with a generic example. Two value graphs, A and B, share a common subgraph C. The method under test creates a new path from A to B, and returns A as the result. When creating the execution trace, A and B are serialized independently before execution, which causes C to be duplicated. The post-execution state of A is also captured and used as the expected value in the test. When executing the test with the reconstructed A and B, each has its own copy of C. After the Act phase, the method result (the actual value) therefore includes both copies. Although the two C instances are structurally equal, they are distinct objects in memory, leading the structural equality assertion between the actual and expected values to fail. To prevent this issue, the execution trace must preserve object identity across captured values so that aliasing is faithfully reconstructed in the generated tests.

4.2.3 Requirements and Challenges of Step 3: Import and Parse Trace Data

MODEST requires that the serialization format preserves **complete representations of runtime values**. If values are omitted, altered, or transformed during serialization, the reconstructed objects may lack necessary information. This problem can arise, for example, when the application itself uses the same library and defines annotations that modify how classes are serialized, such as renaming or ignoring fields. It can also occur when objects are converted into alternative representations, such as dates serialized as human-readable strings or as numeric offsets to an epoch. Other test carving tools [Jaygarl 2010, Tiwari 2022, Deljouyi 2023] report encountering this issue of *incomplete serialization*.

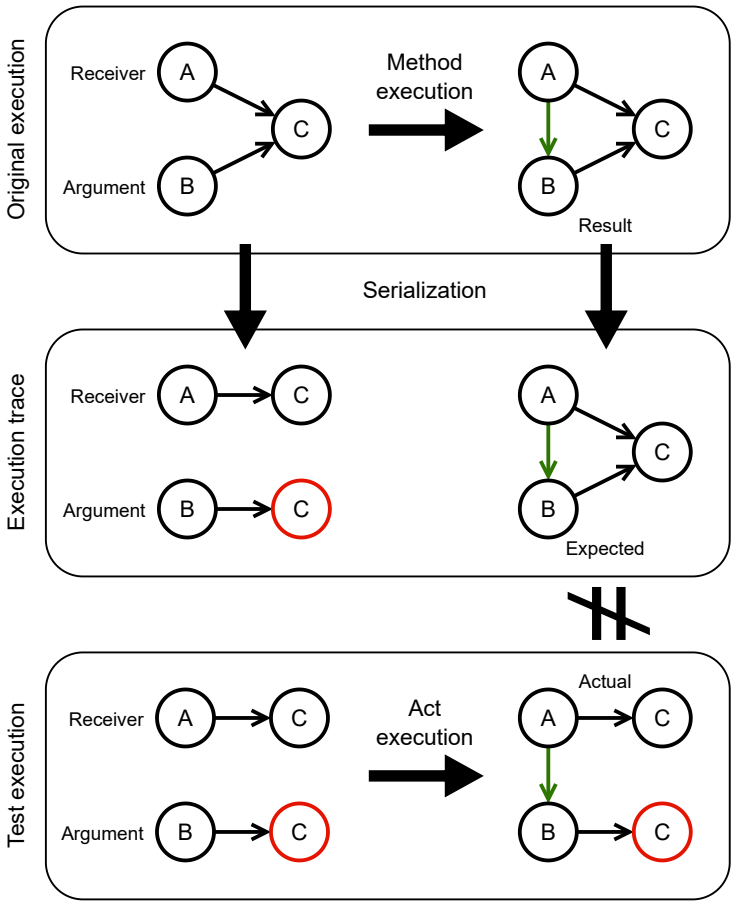

Figure 4.3: Example of *value graph duplication*

4.2.4 Requirements and Challenges of Step 5: Export the Unit Test Model Into Concrete Tests

MODEST requires **mechanisms to reconstruct values as code**, including the ability to instantiate objects and assign their attributes. This reconstruction is a complex problem [Wachter 2024]. To reconstruct objects, MODEST looks for constructors and setters. Class inheritance information can be used to look up these methods, which may be declared in superclasses. However, there may be no constructors or setters, or their visibility (*e.g.*, `private` in Java) can prevent their use in test code. In both cases, the reflexive layer of a language can allow getting around these issues by accessing constructors and fields directly, and bypassing visibility rules.

4.3 From Models to Code

This section details how MODEST turns the Unit Test and Value models into source code. First, we describe the export architecture, which separates concerns between test structure and value reconstruction. Second, we present the mechanisms for reconstructing values, which ensure that runtime data is faithfully expressed in compilable source code. Finally, we discuss strategies for generating assertions, where different oracle choices balance precision against robustness to evolution.

4.3.1 Export Architecture

The export layer separates concerns between two components: the *unit test exporter*, which generates the test structure, and the *value exporter*, which reconstructs concrete values. Both exporters are language-specific, while the unit test exporter is additionally framework-specific (*e.g.*, JUnit vs. TestNG[2]).

The two exporters are implemented using the *visitor* pattern over the `Unit Test` and `Value` metamodels. They cooperate to build an intermediate Abstract Syntax Tree (AST), as illustrated in Figure 4.4. The unit test exporter drives construction of test classes and methods, delegating to the value exporter when data must be expressed. The resulting AST is then passed to a common exporter and rendered into source code by a pretty-printer.

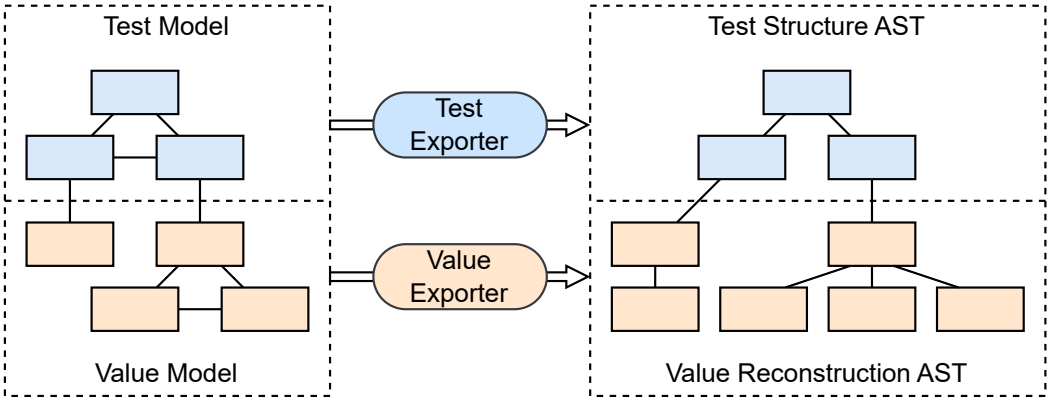

Figure 4.4: Representation of interconnected test and value models and their export to AST

[2] `https://testng.org/`

4.3.2 Reconstructing Values

Reconstructing values is necessary to transform the abstract representations of runtime data into compilable source code. The exporter traverses the Value model, recreating the structure of objects, collections, primitives, and other entities so that they can be instantiated as concrete values in the generated tests. This process must preserve identity, handle cycles, and ensure that values are expressed in the idioms of the target language.

4.3.2.1 Value Reconstruction Overview

Values form a directed graph, which may contain cycles. The exporter visits this graph recursively and assigns each value to a variable, following a single-assignment style: variables are never reassigned once created. Since a value graph can be cyclic, the exporter must keep track of values it has already visited. If a value is visited for the first time, it is exported and bound to a fresh variable. If it has been visited before, the existing variable is reused. Composite values are built by referring to these variables.

Some values are standalone, they correspond to leaf nodes in the graph, and can be exported as a single statement. Other values are *composites*, that is, values composed of other *components*. Reconstructing a composite typically requires multiple statements. The value reconstruction process follows this structure:

1. **Initialization**, where the composite is created (*e.g.*, an empty list) and bound to a variable.

2. For each component:

   (a) **Component construction**, where the component is visited recursively; components may themselves be composites.

   (b) **Composition**, where the reconstructed component is attached to the composite (*e.g.*, inserting an element into a list).

If a composite has no components, its reconstruction reduces to initialization only. A key reason for performing initialization first is that it binds the composite to a variable early. This allows other components to reference it immediately, which is essential in the presence of cycles in the value graph. For instance, if a component has the root composite as one of its subcomponents, it can safely reuse the variable bound to the root even though the root is not yet fully composed. This rule works in the general case, though certain edge cases may exist, for example if a mutator method performs side effects that assume its argument is already fully composed.

The following explains how each value entity is exported into test code.

4.3.2.2 Standalone values

Primitive values, such as numbers, booleans, and strings, are output as literals in the generated code. Enum values are emitted as references to the corresponding enumeration constants. Class references are exported as references to the class object itself, such as `Object.class` in Java.

Although all reconstructed values are initially assigned to variables following a single-assignment style, standalone values do not depend on other components. This makes them candidates for post-processing once the AST has been built: they can be inlined directly at their usage sites. Inlining reduces the number of variables and can improve the readability of the generated code.

4.3.2.3 Collections and dictionaries

Collections and dictionaries are exported using the standard APIs provided by the target language. These APIs typically guarantee an empty constructor and mutator methods for inserting content, *e.g.*, `add(element)` for collections and `put(key, value)` for dictionaries. The exporter first initializes an empty collection or dictionary, then reconstructs and inserts each component in turn. Ordering semantics are preserved when relevant.

4.3.2.4 Objects

Objects are the most complex values to reconstruct because their creation and configuration rely on class-specific APIs rather than standardized ones. Constructors and mutator methods are defined at the discretion of developers, with no guarantee of uniformity across classes. This makes both initialization and attribute assignment more difficult than in the case of collections or dictionaries.

Pharo provides a primitive method (`basicNew`) that allows creating an instance without invoking any developer-defined initialization code. This gives direct access to an uninitialized object and avoids the need to analyze constructor behavior. In Java, creating an instance without running a constructor is technically possible, but only through non-idiomatic mechanisms (*e.g.*, `Unsafe.allocateInstance`). These APIs bypass the normal object-creation rules and are not used in typical application code. Because they fall outside what Java developers normally do, we do not use them and instead rely on constructors. The reconstruction process begins by enumerating all constructors of the object's class found in the code model.

- If no explicit constructor is defined, the implicit empty constructor is used.
- Otherwise, constructors are partitioned by visibility. Public constructors

are preferred; if none exist, non-public constructors are considered and invoked reflectively.

If the chosen set contains a single constructor, it is selected. If multiple candidates remain, each constructor is assigned a score reflecting how many of its parameters correspond to attributes of the object:

$$\text{score} = \frac{\#\text{parameters assigned to attributes}}{\#\text{parameters}}$$

This requires analyzing constructor bodies to trace how parameters are used, in particular whether they are assigned directly or indirectly to attributes. The constructor with the highest score is selected, unless an empty constructor is available, in which case the empty constructor is always preferred. This is because the scoring approach has inherent risks: constructor bodies may contain side effects that complicate reconstruction, such as assumptions about arguments (*e.g.*, non-null), derived attribute values, conditional assignments, or even modifications of the parameters. A full analysis would require control-flow and data-flow reasoning. Relying on an empty constructor when available avoids these issues, at the cost of deferring attribute initialization to setters.

Finally, when a constructor with parameters is chosen, the corresponding components must already have been exported so they can be passed as arguments. If any of these components form a cycle with the object being reconstructed, then reconstruction through this constructor is impossible, since the object must first be initialized and assigned to a variable before it can be used as a component elsewhere.

After the object is initialized, the exporter iterates over its attributes. It excludes attributes that were initialized by the chosen constructor. Each attribute is reconstructed and assigned to the object by invoking a setter, or through reflection when necessary.

We distinguish between value attributes, which denote values described in the trace, and code attributes, which denote program declarations. The value attributes provided in a trace do not necessarily cover all the code attributes of a class.

For each value attribute, MODEST searches for a setter method in the class hierarchy of the object. Because a setter may be declared in a subclass rather than in the class that defines the corresponding code attribute, the search proceeds from the object's class up to and including the field's declaring class. A method is considered a setter if:

- its name begins with `set`,
- it has exactly one parameter,
- its declared return type is `void`, and

- its body accesses the corresponding code attribute.

If a publicly accessible setter is found, the exporter generates a call to it. Otherwise, the exporter assigns the value directly using reflection (*e.g.*, with `Field` in Java). In both cases, the variable created when reconstructing the value attribute is passed as the argument.

Ideally, we want the setter to be pure: it should only assign its parameter to its attribute. However, in practice, some setters may contain additional logic, such as validation or side effects. Such *impure setter* can lead to failures during test execution. Detecting them statically is difficult for the same reason as constructors.

4.3.2.5 Special Objects

Most objects are reconstructed by combining constructor calls and setter invocations based on the code model of the application. However, certain library classes are not present in the code model and cannot be analyzed in this way. A representative case is `java.util.Date`, which is typically serialized as a long encoding the number of milliseconds since the Unix epoch. In principle, static analysis could identify the appropriate constructor (*e.g.*, `Date(long)`), but this requires the Java Runtime Environment (JRE) itself to be part of the model, which is not generally the case. As a result, such objects cannot be reconstructed by generic rules.

To address this, MODEST assigns them to *special objects*, each equipped with dedicated reconstruction logic. During export, the special object emits code that recreates the value through idiomatic factories or constructors, bypassing the need for static analysis of external libraries. For instance, a serialized timestamp is reconstructed as `new Date(epoch)` in Java. Common library classes such as `java.util.Date` and `java.sql.Date` are supported in this way. Extending support for additional special objects requires providing analogous, type-specific reconstruction rules.

4.3.2.6 Closures

Closures are difficult to reconstruct as code because they depend on *lexical binding*: the body may reference free variables whose bindings exist only in the original lexical environment. To emit compilable source for a closure, two ingredients are required: (i) the closure body, and (ii) its lexical environment (the captured variables with their bindings).

In Java, serializing closures (called lambdas) in a format that provides these ingredients is not natively supported, nor is it supported by our chosen serialization library. Moreover, lambdas are not present in our industrial context. For these reasons, MODEST currently does not implement lambda reconstruction as code. However, there are alternative techniques to reconstruct

lambdas at the runtime level (*e.g.*, via `SerializedLambda` or other binary formats). Incorporating such mechanisms could be a potential avenue for future work to extend support for Java lambdas.

In Pharo, closure source code can be recovered, but capturing their lexical environment in traces is currently impossible due to limitations in the chosen instrumentation infrastructure. For now, MODEST only exports *constant* closures, which have no captured variables.

Closure conversion [Appel 1992] is a way to reconstruct closures by generating a class with fields for the captured values and a method that implements the body. The class can then be instantiated with the reconstructed values, producing compilable code without relying on the original lexical scope. However, this process alters both the syntax and the type of the value (from an anonymous function to a class). In Pharo, all closures are instances within the `BlockClosure` class hierarchy, which provides a dedicated API. Replacing closures with generated classes would require either subclassing or emulating this API to preserve functionalities. Exploring closure conversion could be a promising direction for future work, but it is not the focus of MODEST at present.

4.3.3 Asserting Equality

As discussed in Section 2.2.2, most existing test carving approaches rely on default equality for assertions. This choice raises two problems. First, if default equality is not explicitly defined, comparisons reduce to object identity checks, which are too strict. Second, if it is defined by developers, its semantics may not align with the notion of behavioral equivalence expected in a regression oracle.

Structural equality provides an alternative that has received comparatively little attention in test carving. It compares two value graphs by recursively verifying the equality of their nodes and edges. This avoids reliance on developer-defined equality, but it also introduces fragility: structural comparison treats all state as relevant, including fields or representation details that may not matter for observable behavior. As a result, behavior-preserving refactorings or representation changes, such as field renaming or collection reordering, can alter the structure of values and trigger test failures. It may also suffer from technical issues such as *value graph duplication* discussed in Section 4.2.2.

In practice, there is no single universally correct notion of equality. Default equality may be suitable when domain-specific semantics are explicitly defined and considered reliable, but it risks masking relevant behavioral differences if those semantics are incomplete. Structural equality, in contrast, increases precision at the cost of robustness to evolution. The choice of oracle therefore entails a trade-off between stability and sensitivity, and must be guided by the

intended use and context.

For MODEST, we prioritized precision and therefore adopted structural equality as the default strategy. Because structural comparison is not commonly supported natively in languages or test frameworks, we rely on external libraries, such as AssertJ[3] in Java. In the future, custom structural comparison methods could be automatically generated, removing the need for third-party libraries and allowing more control over the comparison semantics. This would make it possible to adjust the level of precision, trading strictness for greater robustness to behavior-preserving changes when appropriate.

4.4 Adapting MODEST to New Ecosystems

MODEST is designed to be adaptable to the programming language, test framework, and serialization format. The import process is affected by the language and the serialization format. The export process is affected by the language and the test framework. To achieve this, MODEST provides the extension points necessary to map language- and framework-specific constructs to the metamodels. Concrete components implement these extension points, each of which fits a specific context.

MODEST can be adapted for the modeling of the application (step 1), the import of execution traces (step 3), and the export of tests (step 5). Steps 2 and 4 do not require adaptation. Producing traces (step 2) is done outside of the MODEST tool. Building the test model (step 4) is implemented as a model-to-model transformation. Because these metamodels abstract away language-specific details, step 4 is entirely agnostic to the programming language or test framework. The metamodeling principles that enable this separation are detailed in Chapter 3.

In the following, we outline the possible adaptations for the other steps. These adaptations show how MODEST can be extended to support new languages and testing environments.

4.4.1 Adapting Step 1: Obtain a Model of the Application

The first step in our test generation process is to model the target application with the Code metamodel. MODEST is part of the Moose platform which models programming languages in a generic way. Languages currently supported by Moose include Java, Pharo, C/C++, Python, Fortran, and TypeScript.

In practice, MODEST does not need a complete model of the application. The model only needs to contain the information listed for step 1 in Section 4.2.

[3] `https://assertj.github.io/`

4.4.2 Adapting Step 3: Import and Parse Trace Data

The third step is to import the execution traces obtained in step 2 into MODEST. It has an extension point that can accommodate different trace serialization formats. The format varies depending on the serialization library and its settings. Currently, MODEST supports importing data in the formats from Jackson (JSON with metadata) and XStream[4] (XML). Supporting new formats requires implementing the same parsing logic as the deserialization processes of the libraries that produce them.

4.4.3 Adapting Step 5: Export the Unit Test Model Into Concrete Tests

The final extensible step in our process is the export of the unit test model to actual tests. At this stage, two key extension points come into play to handle new target languages and environments: the unit test exporter and the value exporter. These components work together to generate the final output in the form of executable tests.

The unit test exporter is responsible for generating the structure of the test cases. It is dependent on the language and test framework. This includes creating the test class, defining methods, and following the AAA pattern that is standard in unit testing. During this process, the unit test exporter interacts with the value exporter, asking it to recreate values for each step in the test that requires them. This includes the initialization of the receiver, the arguments, and the expected result.

The role of the value exporter is to generate the source code necessary to recreate each value. It is only dependent on the language. This ensures that the values used in the tests are the same as those captured during execution. If the execution trace captured a structured object, the value exporter must produce the code that would generate an equivalent object in the target language. As explained in the requirements (Section 4.2), this means finding the constructors and setters, or using the reflective capabilities of the language. Listing 4.3 shows an example of the Java code to recreate the serialized value of Listing 4.2.

In practice, depending on the serialization process, some objects may have a particular representation format that does not map their attributes. For example, a date can be represented by a "yyyy-mm-dd" string, and the idiomatic way to reconstruct it from the string is to use a specific factory method. Import and export components can be specialized to handle such cases.

The extension points throughout these three steps make our approach modular. MODEST can be extended to generate unit tests in new ecosystems

[4] `https://x-stream.github.io/`

Listing 4.3: Code recreating two Java objects in a reference cycle

```
User user = new User();
user.setName("John Doe");
Session session = new Session();
session.setActive(true);
session.setUser(user);
user.setSession(session);
```

by implementing components tailored to specific programming languages, test frameworks, and serialization formats.

4.5 Conclusion

In this chapter, we presented how our test generation approach can be extended to support new programming languages, test frameworks, and serialization formats. While the core of the process is implemented as a language-agnostic model transformation, three steps require adaptation: modeling the code under test, importing execution traces, and exporting tests. We described the responsibilities and requirements of each extension point, and explained how they are handled in MODEST using modular components. By isolating language-specific behavior and relying on metamodels to represent code, values, and tests, our approach promotes reuse, extensibility, and maintainability. This design enables MODEST to be applied across heterogeneous environments, and lays the foundation for integrating support for additional contexts in the future.

In the next chapter, we evaluate our approach by applying it to real-world applications written in Java and Pharo. We present the results of the test generation process, and discuss the readability of the generated tests.

Chapter 5

Empirical Evaluation of Modest

Contents

In this chapter, we present the empirical evaluation conducted to assess the effectiveness and applicability of Modest. The evaluation aims to determine the tool's ability to generate meaningful and correct unit tests across multiple languages and software systems. This chapter is based on our work published at the 2025 edition of the International Conference on Software Analysis, Evolution and Reengineering (SANER) [Darbord 2025].

First, we describe the goal of the evaluation and the experimental protocol, including the subject systems and metrics. Next, we present the results and explain how they support the claims made in this thesis.

5.1 Goals

The goal of this evaluation is to verify that the test carving approach implemented in Modest can generate unit tests from execution traces across a variety of real-world contexts. Specifically, we aim to demonstrate that generated tests:

- are compilable and executable for systems implemented in different programming languages;
- accurately reproduce the observed behavior recorded in execution traces, as measured by their pass or fail outcomes;
- show characteristics supporting readability and maintainability, evaluated through detection of code and test smells [Fowler 1999, van Deursen 2001].

To support these objectives, we applied MODEST to several software systems differing in:

- **Programming languages:** We selected systems written in Java and Pharo.
- **System context and size:** We include both open-source and proprietary systems of varied sizes and application domains.
- **Execution scenarios and traces:** We collected execution traces using different sources, such as unit tests, demonstration scripts, and manually executed scenarios.

We discuss how specific language characteristics, such as typing systems and access control mechanisms, influence the generation process. This chapter covers language-specific constructs, serialization behavior, and typing differences because these factors contributed to the technical challenges we encountered during test generation. These factors also explain certain failures or limitations observed in our results.

The remainder of this chapter first describes the evaluation protocol and study subjects. It then presents the results of test generation and discusses the causes of both successful and failed test generation, as well as the readability and maintainability of the generated tests.

5.2 Evaluation Protocol

This section describes the methodological choices guiding the evaluation, while subsequent sections introduce the study subjects and trace collection procedure.

To demonstrate MODEST's multi-language capabilities, we apply it to software systems developed using different programming languages. We select real-world applications rather than toy examples to assess the applicability of MODEST in realistic scenarios. These applications include both open-source and proprietary (closed-source) systems, which allows us to evaluate MODEST across a broader range of development contexts. This selection serves two purposes. First, it demonstrates MODEST's applicability to industrial

systems, such as those provided by our industrial partner. Second, it allows for reproducibility and comparative analysis through publicly available systems.

Previous studies have shown that open-source and proprietary systems share many engineering characteristics, including complexity metrics and defect-proneness patterns [Paulson 2004, Nguyen-Duc 2017]. This suggests that techniques effective on one type of system can be applied to the other. However, important differences in architectural structures, development practices, and requirements elicitation processes have also been observed [MacCormack 2006]. Opinions vary about experimenting with open and closed-source systems, so we evaluated our tool on both to assess the robustness of our approach in varied contexts. The systems chosen for this study also vary in size and application domain, providing additional diversity in the evaluation.

The completeness of the evaluation is inherently limited by the coverage of the executed scenarios. To avoid bias, we propose to rely exclusively on pre-existing usage scenarios from the documentation, test suites, or examples of each system, instead of constructing scenarios tailored to ensure successful outcomes. This ensures that our evaluation reflects realistic usage patterns and that the results are representative of how MODEST would perform in practical settings, rather than under idealized conditions.

We selected Java and Pharo as the target languages to demonstrate MODEST's ability to generate tests across different programming environments. Supporting Java was an explicit requirement because it is the primary language used by our industrial partner. Pharo, on the other hand, is the language in which MODEST is implemented, as well as the main platform used by our research team. Therefore, supporting Pharo as a target language is both convenient and meaningful. Both are object-oriented languages, but they differ significantly in their type systems and access control. Java is statically typed, and developers can specify the visibility of classes and methods. In contrast, Pharo is dynamically typed, all classes and methods are public, and fields are protected. The lack of declared types in Pharo highlights the importance of type information in execution traces. While type inference techniques exist, reliably inferring the types of parameters, return values, and object fields from the traces alone can be very difficult or even impossible. For MODEST, accurate type information is essential for reconstructing objects and generating valid test code, making this a key challenge in dynamically typed environments.

To evaluate MODEST, we report the number of tests generated and the proportion that pass for each method traced during scenario execution. Our focus is not on maximizing test coverage but on verifying whether MODEST can correctly generate unit tests from the behavior exercised in existing scenarios. This reflects the intended purpose of a test carving tool, which differs fundamentally from search-based approaches, such as EvoSuite [Fraser 2011a], that aim to increase coverage or discover new behaviors. We do not attempt to measure or improve overall application coverage because scenario completeness

is outside the scope of this work.

To evaluate test quality, we use test smell detection tools on the generated suites and report the presence of common maintainability and design issues. We relied on two established tools: SonarQube[1] (version 9) for code smells and JNose [Virgínio 2020] for test smells. This analysis was conducted on the tests generated for Omaje, the main industrial subject of this thesis introduced in Section 5.3, since they are representative of our collaboration with Berger-Levrault and allowed us to discuss with the development team. While this does not constitute a formal user study, it provides insight into the practical usability of the tests.

The following section introduces the systems used in our evaluation.

5.3 Study Subjects

We selected four study subjects for this evaluation, two for each target language, chosen to reflect a range of sizes, application domains, and development contexts, including both industrial and open-source systems:

- **Omaje** is a proprietary Java customer subscription management project with a three-tier architecture. Our industrial partner, Berger-Levrault, uses it internally and has provided us with access to the source code;
- **Traccar**[2] is an open-source Java GPS tracking platform with a three-tier architecture. It has been previously studied in the context of multi-language GUI migration [Verhaeghe 2021];
- **Cormas**[3] is an open-source Pharo agent-based modeling platform with a graphical user interface. It is developed and used by Cirad[4] for modeling socio-ecological systems [Bommel 2016];
- **DataFrame**[5] is an open-source Pharo library that implements a tabular data structure for data manipulation and analysis.

Table 5.1 summarizes the size of each system in terms of number of classes and methods. The two Java projects are significantly larger than the Pharo ones, particularly with respect to the number of classes. However, all systems contain a substantial number of methods, providing a sufficient basis for evaluating the generation of unit tests across different scales.

[1] `https://www.sonarsource.com/products/sonarqube/`

[2] `https://github.com/traccar/traccar/tree/v6.5`

[3] `https://github.com/cormas/cormas/tree/v0.95`

[4] French agricultural research center for international development: `https://www.cirad.fr`

[5] `https://github.com/PolyMathOrg/DataFrame/tree/pre-v3`

Table 5.1: Study subjects: language, version, and size in classes and methods

Project	Language	Version	#Classes	#Methods
DataFrame	Pharo	v6.5	29	1161
Cormas	Pharo	v0.95	59	2036
Traccar	Java	pre-v3	1346	4688
Omaje	Java	–	1198	7006

Having introduced the study subjects, we now describe how execution traces were collected and instrumented for each system.

5.4 Trace Collection

To evaluate the effectiveness of MODEST, we first needed execution traces from the subject systems. This section describes how traces were collected. We detail the instrumentation process used to capture method invocations and serialize runtime values. We then define the scope of test generation for each project, specifying the instrumented classes and methods as well as the scenarios used to produce traces.

5.4.1 Instrumentation Process

To collect execution traces for test generation, we followed the OpenTelemetry specification[6], which defines language-agnostic conventions and a model for recording structured runtime events. We instrumented each application to emit trace data at method boundaries, capturing the method unique identifier, receiver, arguments, and result for each invocation. Tracing logic was inserted dynamically at runtime, which allowed us to observe program execution without modifying the source code of the target systems. Runtime values were serialized in a JSON format and enriched with metadata about runtime types and object identities, following the structure described in Section 4.2.

For the Java applications (Traccar and Omaje), we used the existing OpenTelemetry instrumentation support and the Jackson library for value serialization. For the Pharo applications (DataFrame and Cormas), we implemented our own tracing infrastructure because no standard tracing library exists in this ecosystem. We based it on the OpenTelemetry specification to provide a solid and interoperable foundation, and relied on the native instrumentation mechanism of Pharo (metalinks [Denker 2008, Costiou 2020]). We also developed a serialization library that replicates Jackson's metadata structure. Existing Pharo serializers depend on information available in the code, whereas our format is self-contained so that a trace can be deserialized

[6] `https://opentelemetry.io`

without requiring behavior specific to the original program. Using Jackson as a reference further allowed us to reuse logic from our Java test generation.

5.4.2 Scope of Test Generation

We defined the scope of instrumentation in terms of target classes and methods. This configuration determines the resulting instrumented set, that is, the methods that could produce traces during execution.

For all systems, we focus on public methods in public classes, as explained in Section 4.1.3. Building on this general rule, we explain how the set of tested methods was determined for each study subject. Each project has its own sets of modeled, instrumented, and covered methods, making up their eligible methods for test generation, as defined in Section 4.1. We used JaCoCo[7] to measure the number of covered methods in Java projects, and DrTests [Guerra Calle 2019] for Pharo projects.

The method used to trigger execution traces depended on the availability of existing tests. For Omaje and Cormas, there were few or no existing unit tests. In these cases, we collected traces from the functional usage scenarios provided by the developers. For Cormas, we used a pre-existing demonstration of the agent simulation framework. For Omaje, the developers manually devised scenarios based on their domain expertise, because no functional test suites were readily available. This approach reflects the intended use of MODEST, which is designed to carve fine-grained unit tests from coarse-grained execution scenarios, such as system-level tests, ad-hoc developer interactions, or demonstrations. Also, rather than instrumenting the full codebase, we collaborated with the developers to identify a specific set of relevant classes to test.

In the case of **Omaje**, the selection was focused on a specific architectural layer of the application, chosen to reflect the development team's current testing priorities. Omaje has a three-tier architecture, and the server comprises multiple internal layers. The target was the service layer of the server tier of the application. The service layer was chosen in consultation with the developers because it encapsulates most of the business logic. This decision resulted in the instrumentation of 314 methods across 68 classes.

A developer with domain expertise produced execution traces by manually interacting with a client application connected to an instrumented server instance. The developer performed representative operations corresponding to typical usage scenarios. While this manual execution of scenarios allowed for focused and realistic trace collection, the trace coverage was limited to the behaviors exercised during those sessions. Since the exact execution scenarios were not documented, we cannot determine how many methods were covered.

[7] `https://www.jacoco.org/jacoco/`

The developers acknowledged that the executed scenarios could be extended to cover additional behaviors.

In **Cormas**, we collaborated with the developers to identify three core classes for evaluation: `CMEntity`, `CMSpatialEntity`, and `CMAgent`. These classes are central abstractions in the framework and serve as superclasses for all entities, spatial tiles, and agents in simulation models. Trace collection was based on an existing demonstration model involving forager agents that interact with vegetation tiles in a 27×27 grid. The model simulates the evolution of two types of foragers (restrained and unrestrained) starting with ten entities of each type. To ensure adequate coverage, we ran the simulation for a sufficient number of steps to allow all instrumented methods that could be triggered during normal execution to be exercised at least once.

We applied a filter to the method category to exclude initializers and trivial accessors from instrumentation. Of the 154 methods defined in the three classes, 99 were instrumented. Executing the demonstration scenario led to coverage of 54 methods. The intersection of the instrumented and covered methods yielded 12 methods for which test generation was attempted.

For Traccar and DataFrame, we instead exercised the applications using their existing unit test suites. This is not representative of the typical use case for a test carving tool, which is usually applied to higher-level scenarios such as integration tests. However, for the purposes of our evaluation, this choice was appropriate: it allowed us to generate traces without requiring prior familiarity with the systems, thereby focusing on validating MODEST's functionality rather than scenario design. The unit tests already covered a wide range of behavior, making them a practical and consistent source of execution data.

The **Traccar** codebase contains 1,346 classes, of which 913 are public. We instrumented all public methods in public classes, excluding constructors and accessors, as described above. Among these, 677 classes contain no methods that match the filtering criteria. As a result, instrumentation was applied to 649 methods across the remaining 236 classes. Traccar's unit test suite covers 1,468 methods. Of those, 78 methods were both instrumented and executed during trace collection.

In the **DataFrame** project, instrumentation was limited to the central `DataFrame` class. All 187 of its methods were instrumented. Measuring coverage showed that 144 of these methods were executed by the original test suite. These 144 executed methods produced traces eligible for processing.

Table 5.2 summarizes how many methods were modeled, instrumented, covered, and eligible for test generation for each project.

With the study subjects and trace collection process established, the next section presents the results of applying MODEST to these systems.

Table 5.2: Modeled, instrumented, covered, and eligible methods per study subject

Project	Modeled	Instrumented	Covered	Eligible
DataFrame	1161	187	144	144
Cormas	2036	99	54	12
Traccar	4688	649	1468	78
Omaje	7006	314	–	76
Total	**16891**	**1249**	**>1666**	**310**

5.5 Results

This section presents the success rate of test generation, the pass rate of the generated tests, and the quality of tests in terms of smells.

5.5.1 Success Rate of Test Generation

The success rate of test generation varied considerably across projects. For **DataFrame**, MODEST generated tests for 13 out of 144 traced methods, a rate of 9%. For **Cormas**, 9 out of 12 traced methods produced tests (75%). For **Traccar**, 10 out of 78 traced methods yielded tests (13%). Finally, for **Omaje**, all 76 traced methods produced valid tests, achieving a 100% success rate.

Overall, out of 310 traced methods across all systems, tests were successfully generated for 108 (35%). These outcomes are summarized in Table 5.3, which shows both the number of eligible methods and the number for which at least one compilable test was produced.

Table 5.3: Eligible methods and methods for which MODEST successfully generated tests, per project

Project	Eligible Methods	Successful Tests
DataFrame	144	13 (9%)
Cormas	12	9 (75%)
Traccar	78	10 (13%)
Omaje	76	76 (100%)
Total	**310**	**108 (35%)**

Omaje stands out with its perfect success rate. This result can be attributed to our familiarity with the system and the nature of the targeted methods. Most objects involved in the traces were domain entities with simple structures, making them amenable to serialization and reconstruction. In contrast, the other projects involved more diverse or complex behaviors that introduced

unpredictable failure points in the test generation process. Section 5.6 discusses the causes of failures during test generation.

5.5.2 Pass Rate of Generated Tests

We examine the pass rate of the generated tests, that is, whether the tests produced for each method executed successfully. This complements the success rate, which only measured whether tests could be generated in the first place.

The results are presented in Table 5.4. For each study subject, we report the methods for which tests were generated, the number of tests per method, and the number of passing tests. For each tested method, all tests either passed or failed consistently, so we only report the number of passing tests. Due to confidentiality constraints, we do not disclose method or class names for Omaje, which is a proprietary software system.

In total, we generated 417 tests across all study subjects. Each test corresponds to an execution trace that was successfully processed by the test generation pipeline. These traces originate from 108 distinct methods: 13 from DataFrame, 9 from Cormas, 10 from Traccar, and 76 from Omaje. All generated tests compile, and the overall pass rate is 86.1%. Notably, DataFrame and Omaje achieved a 100% pass rate.

Among the 58 failing tests, 50 originate from Cormas and 8 from Traccar. These failures are concentrated in four specific methods (rows 16, 17, 22, and 27), for which none of the generated tests pass. The causes of these failures are discussed in Section 5.6.2.

5.5.3 Smells in Generated Tests

To evaluate the presence of code smells, we used SonarQube 9 to analyze the tests generated for Omaje. It has 479 rules that cover a wide range of code smells, including those found in tests. The analysis showed that it did not find any smell. SonarQube also reported that the generated tests do not suffer from code duplication, which is a common problem in generated code. This indicates that the generated tests follow standard coding practices, reducing technical debt and making future maintenance easier for developers.

To evaluate the presence of test smells, we used JNose [Virgínio 2020] on the tests generated for Omaje and present our findings in Table 5.5. Most test smells do not arise in the generated tests due to the design choices made in MODEST. We enforce the Arrange-Act-Assert (AAA) structure, so a test has at most one act, which prevents *Eager Test* (a test should invoke only one method of the production object), and one assertion, which prevents both *Assertion Roulette* (a test has multiple non-documented assertions) and *Duplicate Assert* (the same condition is asserted multiple times within the

Table 5.4: Experimental results of test generation with MODEST on the study subjects

#	Class.Method	#Test	#Passing
1	DataFrame.asArrayOfColumns	4	4
2	DataFrame.calculateDataTypes	4	4
3	DataFrame.columnNames	27	27
4	DataFrame.columns	4	4
5	DataFrame.dataTypes	10	10
6	DataFrame.initialize:	2	2
7	DataFrame.initializeColumns:	1	1
8	DataFrame.initializeRows:	1	1
9	DataFrame.numberOfColumns	10	10
10	DataFrame.numberOfRows	5	5
11	DataFrame.privateRowNames:	5	5
12	DataFrame.rowNames	1	1
13	DataFrame.setDefaultRowColumnNames	4	4
	DATAFRAME TOTAL	**78**	**78 (100%)**
14	CMEntity.delete	9	9
15	CMEntity.init	5	5
16	CMEntity.initId	27	0
17	CMSpatialEntity.addOccupant:	5	0
18	CMSpatialEntity.allOccupants	18	18
19	CMSpatialEntity.destroyed:	9	9
20	CMSpatialEntity.initOccupants	18	18
21	CMSpatialEntity.neighbourhood	18	18
22	CMSpatialEntity.neighbourhoodWithNils:	18	0
	CORMAS TOTAL	**127**	**77 (60.6%)**
23	GeofenceCircle.containsPoint	2	2
24	GeofenceCircle.distanceFromCenter	2	2
25	GeofenceCircle.fromWkt	1	1
26	GeofenceCircle.toWkt	1	1
27	GeofencePolyline.containsPoint	8	0
28	GeofencePolyline.fromWkt	4	4
29	GeofencePolyline.toWkt	1	1
30	Network.addCellTower	10	10
31	Network.addWifiAccessPoint	10	10
32	Position.addAlarm	10	10
	TRACCAR TOTAL	**49**	**41 (83.7%)**
	OMAJE TOTAL	**163**	**163 (100%)**
	TOTAL	**417**	**359 (86.1%)**

same test). JNose still found the *Assertion Roulette* smell, but only because the generated assertions do not provide a description.

The generated tests are not subject to the smells *Constructor Initialization* (a test suite should not have a constructor), *Conditional Test Logic* (tests should not have conditional statements), *Default Test* (placeholder test classes created by an integrated development environment), and *Empty Test* (tests should have executable statements). MODEST removes duplicate execution traces, which avoids the *Lazy Test* smell (more than one test with the same fixture that tests the same method). Generating helpers moves the verbosity of value reconstruction out of the test methods, avoiding the *Verbose Test* smell (test method with too many lines). When handling exceptions, generated tests use the appropriate assertions rather than custom try/catch blocks, which avoids the *Exception Handling* smell. Tests do not depend on external resources, so both *Mystery Guest* (tests depend on external resources such as files or databases) and *Resource Optimism* (tests assume an external resource exists without checking) do not arise. Generated tests do not include print statements or waits, thereby avoiding *Redundant Print* (tests include unnecessary print statements) and *Sleepy Test* (tests wait for events). Since we assert structural equality, tests do not suffer from *Redundant Assertion* (assertions that are always true or false) and *Sensitive Equality* (asserting string representation equality).

Some smells were not detected by JNose even though manual inspection reveals their presence. Since we recreate full graphs when reconstructing values, fixtures may contain fields that are unused by tests, which corresponds to the *General Fixture* smell. JNose did not detect the *Unknown Test* smell (tests have no assertions and pass by default if no exception is thrown), although our tool can generate smoke tests without assertions when methods return no value. Finally, generated tests are subject to the *Magic Number Test* smell (numeric literals appear directly in assertions) because MODEST inlines literals throughout the generated code (as explained in Section 4.3.2.2). While inlining literals may be good to reduce the size of helper methods, this should be avoided for test methods in future iterations of our tool. JNose might not have been able to detect this smell because we did not use assertions from JUnit, but from the AssertJ[8] library instead, which it might not recognize.

In the next section, we discuss the scope of the test generation process for each project, the causes of test failures and generation failures, and the readability and maintainability of generated tests.

[8] `https://assertj.github.io/`

Table 5.5: Assessment of test smells in Omaje tests by JNose

Test smell	Presence
Assertion Roulette	YES
Conditional Test Logic	NO
Constructor Initialization	NO
Default Test	NO
Duplicate Assert	NO
Eager Test	NO
Empty Test	NO
Exception Handling	NO
General Fixture	NO
Lazy Test	NO
Magic Number Test	NO
Mystery Guest	NO
Redundant Assertion	NO
Resource Optimism	NO
Redundant Print	NO
Sensitive Equality	NO
Sleepy Test	NO
Unknown Test	NO
Verbose Test	NO

5.6 Discussion

This section interprets the results presented above. We examine the causes of failures that occurred during test generation and during test execution, and reflect on their practical implications. Finally, we discuss the quality of the generated tests and their usability in practice.

5.6.1 Failures During Test Generation

To investigate errors that occured during test generation, we analyzed the logs produced during the generation process to identify the points at which they occurred and their underlying causes. Different errors can happen during three phases of our approach: trace collection, trace parsing, and test export.

When evaluating Traccar, we encountered an example of the *missing type* error. The class `java.util.ImmutableCollections$SetN`, an internal JDK implementation for small immutable sets, was missing from our code model. Ideally, recreating such collections would involve invoking the correct factory methods. However, supporting all such cases would require extensive specialization.

In both Java and Pharo systems, we encountered *language-specific constructs* that our value reconstruction process could not handle, namely Java

lambdas and Pharo blocks (see Section 4.1.6). We also found cases of *unserializable value* (Section 4.1.3), *incomplete type* (Section 4.2.1), and *incomplete serialization* (Section 4.2.3) errors.

The errors encountered during test generation stem from language-specific constructs, limitations in serialization, and gaps in the code model that make value reconstruction more complex. While some of these limitations can be addressed through additional specialization, this remains challenging in a language-independent context. Next, we turn to the failures that occurred during test execution.

5.6.2 Failures During Test Execution

Among the 417 tests generated across all projects, 58 failed during execution. As previously reported, 50 of these failures originate from Cormas and 8 from Traccar. Notably, the failures are concentrated in four specific methods, for which none of the corresponding tests pass. This systematic pattern shows that the failures stem from fundamental issues in how input data was captured and reconstructed for these methods.

To understand these failures, we relied primarily on the following sources of information:

- The execution reports, to identify exceptions or assertion failures and their causes.
- The generated test code, to verify its structure and semantic correctness.
- The implementation of the tested methods, to understand their behavior and how they led to the failure.
- The original execution traces, to verify that relevant information was captured and serialized accurately.

We identified the causes of the failures and present our findings for each of the four affected methods below.

`CMEntity.initId` (Cormas): This method relies on a static field to assign unique identifiers to instances, which corresponds to the *external state* error (Section 4.1.3). Addressing such cases is currently outside the scope of our approach.

`CMSpatialEntity.neighbourhoodWithNils:`/`addOccupant:` (Cormas): Tests for these methods fail due to a mismatch in object references during the assertion phase, which corresponds to the *value graph duplication* error (Section 4.2.2). This happens when the expected result contains a reference to an object that has already been serialized elsewhere in the trace, whereas the actual result contains a structurally equal but distinct object. This error occurs

in particular with Cormas because its data model involves many interconnected entities. We leave managing these problematic object graphs for future work.

`GeofencePolyline.containsPoint` (Traccar): Failures for this method are caused by an exception raised during the reconstruction of an input argument. The execution trace records an argument of type `Geofence` whose `area` field is `null`. To reconstruct this object, MODEST selected the setter method `setArea(String)`. When invoked with `null`, the method raises a `NullPointerException`. This corresponds to the *impure setter* problem discussed in Section 4.3.2.4.

In summary, the failures observed in our evaluation stem from three main factors: (i) the absence of reconstructed global or external state, (ii) inconsistencies in object graphs between receivers and arguments due to separate serialization, and (iii) the difficulty of selecting suitable reconstruction methods.

In the next section, we turn our attention to the readability and maintainability of the generated tests, examining whether the generated code meets the practical expectations of developers.

5.6.3 Readability of Generated Tests

Readability was identified as one of our evaluation criteria in Section 5.1 and provides insight into the potential for developer adoption of the generated tests.

One indication of readability comes from Omaje, where developers modified some generated tests. For instance, one test was adapted to assert only the filename within a file path rather than the full absolute path. Such modifications indicate that the generated tests were sufficiently clear and maintainable to be reused and adjusted as needed. We provide examples of generated tests in Listing 5.1, Listing 5.2, Listing 5.3, and Listing 5.4.

Listing 5.1 shows a test generated for the Java project Traccar, and the associated helper method is presented in Listing 5.2. The helper uses the simplest constructor available, an empty constructor, because no other appropriate constructor could be automatically identified. When the constructor does not initialize all fields, MODEST attempts to use setter methods. If no suitable setters are found, the tool resorts to reflection, using the generated `setField` method to assign values directly to fields. In this particular case, using a constructor that takes field values as arguments would have been preferable, but such a constructor was either absent or not detected. Reconstructing objects as plain source code is a non-trivial problem, especially when dealing with classes that expose limited construction or accessing methods. Improving constructor and setter inference is part of our future work, and relates to ongoing challenges discussed by [Wachter 2024].

Despite these limitations, the use of helper methods improves the readability and maintainability of generated tests. They abstract away the complexity of

Listing 5.1: Java test generated for Traccar

```
@Test
public void testDistanceFromCenter() {
    /* ARRANGE */
    GeofenceCircle receiver =
       given_receiver_for_testDistanceFromCenter();
    /* ACT */
    double actual = receiver.distanceFromCenter(
       55.75545, 37.61921);
    /* ASSERT */
    assertThat(actual).isEqualTo(163.77736255543593);
}
```

Listing 5.2: Java helper generated for the test of Listing 5.1

```
public static GeofenceCircle
   given_receiver_for_testDistanceFromCenter() {
    GeofenceCircle geofenceCircle = new GeofenceCircle();
    setField(geofenceCircle,
       GEOFENCECIRCLE_CENTERLATITUDE, 55.75414);
    setField(geofenceCircle,
       GEOFENCECIRCLE_CENTERLONGITUDE, 37.6204);
    setField(geofenceCircle,
       GEOFENCECIRCLE_RADIUS, 100.0);
    return geofenceCircle;
}
```

object creation and encapsulate the use of reflection, allowing the body of the test to focus on behavior rather than setup. Although the example in Listing 5.2 is relatively simple, helper methods are designed to handle arbitrarily complex object graphs, sometimes involving dozens of fields or nested objects. They are also responsible for catching and rethrowing exceptions that may occur during reflective access or construction, ensuring that the generated tests compile and execute without interruption. This design aligns with the principles of clear test arrangement advocated in prior work [Khorikov 2020, Wachter 2024].

The Pharo test for DataFrame shown in Listing 5.3 also follows the AAA structure, with a clear separation of phases. As with Java, helper methods are used to simplify the arrangement phase and improve test readability. A particular challenge in Pharo is the limitation on the number of temporary variables in a method, imposed by the virtual machine's stack size. When recreating complex objects, each intermediate value may require its own variable, which can easily exceed this limit. As a workaround, all intermediate values

Listing 5.3: Pharo test generated for DataFrame

```
testRowNames
    | actual expected anArray receiver |
    "ARRANGE"
    expected := self given_expected_for_testRowNames.
    anArray := self
       given_anArray_argument_for_testRowNames.
    receiver := self given_receiver_for_testRowNames.
    "ACT"
    actual := receiver rowNames: anArray.
    "ASSERT"
    self assert: actual deepEquals: expected
```

can be stored in a single dictionary. While this avoids stack overflow errors, it significantly reduces the clarity of the generated test code. This example illustrates how language-specific constraints can impact test generation and may require compromises between correctness, completeness, and readability.

In summary, the readability of our generated tests can be assessed against the four axes introduced in Section 2.2.1. **Naming** in generated tests follows a systematic scheme that distinguishes test methods, helper methods, and variables. This ensures that generated identifiers are unique, descriptive of their role (*e.g.*, helper constructing the receiver, variable holding the expected value), and predictable for developers. While the names remain mechanically derived from traces rather than idiomatic to the domain, the scheme addresses clarity and avoids ambiguity. Regarding **code and test smells**, automated analyses with SonarQube and JNose found no code smells and only a few test smells, indicating that most common issues are avoided by design. **Value reconstruction readability** is supported through code-based reconstruction, though reflection and Pharo's temporary variable limits introduce challenges. Finally, **scaffolding** is provided through helpers that encapsulate complexity and improve the clarity of test methods. Taken together, these aspects show that our approach addresses readability concerns more explicitly than many existing carving tools, while still leaving room for improvement in idiomaticity and language-specific constraints.

Listing 5.4: Pharo helper generated for the test of Listing 5.3

```
given_receiver_for_testRowNames
    | contents3 contents2 contents rowNames columnNames
       dataTypes dataFrame |
    contents3 := { 41.58. 20.79. 2.31. 55.03.
       18.343333333333334. 2.8233333333333337 }.
    contents2 := Array2D new
        instVarNamed: #contents put: contents3;
        numberOfColumns: 3;
        numberOfRows: 2;
        yourself.
    contents := DataFrameInternal new
        instVarNamed: #contents put: contents2;
        yourself.
    rowNames := OrderedCollection withAll: { 1. 2 }.
    columnNames := OrderedCollection withAll: { 1. 2. 3 }.
    dataTypes := Dictionary newFrom: {
        (1 -> SmallFloat64).
        (2 -> SmallFloat64).
        (3 -> SmallFloat64) }.
    dataFrame := DataFrame new
        instVarNamed: #contents put: contents;
        instVarNamed: #rowNames put: rowNames;
        instVarNamed: #columnNames put: columnNames;
        dataTypes: dataTypes;
        yourself.
    ^ dataFrame
```

5.7 Threats to Validity

Internal validity concerns the extent to which observed outcomes can be attributed to the experimental treatment, rather than to other confounding factors.

To minimize such threats, we applied a consistent methodology across all study subjects. The trace collection, test generation, and result analysis processes followed the same procedure for each project. All execution scenarios used to generate traces were derived from pre-existing project artifacts: unit tests, functional tests, or demonstrations, rather than scenarios crafted specifically for this study. While this ensures realistic usage, it may introduce a bias based on what the original developers chose to exercise. However, we argue that this bias reflects real-world priorities, as the developers' scenarios naturally focus on critical functionality. We believe this is consistent with how MODEST would be used in practice: to generate tests for behavior that developers have deemed important enough to exercise.

External validity concerns the extent to which the findings can be generalized beyond the specific study subjects.

To address this, we selected a diverse set of systems, varying in programming language (Java and Pharo), application domain (*e.g.*, GPS tracking, simulation, data processing), size, and licensing model (open-source and proprietary). Nonetheless, all systems were implemented in object-oriented languages. This limits the generalizability of our results to other programming paradigms, such as procedural or functional languages. Expanding the approach to non-object-oriented targets is left as future work.

Construct validity relates to whether the evaluation accurately measures the intended properties. In this case, correctness and readability of the generated tests.

Correctness is measured using test pass rates, which provide an objective and reproducible metric. Readability, by contrast, is inherently subjective. We did not employ a formal user study or automated readability metric. Instead, our assessment is based on informal feedback and post-hoc modifications made by the developers of the Omaje system. For instance, some generated assertions were manually adjusted for conciseness or relevance. We interpret this as evidence that the generated tests were sufficiently readable and maintainable for practical use, although a formal study involving external developers would be necessary to draw broader conclusions.

Reliability concerns the extent to which the results can be reproduced when the research is repeated under the same conditions.

To reduce the risk of introducing bias or artifacts specific to our study, we used existing execution scenarios for all subject systems. These scenarios were not created for the purpose of test generation with MODEST, but were already available within the projects, such as unit tests, functional tests, or

interactive demonstrations. As such, they reflect realistic usage and provide a stable foundation for replication. We include the generated tests and detailed instructions for reproducing the experiment in a replication package[9], specifying the versions of each subject system used. The only exception is Omaje, which we had to exclude due to its proprietary nature.

5.8 Conclusion

This chapter presented an empirical evaluation of MODEST, focusing on its ability to generate unit tests from execution traces collected in real-world systems written in different programming languages. Our results show that MODEST can successfully produce compilable and readable tests for several projects, including both industrial and open-source systems in Java and Pharo. The evaluation confirms that generated tests capture realistic system behaviors and follow clear structuring principles, while also avoiding most common code and test smells.

The evaluation further illustrates how metamodel-based abstractions make it possible to apply the same approach across different technical contexts, despite differences in language features, type systems, and runtime environments. While the evaluation highlighted areas where reconstruction or serialization remains challenging, the overall outcome demonstrates that the approach is feasible.

In the next chapter, we conclude this thesis by revisiting its main contributions, discussing their implications, and outlining directions for future work.

[9] `https://doi.org/10.5281/zenodo.13921267`

Chapter 6

Conclusion

Contents

This chapter concludes the thesis by summarizing our contributions and results obtained, and by outlining directions for future research.

6.1 General Conclusion

This thesis addressed the problem of generating an automated regression test suite for legacy systems that lack unit tests and formal specifications. This problem is particularly relevant in industrial settings such as those of our partner Berger-Levrault, where monolithic applications with thick clients are being migrated to modern Software-as-a-Service architectures with thin, web-based frontends. The central challenge was to obtain regression tests at the unit test level without substantial manual effort, while ensuring that they reflect realistic executions, are readable to developers, and can be maintained as part of regular practice. In addition, because Berger-Levrault maintains systems in multiple languages, the solution needed to be applicable across heterogeneous technical contexts.

To this end, we proposed a metamodel-driven approach to test carving. Execution traces are used as a source of behavioral ground truth, and transformed into reproducible unit tests that serve as regression oracles. By introducing explicit metamodels for code, values, and tests, we decouple the generation process from language- and framework-specific details. This design allows a single tool architecture to be reused and specialized to target different programming languages and test frameworks, without reimplementing the approach for each ecosystem. Runtime values are reconstructed as source code to improve readability, so that generated tests are not opaque artifacts but units that developers can understand and extend.

We implemented this approach in MODEST, a tool that supports Java (JUnit 4/5) and Pharo (SUnit) through modular extension points for modeling, importing traces, exporting values, and generating tests. The tool provides assertions based on both default equality and structural equality, and generates tests structured to minimize reliance on external state. Our empirical evaluation on industrial and open-source systems showed that MODEST can generate hundreds of executable and developer-readable tests, demonstrating feasibility in practice. The study also identified limitations related to serialization, complex object reconstruction, and handling of external state, which define priorities for future work.

Together, these contributions demonstrate that metamodel-driven test carving can provide a practical and extensible solution to bootstrap regression test suites for legacy backends undergoing migration. By combining regression oracles grounded in real executions, language-agnostic abstractions, and developer-oriented readability, the approach bridges the gap between automatic test generation and long-term adoption in software maintenance workflows.

6.2 Future Work

While this thesis demonstrated the feasibility and applicability of metamodel-driven test carving, several avenues remain open for further investigation and improvement. The following subsections outline three directions that we identify as particularly promising.

6.2.1 Evaluation of Assertion Strategies

A direction for future work is to investigate the impact of different assertion strategies. As reviewed in the state of the art, existing carving tools rely on a variety of oracles, with default equality being the most common. In our empirical evaluation of MODEST, we reconstructed values fully and adopted structural equality, *i.e.*, comparing object graphs by value. This increases sensitivity to evolution: behavior-preserving refactorings or representation changes (*e.g.*, collection ordering, field renaming) can alter the object graph and trigger failures. Part of this sensitivity comes from the oracle itself (by-value comparison), and part from the value reconstruction step (complete instead of partial). The choice of oracle therefore represents a trade-off that depends on the intended use and context.

A systematic evaluation could be carried out by generating identical test suites with different assertion strategies, and comparing them along multiple dimensions. These dimensions may include fault detection capability, stability under code evolution, and developer comprehension. Such a study would clarify the trade-offs between simplicity, expressiveness, and maintainability,

and would guide practitioners in choosing or configuring assertion strategies that best fit their context.

6.2.2 Test Suite Reduction

A characteristic of test carving is that it often produces a large number of tests for the same method. Although these tests may differ in their concrete inputs, many of them can be semantically redundant in the sense that they achieve the same coverage. This redundancy inflates the size of the generated suite without necessarily increasing its fault detection potential, and it places an unnecessary maintenance burden on developers. For this reason, several carving approaches either integrate test suite reduction into their process [Elbaum 2006, Thummalapenta 2010, Křikava 2018], or highlight it as an essential area of future work [Gambi 2023, Alshahwan 2024].

In our setting, a requirement is that reduction should remain language-agnostic, so that it can be applied to generated tests regardless of the target language. This is possible because generated tests, execution traces, and coverage information can all be represented at the model level. Reduction can therefore be expressed as a language-agnostic transformation applied after test generation. The goal of such reduction is to preserve the maximal behavioral coverage of the suite while eliminating redundant tests. This would extend the applicability of MODEST by allowing developers to bootstrap a regression test suite that is more concise and easier to maintain.

6.2.3 Handling of External State

A further line of work concerns the treatment of external state, in particular persistent data managed by databases. In our industrial context, many methods interact with the database through create, update, or delete operations. Unlike pure computations that return values, these operations produce side effects that are not captured by return values but are central to the correct functioning of the system.

Future work could extend carving to include assertions over the database state. For instance, after an insertion, a test could verify that an additional row exists; after a deletion, that the row count decreases; or after an update, that the expected values have been changed. Such assertions would capture effects that are crucial in practice but are otherwise invisible in generated tests. Achieving this would require a way to represent relevant aspects of the database state and to translate them into verifiable test code.

Bibliography

[Alshahwan 2024] Nadia Alshahwan, Mark Harman, Alexandru Marginean, Rotem Tal and Eddy Wang. *Observation-based unit test generation at Meta*, 2024.

[Anquetil 2020] Nicolas Anquetil, Anne Etien, Mahugnon Honoré Houekpetodji, Benoît Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatija Djareddir, Jérôme Sudich and Mustapha Derras. *Modular Moose: A new generation of software reengineering platform.* In International Conference on Software and Systems Reuse (ICSR'20), numéro 12541 de LNCS, pages 119–134, December 2020.

[Appel 1992] Andrew W. Appel. Compiling with continuations. Cambridge University Press, 1992.

[Artzi 2008] Shay Artzi, Sunghun Kim and Michael D. Ernst. *ReCrash: Making Software Failures Reproducible by Preserving Object States*. In Jan Vitek, editeur, ECOOP 2008 – Object-Oriented Programming, pages 542–565, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Baudry 2015] B. Baudry, Simon Allier, M. Rodriguez-Cancio and Monperrus Martin. *DSpot: Test Amplification for Automatic Assessment of Computational Diversity.* ArXiv, vol. abs/1503.05807, 2015.

[Beck 2000] Kent Beck. Extreme programming explained: Embrace change. Addison Wesley, 2000.

[Beck 2002] Kent Beck. Test driven development: By example. Addison-Wesley Longman, 2002.

[Bommel 2016] Pierre Bommel, Nicolas Becu, Christophe Le Page and François Bousquet. *Cormas: an agent-based simulation platform for coupling human decisions with computerized dynamics.* In Simulation and gaming in the network society, pages 387–410. Springer, 2016.

[Cadar 2006] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill and Dawson R. Engler. *EXE: automatically generating inputs of death.* In Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS '06, pages 322–335, New York, NY, USA, 2006. Association for Computing Machinery.

[Cadar 2008] Cristian Cadar, Daniel Dunbar and Dawson Engler. *KLEE: unassisted and automatic generation of high-coverage tests for complex*

systems programs. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, pages 209–224, USA, 2008. USENIX Association.

[Chen 2010] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel and T. H. Tse. *Adaptive Random Testing: The ART of test case diversity.* Journal of Systems and Software, vol. 83, no. 1, pages 60–66, January 2010.

[Chen 2024] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng and Jianwei Yin. *ChatUniTest: A Framework for LLM-Based Test Generation.* In Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, pages 572–576, New York, NY, USA, 2024. Association for Computing Machinery.

[Claessen 2000] Koen Claessen and John Hughes. *QuickCheck: a lightweight tool for random testing of Haskell programs.* In Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP '00, pages 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

[Costiou 2020] Steven Costiou, Vincent Aranega and Marcus Denker. *Sub-method, partial behavioral reflection with Reflectivity: Looking back on 10 years of use.* The Art, Science, and Engineering of Programming, vol. 4, no. 3, February 2020.

[Daka 2014] Ermira Daka and Gordon Fraser. *A Survey on Unit Testing Practices and Problems.* In 2014 IEEE 25th International Symposium on Software Reliability Engineering, pages 201–211, 2014.

[Daka 2015] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn and Westley Weimer. *Modeling readability to improve unit tests.* In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pages 107–118, New York, NY, USA, 2015. Association for Computing Machinery.

[Danglot 2019a] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus and Benoit Baudry. *A snowballing literature study on test amplification.* Journal of Systems and Software, vol. 157, November 2019.

[Danglot 2019b] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry and Martin Monperrus. *Automatic test improvement with DSpot: a study with ten mature open-source projects.* Empirical Softw. Engg., vol. 24, no. 4, pages 2603–2635, August 2019.

[Darbord 2023] Gabriel Darbord, Anne Etien, Nicolas Anquetil, Benoit Verhaeghe and Mustapha Derras. *A Unit Test Metamodel for Test Generation*. In Proceedings of the 2023 International Workshop on Smalltalk Technologies. CEUR Workshop Proceedings, August 2023.

[Darbord 2024] Gabriel Darbord, Fabio Vandewaeter, Anne Etien, Nicolas Anquetil and Benoit Verhaeghe. *Modest-Pharo: Unit Test Generation for Pharo Based on Traces and Metamodels*. In IWST 2024: International Workshop on Smalltalk Technologies, July 2024.

[Darbord 2025] Gabriel Darbord, Nicolas Anquetil, Benoit Verhaeghe and Anne Etien. *A Multi-Language Tool for Generating Unit Tests from Execution Traces*. In 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). Institute of Electrical and Electronics Engineers, 2025.

[Deljouyi 2023] Amirhossein Deljouyi and Andy Zaidman. *Generating Understandable Unit Tests through End-to-End Test Scenario Carving*. In 2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 107–118, 2023.

[Denker 2008] Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008.

[Elbaum 2006] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer and Jonathan Dokulil. *Carving differential unit test cases from system test cases*. In Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, pages 253–264, New York, NY, USA, 2006. Association for Computing Machinery.

[Fowler 1999] Martin Fowler. Refactoring: improving the design of existing code. Addison-Wesley Professional, 1999.

[Fraser 2011a] Gordon Fraser and Andrea Arcuri. *EvoSuite: Automatic Test Suite Generation for Object-Oriented Software*. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. Association for Computing Machinery.

[Fraser 2011b] Gordon Fraser and Andreas Zeller. *Generating parameterized unit tests*. In Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11, pages 364–374, New York, NY, USA, 2011. Association for Computing Machinery.

[Fraser 2012] Gordon Fraser and Andreas Zeller. *Mutation-Driven Generation of Unit Tests and Oracles.* IEEE Transactions on Software Engineering, vol. 38, no. 2, pages 278–292, 2012.

[Fraser 2013] Gordon Fraser and Andrea Arcuri. *Whole Test Suite Generation.* IEEE Transactions on Software Engineering, vol. 39, no. 2, pages 276–291, 2013.

[Gambi 2023] Alessio Gambi, Hemant Gouni, Daniel Berreiter, Vsevolod Tymofyeyev and Mattia Fazzini. *Action-Based Test Carving for Android Apps.* In 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pages 107–116, 2023.

[Godefroid 2005] Patrice Godefroid, Nils Klarlund and Koushik Sen. *DART: directed automated random testing.* In Proceedings of Programming Language Design and Implementation (PLDI'05), pages 213–223. ACM, 2005.

[Godefroid 2012] Patrice Godefroid, Michael Y. Levin and David Molnar. *SAGE: whitebox fuzzing for security testing.* Commun. ACM, vol. 55, no. 3, pages 40–44, March 2012.

[Guerra Calle 2019] Dayne Guerra Calle, Julien Delplanque and Stéphane Ducasse. *Exposing Test Analysis Results with DrTests.* In International Workshop on Smalltalk Technologies, Cologne, Germany, August 2019.

[Jaygarl 2010] Hojun Jaygarl, Sunghun Kim, Tao Xie and Carl K. Chang. *OCAT: object capture-based automated testing.* In Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10, pages 159–170, New York, NY, USA, 2010. Association for Computing Machinery.

[Jones 1986] C.B. Jones. Systematic software development using VDM. Prentice Hall International, 1986.

[Khorikov 2020] Vladimir Khorikov. Unit testing principles, practices, and patterns. Simon and Schuster, 2020.

[Khorram 2022] Faezeh Khorram. *A testing framework for executable domain-specific languages.* Theses, Ecole nationale supérieure Mines-Télécom Atlantique, December 2022.

[Křikava 2018] Filip Křikava and Jan Vitek. *Tests from traces: automated unit test extraction for R.* In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, pages 232–241, New York, NY, USA, 2018. Association for Computing Machinery.

[Lakhotia 2010] Kiran Lakhotia, Mark Harman and Hamilton Gross. *AUSTIN: A Tool for Search Based Software Testing for the C Language and Its Evaluation on Deployed Automotive Systems.* In 2nd International Symposium on Search Based Software Engineering, pages 101–110, 2010.

[Leavens 2004a] Gary Leavens and Yoonsik Cheon. *Design by contract with JML.* 07 2004.

[Leavens 2004b] Gary Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok and Joseph Kiniry. *JML reference manual*, 05 2004.

[Lévesque 2009] Martin Lévesque. *A metamodel of unit testing for object-oriented programming languages.* arXiv preprint arXiv:0912.3583, 2009.

[MacCormack 2006] Alan MacCormack, John Rusnak and Carliss Y. Baldwin. *Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code.* Management Science, vol. 52, no. 7, pages 1015–1030, 2006.

[Mao 2016] Ke Mao, Mark Harman and Yue Jia. *Sapienz: multi-objective automated testing for Android applications.* In Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, pages 94–105, New York, NY, USA, 2016. Association for Computing Machinery.

[Meszaros 2007] Gerard Meszaros. Xunit test patterns – refactoring test code. Addison Wesley, June 2007.

[Nguyen-Duc 2017] Anh Nguyen-Duc. *The impact of software complexity on cost and quality - A comparative analysis between Open source and proprietary software.* CoRR, vol. abs/1712.00675, 2017.

[Oliveira 2023] Delano Oliveira, Reydne Santos, Fernanda Madeiral, Hidehiko Masuhara and Fernando Castor. *A systematic literature review on the impact of formatting elements on code legibility.* Journal of Systems and Software, vol. 203, September 2023.

[Oriat 2005] Catherine Oriat. *Jartege: A Tool for Random Generation of Unit Tests for Java Classes.* Lecture Notes in Computer Science, 01 2005.

[Pacheco 2005] Carlos Pacheco and Michael D. Ernst. *Eclat: automatic generation and classification of test inputs.* In Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP'05, pages 504–527, Berlin, Heidelberg, 2005. Springer-Verlag.

[Pacheco 2007] Carlos Pacheco and Michael D. Ernst. *Randoop: feedback-directed random testing for Java.* In Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. Association for Computing Machinery.

[Paulson 2004] James Paulson, Giancarlo Succi and Armin Eberlein. *An empirical study of open-source and closed-source software products.* Software Engineering, IEEE Transactions on, vol. 30, pages 246–256, 05 2004.

[Pires 2018] Joao Pires and Fernando Brito e Abreu. *Knowledge Discovery Metamodel-Based Unit Test Cases Generation.* In 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), pages 432–433, April 2018.

[Schärli 2003] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz and Andrew P. Black. *Traits: Composable Units of Behavior.* In Proceedings of European Conference on Object-Oriented Programming, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[Sen 2005] Koushik Sen, Darko Marinov and Gul Agha. *CUTE: A Concolic Unit Testing Engine for C.* In Proceedings of the European Software Engineering Conference (ESEC), pages 263–272, New York, NY, USA, 2005. Association for Computing Machinery.

[Shore 2021] James Shore and Shane Warden. The art of agile development. O'Reilly Media, Inc., 2021.

[Siddiq 2024] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat and Vinícius Carvalho Lopes. *Using Large Language Models to Generate JUnit Tests: An Empirical Study.* In Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE '24, page 313–322, New York, NY, USA, 2024. Association for Computing Machinery.

[Siddiqui 2021] Saleem Siddiqui. Learning test-driven development. O'Reilly Media, Inc., 2021.

[Spivey 1989] J. Spivey. The Z notation: A reference manual. Prentice-Hall, 1989.

[Thummalapenta 2010] Suresh Thummalapenta, Jonathan de Halleux, Nikolai Tillmann and Scott Wadsworth. *DyGen: automatic generation of high-coverage tests via mining gigabytes of dynamic traces.* In Proceedings of

the 4th International Conference on Tests and Proofs, TAP'10, pages 77–93, Berlin, Heidelberg, 2010. Springer-Verlag.

[Tiwari 2022] Deepika Tiwari, Long Zhang, Martin Monperrus and Benoit Baudry. *Production Monitoring to Improve Test Suites*. IEEE Transactions on Reliability, vol. 71, no. 3, pages 1381–1397, September 2022.

[Tiwari 2024] Deepika Tiwari, Martin Monperrus and Benoit Baudry. *Mimicking Production Behavior With Generated Mocks*. IEEE Transactions on Software Engineering, vol. 50, no. 11, pages 2921–2946, November 2024.

[van Deursen 2001] Arie van Deursen, Alex Leon Moonen andvan den Bergh and Gerard Kok. *Refactoring Test Code*. In M. Marchesi, editeur, Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001), pages 92–95. University of Cagliari, 2001.

[Verhaeghe 2021] Benoît Verhaeghe, Anas Shatnawi, Abderrahmane Seriai, Anne Etien, Nicolas Anquetil, Mustapha Derras and Stephane Ducasse. *From GWT to Angular: An Experiment Report on Migrating a Legacy Web Application*. IEEE Software, 2021.

[Virgínio 2020] Tássio Virgínio, Luana Martins, Larissa Rocha, Railana Santana, Adriana Cruz, Heitor Costa and Ivan Machado. *JNose: Java Test Smell Detector*. In Proceedings of the XXXIV Brazilian Symposium on Software Engineering, SBES '20, pages 564–569, New York, NY, USA, 2020. Association for Computing Machinery.

[Wachter 2024] Julian Wachter, Deepika Tiwari, Martin Monperrus and Benoit Baudry. *Serializing Java Objects in Plain Code*, 2024.

[Yang 2024] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang and Junjie Chen. *On the Evaluation of Large Language Models in Unit Test Generation*. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE '24, pages 1607–1619, New York, NY, USA, 2024. Association for Computing Machinery.

[Zander 2011] Justyna Zander, Ina Schieferdecker and Pieter Mosterman. Model-based testing for embedded systems. 09 2011.

[Zhang 2012] Pingyu Zhang and Sebastian Elbaum. *Amplifying tests to validate exception handling code*. In 2012 34th International Conference on Software Engineering (ICSE), pages 595–605, 2012.

[Zhang 2014] Pingyu Zhang and Sebastian Elbaum. *Amplifying Tests to Validate Exception Handling Code: An Extended Study in the Mobile Application Domain.* ACM Trans. Softw. Eng. Methodol., vol. 23, no. 4, September 2014.

[Zhang 2016] Jie Zhang, Yiling Lou, Lingming Zhang, Dan Hao, Lu Zhang and Hong Mei. *Isomorphic regression testing: executing uncovered branches without test augmentation.* In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, page 883–894, New York, NY, USA, 2016. Association for Computing Machinery.

[Zimmerman 2010] Daniel M. Zimmerman and Rinkesh Nagmoti. *JMLUnit: the next generation.* In Proceedings of the 2010 International Conference on Formal Verification of Object-Oriented Software, FoVeOOS'10, pages 183–197, Berlin, Heidelberg, 2010. Springer-Verlag.