Automated Software Engineering (2026) 33:64
https://doi.org/10.1007/s10515-026-00613-0

Developing, testing, building and maintaining an Eclipse project based on EMF and Xtext

Lorenzo Bettini[1]

Received: 28 October 2025 / Accepted: 28 February 2026

Abstract

The study addresses a recurring practical problem: how to obtain high confidence in the correctness of an Eclipse/Xtext-based DSL toolchain—spanning language services, IDE integration, and UI workflows—while keeping builds and releases reproducible and CI-friendly across platforms. We present a general, practice-oriented testing and CI framework for such toolchains and instantiate it in Edelta, an open-source framework and Xtext-based DSL for the evolution and refactoring of EMF metamodels and models. We describe a comprehensive testing strategy, based on test-driven development, spanning fast unit tests for the runtime and refactoring libraries, headless DSL tests for parsing, scoping, validation, type checking, interpretation, and code generation, IDE integration tests for the Eclipse editor, and end-to-end SWTBot tests that exercise complete user workflows. These tests are orchestrated in a single Maven/Tycho build and continuously executed on GitHub Actions across platforms, with SonarCloud providing code-quality and coverage feedback. Our CI setup also builds and verifies distributable Eclipse products and a self-contained p2 update site. An empirical analysis of a representative CI run shows a clear testing-pyramid distribution, with most checks implemented at the fastest layers, keeping the overall CI turnaround time low while providing high confidence in correctness. We summarize lessons learned, actionable guidelines, and pitfalls to avoid when engineering Eclipse- and Xtext-based research software that must be both feature-rich and continuously releasable.

Keywords Metamodels · EMF · Xtext · DSL · Testing · Continuous integration · Eclipse · Build automation

✉ Lorenzo Bettini
lorenzo.bettini@unifi.it

[1] Dipartimento di Statistica, Informatica, Applicazioni, Università degli Studi di Firenze, V.le Morgagni 65, Firenze 50134, Italy

Published online: 17 March 2026

1 Introduction

Engineering and sustaining Eclipse-based toolchains for domain-specific languages (DSLs)—especially when implemented with Xtext/Xbase—requires addressing quality concerns that span multiple layers: runtime libraries, language services (e.g., scoping, validation, typing), IDE integration in an OSGi environment, and end-to-end user workflows in the generated tooling. While language workbenches and DSL engineering have been widely discussed in the literature (Mernik et al. 2005; Fowler 2010), practitioners still face substantial effort in making such toolchains *systematically testable* and *continuously releasable* in a cross-platform setting.

The research problem we address is therefore pragmatic but recurring: *how can one design a testing and CI methodology that provides high confidence for an Eclipse/Xtext-based DSL toolchain, without making builds prohibitively slow or fragile?* This problem is amplified by the fact that correctness cannot be assessed only at the parsing level: language-level services can fail in subtle ways, IDE-level behavior often requires integration tests inside Eclipse, and realistic user workflows may only be validated via true UI end-to-end tests. Moreover, sustaining such systems typically requires reproducible build and release automation that produces verifiable artifacts (e.g., Eclipse products and p2 update sites) across platforms. Systematic mapping studies also indicate that testing remains a challenging aspect of DSL development and that tool support is often limited (Kosar et al. 2016; Iung et al. 2020), while experience reports on Eclipse/Tycho builds highlight recurring "pain points" and best practices for long-lived Eclipse ecosystems (Grebien 2012; Müller et al. 2017; Seifermann and Krach 2020). However, detailed, end-to-end accounts that connect *multi-level testing* (unit → integration → UI/end-to-end) with an *integrated CI/CD pipeline* for an Xtext toolchain delivered as Eclipse artifacts are still relatively scarce.

In this paper, we contribute an experience-driven methodology that operationalizes this objective. We document concrete techniques and reusable patterns to test Xtext-based languages and Eclipse plug-ins end to end, and to automate build and release tasks using Maven/Tycho and GitHub Actions. At the unit level, we adopt test-driven development (Beck 2003; Maximilien and Williams 2003; Williams et al. 2003; Desai et al. 2008; Cauevic et al. 2012; Munir et al. 2014) for runtime APIs and language-level behavior, relying on dependency injection (Prasanna 2009) and Xtext testing utilities for fast, deterministic checks. We complement these with IDE integration tests and SWTBot-based end-to-end tests that run against our product distribution within the same Tycho-driven build. Our CI executes all tests across platforms, while SonarCloud tracks coverage and quality trends. Empirical data from representative CI runs confirms a testing-pyramid distribution (Cohn 2009; Fowler 2013): most checks live at the fastest layers, keeping the total build time low without sacrificing confidence.

We illustrate and validate the proposed methodology on a non-trivial case study drawn from a project maintained over several years: *Edelta*, an open-source framework and Xtext-based DSL for the refactoring, evolution, and co-evolution of EMF metamodels and models (Bettini et al. 2017, 2019, 2020; Barriga et al. 2021; Bettini et al. 2022a, b, 2024a, b). In Model-Driven Engineering (MDE), metamodels define the shape and well-formedness of models, and models become first-class develop-

ment artifacts (Schmidt 2006; Steinberg et al. 2008). As domains evolve, metamodels and the corresponding models must co-evolve to remain consistent (Paige et al. 2016; Hebig et al. 2017)—a setting that naturally yields a feature-rich toolchain with complex interactions between language services, IDE features, and generated artifacts.[1] Edelta comprises a runtime library on top of EMF, a DSL implemented with Xtext (Bettini 2016) and Xbase (Efftinge et al. 2012), an Eclipse editor and IDE with advanced editor services, and a refactoring library including bad-smell detection and resolution. A distinctive aspect is the use of interpretation during compilation, which provides a live, context-aware editor that surfaces errors and quick fixes as the developer types.

In summary, this paper makes the following contributions. (1) Several testing strategies and methodologies for Xtext-based DSLs that cover runtime APIs, language services, IDE integration, and user workflows. (2) A unified Maven/Tycho build that executes unit, UI, and SWTBot tests, also producing and validating distributable Eclipse products and a self-contained p2 update site. (3) A CI process based on GitHub Actions and SonarCloud for cross-platform verification and continuous quality analysis. (4) Lessons learned and pitfalls to avoid when engineering Eclipse- and Xtext-based research software that must be both feature-rich and continuously releasable.

The rest of the paper is structured as follows. Section 2 recalls Edelta main features we rely on throughout the paper. Section 3 presents our general testing and CI framework for Eclipse/Xtext-based toolchains. Section 4 presents unit tests for the runtime and the DSL, including parser, scoping, validation, typing, interpretation, and generation checks. Section 5 covers IDE integration tests for our Eclipse-based editor and services. Section 6 details SWTBot end-to-end tests that validate complete user workflows on the product distribution. Section 7 describes build automation with Maven/Tycho, the packaging of Eclipse products and the p2 site, and our CI setup with GitHub Actions and SonarCloud. Section 8 reports an empirical analysis of a representative CI run and discusses practical implications, lessons learned and best practices, challenges/trade-offs/limitations, and portability and reuse across Eclipse/Xtext projects. Section 9 discusses related work. Section 10 concludes the paper.

2 Edelta

In this section, we summarize the main features of Edelta. We focus only on those aspects that are relevant to the contributions of this paper, referring the reader to previous publications for more details on specific aspects of the framework (Bettini et al. 2017, 2019, 2020; Barriga et al. 2021; Bettini et al. 2022a, b, 2024a, b).

Edelta is an open-source framework[2] designed for the refactoring, evolution, and co-evolution of EMF (Eclipse Modeling Framework) metamodels and models. The framework addresses the critical need for managing metamodel changes in a con-

[1] Throughout the paper, we use "Ecore models" and "metamodels" interchangeably; when we only mention "models", we refer to instances of metamodels.

[2] https://github.com/LorenzoBettini/edelta

trolled, safe, and automated manner while ensuring that existing models remain compatible with evolved metamodels. Edelta provides these main components: a runtime library, a domain-specific language (DSL), an integrated development environment (IDE) based on Eclipse with advanced features, and a library for refactorings and bad-smell detectors and resolvers. Figure 1 summarizes how these components fit together and how each is covered by a corresponding testing layer in our build and CI pipeline.

Edelta follows an operation-based approach where developers specify evolution operations at a high level, while the framework takes care of the low-level EMF manipulation and supports integrated model migration. In the MDE co-evolution landscape, this complements transformation- and pattern-centric techniques such as COPE (Herrmannsdoerfer et al. 2009) and aligns with best practices surveyed in the literature on model evolution and co-evolution (Paige et al. 2016; Hebig et al. 2017).

Edelta has been continuously developed and extended since its first introduction (Bettini et al. 2017, 2019, 2020; Barriga et al. 2021; Bettini et al. 2022a, b, 2024a, b).

2.1 Edelta runtime library

The Edelta runtime library provides the foundational Java API for manipulating Ecore metamodels and models on top of EMF. It implements the core evolution primitives used by both the DSL and the refactorings library (Fig. 1), and it supports co-evolution by allowing developers (or higher-level refactorings) to specify model

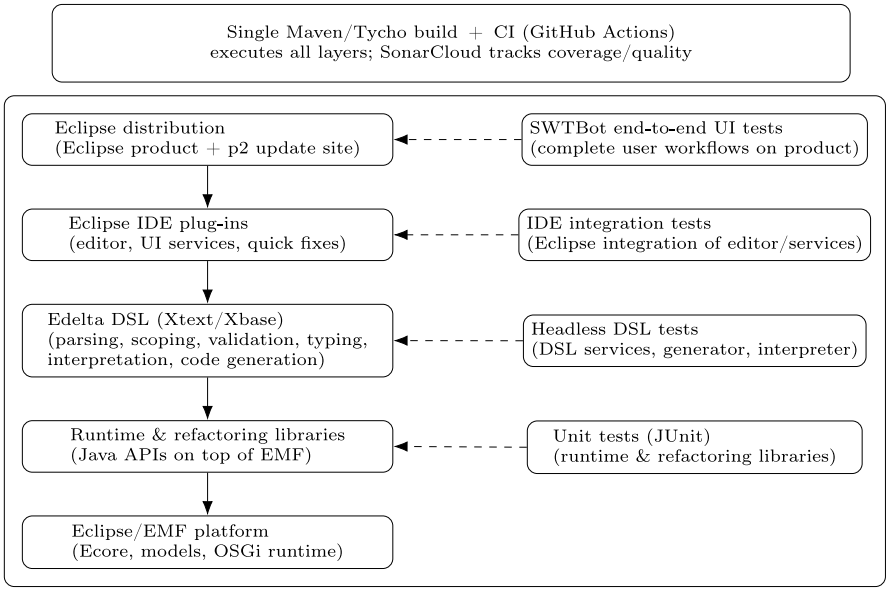


Fig. 1 High-level architecture of Edelta and correspondence to the multi-level testing strategy (the architecture is shown for Edelta but reflects a general toolchain structure)

migration steps alongside metamodel changes (Di Ruscio et al. 2012; Rose et al. 2012; Cicchetti et al. 2009; Wachsmuth 2007; Taentzer et al. 2013; Paige et al. 2016).

2.2 Edelta domain-specific language

The Edelta DSL provides a concise, Java-like syntax for specifying metamodel evolution operations. It is implemented with Xtext (Bettini 2016) and reuses Xbase (Efftinge et al. 2012) for expressions and seamless Java interoperability; for details of the Xbase expression language we refer the reader to the Xbase literature (Efftinge et al. 2012) and to previous Edelta publications (Bettini et al. 2017, 2020).

Listing 1 Example of an Edelta program (schematic).

```
1  metamodel "..."
2  metamodel "..."
3
4  migrate "http://www.mypackage.org/v1" to "http://www.mypackage.org/v2"
5  migrate "http://www.otherpackage.org/v1" to "http://www.otherpackage.org/v2"
6  ...
7  use ... as ...
8  use ... as ...
9  ...
10 def <name>(<... parameters ...>) : <returntype> { <body> }
11 def <name>(<... parameters ...>) : <returntype> { <body> }
12 ...
13 modifyEcore <name> epackage <EPackage name> { <body> }
14 modifyEcore <name> epackage <EPackage name> { <body> }
15 ...
```

At a high level, an Edelta program (Listing 1) imports one or more metamodels, optionally declares versioned migration pairs (via `migrate` Bettini et al. 2024b), and then specifies evolution operations inside `modifyEcore` blocks; reusable functions and libraries can be imported and invoked as needed.

A crucial feature of the Edelta DSL is the `ecoreref(...)` construct (we extended the Xbase expression syntax with this construct), which provides statically typed access to Ecore elements. This allows developers to refer directly to metamodel elements by their fully qualified names or simple names (when unambiguous) without requiring generated Java model code. The Edelta editor (Section 2.3) provides content assist for these references and can navigate directly to the corresponding Ecore elements.

For example, Listing 2 shows a simple refactoring operation that renames attributes (`firstname` and `lastname`) in a metamodel (`persons`) to use camel case (there is no need to specify the EClass containing the attributes since, in this case, `ecoreref` can resolve them unambiguously).

Listing 2 Edelta program that renames attributes to camel case.

```
1  modifyEcore renameAttributes epackage persons {
2      ecoreref(firstname).name = "firstName"
3      ecoreref(lastname).name = "lastName"
4  }
```

The Edelta compiler generates Java code that depends only on the Edelta runtime library and standard EMF APIs, and can therefore be executed headlessly.

Another distinctive aspect is *live interpretation* during validation: evolution operations are evaluated on an in-memory copy of the imported metamodels so that the compiler and the IDE can reason about the *current* (evolved) state while type checking and validating the program. This enables context-aware editor services (e.g., content assist and quick fixes) and supports UI features that reflect dynamically created/renamed elements.

For instance, newly created Ecore elements become immediately available for subsequent `ecoreref` expressions during validation, as shown in Listing 3. Since the compiler interprets operations during validation, it can correctly resolve the `ecoreref` references and verify that the last expression is correct, knowing that `ecoreref(ANewClass)` refers to an `EClass`.

Listing 3 Edelta program that creates a new abstract EClass.

```
1 modifyEcore createAndModify epackage mypackage {
2     addNewEClass("NewClass")
3     ecoreref(NewClass).abstract = true
4 }
```

2.3 Edelta integrated development environment

The Edelta IDE, built on Eclipse, provides an editor and IDE services for the DSL. The editor offers standard Xtext-based services (syntax highlighting, validation, content assist, navigation, and quick fixes) and integrates with the Eclipse workspace and OSGi runtime, which motivates IDE integration tests and end-to-end UI tests (Fig. 1).

The live-interpretation mechanism described above is used to make these services context-aware with respect to the evolving metamodel state, e.g., by proposing only elements that are valid at the current point in the program and by supporting quick fixes when references become stale due to evolution operations. An example quick fix is shown in Fig. 2.

For elements created dynamically during interpretation, we also customize hyperlinking so that navigation targets the Edelta expression that introduced the element (rather than an original Ecore declaration).

The IDE provides an Outline view summarizing both imported metamodel elements and dynamically introduced ones. An example is shown in Fig. 3.

Finally, the Edelta IDE provides a project wizard for creating correctly configured Edelta projects and launch contributions to run/debug Edelta files as Java applications directly from the Eclipse UI.

2.4 Edelta refactorings library

Edelta provides a refactorings library implementing a catalog of reusable metamodel evolution patterns and operators (Bettini et al. 2022b; Herrmannsdoerfer et al. 2010). These refactorings are packaged as reusable Edelta functions, typically including pre-

Fig. 2 The Edelta editor: the quick fix for a reference after an element has been renamed

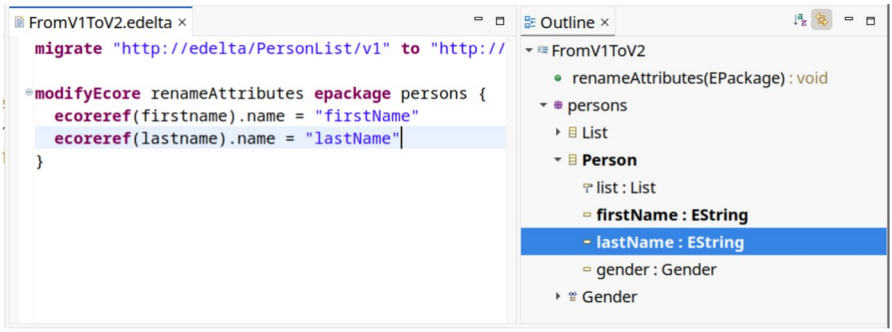


Fig. 3 The Edelta editor and the outline view showing the evolved metamodel structure

condition checks to ensure safe application, and they can also prescribe corresponding model migration steps to enable seamless co-evolution. In addition, the library integrates metamodel quality assessment through bad smell detection and resolution capabilities (Bettini et al. 2019), which can be invoked from Edelta programs and tested at the library level (Fig. 1).

3 A general testing and CI framework for Eclipse/Xtext toolchains

Eclipse/Xtext-based language toolchains typically comprise multiple layers that must work together: (i) *runtime and domain libraries* (plain Java), (ii) *language services* (parsing, linking/scoping, validation, typing, interpretation and/or code generation), (iii) *IDE integration* (Eclipse/OSGi plug-ins implementing editor and IDE services), and (iv) *packaging and distribution* (Eclipse products and p2 update sites). A central challenge is to obtain high confidence in correctness across these layers while keeping builds reproducible and CI-friendly across platforms. This section summarizes a

generalizable framework that structures testing and automation for such toolchains; Sections 4–7 then show how we instantiate it for our case study.

3.1 Principles: Layer the tests, keep fast feedback dominant

Our approach follows the testing-pyramid principle (Cohn 2009; Fowler 2013): maximize the proportion of fast, deterministic tests at the lowest layers and reserve UI-driven end-to-end tests for a small number of critical user workflows. Concretely:

- **Test lowest layers first:** validate runtime/domain libraries with unit tests (JUnit) and treat these as the primary regression net.
- **Test language services headlessly:** test parsing, linking/scoping, validation, typing, interpretation and code generation using Xtext's JUnit-based testing infrastructure and dependency injection (Prasanna 2009; Bettini 2016) in a headless environment (outside the Eclipse UI/workbench).
- **Add IDE integration tests selectively:** test Eclipse workbench integration and editor services where a running IDE context is required (e.g., workspace/build, content assist, hyperlinking, quick fixes, outline view).
- **Minimize end-to-end UI tests:** keep only a small set of SWTBot tests that cover essential workflows that cannot be validated reliably at lower layers (e.g., project creation wizards, launch configurations).

Test-driven development (TDD) (Beck 2003; Maximilien and Williams 2003; Williams et al. 2003; Desai et al. 2008; Cauevic et al. 2012; Munir et al. 2014) complements this pyramid by ensuring that new features and bug fixes are introduced together with executable regression tests.

3.2 Mapping toolchain components to test layers

A practical way to operationalize the framework is to explicitly map *each deliverable component* to its primary test layer(s). Figure 1 (Section 2) illustrates this mapping for our toolchain, but the same structure applies to similar Eclipse/Xtext projects:

- **Runtime and domain libraries:** unit tests focus on API contracts and transformation/migration effects on in-memory models; when the output is structured (e.g., generated models/code), golden-file comparisons can serve as an oracle.
- **Language services (headless):** JUnit tests exercise the Xtext pipeline end-to-end in a headless environment: parse inputs, assert linking/validation diagnostics, check typing/type inference, and validate generator/interpreter outcomes.
- **IDE plug-ins:** UI integration tests verify that editor services behave correctly inside the Eclipse workbench (workspace build integration, content assist proposals, hyperlink targets, quick-fix availability/application, outline structure).
- **Product and distribution artifacts:** build-level checks verify that the produced Eclipse product and p2 update site are complete and installable/reproducible, and that they embed the expected plug-ins and dependencies.

This explicit mapping is useful both for communication (it clarifies what is being validated at which level) and for maintenance (it prevents UI tests from growing into a catch-all for problems better addressed by unit/headless tests).

3.3 Test oracles and test data management

Multi-level testing requires choosing appropriate *oracles* (i.e., how to decide whether a test passed Barr et al. 2015) and managing test data so that tests remain readable and stable.

Oracles by layer At the library level, assertions are typically structural (e.g., properties of an in-memory model after applying an operation). For language services, oracles include expected diagnostics (errors/warnings), expected resolved references (linking/scoping), expected inferred types, and expected generated artifacts. For generators and migration pipelines, golden outputs (expected files) can be used when stable; when introducing a new expected output, starting from an empty baseline helps ensure that the first run fails and the produced artifacts are reviewed before being committed as expected results. For IDE tests, oracles typically assert the presence/shape of editor services (proposal sets, hyperlink targets, available quick fixes, outline labels) rather than pixel-perfect UI state, to reduce fragility.

Shared test data When multiple test suites need the same inputs and expected outputs (e.g., library tests and DSL tests), placing them in a shared test-data module avoids duplication and helps keep tests consistent across layers. Conversely, when tests mainly need short language snippets, embedding them as strings (e.g., Java text blocks) keeps tests self-contained and readable.

3.4 Build integration and CI execution

For Eclipse-based toolchains, the build is part of the quality strategy: it must "prove" that the toolchain is reproducible, installable, and testable.

Single build that runs all layers A key practice is to orchestrate all test layers in a unified Maven/Tycho build: unit tests for plain Java modules, headless Xtext tests for language services, IDE integration tests for UI plug-ins, and SWTBot tests for end-to-end workflows. Keeping the build unified ensures that CI continuously validates the same artifact structure that users receive, and it prevents divergence between "developer builds" and "release builds."

CI as cross-platform verification Running the build on a cross-platform CI matrix increases confidence in Eclipse/OSGi dependency resolution and in UI behavior across operating systems. CI should (i) execute the full test pyramid, (ii) publish actionable logs/artifacts for debugging failures, and (iii) integrate quality tracking (e.g., coverage/quality trends) as continuous feedback rather than a one-time release check. In practice, fast unit and headless language tests are typically part of the developer's local workflow, whereas IDE integration and SWTBot end-to-end tests are

slower; therefore, the CI pipeline provides the authoritative cross-platform execution point for these heavier tests, ensuring that UI-level regressions are detected before artifacts are considered releasable.

3.5 How the framework is instantiated in our case study

Sections 4–7 apply this framework to our case study. We start from unit tests for runtime and refactoring libraries (Section 4.1), then cover headless DSL tests for language services (Section 4.2), proceed with IDE integration tests for editor services (Section 5), and finally report a small set of SWTBot tests for critical end-to-end workflows (Section 6), all executed within a single Maven/Tycho build and CI pipeline (Section 7). In each subsequent section, we first state the general testing intent and then show how we implement it in our toolchain.

4 Unit tests

In this section, we describe the unit tests we have developed to get confidence on the correctness and reliability of both the Edelta runtime and the Edelta DSL. These tests cover a wide range of scenarios, from basic functionality to complex transformations. We will show a few representative examples of the tests we have implemented, highlighting the key aspects of our testing strategy, which we believe can be useful for other developers working on similar projects.

In many of the code snippets we show in this section, we omit some details for brevity, such as import statements, class declarations, and setup/teardown methods. The complete test classes can be found in the Edelta source code repository. We focus on the most relevant parts of each test, highlighting the key assertions and operations that demonstrate the testing strategies employed. Moreover, we use many utility functions of our testing library (which is also part of the Edelta framework) to simplify test creation, assertions, and improve readability.

For writing our tests, we have followed a test-driven development (TDD) approach (Beck 2003), where we write a failing test before implementing new features or fixing bugs. This is crucial to have confidence in the quality of our tests, ensure that they cover the intended functionality and rule out false positives (Maximilien and Williams 2003; Williams et al. 2003; Desai et al. 2008; Cauevic et al. 2012; Munir et al. 2014). More generally, we adopt TDD because it is an effective unit-testing methodology independent of the specific domain, helping to drive test design from observable requirements and to prevent regressions via executable specifications (Beck 2003; Maximilien and Williams 2003; Williams et al. 2003; Desai et al. 2008; Cauevic et al. 2012; Munir et al. 2014). In our context, TDD is also particularly suitable because much of the critical behavior of an Eclipse/Xtext toolchain resides in fast, deterministic components (runtime/refactoring libraries and headless language services), where a failing-test-first workflow provides quick feedback and keeps the base of the testing pyramid strong, reducing reliance on slower and more fragile UI-level tests. Finally, many defects manifest as subtle corner cases in metamodel/model structures or in language service interactions; capturing a minimal reproducer

as a regression test before implementing the fix helps ensure that the issue is fully understood and that the fix is effective.

4.1 Edelta runtime unit tests

First, we must ensure that the Edelta runtime, based on EMF, is functioning correctly. To achieve this, we have developed a suite of unit tests that cover the core mechanisms of the runtime. Moreover, we also test the refactorings library (including bad smells detection and resolution) to ensure that the provided operations work as intended. These unit tests are located in the `edelta.lib.tests` and `edelta.refactorings.lib.tests` projects, which only test the Edelta runtime library (Section 2.1) and the Edelta refactorings library (Section 2.4), respectively.

Testing the base runtime behavior includes exercising package loading, model element retrieval, and error handling mechanisms. The tests verify that the runtime can correctly load Ecore files, retrieve packages and classifiers by name, and handle inheritance hierarchies including subpackages. They focus on model transformation operations, covering both metamodel refactoring and model migration scenarios. The tests validate operations for adding new model elements (classes, attributes, references, enums), copying and moving elements between containers, and establishing relationships like inheritance and opposites. Such tests can be implemented by creating in-memory Ecore models, applying transformation operations, and then asserting that the resulting models have the expected structure and properties.

Listing 4 shows an example of a test that verifies the behavior of our library method for removing an element like an EClass: the standard EMF API does not automatically clean up cross-references to the removed element, while our library method does.

Listing 4 Edelta Runtime Unit Test Example

```
1  @Test
2  public void test_removeElement() {
3      var superClass = ecoreFactory.createEClass();
4      var subClass = ecoreFactory.createEClass();
5      subClass.getESuperTypes().add(superClass);
6
7      EdeltaUtils.removeElement(superClass);
8      // Verify references to removed class are cleaned up
9      assertThat(subClass.getESuperTypes()).isEmpty();
10 }
```

More sophisticated tests cover complete model migration workflows, such as renaming classes, changing attribute types with custom value transformations, and modifying reference types while maintaining model consistency. These tests load test models files, apply transformations, and then verify the results against expected output files, ensuring that both the metamodel changes and the corresponding model instance migrations are correctly executed. An example of such a test is shown in Listing 5.

Listing 5 Edelta Runtime Unit Test Example for Model Migration

```
1 @Test
2 public void testCreationAndExecution() throws Exception {
3     var engine = new EdeltaEngine(/*... omitted for brevity ...*/);
4     engine.loadEcoreFile(TESTDATA + SIMPLE_TEST_DATA + MY_ECORE);
5     engine.loadModelFile(TESTDATA + SIMPLE_TEST_DATA + MY_CLASS);
6     engine.execute();
7     engine.save(OUTPUT + subdir);
8     assertGeneratedFiles(subdir, MY_ECORE, MY_CLASS);
9 }
```

We have a common project with shared test data: this contains input Ecore and model files, as well as expected output files for comparison after transformations are applied. Many of these input and expected output files are shared by other test projects, including the Edelta DSL tests, thus, having them in a common location avoids duplication and ensures consistency across tests.

When writing such tests, we start with empty expected output files, so that initially the test fails, and then we can inspect the generated files to verify they are correct before committing them as the expected results.

Summarizing, for testing simple operations, we create in-memory Ecore models and apply operations, and then, we assert the resulting model structure. Such assertions are simple. Typically, a single assertion is sufficient to verify the expected behavior, as in Listing 4. Several assertions might be needed for slightly more complex scenarios, but they remain simple and limited in number. For testing complete model migration workflows, we load input files, apply transformations, save the results, and compare them against expected output files. For such tests, writing assertions on the structure of the resulting models would be much more complex and error-prone. Moreover, they would make the tests less readable and maintainable.[3] Note that examining the XMI files directly is not a significant issue for small metamodels and models, as they are generally readable and understandable. For larger models, we can use the EMF Reflective editor to open and visually inspect them. Thus, we find it more practical and effective to verify the correctness of the generated XMI files by checking the actual output against expected results since this can capture a broader range of potential issues than structure-based and property-based checks alone.

It is worth emphasizing that we carefully craft our test input metamodels and models to thoroughly exercise the Edelta runtime library, including both the core operations and the refactorings library. These inputs are designed not only to test typical usage scenarios but also to cover edge cases and corner cases, such as metamodels with complex inheritance hierarchies, bidirectional references, and deeply nested containment structures. By systematically testing against such carefully designed inputs (covering a wide range of metamodel and model configurations), by applying TDD, and by checking code coverage (Section 7.6), we gain confidence that our

[3] In the future, we might consider using OCL (Warmer and Kleppe 2003) to write more concise assertions on the structure of the resulting models, but we have not explored this option yet. Whether assertions with Java or OCL can be generated automatically from Edelta programs is an interesting research question that we leave for future work.

runtime library works correctly. Creating such input metamodels and models with the EMF Reflective editor is straightforward and efficient, allowing us to set up the necessary test scenarios quickly. In particular, with such an editor's tooling, we can easily validate that the created metamodels and models are well-formed according to EMF constraints before using them in our tests. Note that these input metamodels and models are minimal since they only need to include the elements necessary to test the specific features and scenarios we want to cover. Due to the shape and structure of Ecore metamodels and models, as well as the existing EMF tooling, creating such minimal test inputs is not a significant challenge. For these reasons, we did not feel the need to generate input data with automatic tools and techniques (see, e.g., Wu et al. 2012; Ehrig et al. 2009; Bertolino et al. 2007; Nassar et al. 2020).

4.2 Edelta DSL unit tests

Next, we need to ensure that the Edelta DSL itself is functioning correctly. To achieve this, we have developed a suite of unit tests that cover the core functionalities of the DSL. The Edelta DSL unit tests are located in the `edelta.tests` project and demonstrate all these testing strategies in practice.

When developing a Domain-Specific Language (DSL) with Xtext, several key components require thorough testing to ensure the language behaves correctly. The main strategies for unit testing Xtext DSLs focus on testing the different phases of language processing: parsing (Section 4.2.1), scoping and linking (Section 4.2.2), validation (Section 4.2.3), and code generation (Section 4.2.4). Besides, many DSLs, including Edelta, implement additional components that also require testing. In particular, since the Edelta DSL relies on Xbase (Efftinge et al. 2012), we also need to test the *type system* to ensure that types are correctly inferred and checked according to the language semantics (Section 4.2.5). Moreover, since the Edelta DSL uses an *interpreter* to execute metamodel transformations on-the-fly during compilation, we also need to test the interpreter to ensure that it correctly applies the transformations defined in the DSL programs (Section 4.2.6).

Xtext itself provides a robust testing framework that integrates with JUnit, making it straightforward to write and run tests for these components (Bettini 2016). A key advantage of Xtext's architecture is its extensive use of *dependency injection* (Prasanna 2009), which greatly facilitates testing. Xtext components are wired together using Google Guice, allowing test classes to easily inject the specific services they need to test. This dependency injection mechanism works seamlessly in test environments, enabling fine-grained testing of language services without requiring a full Eclipse runtime. In fact, all the mechanisms mentioned above (parsing, linking, validation, code generation) can be tested in a headless environment, without the need for a running Eclipse IDE. This makes the tests fast and reliable, as they do not depend on external factors like UI components or workspace state. Of course, we will also need to test the IDE integration of the DSL (Section 5).

Although these tests exercise multiple language components working together (parser, linker, validator, generator), they can still be considered *unit tests* rather than integration tests. This is because they test small, focused aspects of the language that do not require a running Eclipse IDE. Each test typically focuses on a specific

language feature or scenario, making them effective for rapid feedback during development and regression testing. These DSL tests can be run quickly and frequently, though they are a bit slower than typical unit tests due to the multiple components involved (Section 8.2). We must also consider that all the DSL components depend on each other, so it is not always possible to test a single component in complete isolation. Therefore, a bug in one component (e.g., the parser) can affect the behavior, and thus the tests, of other components (e.g., the validator).

In the following, we provide representative examples of each of these testing strategies as applied to the Edelta DSL. As for the previous unit tests, we omit some details for brevity, focusing on the most relevant parts of each test. Moreover, we use many utility functions of our testing library to simplify test creation, assertions, and improve readability. In general, for the tests of the DSL, instead of input files, we use input strings that contain the DSL code to be tested by relying on Java text blocks for better readability. Similarly, for expected outputs. This should make it easier to understand the tests without having to look for external files. We extensively use our utility function `parseWithTestEcore(...)` to parse Edelta programs that reference test Ecore metamodels. This function automatically loads the test Ecore metamodels into the resource set used for parsing, so that the tests can focus on the Edelta code without worrying about loading the referenced metamodels. We also use other utility functions to navigate the AST and retrieve relevant nodes for testing. For example, `getLastEcoreReferenceExpression(program)` retrieves the AST node corresponding to the last `ecoreref(...)` expression in the parsed program. In fact, many tests perform assertions and verifications on the last `ecoreref(...)` expression and its argument, as it is often the most relevant for most test scenarios (parsing, scoping, validation, and type system tests).

4.2.1 Parser testing

The first strategy involves testing the *parser* to ensure that the grammar correctly recognizes valid input and rejects invalid syntax. Parser tests typically load DSL programs as strings or files and verify that the resulting Abstract Syntax Tree (AST) has the expected structure. Moreover, it is crucial to verify that the structure and shape of the AST are as expected, since subsequent phases (linking, validation, code generation) depend on a correct AST.

Since our grammar inherits from Xbase the Java-like syntax, we mostly focus on testing the Edelta-specific constructs, relying on the correctness of the Xbase constructs. In particular, we extended the Xbase syntax with new expressions for referencing Ecore elements (`ecoreref(...)`), which are crucial for the Edelta DSL (Section 2.2). Thus, we must ensure the parts of the reference are correctly parsed with the correct precedence. For example, the test in Listing 6 verifies that a qualified Ecore reference like `foo.FooClass` is correctly parsed into its qualification (`foo`) and its element (`FooClass`). The test parses a minimal Edelta program containing a `modifyEcore` operation with an `ecoreref(...)` expression. Then, it retrieves the AST node corresponding to the last `ecoreref(...)` expression and extracts its argument, which should be an instance of `EdeltaEcoreQualifiedArgument`. Finally, it asserts that the qualification and name parts of the qualified reference are as expected (i.e., like in Fig. 4).

Listing 6 Edelta DSL Parser Test Example

```
@Test
public void testQualifiedEcoreReference() throws Exception {
    var prog = parseWithTestEcore("""
          metamodel "foo"
          modifyEcore aTest epackage foo {
              ecoreref(foo.FooClass)
          }
      """);
    var arg = (EdeltaEcoreQualifiedArgument)
        getLastEcoreReferenceExpression(prog).getArgument();
    assertEquals("foo", arg.getQualification().getElement().getName());
    assertEquals("FooClass", arg.getElement().getName());
}
```

Besides a few more tests for the other Edelta-specific constructs, e.g., for importing metamodels and defining `modifyEcore` operations, we do not need many parser tests, relying on the correctness of the Xbase grammar concerning its expressions.

4.2.2 Scoping and linking testing

The second strategy focuses on testing the *scoping and linking* mechanisms. Scoping tests verify that references to elements (such as variables, types, or imported modules) are resolved to the correct target elements within the appropriate scope. In Xtext, scoping is delegated to an implementation of a scope provider that defines which elements are visible at a given point in the DSL program. The scope provider main method is

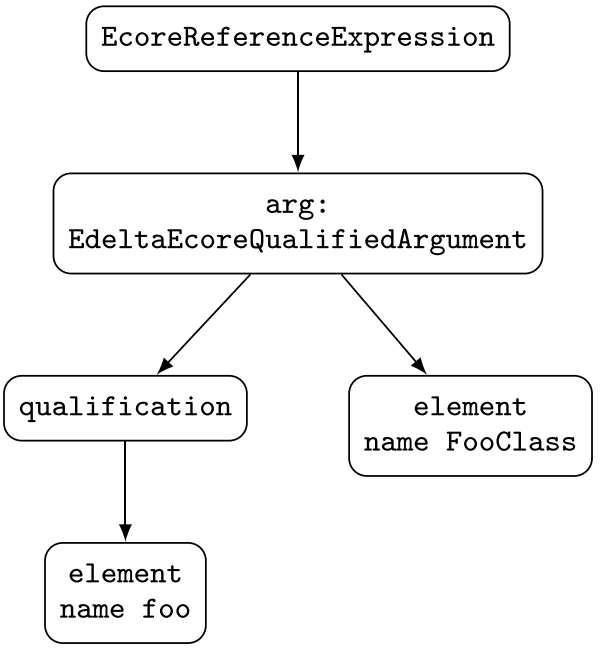


Fig. 4 Subtree for `ecoreref(foo.FooClass)`

`getScope(EObject context, EReference reference)` that, given a context and a reference, returns the collection of candidate elements that can be referenced from that context. By using the scope provider, the Xtext's linking mechanism automatically resolves cross-references by looking up names in the appropriate scopes, matching them against the available candidates. Typically, as in our case, customizing the scope provider is sufficient to implement the desired name resolution behavior: the default linking mechanism does not need to be customized, as it relies on the scopes provided.

Again, due to the use of Xbase, we can assume that all the references to Java types are correctly resolved, so we mostly focus on testing the Edelta-specific references to Ecore model elements (`ecoreref`). As detailed above, the scope provider must return the correct set of candidate elements for a given context in the input program. In these tests, the verification is done by checking the names of the elements in the returned scope.

More complex scoping scenarios test qualified references to Ecore elements created by the interpreter during the execution of `modifyEcore` operations are correctly resolved. For example, the test in Listing 7 verifies that after renaming an EClass within a `modifyEcore` operation, the new name becomes available in the scope for subsequent references; it also verifies that, in that program context, the name of the original renamed class is not available anymore. We use several utility functions to navigate the AST and retrieve the relevant node for testing the scope of its reference. The `assertScope(...)` utility function verifies that the scope of the given context (first argument) and reference (second argument) contains exactly the expected elements (passed as the last argument as a string with one name per line).

Listing 7 Edelta DSL Scoping Test Example

```
@Test
public void testScopeForRenamedEClassInModifyEcore() throws Exception {
    assertScope(
        // given the last ecoreref expression in the modifyEcore
        // i.e., ecoreref(foo.RenamedClass) ...
        getEdeltaEcoreReferenceExpression(
            getLastModifyEcoreOperationLastExpression(
                parseWithTestEcore("""
                        metamodel "foo"
                        metamodel "bar"
                        modifyEcore aTest epackage foo {
                            ecoreref(foo.FooClass).name = "RenamedClass"
                            ecoreref(foo.RenamedClass)
                        }
                        """)
                )
            ).getReference(),
        // ... if we compute the scope of its argument ...
        getEcoreReferenceArgumentReference(),
        // ... these should be the available elements:
        """
            RenamedClass
            FooDataType
            FooEnum
            """);
    // Note: the RenamedClass is there (so we can resolve it),
    // while the original FooClass is not in this program's context anymore
}
```

If we test the correctness of the scope provider, we can rely on Xtext's linking mechanism to correctly resolve the reference. Thus, we do not need to write specific tests for the linking phase.

4.2.3 Validation testing

The third strategy involves testing the *validation* rules that enforce the semantic constraints of the language. Validation tests ensure that the DSL correctly identifies semantic errors, warnings, and information messages according to the language specification. These tests typically check that valid programs produce no validation errors, while invalid programs generate the expected error messages at the correct locations. The Xtext testing framework provides a validation test helper, `ValidationTestHelper`, which simplifies the assertions in validation tests. This helper allows us to easily check for the presence of validation errors and warnings on specific AST nodes with several overloaded `assertError` and `assertWarning` methods. Similarly, it allows us to verify that no validation issues are present in valid programs with methods `assertNoErrors` and `assertNoWarnings`. It is important, for a nice user experience, that the error is shown at the correct location, so that the developer can easily understand and fix the problem. (The location will be used by the IDE to highlight the error in the editor and show the error message in the problems view.) For instance, the test in Listing 8 verifies that duplicate metamodel imports are correctly detected and reported as validation errors. In particular, it verifies that the error message is shown at the correct location in the input program, that is, only the duplicate import string `"foo"` is marked as erroneous, not the entire line.

Listing 8 Edelta DSL Validation Test Example

```
1  @Test
2  public void testDuplicateMetamodelImport() throws Exception {
3      var input = """
4        metamodel "foo"
5        metamodel "bar"
6        metamodel "foo"
7        """;
8      var prog = parseWithTestEcore(input);
9      validationTestHelper.assertError(prog,
10             prog.eClass(),
11             EdeltaValidator.DUPLICATE_METAMODEL_IMPORT,
12             input.lastIndexOf("\"foo\""), "\"foo\"".length(),
13             "Duplicate metamodel import \"foo\"");
14 }
```

The `assertError` method, in this version of one of its overloaded forms, requires six arguments: the first is the root AST node of the parsed program, the second is the type of the AST node (specified as an `EClass`) where we expect the error to be reported (in this case, the root node of the program), the third is the error code defined in the validator class, the fourth and fifth specify the location of the error in the input string (offset and length), and the sixth is the expected error message.

We also have validation tests that verify the validity of programs that refer to Ecore elements created by the interpreter during the execution of `modifyEcore` operations. While this looks like a repetition of the scoping tests, it is not, because the validation tests verify that the program is valid according to the language semantics, while the scoping tests only verify that the scope provider returns the correct set of candidate elements for a given context.

Moreover, due to the dynamic nature of the Edelta DSL, where metamodel elements can be created and modified (i.e., renamed or removed) during the execution of `modifyEcore` operations, we implemented custom validation rules to provide meaningful and useful error messages when the program attempts to reference elements that do not exist anymore (e.g., after a rename operation) or that are not yet created at that point in the program. Our validation tests verify that these custom rules work as intended and that the error messages are properly displayed at the correct locations. An example of such a test is shown in Listing 9, which verifies that referencing an EClass that has been removed earlier in the same `modifyEcore` operation produces the expected validation error. Differently from the previous test (Listing 8), here the error is expected to be reported on the `EdeltaEcoreReferenceExpression` AST node corresponding to the `ecoreref(...)` expression, as it is the expression that attempts to access a removed element. Its type (`EClass`) is obtained from the Edelta package literals, which is the standard way to reference Ecore meta-objects in EMF-based generated code.

Listing 9 Edelta DSL Validation Test Example for Removed Ecore Element

```
@Test
public void testReferenceToEClassRemoved() throws Exception {
    var input = """
      metamodel "foo"

      modifyEcore aTest epackage foo {
          EClassifiers -= ecoreref(FooClass)
          ecoreref(FooClass).abstract // this doesn't exist anymore
      }
      """;
    var prog = parseWithTestEcore(input);
    validationTestHelper.assertError(prog,
        EdeltaPackage.Literals.EDELTA_ECORE_REFERENCE_EXPRESSION,
        EdeltaValidator.INTERPRETER_ACCESS_REMOVED_ELEMENT,
        input.lastIndexOf("FooClass"), "FooClass".length(),
        "The element is not available anymore in this context: \'FooClass\'");
}
```

4.2.4 Code generation testing

The fourth strategy covers testing the *code generator* that transforms DSL programs into executable artifacts, in this case, Java code. Such tests typically consist in comparing generated files against expected outputs or executing the generated code to verify its behavior. Of course, we must also ensure that the generated Java code is valid Java code. In particular, the Xtext testing infrastructure provides support for compiling the generated Java code, instantiating reflectively and running it as part of the test suite, allowing for end-to-end verification of the DSL's functionality.

These tests compile complete Edelta programs and compare the generated Java code against expected outputs (in the input program, `modelMigration` defines the model migration rules associated with the metamodel transformation defined in the `modifyEcore` block). An example of such a test is shown in Listing 10.

Listing 10 Edelta DSL Code Generation Test Example

```
@Test
public void testCompilationOfModelMigration() throws Exception {
    checkCompilation("""
      package foo;

      metamodel "foo"

      modifyEcore aTest epackage foo {
          val fooClass = ecoreref(FooClass)
          val mySubClass = fooClass.addNewSubclass("MySubClass")
          fooClass.abstract = true
          modelMigration[
              createInstanceRule(
                  isRelatedTo(fooClass),
                  [oldObj|
                      return createInstance(mySubClass)
                  ]
              )
          ]
      }
      """, /* expected Java code omitted for brevity */);
}
```

The `checkCompilation` method not only verifies that valid Java code is generated as expected but also compiles it with the Java compiler to ensure its validity.

Checking the generated code against precise expected outputs is known to require a maintenance effort, as even minor changes in the code generation logic can lead to significant differences in the generated code. Even formatting changes can cause tests to fail, making maintenance cumbersome. However, having full control on the generated code (which, in the end, is the code that will be effectively executed) is crucial for our use case, and in general for DSLs that generate code to be integrated into larger projects. Thus, we accept this maintenance effort. Moreover, such changes in the generated code are relatively infrequent, as the code generation logic is quite stable once the DSL is mature. Finally, our choice to test the code generator by checking its output against a precomputed expected result follows established practice in both textbook treatments of compiler correctness (Aho et al. 2006; Fraser and Hanson 1995) and empirical studies of compiler testing (Sheridan 2007; Chen et al. 2020; Tang et al. 2020).

In other tests related to code generation, which we do not show here for brevity, we also instantiate the compiled Java code and execute it to verify its behavior. We verify that the generated code, when executed, produces the expected results, such as correctly transforming metamodels and migrating models according to the defined rules.

4.2.5 Type system testing

Besides the grammar for Java-like syntax, Xbase also provides a powerful type system that can be extended to support custom types and type inference rules. Such a

type system takes care of type checking and type inference for expressions according to the Java type system. Since we introduced new expressions for referencing Ecore elements, we also need to extend the Xbase type system to support these new expressions. For example, the `ecoreref` expression should have the Java type of the referenced Ecore element (e.g., `EClass`, `EAttribute`, etc.). Type computation tests verify that the Edelta type system correctly infers types for `ecoreref` expressions, especially for dynamically created or modified metamodel elements. An example of such a test is shown in Listing 11, which verifies that after renaming an EClass within a `modifyEcore` operation, the type system correctly infers the type of a subsequent `ecoreref` expression that references the renamed class as `EClass`.

Listing 11 Edelta DSL Type System Test Example

```
@Test
public void testTypeForRenamedEClassInModifyEcore() throws Exception {
    var prog = parseWithTestEcore(
        """
                metamodel "foo"

                modifyEcore aTest epackage foo {
                    ecoreref(foo.FooClass).name = "RenamedClass"
                    ecoreref(RenamedClass)
                }
            """
    );
    var ecoreRefExp = getLastEcoreReferenceExpression(prog);
    assertType(ecoreRefExp, EClass.class);
}
```

If we test the correctness of the type system with our custom implementation, we can rely on Xbase's type system to ensure that types are correctly inferred and checked according to the Java type system. Moreover, we let Xbase handle type errors, e.g., when an expression has an incompatible type, so we do not need to write specific tests for type errors.

4.2.6 Interpreter testing

Interpreter tests verify that executing Edelta programs produces the expected metamodel modifications *in memory*. Concretely, our test harness (i) parses a small Edelta snippet, (ii) runs it through the interpreter, and (iii) passes the resulting `EPackage` (an in-memory copy of the referenced metamodel) to a lambda expression for assertions. These tests are complementary to scoping/validation tests: since Edelta uses interpretation to create or modify Ecore elements that are then consumed by other language services, interpreter regressions can otherwise surface indirectly as failures in later components.

Listing 12 shows a representative example. The interpreted `modifyEcore` block creates a new `EClass` and adds it to the in-memory `EPackage`; the assertions check that the class was created with the expected name.

Listing 12 Edelta DSL Interpreter Test Example

```
@Test
public void testCreateEClass() throws Exception {
    // Parse + interpret the snippet below on an in-memory copy of the "foo" EPackage
    assertAfterInterpretationOfEdeltaModifyEcoreOperation("""
      metamodel "foo"

      modifyEcore aTest epackage foo {
          addNewEClass("NewClass") // add a new EClass to the package
      }
      """,
    // Pass the in-memory EPackage to the lambda expression for assertions
    ePackage -> {
        // Verify that the last EClassifier is the newly created EClass
        var classifiers = ePackage.getEClassifiers();
        var last = (EClass) classifiers.get(classifiers.size() - 1);
        assertEquals("NewClass", last.getName());
    });
}
```

Note that all the interpreter tests also ensure that the interpreter effectively acts on an in-memory copy of the referenced Ecore metamodels, leaving the original files unchanged. This is crucial to avoid side effects that could affect subsequent tests or the developer's workspace (e.g., through unintended writes to the original metamodel resources or global EMF state). To check that, in interpreter tests we attach an EMF resource set listener that fails the test if any change is inadvertently made to the resources containing the original Ecore metamodels. This allowed us to catch several bugs in the first versions of the interpreter implementation and will keep protecting us from future regressions.

5 User interface tests

Beyond the DSL unit tests that focus on the core language components, we also have some UI integration tests that verify the Eclipse-based development experience. These tests ensure that the DSL works correctly within the Eclipse IDE environment, including features like workspace integration, content assist, hyperlinking, quick fixes, and outline views.

The UI tests are located in the `edelta.ui.tests` project and leverage Xtext's built-in UI testing framework, which provides utilities and base classes for testing Eclipse UI components without requiring manual interaction. Thus, these tests remain automated and can be executed as part of the continuous integration pipeline (Section 7). As in Section 4, we use many utility functions of our testing library (which is also part of the Edelta framework) to simplify test creation, assertions, and improve readability.

In particular, in our UI test utilities, we provide methods to programmatically import Eclipse projects. The Xtext testing framework provides methods to create projects programmatically, but we found that importing existing projects from the file system is more convenient. This approach allows us to easily create Eclipse test projects manually, with all the necessary configurations, and then reuse them across multiple UI tests as these test projects are part of our Git repository. On the contrary, creating projects programmatically is error-prone and would require much more effort to correctly create the project structure, configurations, and content, generating friction while writing tests.

5.1 Workbench integration testing

Workbench integration tests verify that Edelta files can be created, edited, and built correctly within Eclipse projects.

For example, the test in Listing 13 creates an Edelta file in an Eclipse project (previously imported with the mechanisms described above), triggers a build, verifies that no compilation errors occur, and confirms that the expected Java code is generated in the appropriate source generation folder. In the test, we use some utility methods provided by the Xtext testing framework (e.g., `waitForBuild`) as well as some of our own utilities (e.g., `assertNoErrors`, which retrieves Eclipse markers and checks for errors, and `assertSrcGenFolderFile`, which verifies the presence of a generated file in the source generation folder). The test validates the entire toolchain from DSL source to generated artifacts, and that everything compiles without error (including generated Java code).

Listing 13 Edelta UI Test example: integration test for a valid Edelta project in Eclipse.

```
1  @Test
2  public void testValidProject() throws Exception {
3      createFile(
4          TEST_PROJECT + "/src/Test.edelta",
5          """
6              package foo
7
8              metamodel "mypackage"
9
10             modifyEcore aTest epackage mypackage {
11                 ecoreref(MyClass)
12             }
13             """);
14     waitForBuild();
15     PluginProjectHelper.assertNoErrors();
16     assertSrcGenFolderFile("foo", "Test.java");
17 }
```

We do not need many tests for workspace integration because most of the functionality is already covered by DSL unit tests (Section 4.2). However, having a few of such tests helps increasing the confidence that the integration between the Edelta language services and the Eclipse workspace functions correctly, as some issues may arise only in the full IDE context.

5.2 Content assist testing

Content assist tests verify that the editor provides appropriate code completion suggestions based on the current program context. These tests are particularly important for DSLs like Edelta where the available completion options depend on dynamic metamodel state based on the on-the-fly interpretation of transformation operations.

For example, the test in Listing 14 verifies that after renaming an `EClass` in an Edelta transformation, the content assist suggests the new name and not the old one in the subsequent `ecoreref` expressions. The assert methods provided by the Xtext testing framework allow us to specify the expected proposals at the cursor position

represented by the <|> marker. The marker is only used in the test input string to indicate where content assist should be invoked; it is not part of the actual Edelta code.

Listing 14 Edelta UI Test example: content assist test for dynamically renamed EClass.

```
@Test
public void testEClassifierAfterRenamingAnEClass() throws Exception {
    var proposalTester = newBuilder().append(
        """
            metamodel "mypackage"

            modifyEcore aTest epackage mypackage {
                ecoreref(MyClass).name = "Renamed"
                ecoreref(<|>)
            """);
    proposalTester.assertProposalAtCursor("Renamed");
    proposalTester.assertNoProposalAtCursor("MyClass");
}
```

We also have tests that verify that newly created elements are proposed by content assist and removed elements are not proposed anymore. These tests ensure that the content assist is context-aware and reflects the current state of the metamodel as modified by the Edelta transformations during the interpretation.

5.3 Hyperlinking testing

Hyperlinking tests verify that users can navigate from references to their declarations by clicking on elements in the editor. This functionality is crucial for understanding metamodel transformations.

For elements created dynamically during the interpretation of Edelta operations, we customized the hyperlinking mechanism to point to the Edelta expression that created the element (Section 2.3). The test in Listing 15 verifies that the hyperlink on the `ecoreref` argument (the dynamically created EClass `NewClass`) points to the `addNewEClass` expression that created it.

Listing 15 Edelta UI Test example: hyperlinking test for dynamically created EClass.

```
@Test
public void hyperlinkOnCreatedEClass() throws Exception {
    hasHyperlinkTo("""
            metamodel "mypackage"

            modifyEcore aTest epackage mypackage {
                addNewEClass("NewClass")
                ecoreref(<|>NewClass<|>)
            }
            """,
        "addNewEClass");
}
```

Again, we rely on the Xtext testing framework providing dedicated API for testing hyperlinks and a useful assertion method. (Note the delimiter syntax <|> ... <|> to indicate the text region for invoking the hyperlinking action.) In particular, we only verify the shape of the hyperlink target (its string representation): we rely on the Xtext framework to correctly open the target in the proper editor when the developer clicks on that.

5.4 Quick fix testing

Edelta provides several context-aware quick fixes through a custom quick fix provider that extends Xtext's base functionality. These quick fixes are designed to help developers resolve common errors in metamodel transformations. The quick fixes range from simple text replacements to complex program transformations that require deep understanding of the program's execution context.

By leveraging the Xtext testing framework, we can create tests that simulate error scenarios and verify that the appropriate quick fixes are suggested and applied correctly. In particular, we verify that the expected quick fixes are available for specific validation issues and that applying the fixes results in the correct program transformation.

The most sophisticated quick fixes are those that interact with the interpreter and the in-memory metamodel state. For instance, if an `ecoreref` points to an element that has been renamed in that program context, the quick fix can update the reference to use the new name (see Fig. 2).

Similarly, when an `ecoreref` points to an element that does not yet exist in the current program context, the quick fix can reorder statements to move the reference after the creation site. The test in Listing 16 verifies that when an `ecoreref` refers to a not-yet-existing element (`NewClass`, created in the subsequent statement), the quick fix to move the `ecoreref` to the correct position, after the creation statement, is suggested and after applying it, the program is correctly transformed as expected.

Listing 16 Edelta UI Test example: quick fix test for moving `ecoreref` to the right position.

```
@Test
public void fixMoveToRightPosition() {
    testQuickfixesOn(
        // the initial DSL text
        """
            metamodel "foo"

            modifyEcore creation epackage foo {
                ecoreref(NewClass).abstract = true
                addNewEClass("NewClass")
                ecoreref(NewClass).ESuperTypes += ecoreref(FooClass)
            }
            """,
        // the expected validation issue to fix
        EdeltaValidator.INTERPRETER_ACCESS_NOT_YET_EXISTING_ELEMENT,
        new AbstractQuickfixTest.Quickfix(
            "Move to the right position", // the expected quick fix label
            // the expected DSL text after applying the quick fix
            """
                metamodel "foo"

                modifyEcore creation epackage foo {
                    addNewEClass("NewClass")
                    ecoreref(NewClass).abstract = true
                    ecoreref(NewClass).ESuperTypes += ecoreref(FooClass)
                }
                """));
}
```

5.5 Outline view testing

Even concerning the outline view, we provide a custom outline tree provider that extends Xtext's default implementation to correctly represent both the static structure of Edelta files and the dynamic changes introduced by metamodel transformations (see Fig. 3). The Xtext testing framework includes utilities to interact with the outline view and assert the presence and structure of elements; the expected structure is provided through a string representation with proper indentation.

The test in Listing 17 verifies that the outline view correctly displays both the original metamodel elements and the dynamically created `NewClass` EClass.

Listing 17 Edelta UI Test example: outline view test for dynamically created EClass.

```
@Test
public void testOutlineWithCreateEClassInModifyEcore() throws Exception {
    assertAllLabels("""
      metamodel "mypackage"

      modifyEcore aModification epackage mypackage {
          EClassifiers += newEClass("NewClass")
      }
      """,
    """
      test
        aModification(EPackage) : void
        mypackage
          MyClass
            myAttribute : EString
            myReference : EObject
          MyDataType [java.lang.String]
          NewClass
      """);
}
```

Our custom outline also uses the boldface font style to highlight elements that are created dynamically during the interpretation of Edelta operations (see Fig. 3). Unfortunately, the Xtext testing framework does not provide a way to verify font styles in the outline view. To test the correct application of the font style, we had to create dedicated tests for our custom label provider that is used by our outline tree provider. In such tests, we directly invoke the label provider methods to verify that the correct font style is applied to dynamically created elements. We omit the code snippet here for brevity.

6 End-to-end tests with SWTBot

While the DSL unit tests and UI integration tests provide comprehensive coverage of individual components and their interactions, there is still a need to test the complete user experience from a black-box perspective. End-to-end tests simulate real user interactions with the Eclipse IDE to verify that the entire Edelta development workflow functions correctly. For this purpose, we employ SWTBot,[4] a Java-based UI testing tool specifically designed for Eclipse applications.

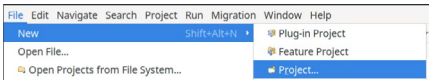
[4] https://projects.eclipse.org/projects/technology.swtbot

SWTBot provides an API to interact with Eclipse's Standard Widget Toolkit (SWT) user interface elements, allowing tests to locate and manipulate UI components based on their labels, types, and hierarchical relationships and to click buttons, select menu items, type text, and navigate tree structures just as a human user would. This API makes such tests maintainable and readable. SWTBot tests help identify issues that might arise from the complex interactions between different Eclipse plugins, UI components, and the underlying Edelta language infrastructure. Such issues might be missed by more focused unit tests or UI integration tests. However, SWTBot tests are significantly slower to execute than unit and UI tests (see Section 8.2), require a full Eclipse IDE environment, and can be fragile due to their dependence on UI timing and layout. Therefore, we maintain only a small number of carefully selected SWTBot tests that focus on the most critical user-facing features that cannot be tested with other kinds of tests. The Edelta SWTBot tests are located in the `edelta.swtbot.tests` project.

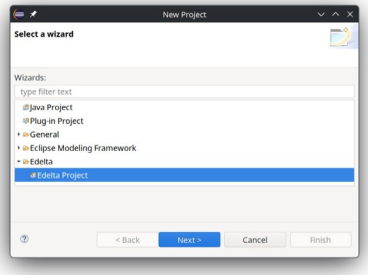
Edelta provides a project wizard that guides users through the process of setting up a new Edelta project with the necessary configurations and some example files. The test in Listing 18 verifies the complete project creation wizard workflow (in particular, it also verifies that the generated project structure is correct and that the build process completes without errors). The test simulates user interactions with the Eclipse UI to create a new Edelta project using the wizard, as illustrated in Fig. 5. In the listing, we added comments that refer to the corresponding sub-figures to help understand the workflow steps.

Listing 18 SWTBot test for creating a new Edelta project.

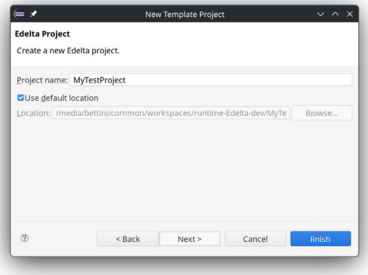
```
@Test
public void canCreateANewExampleProject() throws OperationCanceledException {
    // See Fig.5(a)
    bot.menu("File").menu("New").menu("Project...").click();

    // See Fig.5(b)
    var shell = bot.shell("New Project");
    shell.activate(); // Focus on the new project wizard dialog
    bot.tree().expandNode("Edelta").select("Edelta Project");
    bot.button("Next >").click();

    // See Fig.5(c)
    // Fill the first page of the wizard with the project name
    bot.textWithLabel("Project name:").setText("MyTestProject");
    bot.button("Next >").click();

    // See Fig.5(d)
    // Select the example project template in the second page of the wizard
    bot.table().select("Edelta Example Project");
    // Finish the wizard
    bot.button("Finish").click();

    // Wait for the project to be built and verify no errors occurred
    assertTrue(isProjectCreated("MyTestProject"));
    waitingForBuild();
    PluginProjectHelper.assertNoErrors();

    // See Fig.5(e)
    // Verify that Java code is generated by expanding the project tree
    getProjectTreeItem("MyTestProject").expand()
        .expandNode("edelta-gen", "com.example", "Example.java");
}
```



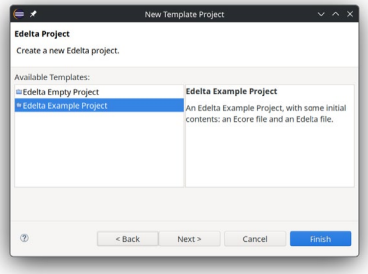
(a) Selecting the menu for creating a new project.



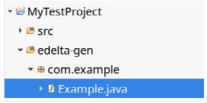
(b) Selecting the Edelta Project wizard.



(c) Filling in the project name.



(d) Selecting the project template.



(e) Expanding the project tree up to the generated Java file.

Fig. 5 The operations for creating a new Edelta project with its wizard performed by the SWTBot test

Additional SWTBot scenarios cover other regression-prone workflows (e.g., launch-configuration-based execution); we omit the corresponding listing for brevity.

7 Automated building and continuous integration

In this section, we describe the automated building infrastructure and continuous integration practices we employ for the Edelta project. These practices are crucial for maintaining code quality, ensuring correctness across multiple platforms, and streamlining the release process. The entire build pipeline is automated and runs on every commit, providing feedback on code changes.

7.1 Maven and Tycho build system

We use Apache Maven as our build automation tool, combined with Eclipse Tycho[5], a set of Maven plugins specifically designed for building Eclipse plugins and OSGi bundles. While Gradle has gained popularity in recent years (Prakash 2022; Medeiros et al. 2025), Maven with Tycho remains the de facto standard for Eclipse plugin development, offering several critical advantages.

First, Tycho seamlessly integrates with the OSGi runtime, allowing us to execute tests that require Eclipse platform services. The `tycho-surefire-plugin` executes JUnit tests within an OSGi runtime, which is necessary for testing components that depend on Eclipse platform services. This plugin configuration allows us to run both unit tests (which do not require a full Eclipse workbench, but are tested with an OSGi runtime) and UI and SWTBot tests (which do require a running Eclipse instance) as part of the automated build. This is essential for our integration and UI tests, which need to run within an Eclipse instance. Second, Tycho automatically resolves dependencies from p2 repositories (Eclipse update sites). Third, Tycho supports building complete Eclipse products and p2 repositories, which we leverage for our distribution strategy (Sections 7.3 and 7.4).

Our project follows a multi-module Maven structure, with a parent POM that defines common configurations and properties for all modules. The parent POM specifies the Tycho version, target platform definition, and build plugin configurations. The target platform[6] configuration is particularly important, as it defines which Eclipse and Xtext versions our plugins are compatible with, as well as the environments (operating systems and architectures) we support. We build for several environments: Linux x86_64 and aarch64, Windows x86_64, and macOS x86_64 and aarch64 (Apple Silicon). This ensures our Eclipse product distributions work correctly on all major desktop platforms.

7.2 Continuous integration with GitHub Actions

We leverage GitHub Actions for our continuous integration pipeline. GitHub Actions is tightly integrated with GitHub repositories and is free for public repositories, making it an ideal choice for our needs. Our CI workflow is triggered on every push and pull request, ensuring that all code changes are automatically tested before being merged.

The main workflow performs several important tasks:

- Building the entire project with Maven/Tycho
- Running all tests (unit, integration, UI, and end-to-end tests) on multiple platforms
- Computing test coverage metrics
- Performing code quality analysis with SonarCloud

[5] https://eclipse.dev/tycho/

[6] A target platform defines the set of bundles and features that an Eclipse plugin is built against; these are downloaded from specified p2 repositories. A target platform is specified in a separate .target textual file, which is part of the source code and versioned alongside the project.

The workflow uses a matrix strategy to run tests on Linux, Windows, and macOS, ensuring cross-platform compatibility. This is particularly important for Eclipse plugins, as platform-specific issues can arise from differences in file systems, line endings, and windowing systems. The matrix approach allows us to catch such issues early in the development cycle.

One of the key aspects of our CI pipeline is the separation of concerns between different jobs. The main build and test job runs on all platforms, while code coverage and code quality are computed only once (on Linux) to save resources and to avoid redundant computations and reports.

Additionally, we have separate deployment jobs that are manually triggered only on the master branch. These deployment jobs handle the publishing of Eclipse product distributions to SourceForge (Section 7.3), the p2 repository to GitHub Pages (Section 7.4), and the deployment of Maven artifacts to Maven Central (Section 7.5). Authentication tokens, GPG keys, and other sensitive information required for deployment are securely stored as GitHub Secrets.

Eclipse users can choose whether to install Edelta via the p2 repository in their existing Eclipse installations or to download a complete Eclipse distribution with Edelta pre-installed. Our Eclipse distributions are also configured to be updatable through our p2 repository, ensuring that users can easily receive updates without needing to download new distributions.

Currently, we do not apply continuous delivery (Humble and Farley 2010) for each commit to the master branch. In this project, we prefer to have manual control over releases by having separated GitHub Actions workflows that are triggered manually, since we do not expect very frequent releases and we would not have benefits or problems from continuous delivery (Chen 2015; Laukkanen et al. 2017; Shahin et al. 2017). Note however that the deployment jobs are fully automated and require no manual intervention once triggered.

The deployment on Maven Central for the Maven artifacts is given for granted, since it is a standard practice and a common repository for Java libraries. For Eclipse products and p2 repositories, there are no standard repositories, so we had to choose where to publish them. For Eclipse products, we chose SourceForge (Section 7.3), since it is a well-known hosting service for open-source software, and it provides a reliable platform for big artifacts like our Eclipse distributions (each one is about 300 MB in size). For the p2 repository, we chose GitHub Pages (Section 7.4), instead of SourceForge, because the latter relies on mirrors that are automatically selected after a few seconds of delay for each downloaded artifact. That would be inconvenient for an update site that serves several files for each installation. Moreover, since GitHub Pages are updated through Git, we can easily automate the deployment of our composite p2 repository structure, which first requires cloning the existing repository, adding the new version files, and pushing back the changes.

7.3 Building and deploying Eclipse products

A crucial part of our infrastructure is the automated building and deployment of Eclipse product distributions. Using Tycho's `tycho-p2-director-plugin`

and `tycho-p2-repository-plugin`, we create standalone Eclipse installations with Edelta pre-installed for different operating systems and architectures.

The product build process materializes complete Eclipse distributions from our p2 repository and package them into zip archives. These distributions include not only our Edelta plugins but also all required dependencies, such as Xtext, EMF, and the Eclipse platform itself. This provides users with a ready-to-use environment without requiring manual plugin installation. We provide distributions for Linux (x86_64 and aarch64), Windows (x86_64), and macOS (x86_64 and aarch64).

The built products are automatically uploaded to SourceForge[7] using GitHub Actions, making them immediately available to users. Instead of using a dedicated GitHub action for SourceForge deployment, we perform the upload through `rsync` over SSH, directly from our Maven build. The rsync commands are executed through the `exec-maven-plugin`[8] during the Maven build. This approach gives us more control over the upload process and allows us to easily test it locally. In particular, we could also perform the upload from our local machine though we prefer to automate it through GitHub Actions.

Moreover, during the build, we run the SWTBot tests (Section 6) against the Eclipse distribution produced for the current platform. This is made possible by the `tycho-surefire-plugin`, which allows us to run tests against a provisioned Eclipse instance. By default, the plugin would run the tests against an Eclipse instance built from the current target platform. By specifying `<testRuntime>p2Installed</testRuntime>` in the plugin configuration, we can instruct Tycho to run the tests against the Eclipse installation produced by the product build. This ensures that the built Eclipse distribution is fully functional and that all plugins are correctly installed and integrated.

7.4 P2 repository management and GitHub Pages

We maintain a p2 repository (Eclipse update site) that allows users to install Edelta into their existing Eclipse installations. This is a composite repository[9] structured to support multiple versions of Edelta. The p2 repository is built using Tycho and includes repository references[10] to external dependencies (such as Xtext and EMF update sites), ensuring that all transitive dependencies are resolved during installation. Thus, our p2 repository is self-contained and does not require users to manually add additional update sites.

For adding the produced p2 repository to the composite repository structure, we use the dedicated Ant task `p2.composite.repository` provided by the

[7] https://sourceforge.net/projects/edelta/files/products

[8] The https://www.mojohaus.org/exec-maven-plugin/ allows executing system programs from Maven builds.

[9] https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fp2_composite_repositories.

[10] A repository reference is a pointer to another p2 repository, allowing the p2 installer to resolve dependencies from multiple sources.

Eclipse p2 tools. The Ant task is executed from our Maven build using the `maven-antrun-plugin`, allowing us to integrate it seamlessly into the build process.

The p2 repository follows a version-based structure: `updates/major.x/major.minor.x/`.

This allows users to use a specific major or minor version stream and receive automatic updates for that stream when they are available. For example, using https://lorenzobettini.github.io/edelta-releases/updates/3.x/ ensures users receive all updates for Edelta 3.x, but not updates for Edelta 4.x (when released).

The repository is deployed to GitHub Pages using a dedicated workflow that runs on the master branch. GitHub Pages provides free hosting with a CDN, ensuring fast and reliable access to our update site worldwide. We have a separate GitHub repository only for hosting the GitHub Pages site, which contains the p2 repository files. This separation keeps the main Edelta repository clean and focused on development.

Similarly to what we described in Section 7.3, and for the same reasons, we perform the upload through standard Git commands from our Maven build instead of using a dedicated GitHub Actions action. The Git commands executed to first clone the current GitHub Pages repository, copy the newly built p2 repository files, and push the changes back to GitHub are executed through the `exec-maven-plugin` during the Maven build.

An important quality assurance step in our CI pipeline is testing the p2 repository installation. We use Tycho's director application to install our plugins from the newly built p2 repository into a fresh Eclipse instance, verifying that all dependencies are correctly resolved. This test catches issues such as missing dependencies or incorrect repository references before the update site is deployed to users.

7.5 Maven Central deployment

We also publish Edelta Maven artifacts to Maven Central[11]. These artifacts include the Edelta compiler and runtime libraries, allowing users to integrate Edelta into pure Maven Java projects without requiring Eclipse or Tycho. The published Maven artifacts can also be used in Gradle projects, as Gradle can consume Maven artifacts seamlessly.

The deployment to Maven Central is automated through GitHub Actions and uses the standard Maven plugin `central-publishing-maven-plugin` for publishing. The workflow signs artifacts using GPG and deploys them to the Central Maven repository. As for the other publishing tasks described above, the deployment job is triggered manually to maintain control over the release process.

Users can then include Edelta artifacts in their Maven projects and use the `xtext-maven-plugin` to compile `.edelta` files into Java code during the build. An example Maven project demonstrating this integration is available in the repository at `edelta.maven.example`.

[11] Group ID: io.github.lorenzobettini.edelta

7.6 Code quality analysis with SonarCloud

We integrate SonarCloud[12] into our CI pipeline for continuous code quality monitoring. SonarCloud is a cloud-based static analysis service that is free for open-source projects and provides detailed metrics on code smells, bugs, vulnerabilities, test coverage, and code duplication.

The SonarCloud integration is configured in our parent POM and runs automatically during the Maven build on Linux (we only run it once per build to save resources). The `sonar-maven-plugin` collects various metrics during the build, including test coverage from JaCoCo reports (the code coverage information must be computed during the Maven build when running tests, through the JaCoCo Java code coverage tool[13]), and sends them to SonarCloud for analysis. SonarCloud then provides feedback directly on GitHub pull requests, allowing reviewers to see code quality metrics alongside the code changes.

Key metrics we monitor include:

- **Code Coverage:** We aim for 100% test coverage to have some guarantees that our code is well-tested.
- **Code Smells:** SonarCloud identifies maintainability issues, such as overly complex methods or bad code practices.
- **Bugs and Vulnerabilities:** The tool flags potential runtime errors and security issues.
- **Duplications:** The tool identifies code duplication throughout the code base, and we want to avoid code duplication to improve maintainability.

The quality gate configured in SonarCloud enforces minimum standards for new code. If a pull request introduces code that does not meet these standards (e.g., insufficient test coverage or critical bugs), the quality gate fails, and the pull request cannot be merged until the issues are addressed. This automated enforcement ensures that code quality does not degrade over time. In particular, since we have been using SonarCloud from the very beginning of the project, we have been able to easily reach 0 technical debt (Kruchten et al. 2012; Tom et al. 2013; Li et al. 2015) and maintain it over time.

Code coverage can be a faulty metric (Marick et al. 1999; Hemmati 2015), and reaching 100% can often be considered too much work with no guarantees, still it is valuable (Chen et al. 2002; Ivanković et al. 2019) if used wisely. First of all, we configure the scope of code coverage to only consider real code, with effective logic, containing the important parts of the implementation. For example, we exclude generated code, model definitions, and trivial getters/setters from coverage analysis. Second, we use code coverage as a guide to identify untested parts of the codebase, rather than as an absolute measure of quality. We focus on writing meaningful tests that cover critical paths and edge cases, rather than aiming for high coverage numbers alone. However, since, as we have already said, we have applied test-driven develop-

[12] https://sonarcloud.io/project/overview?id=io.github.lorenzobettini.edelta%3Aedelta.parent

[13] https://www.eclemma.org/jacoco/

ment since the very beginning of the project, writing a test before the code natually leads to 100% coverage of the implemented code with great guarantees that the code is well-tested.

Mutation testing (Woodward 1993; Papadakis et al. 2019) is another valuable technique for assessing test suite effectiveness. By introducing small changes (mutations) to the code and checking if the existing tests catch these changes, we can identify weaknesses in our tests. While we have not yet integrated mutation testing into our CI pipeline, it is a consideration for future improvements to further enhance our testing strategy. For Java, we could use the PIT mutation testing tool[14]. However, it does not support OSGi/Tycho projects and tests that require an Eclipse runtime. Moreover, mutation testing can significantly increase build times, so careful consideration and configurations are needed before integrating it into the development workflow and the CI pipeline (Wong and Mathur 1995; Petrović and Ivanković 2018; Parsai and Demeyer 2020). Due to the way we write tests, as described above, we have not felt the need to use mutation testing so far and we are quite confident about the effectiveness of our test suite.

Throughout these years, we found that when new bugs arise in Edelta, even in the presence of 100% code coverage, they are often due to missing special cases about the structure of the models being processed (being the EMF model of the AST of Edelta code, or the Ecore models being used by our code and libraries). In that respect, even a mutation testing tool would not help much, since the problem is not that the tests do not check the behavior correctly, but that the tests do not cover the specific model structures that trigger the bugs. Even a mutation testing tool would not catch such problems.

8 Discussion

In this section, we reflect on the practical implications of our multi-level testing strategy and CI infrastructure for the Edelta toolchain. We first analyze a representative CI run to provide *operational* evidence about feasibility and cost distribution, then distill lessons learned and best practices; finally, we discuss challenges/trade-offs and clarify how much of the approach can be reused when porting it to other Eclipse/Xtext-based projects.

8.1 Practical implications of the testing strategy

We emphasize that the evidence we can substantiate quantitatively in this paper is primarily *operational*: a comprehensive, multi-level test strategy for an Eclipse/Xtext toolchain can be executed continuously with reasonable turnaround time (the full pipeline completes in about 8 minutes while executing 883 tests across all layers as shown in Section 8.2 and Table 1). This makes it realistic to apply the strategy routinely (e.g., on every pull request), including when upgrading external dependencies.

[14] https://pitest.org/

Accordingly, we phrase the following points as observations from our case study rather than as causal claims about developer productivity.

Testing the runtime library and the refactorings library with fast unit tests, following a test-driven development approach, allowed us to focus on single small tasks and implement them correctly from the start. While we do not attempt to quantify defect reduction over time, the unit-test base provides a fast regression net that catches behavioral changes immediately when refactoring or extending the libraries (in the CI run of Section 8.2, 474 library-level unit tests execute in about 13 seconds; on a local modern computer, they are even faster).

The DSL tests provided confidence that the language features worked as intended and that changes to the language infrastructure did not introduce regressions. In particular, testing and focusing on the single DSL components was crucial for the advanced features of the Edelta DSL (Section 2.2). In that respect, the hardest part to implement was the interpretation mechanism that had to be inserted in the compilation pipeline. Having a solid suite of DSL tests allowed us to easily and quickly experiment with the interpretation and check whether the other language features were still working as expected leveraging the additional information provided by the interpretation in the proper program contexts. These middle-layer tests remain fast as well (about 40 seconds in the CI, and, again, even less on a local machine), which helped us validate language-level changes without relying on UI-driven checks as the primary oracle.

The UI integration tests ensured that the language services integrated correctly with the Eclipse IDE. Again, this allowed us to verify that the editor provides a "live" experience with real-time feedback as users write Edelta code, using also the information provided by the interpretation. Finally, the end-to-end SWTBot tests validated the complete user workflows, ensuring that users could create projects, write Edelta code, and generate Java code without issues. In particular, as stressed in Sections 7.3 and 7.4, having automated tests during the build that verify the Eclipse products and the p2 repository is crucial to ensure that users can install and use Edelta without problems. Testing such artifacts manually would be error-prone and time-consuming. As expected, UI and SWTBot tests are costlier, which motivates keeping them deliberately small and focused, while most checks remain at the faster layers.

All these tests took time to implement and maintain, especially the UI and end-to-end tests. However, with the measured execution cost remaining under 8 minutes for the full pipeline (Section 8.2), running the complete suite continuously provides practical feedback on whether a change affects the runtime libraries, language services, IDE integration, or end-to-end workflows. Moreover, we can deal with external dependencies updates with more confidence, knowing that our comprehensive test suite will likely catch any regressions. Keeping dependencies up to date is a crucial software maintenance task (Jaime et al. 2024; Kula et al. 2018) but might become a burden without existing automated tests. In our experience, the fact that the whole pipeline is routinely executed on CI with predictable cost is what makes such updates manageable in practice: upgrades can be validated by the same multi-level suite that is also used for feature development. With our test suite, we can regularly update dependencies and ensure compatibility without fear of breaking existing functionality.

We leave a controlled study of defect trends or developer productivity (e.g., issue lead time, regression rates) to future work; the present paper focuses on techniques and on the operational evidence reported in Section 8.2.

8.2 Build execution analysis

This subsection provides quantitative, *operational* evidence about our testing and CI strategy: how the overall CI cost is distributed across testing layers, and whether the full multi-level suite is feasible to execute continuously. While such timing results do not directly measure reliability or maintainability outcomes, they are a necessary precondition for *effectiveness in practice*, since regressions can only be caught routinely if the complete verification pipeline can run on every change with predictable turnaround time.

We analyzed a complete build execution from our GitHub Actions CI pipeline. The build was executed on September 4, 2025, on an Ubuntu 24.04.3 LTS runner, for a total duration of *7 minutes and 56 seconds*.

The build comprises 29 Maven modules, including 5 core test projects that collectively execute comprehensive test suites across all levels of our testing strategy. Table 1 summarizes the test execution results for each test project. The test execution data reveals several important characteristics of our testing approach:

Unit Test Performance: The runtime library tests (`edelta.lib.tests`, Section 4.1) demonstrate excellent performance with 345 tests executing in just 6.4 seconds, averaging 19 milliseconds per test. These tests validate the core EMF-based runtime functionality and represent the fastest category of tests, confirming our strategy of implementing most functionality testing at this level. Concerning the tests for the refactorings library (`edelta.refactorings.lib.tests`), 129 tests execute in 6.6 seconds, averaging 51 milliseconds per test. While slightly slower than the core library tests, this performance remains efficient, reflecting the additional complexity involved in testing refactoring operations, including bad smell detection and resolution.

DSL Test Performance: The DSL unit tests (`edelta.tests`, Section 4.2) execute 345 tests in 40.8 seconds, averaging 118 milliseconds per test.[15] This represents a 6× slowdown compared to runtime tests, which is expected given that these tests involve the full Xtext infrastructure including parsing, validation, type checking, and code generation. However, the total execution time remains reasonable for comprehensive language testing.

UI Test Performance: The UI integration tests (`edelta.ui.tests`, Section 5) show significantly longer execution times, with 59 tests taking 73.4 seconds (1.2 seconds per test on average). This performance profile reflects the overhead of Eclipse workspace operations, including project creation, file manipulation, and automated building processes. Despite the longer execution time, these tests provide essential coverage of IDE integration features that cannot be tested at lower levels.

End-to-End Test Performance: The SWTBot tests (`edelta.swtbot.tests`, Section 6) represent the most time-intensive category, with only 5 tests requiring

[15] It is a pure coincidence that the number of tests in edelta.lib.tests and edelta.tests is the same (345).

Table 1 Test execution summary by project (totals are in boldface)

Test Project	Tests	Time (s)	Avg (ms/test)
`edelta.lib.tests`	345	6.4	19
`edelta.refactorings.lib.tests`	129	6.6	51
`edelta.tests`	345	40.8	118
`edelta.ui.tests`	59	73.4	1,244
`edelta.swtbot.tests`	5	38.5	7,700
Total	**883**	**165.6**	**188**

38.5 seconds (7.7 seconds per test). While expensive in execution time, these tests provide validation of complete user workflows that are otherwise difficult to exercise automatically.

Build Infrastructure Efficiency: The SonarCloud analysis (Section 7.6) adds an additional 40.3 seconds to the build, providing code quality metrics including coverage analysis, bug detection, and vulnerability scanning.

Testing pyramid validation The total build time of under 8 minutes demonstrates that comprehensive testing across all levels can be achieved with reasonable CI resource consumption. This empirical analysis provides concrete validation of the well-known testing pyramid (Cohn 2009; Fowler 2013) principle in the context of DSL and Eclipse plug-ins development. Our test distribution clearly follows the pyramid structure (visually depicted in Fig. 6): 474 unit tests (runtime library and refactorings library) execute in 13.0 seconds and form the broad base, 345 DSL tests (which can be considered unit tests for language features though they integrate several language services) execute in 40.8 seconds in the middle layer, 59 integration tests (UI tests) take 73.4 seconds, and 5 end-to-end tests (SWTBot) require 38.5 seconds at the apex.

Efficiency vs. effectiveness The timing results primarily quantify efficiency (cost), but they also connect to effectiveness in an important practical sense: because the full multi-level suite completes in under 8 minutes, it can serve as an *authoritative CI gate* for changes that may affect reliability and maintainability (including dependency updates and release preparation). In other words, these measurements show that comprehensive verification is not reserved for occasional release builds but can be applied routinely, which is what allows regressions to be caught early and prevents "bit rot" in long-lived Eclipse/Xtext toolchains. In our pipeline, this includes verifying packaging artifacts (Eclipse products and the p2 repository) and critical IDE workflows (wizard and launch configurations), which are areas where regressions are particularly costly (because they affect installation and core workflows) and therefore motivated dedicated checks. We do not claim a measured improvement in defect rates or maintenance effort; rather, the evidence here is that the verification strategy is continuously executable and thus actionable in everyday development.

Summary The performance characteristics support our architectural decision to implement most testing logic in fast, isolated unit tests while selectively using more expensive integration and UI tests only where necessary. The test execution times

demonstrate a clear inverse relationship between test count and execution cost: unit tests average 27 milliseconds each, DSL tests average 118 milliseconds, integration tests 1.2 seconds, and end-to-end tests 7.7 seconds. This $\approx 285\times$ performance difference between the fastest and slowest test categories justifies the pyramid approach and validates our strategy of comprehensive testing at the unit level with selective coverage at higher levels.

8.3 Lessons learned and best practices

Through our experience maintaining the Edelta CI infrastructure, we have identified several best practices:

1. **Matrix Builds:** Test on multiple platforms to catch platform-specific issues early. Even if the primary development happens on one platform, users will run the software on various systems.
2. **Separate Deployment:** Keep deployment steps separate from build and test jobs. This prevents accidental deployments and allows for manual intervention if needed.
3. **Cache Dependencies:** Use caching mechanisms (like GitHub Actions cache) to speed up builds by avoiding redundant downloads of Maven dependencies. This is particularly important for Tycho builds when resolving target platform dependencies. An Eclipse project like Edelta, based on Xtext and EMF, has a few hundred megabytes of dependencies to download, so caching is essential to have reasonable build times.
4. **Self-Contained Artifacts:** Ensure that deployed artifacts like p2 repositories are self-contained so that users only have to specify a single p2 repository and not the repositories of transitive dependencies.
5. **Automated Quality Gates:** Integrate code quality tools directly into the CI pipeline and enforce quality standards automatically. In our experience, this helps prevent gradual quality decay and makes regressions more visible early, even though it requires sustained discipline and occasional refactoring effort.

In the whole automated build and CI infrastructure, the build tool plays a central role. We keep our Maven/Tycho build as the single source of truth for building, testing, and deploying the project. We keep the build as reproducible as possible also in the local development environment. Debugging a failing build is much easier if developers can reproduce the same build locally. Doing that when the build runs on GitHub Actions is not always straightforward instead.

In particular, our build is self-contained and executes all the needed external commands through Maven plugins, so that developers can run the same build locally as in CI. Of course, we assume that the needed system commands are available in the developer's environment (for example, Git and rsync). A single Maven command is enough to run the typical standard build and test process. We created additional Maven profiles to perform more complex tasks, such as deploying the p2 repository or the Eclipse products.

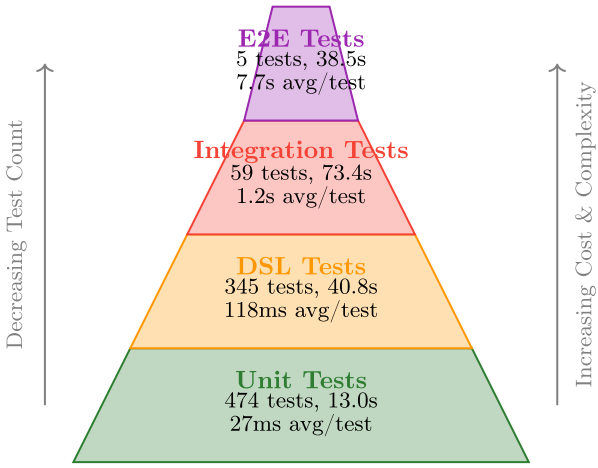


Fig. 6 Edelta Testing Pyramid: Distribution of tests across testing levels with execution times and average cost per test

Running system commands through the `exec-maven-plugin` allows us to keep the build logic in Maven. Running such commands from Maven might seem cumbersome at first, but it pays off in the long run. In particular, the system commands for Git, rsync and the Ant tasks for maintaining the p2 composite repository need information related to the current version being released (including the typical build qualifiers transformed into timestamps). Such information is naturally available in the Maven build through properties, so that we can avoid duplicating versioning logic in scripts outside Maven. On the contrary, having separate steps outside Maven in the CI configuration would require duplicating such logic (for example, to retrieve the above-mentioned information by scanning the POM files or reading produced artifacts), making the build more error-prone and harder to maintain. Not to mention, that, in case of switching the CI platform in the future, having the whole build logic in Maven would make the migration easier (the single Maven command would be the same). Hardcoding such logic in the CI configuration would make the migration more difficult.

Secrets are injected in the Maven build through environment variables and properties, so that developers can also test deployment steps locally if needed, as long as they have such secrets on their computers. In GitHub Actions, secrets are stored as GitHub Secrets and passed to the Maven build as environment variables.

The combination of Maven/Tycho for building, GitHub Actions for CI/CD, and SonarCloud for quality analysis provides a robust infrastructure that supports the entire Edelta development lifecycle. This infrastructure has proven essential for maintaining code quality, ensuring correctness across platforms, and delivering reliable releases to users.

Two additional lessons emerged over the years that are worth highlighting, as they directly impacted both our testing/build practices and the long-term maintainability of the DSL: treating discovered bugs as opportunities to strengthen the test suite, and adopting an architecture that minimizes grammar churn while allowing the set of evolution operations to grow.

Users can report bugs and request improvements through the Edelta GitHub repository, via the issue tracker.[16] Whenever a bug is reported (or discovered during development), we first translate it into an automated regression test that reproduces the failure, and only then implement the fix until the new test—and the full existing suite—passes, in line with our test-driven development workflow. This practice is especially valuable because, even with 100% code coverage, several bugs may stem from previously untested corner cases in the structure of the processed models; capturing those cases as focused regression tests ensures they remain covered over time.

Early versions of the Edelta DSL encoded several metamodel-evolution primitives directly in the grammar (e.g., dedicated constructs for creating/removing `EClass`, `EAttribute`, etc.), which increased the size of the generated Ecore metamodel for the DSL AST and made language evolution more invasive: introducing a new operation required changing the grammar (and thus the AST metamodel), and validating the implementation often required end-to-end tests through the DSL and its generator. Inspired by the *"libraries instead of keywords"* approach advocated by Zarnekow (2012), i.e., keeping the DSL syntax small and stable while offering domain operations through a library API rather than dedicated grammar constructs, we later (Bettini et al. 2020) moved these evolution primitives into the Edelta Java runtime library and kept the concrete syntax minimal by reusing Xbase's seamless integration with Java APIs (Efftinge et al. 2012). This choice is consistent with well-established DSL engineering guidance that favors leveraging a host language and libraries (embedded/internal DSLs) over continuously extending the surface syntax with new constructs (Hudak 1998; Fowler 2010). In the current design, the expression grammar only contains the domain-specific adaptation we need (notably the `ecoreref` expression), while evolution operations are provided as regular library methods. This shift made the primitives straightforward to unit-test independently from the DSL and allowed us to extend Edelta with new operations without modifying the grammar. Since adopting this architecture, the grammar and its generated Ecore metamodel have changed only rarely, which has substantially smoothed Edelta's evolution over the years. In particular, one of the most significant new features recently added to Edelta was co-evolution support (Bettini et al. 2024a), allowing models to be evolved together with their metamodels. This feature was implemented entirely in the Edelta Java runtime library, without any change to the grammar. This way, we were able to implement and test the new feature quickly, focusing on the co-evolution runtime engine, reusing the existing DSL syntax, and writing unit tests for the new library methods.

8.4 Challenges, trade-offs, and limitations

The practices discussed above are not free: adopting a comprehensive, multi-level test strategy and CI pipeline involves several practical trade-offs that we found important to acknowledge.

[16] https://github.com/LorenzoBettini/edelta/issues

Up-front and ongoing investment Building a broad unit-test base, curating representative test fixtures (metamodels, models, example projects), and maintaining CI scripts requires sustained effort. This effort is front-loaded when establishing the infrastructure, but it also persists over time as the toolchain evolves (e.g., new language features, new Eclipse/Xtext/EMF versions, and new platform constraints).

UI and end-to-end tests are valuable but fragile IDE integration and SWTBot tests exercise critical workflows that cannot be validated reliably at lower layers, but they are also the most sensitive to environmental issues and timing (e.g., workspace builds, UI focus, background jobs, and platform-specific behavior). To manage this trade-off, we deliberately keep UI/SWTBot suites small and focused, and we prefer to push as many checks as possible to deterministic unit/headless tests, using UI-level tests primarily to guard the most user-visible workflows.

Strict quality gates require discipline Enforcing coverage and static-analysis rules via tools such as SonarCloud can introduce short-term friction (e.g., refactoring code to satisfy rules, occasionally adjusting thresholds, or writing additional tests for edge cases). However, we found that treating these checks as part of the definition of done helps prevent gradual quality decay in long-lived research software, where small compromises can accumulate over years.

Avoiding "symptom fixes" without a reproducer A recurring temptation when addressing failures is to apply small defensive changes (e.g., adding a null check or a conditional) that appear to fix the observed issue but do not establish whether the root cause is understood. We therefore try to make bug fixing test-driven whenever feasible: first capture a minimal failing scenario as a regression test, then implement the fix until the new test and the existing suite pass. This discipline can be slower in the short term, but it reduces the risk of masking deeper issues and helps prevent regressions.

What the analysis does and does not claim Finally, while we provide operational evidence that the full verification pipeline is continuously executable (Section 8.2), we do not present a controlled longitudinal study of defect trends or developer productivity. Our contribution is to document the techniques and engineering choices that make multi-level verification practical and repeatable for Eclipse/Xtext-based toolchains, and to provide quantitative evidence about the cost profile of this strategy.

8.5 Portability and reuse across Eclipse/Xtext projects

Although our experience is reported through the Edelta case study, most of the testing and CI setup follows recurring patterns that can be transferred to other Eclipse/Xtext-based DSL toolchains. In practice, some parts are reusable largely unchanged (as templates), while others require adaptation to the language- and IDE-specific behavior.

What Xtext provides out of the box When creating a new Xtext-based DSL, the Xtext wizard generates several projects, separating core logic from IDE integration, and it also generates separate projects for unit tests and UI tests. The wizard also generates a project for the p2 repository, although additional configuration is needed to make it self-contained. Moreover, it also generates the Maven/Tycho build structure, including a parent POM and child modules for the various projects. As we saw throughout the sections related to DSL tests (Section 4.2), Xtext also provides testing utilities that simplify writing tests for parsing, validation, and code generation in the form of helper classes based on dependency injection. The Java code generated by Xtext also includes specific Google Guice modules to configure the language services and their testing counterparts so that the above helper classes can be injected into test classes without further configurations. For UI tests (Section 5), Xtext provides several base classes for testing specific language services in the Eclipse IDE, such as content assist, quick fixes, hyperlinking, and outline. Typically, it is enough to extend such base classes and use the inherited assertion methods to write UI tests for that specific language service. These base classes also handle setting up and tearing down the Eclipse workbench environment required to run such tests.

However, there is no support out of the box for end-to-end SWTBot tests (Section 6), and Xtext does not provide built-in mechanisms for packaging and validating Eclipse products or for managing self-contained p2 repositories (Sections 7.3 and 7.4). Finally, CI pipelines and code-quality analysis (Sections 7.2 and 7.6) are outside the scope of Xtext and must be configured separately.

Reusable building blocks (often transferable with minor edits) Beyond what Xtext generates, the following elements typically generalize well across Xtext projects: (i) the layered test organization aligned with the testing pyramid (unit tests for libraries, headless language-service tests, IDE integration tests, and a small SWTBot suite); (ii) the headless Xtext test harness based on dependency injection and the standard Xtext testing utilities (for parsing, linking/scoping, validation diagnostics, and type checking); (iii) a unified Maven/Tycho build that orchestrates all layers, including packaging of Eclipse artifacts (products and p2 repositories) and their verification steps; (iv) the CI configuration structure (matrix builds, caching, artifact/log publication, and integration of quality gates such as coverage and static analysis); and (v) general UI-testing scaffolding (project import utilities, workspace build synchronization, and common assertions over markers, proposals, hyperlinks, and outline labels). In our experience, once these "non-Xtext" mechanisms (product/p2 packaging and CI/CD plumbing) are established in one project, they can often be transferred to other projects with minimal adjustments (mainly project names, artifact IDs, and packaging details).

Project-specific adaptation points Other parts are necessarily domain- and implementation-specific: (i) test fixtures (representative metamodels/models and example DSL programs) and the expected diagnostics or generated artifacts; (ii) custom language semantics (validators, scoping/linking rules, typing rules, interpreter/generator behavior) and therefore the assertions that encode the intended behavior; (iii) UI

customizations (quick-fix logic, hyperlinking targets, outline structure and labeling, wizards, launch configurations), where tests must refer to project-specific identifiers, menu labels, and workflow steps; and (iv) any additional verification that depends on the delivered packaging structure (e.g., product features, plug-in sets, update-site layout), which varies across toolchains. Concerning SWTBot, while the overall structure of tests (workbench manipulation, editor interaction, and common synchronization patterns) can be reused, the concrete workflows and assertions tend to be highly project-specific and usually require the largest adaptation effort.

What tends to recur across DSLs A useful rule of thumb is that tests for the *Xtext pipeline mechanics* (parsing, reference resolution, and diagnostic collection) can often be reused with similar structure across languages, whereas tests for *domain rules* (custom validation constraints, refactoring/migration semantics, generator-specific oracles) must be adapted.

In our experience, keeping the reusable harness and CI skeleton stable while evolving only the language-specific fixtures and assertions helps scale the approach to multiple Xtext/Xbase DSLs; we have applied the same general strategy in several other Xtext-based DSL projects (see, e.g., Bettini 2016; Bettini and Damiani 2017; Bettini and Crescenzi 2016; Bettini 2016; Bettini et al. 2019).

9 Related work

In this Section, we review related work in the areas of DSL and Eclipse plugin development and testing, and software quality practices including continuous integration and automated deployment. For related work on Edelta itself, we refer the reader to the previous publications on Edelta (Bettini et al. 2017, 2019, 2020; Barriga et al. 2021; Bettini et al. 2022a, b, 2024a, b).

9.1 Domain-specific language development

The development of domain-specific languages has been extensively studied in the literature. Mernik et al. (2005) provide foundational guidance on when and how to develop DSLs, discussing decision patterns, design patterns, and implementation patterns. Their work emphasizes the importance of proper language design and testing, though it predates modern DSL frameworks like Xtext. Fowler's comprehensive book (Fowler 2010) establishes key concepts and patterns for DSL development, distinguishing between internal and external DSLs and discussing various implementation approaches. Complementing these foundations with a strongly practice-oriented perspective, Völter et al. present an extensive account of DSL engineering with modern language workbenches, including guidance on tool construction, evolution, and validation activities that practitioners typically institutionalize as regression test suites and automated builds (Voelter et al. 2013). While these works provide theoretical foundations, our paper complements them by presenting concrete testing methodologies and automation practices for a production DSL implemented with Xtext.

Recent systematic mapping studies have analyzed the state of DSL development tools and methodologies. Kosar et al. (2016) conducted a systematic mapping study identifying research trends in DSL development, while Iung et al. (2020) specifically focused on DSL development tools, analyzing their capabilities including testing support. Their study found that testing remains a challenging aspect of DSL development, with many tools providing limited support for automated testing. To further contextualize the tool landscape, we now compare engineering practices for developing and testing DSLs across representative language workbenches and language-oriented programming platforms.

9.2 Engineering practices across DSL platforms and language workbenches

Language workbenches provide integrated support for language definition and tool generation, typically covering aspects such as syntax, static semantics, transformations, code generation, and IDE services. A broad comparative perspective on this ecosystem is provided by surveys and overviews of language workbenches (Erdweg et al. 2013). Across platforms, differences in language definition style (textual vs. projectional vs. library-based), composition mechanisms, and testing support lead to distinct engineering practices.

Spoofax targets the development of *textual* DSLs and IDE services from high-level, largely declarative specifications (Kats and Visser 2010). A distinctive feature is explicit support for test-driven language development: integrated language definition testing enables regression tests for parsing, static analyses, transformations, and generated artifacts, making the "language definition" itself a testable unit (Kats et al. 2011). In practice, this encourages developers to maintain executable test suites alongside grammars and semantic rules, and to use golden outputs or assertions over intermediate representations to detect regressions early.

JetBrains MPS is centered around projectional editing and language composition, with generators that translate abstract syntax trees into target code (Pech et al. 2013). Industrial-scale experience is reported for the mbeddr ecosystem, an MPS-based, extensible set of integrated DSLs and supporting tooling for embedded software development (notably for C), which demonstrates that MPS-based language engineering can scale to large, evolving DSL families and emphasizes extensive automated test suites across language front-end and back-end concerns (Voelter et al. 2013; Ratiu et al. 2018; Voelter et al. 2019).

Rascal is a meta-programming language designed for source code analysis, transformation, and DSL implementation (Klint et al. 2009). Engineering practices in Rascal often focus on making language implementations modular and testable by construction: concrete syntax, analysis, transformations, and code generation are typically organized as separate components with explicit interfaces, supporting independent evolution and unit-style testing of language features. These practices are illustrated in experience reports on modular language implementation in Rascal (Basten et al. 2015).

Racket represents a different point in the design space, promoting language-oriented programming by enabling new languages to be implemented as libraries and macros (e.g., via `#lang`) (Felleisen et al. 2018). Because DSLs are "just programs"

in the host ecosystem, they naturally adopt mainstream software engineering workflows such as unit and regression testing, CI, and modular reuse. Foundational macro technology underpins staged expansion and compositional extension, which is essential for building and evolving such embedded language layers (Flatt 2002).

Across these platforms, a recurring theme is the need for multi-level testing: fast checks for front-end behavior (syntax, editor services, static semantics), regression tests for transformations and generators (often using reference outputs or structured assertions), and end-to-end tests over the produced tooling. However, the "out-of-the-box" testing story and the practical integration into CI/CD pipelines vary widely by platform, and experience reports highlight non-trivial effort to make automated testing systematic and maintainable at scale (Ratiu et al. 2018). Within this landscape, and in line with the challenges highlighted by mapping studies (Iung et al. 2020), our work aims at demonstrating comprehensive testing strategies for an Xtext-based DSL across multiple levels—from unit tests for the runtime library to end-to-end tests for the complete IDE—all integrated into an automated CI/CD pipeline.

9.3 Testing domain-specific languages

The specific challenge of testing DSLs has received attention from researchers. Wu and Gray (2005) propose an approach for automated generation of testing tools for DSLs, including debuggers and test engines, from language specifications. While their work focuses on test tool generation, our approach emphasizes manual creation of comprehensive test suites that leverage existing frameworks (JUnit, SWTBot) and Xtext's testing infrastructure, providing more control over test scenarios and better coverage of edge cases.

Testing support has also been integrated directly into language engineering toolchains. For example, Kats et al. propose integrated language definition testing to support test-driven language development, enabling systematic regression testing of parsers, analyses, transformations, and generated artifacts as part of the language definition workflow (Kats et al. 2011).

Orthogonal to functional testing, mutation testing has been adapted to the DSL setting to assess the fault-detection capability of DSL test suites. Gómez-Abajo et al. propose Wodel-Test, a model-driven framework that synthesizes mutation testing environments from a high-level mutation specification, with the goal of reducing the effort of constructing mutation testing tools for new (meta-model-defined) languages (Gómez-Abajo et al. 2021).

A recurring challenge in DSL testing is the *test oracle problem*, i.e., determining whether the observed output for a given input is correct; in many realistic settings, such an oracle may be unavailable or too costly to apply systematically (Weyuker 1982; Barr et al. 2015; Zhou et al. 2018). This motivates approaches that reduce dependence on explicit expected outputs, such as metamorphic testing.

Cañizares et al. present a "new ideas" approach to automatically engineer metamorphic testing environments for DSLs, leveraging metamorphic relations to derive follow-up tests and expected relationships between outputs when explicit expected outputs are hard to specify (Cañizares et al. 2021).

Finally, in the context of executable DSLs, Khorram et al. adapt the ETSI Test Description Language (TDL) to provide generic testing support for executable models, including mechanisms to execute models and define oracles over runtime state (e.g., via model comparison or OCL queries) (Khorram et al. 2021). Compared to these approaches, our work focuses on an industrial-strength Xtext toolchain and demonstrates how a multi-level test strategy (unit, integration, UI, and end-to-end) can be operationalized within a single automated CI/CD pipeline for Eclipse-delivered DSL tooling.

In addition to these DSL-specific techniques, when a DSL relies on translational semantics (i.e., compilation), established compiler-testing practices inform suitable test oracles. Focusing on testing the correctness of code generation in DSLs, using the expected result of compilation as a test oracle is consistent with how compiler validation is described in the literature. Classic compiler texts treat code generation as a mapping from source programs to target code whose correctness must ultimately be checked on representative test suites (Aho et al. 2006; Fraser and Hanson 1995). In industrial case studies, the correctness of a production compiler is routinely assessed by comparing its generated code and diagnostics against a trusted reference output (Sheridan 2007). Closely related techniques such as differential testing (McKeeman 1998) and randomized differential testing of C compilers (Yang et al. 2011) also use the (dis)agreement between outputs of different implementations as an oracle. Recent surveys of compiler-testing techniques explicitly classify comparison against a reference output—whether from a trusted baseline compiler or from multiple implementations—as a standard and mature family of test oracles in practice (Chen et al. 2020; Tang et al. 2020).

9.4 Eclipse plugin development and testing

Beyond language-level testing, Xtext-based DSL toolchains are typically delivered as Eclipse plug-ins, where correctness also depends on OSGi packaging, p2/target-platform resolution, and the behavior of generated IDE components under UI interaction. Consequently, engineering practices for Eclipse plug-in development and testing are directly relevant to establishing an end-to-end, CI-friendly quality assurance workflow for DSL tooling.

Blewitt (2016) provides practical guidance on Eclipse plugin development, including automated builds using Maven Tycho, though with limited coverage of comprehensive testing strategies. Mazurkiewicz (2010) specifically investigates GUI test automation with SWTBot, demonstrating its integration with continuous integration tools. Our work builds on these foundations by showing how SWTBot tests can be effectively integrated into a complete testing strategy alongside unit and integration tests, all managed through a single Maven/Tycho build.

The challenges of building Eclipse-based academic and research projects are discussed by Grebien (2012) and Müller et al. (2017). Müller et al.'s work is particularly relevant as it documents best practices for using Tycho, Maven, p2 repositories, and target platforms, addressing the "pain points" that developers commonly encounter. Our paper extends this line of work by demonstrating how these build automation practices integrate with comprehensive testing at multiple levels and continuous integration workflows, providing empirical evidence of their effectiveness through concrete build time measurements and test execution analysis.

9.5 Software quality and continuous integration

The importance of comprehensive testing, build automation, and continuous integration for research software has been increasingly recognized in the literature. Basile and ter Beek (2022) present the Contract Automata Library, a software artefact for service composition and controller synthesis based on behavioural contracts. Their work emphasizes similar software engineering practices to those we adopt for Edelta, including the use of Maven for build automation, JUnit for unit testing, GitHub Actions for continuous integration, and SonarCloud for code quality analysis. Like our approach, they demonstrate how comprehensive testing and automated quality checks can be integrated into the development workflow of research software, ensuring maintainability and reliability. Their experience confirms that such practices, while adding initial development overhead, significantly reduce maintenance effort and improve software quality over time, aligning with our findings in the context of DSL development with Xtext and Eclipse.

Seifermann and Krach report on modernizing CI pipelines for the Palladio ecosystem (Seifermann and Krach 2020). Their experience report highlights challenges and solutions when evolving a large, long-lived research codebase towards current CI best practices, including improved traceability of failures and maintainability of build configurations. This complements our contribution by focusing on CI pipeline structure and operability at scale, whereas we provide a deep, data-driven account of testing strategies for Xtext-based DSLs and Eclipse plug-ins, including the quantification of the testing pyramid and the integration of Tycho-driven UI and SWTBot end-to-end tests.

Our work provides empirical validation of the testing pyramid principle (Cohn 2009; Fowler 2013) specifically in the context of DSL and Eclipse plugin development. While the testing pyramid is well-established in software engineering, its application to DSL development with frameworks like Xtext has not been thoroughly documented with empirical data. We show that the pyramid structure—with a broad base of fast unit tests, fewer integration tests, and selective end-to-end tests—is not only applicable but essential for maintaining reasonable CI build times while ensuring comprehensive quality assurance across all aspects of a complex DSL toolchain.

10 Conclusions

We presented a comprehensive, experience-based account of how we develop, test, build, and maintain Edelta, an Eclipse- and Xtext-based framework and DSL for EMF metamodel and model evolution. While developing Edelta, we learned several lessons that may help researchers and practitioners facing similar challenges. Our approach combines multiple testing layers with fully automated build and deployment pipelines to sustain a complex toolchain over time. To make the contribution reusable beyond Edelta, we also distilled the underlying methodology into a general testing and CI framework for Eclipse/Xtext toolchains and discussed portability and reuse across projects.

The key outcome is that a disciplined, multi-level test strategy is feasible for Eclipse/Xtext-based DSL toolchains and can be continuously enforced in CI with predictable cost. Fast unit tests cover most logic in the runtime and refactorings libraries, while focused language tests verify parsing, scoping, linking, validation, type checking, interpretation, and code generation. Integration UI tests exercise the IDE and workspace behavior, and a small set of SWTBot end-to-end tests validates complete user workflows on the product distribution. Our single Maven/Tycho build executes all these tests, produces Eclipse products and a self-contained p2 site, and enables cross-platform CI on GitHub Actions with SonarCloud for continuous quality feedback. On a representative CI run, the full pipeline completes in under 8 minutes while executing 883 tests across all layers, which makes it realistic to apply as an authoritative CI gate on everyday changes.

Threats to validity and limitations include the specificity of the Eclipse/Xtext ecosystem, the characteristics of the Edelta domain, and the costs and fragility trade-offs of IDE- and UI-level testing. Nevertheless, we believe the practices reported here generalize to many language workbenches and plugin-based toolchains that share similar architectural traits (modular services, dependency injection, headless compilation, product packaging), with project-specific adaptation mainly required for language semantics, test fixtures, and UI workflows.

During the development of Edelta, we discovered a few bugs and limitations in Xtext. We fixed those bugs in Edelta by overriding the behavior of the corresponding Xtext components, and we then backported the fixes to Xtext itself as contributions to the open-source project. Similarly, we found a few ways to make the Xtext testing framework more usable, and we also contributed those improvements back to the Xtext project (e.g., for content assist and outline testing). Finally, the techniques and strategies learned while working on the build automation of Edelta, including deployment techniques for p2 sites, were used by the author, as a committer to the Xtext project, to improve the Xtext build and release process itself. In particular, Xtext code base used to be split into 6 separate Git repositories. The core repositories were using Gradle as the build system, while the others, related to Eclipse plugins, were using Maven/Tycho. While this allowed for easy testing of a single core repository, containing only the core Xtext components, it made it hard to test the whole Xtext toolchain end to end. Testing the changes in a core repository in downstream repositories required complex cross-repository build setups, including some manual adaptations to point downstream repositories to local builds of the core repository (such manual adaptations had to be manually removed before committing the changes to the downstream repositories). Thanks to the experience gained while working on the Edelta build and release process, the author personally managed to unify all Xtext repositories into a single Git monorepo (Brousse 2019) and to use Maven/Tycho as the build system for all components, including a completely automated deployment process into a single Maven command. This transformed the original Jenkins configuration file[17], weighing more than a few thousand lines of code, into less than 200 lines of code. This allowed Xtext to survive: maintainers can now focus on the development of new features and bug fixes, rather than fighting with the build system and the release process.

[17] Official Eclipse projects must use the Eclipse foundation Jenkins infrastructure for releasing.

Acknowledgements The author is grateful to Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio for their work on the Edelta project.

Author Contributions Lorenzo Bettini, as the only author, wrote the entire manuscript and also authored all the code shown in the paper.

Funding Open access funding provided by Università degli Studi di Firenze within the CRUI-CARE Agreement.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing Interests The authors declare no competing interests.

References

Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley, Boston, MA, USA (2006)

Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Trans. Software Eng. **41**(5), 507–525 (2015)

Barriga, A., Bettini, L., Iovino, L., Rutle, A., Heldal, R.: Addressing the trade off between smells and quality when refactoring class diagrams. J. Obj. Technol. **20**(3), 1–115 (2021)

Basile, D., ter Beek, M.H.: Contract automata library. Sci. Comput. Program. **221**, 102841 (2022)

Basten, B., Bos, J., Hills, M., Klint, P., Lankamp, A., Lisser, B., Ploeg, A., Storm, T., Vinju, J.: Modular language implementation in rascal-experience report. Sci. Comput. Program. **114**, 7–19 (2015)

Beck, K.: Test Driven Development: By Example. Addison-Wesley, USA (2003)

Bertolino, A., Gao, J., Marchetti, E., Polini, A.: Automatic test data generation for XML schema-based partition testing. In: Second International Workshop on Automation of Software Test, pp. 4–4 (2007). IEEE

Bettini, L., Merelli, E., Tiezzi, F.: X-Klaim is back. In: Models, Languages, and Tools for Concurrent and Distributed Programming: Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday. LNCS, vol. 11665, pp. 115–135. Springer, Cham (2019)

Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, 2nd edn. Packt Publishing, Birmingham, UK (2016)

Bettini, L.: Implementing Type Systems for the IDE with Xsemantics. J. Log. Algebr. Methods Program. **85**(5, Part 1), 655–680 (2016)

Bettini, L.: Jbase: Embedding java expressions in Xtext DSLs. In: Proceedings of the International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools. PPPJ, pp. 16–1166. ACM, New York, NY, USA (2016)

Bettini, L., Crescenzi, P.: An eclipse IDE for teaching java–. In: Software Technologies, pp. 63–78. Springer, Cham (2016)

Bettini, L., Damiani, F.: Xtraitj: Traits for the Java platform. J. Syst. Softw. **131**, 419–441 (2017)

Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Edelta: An approach for defining and applying reusable metamodel refactorings. In: Procs of MODELS 2017 Satellite Event, pp. 71–80 (2017)
Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Quality-Driven detection and resolution of metamodel smells. IEEE Access **7**, 16364–16376 (2019)
Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Edelta 2.0: supporting live metamodel evolutions. In: MODELS, pp. 1–10. ACM/IEEE, New York, NY, USA (2020)
Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Supporting safe metamodel evolution with Edelta. Int. J. Softw. Tools Technol. Transfer **24**, 247–260 (2022)
Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: An executable metamodel refactoring catalog. Softw. Syst. Model. **21**, 1689–1709 (2022)
Bettini, L., Di Salle, A., Iovino, L., Pierantonio, A.: Supporting reusable model migration with Edelta. J. Syst. Softw. **212**, 112012 (2024a)
Bettini, L., Di Ruscio, D., Di Salle, A., Iovino, L., Pierantonio, A.: Keeping clients' models up-to-date with Edelta. In: Proc. of MODELS, pp. 46–50. ACM, New York, NY, USA (2024)
Blewitt, A.: Eclipse Plug-in Development: Beginner's Guide, 2nd edn. Packt Publishing, Birmingham, UK (2016)
Brousse, N.: The issue of monorepo and polyrepo in large enterprises. In: Companion Proceedings of the International Conference on the Art, Science, and Engineering of Programming. ACM, New York, NY, USA (2019)
Cañizares, P.C., Gómez-Abajo, P., Núñez, A., Guerra, E., Lara, J.: New ideas: automated engineering of metamorphic testing environments for domain-specific languages. In: Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2021), pp. 49–54. ACM, New York, NY, USA (2021)
Cauevic, A., Punnekkat, S., Sundmark, D.: Quality of testing in test driven development. In: Proc. Conference on the Quality of Information and Communications Technology, pp. 266–271. IEEE, Lisbon, Portugal (2012)
Chen, L.: Continuous delivery: Huge benefits, but challenges too. IEEE Softw. **32**(2), 50–54 (2015)
Chen, M.-H., Lyu, M.R., Wong, W.E.: Effect of code coverage on software reliability measurement. IEEE Trans. Reliab. **50**(2), 165–170 (2002)
Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., Zhang, L.: A survey of compiler testing. ACM Comput. Surv. **53**(1), 1–36 (2020)
Cicchetti, A., Ruscio, D., Pierantonio, A.: Managing dependent changes in coupled evolution. In: ICMT. LNCS, vol. 5563, pp. 35–51. Springer, Berlin, Heidelberg (2009)
Cohn, M.: Succeeding with Agile: Software Development Using Scrum. Addison-Wesley Professional, Boston, MA, USA (2009)
Desai, C., Janzen, D., Savage, K.: A survey of evidence for test-driven development in academia. ACM SIGCSE Bulletin **40**(2), 97–101 (2008)
Di Ruscio, D., Iovino, L., Pierantonio, A.: Evolutionary togetherness: How to manage coupled evolution in metamodeling ecosystems. In: Graph Transformations. LNCS, vol. 7562, pp. 20–37. Springer, Berlin, Heidelberg (2012)
Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., Hasselbring, W., Massow, R.: Xbase: Implementing Domain-Specific Languages for Java. In: GPCE, pp. 112–121. ACM, New York, NY, USA (2012)
Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Softw. Syst. Model. **8**(4), 479–500 (2009)
Erdweg, S., Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G.D.P., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V.A., Visser, E., Vlist, K., Wachsmuth, G.H., Woning, J.: The state of the art in language workbenches. In: Software Language Engineering (SLE 2013). LNCS, vol. 8225, pp. 197–217. Springer, Cham (2013)
Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., Tobin-Hochstadt, S.: A programmable programming language. Commun. ACM **61**(3), 62–71 (2018)
Flatt, M.: Composable and compilable macros: You want it when? In: Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP 2002), pp. 72–83. ACM, New York, NY, USA (2002)
Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional, Boston, MA, USA (2010)
Fowler, M.: Test Pyramid (2013). https://martinfowler.com/bliki/TestPyramid.html. Accessed: 21 Oct 2025

Fraser, C.W., Hanson, D.R.: A Retargetable C Compiler: Design and Implementation. Benjamin/Cummings, Redwood City, CA, USA (1995)
Gómez-Abajo, P., Guerra, E., Lara, J., Merayo, M.G.: Wodel-test: a model-based framework for language-independent mutation testing. Softw. Syst. Model. **20**(3), 767–793 (2021)
Grebien, T.: Managing academic eclipse-based projects. Diploma thesis, Christian-Albrechts-Universität zu Kiel (2012)
Hebig, R., Khelladi, D.E., Bendraou, R.: Approaches to co-evolution of metamodels and models: A survey. Trans. on Soft. Eng. **43**(5), 396–414 (2017)
Hemmati, H.: How effective are code coverage criteria? In: International Conference on Software Quality. Reliability and Security, pp. 151–156. IEEE, USA (2015)
Herrmannsdoerfer, M., Benz, S., Jürgens, E.: COPE - automating coupled evolution of metamodels and models. In: ECOOP. LNCS, vol. 5653, pp. 52–76. Springer, Berlin, Heidelberg (2009)
Herrmannsdoerfer, M., Vermolen, S., Wachsmuth, G.: An extensive catalog of operators for the coupled evolution of metamodels and models. In: SLE. LNCS, vol. 6563, pp. 163–182. Springer, Berlin, Heidelberg (2010)
Hudak, P.: Domain-specific languages. In: Salus, P.H. (ed.) Handbook of Programming Languages, Volume III: Little Languages and Tools, pp. 39–60. Macmillan Technical Publishing, Indianapolis, IN (1998). Chap. 3
Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Pearson Education, Boston, MA, USA (2010)
Iung, A., Carbonell, J., Marchezan, L., Rodrigues, E., Bernardino, M., Basso, F.P., Medeiros, B.: Systematic mapping study on domain-specific language development tools. Empir. Softw. Eng. **25**(6), 4205–4249 (2020)
Ivanković, M., Petrović, G., Just, R., Fraser, G.: Code coverage at Google. In: Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 955–963. ACM, USA (2019)
Jaime, D., Poizat, P., El Haddad, J., Degueule, T.: Balancing the quality and cost of updating dependencies. In: Proc. Conference on Automated Software Engineering. ASE, pp. 1834–1845. ACM, New York, NY, USA (2024)
Kats, L.C.L., Visser, E.: The Spoofax Language workbench: Rules for declarative specification of languages and IDEs. In: Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010), pp. 444–463. ACM, New York, NY, USA (2010)
Kats, L.C.L., Vermaas, R., Visser, E.: Integrated language definition testing: Enabling test-driven language development. In: Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2011), pp. 139–154. ACM, New York, NY, USA (2011)
Khorram, F., Bousse, E., Mottu, J.-M., Sunyé, G.: Adapting TDL to provide testing support for executable DSLs. J. Obj. Technol. **20**(3), 6–1615 (2021)
Klint, P., Storm, T., Vinju, J.J.: RASCAL: A domain specific language for source code analysis and manipulation. In: Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009), pp. 168–177. IEEE, USA (2009)
Kosar, T., Bohra, S., Mernik, M.: Domain-specific languages: A systematic mapping study. Inf. Softw. Technol. **71**, 77–91 (2016)
Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: From metaphor to theory and practice. IEEE Softw. **29**(6), 18–21 (2012)
Kula, R.G., German, D.M., Ouni, A., Ishio, T., Inoue, K.: Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. Empir. Softw. Eng. **23**(1), 384–417 (2018)
Laukkanen, E., Itkonen, J., Lassenius, C.: Problems, causes and solutions when adopting continuous delivery - A systematic literature review. Inf. Softw. Technol. **82**, 55–79 (2017)
Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. J. Syst. Softw. **101**, 193–220 (2015)
Marick, B., Smith, J., Jones, M.: How to misuse code coverage. In: Interational Conference on Testing Computer Software, pp. 16–18 (1999)
Maximilien, E.M., Williams, L.: Assessing test-driven development at IBM. In: Proc. Conference on Software Engineering, pp. 564–569. IEEE, Portland, Oregon, USA (2003)

Mazurkiewicz, M.: GUI test automation with SWTBot. Master's thesis, VAMK University of Applied Sciences (2010)
McKeeman, W.M.: Differential testing for software. Digit. Tech. J. **10**(1), 100–107 (1998)
Medeiros, S.Q., Lefeuvre, R., Combemale, B., Perez, Q.: Evaluating the energy profile of tasks managed by build automation tools in continuous integration workflows: The case of apache maven and gradle. In: ICT4S, pp. 1–11. IEEE, Dublin, Ireland (2025)
Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4), 316–344 (2005)
Müller, S., Fischer, P.M., Schlauch, T.: About tycho, maven, p2 and target-platforms: From pain to best practice. EclipseCon (2017)
Munir, H., Moayyed, M., Petersen, K.: Considering rigor and relevance when evaluating test driven development: A systematic review. Inf. Softw. Technol. **56**(4), 375–394 (2014)
Nassar, N., Kosiol, J., Kehrer, T., Taentzer, G.: Generating large EMF models efficiently: A rule-based, configurable approach. In: Fundamental Approaches to Software Engineering, pp. 224–244 (2020). Springer
Paige, R.F., Matragkas, N., Rose, L.M.: Evolving models in model-driven engineering: State-of-the-art and future challenges. J. Syst. Softw. **111**, 272–280 (2016)
Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M.: Mutation testing advances: an analysis and survey. In: Advances in Computers, vol. 112, pp. 275–378. Elsevier, USA (2019)
Parsai, A., Demeyer, S.: Comparing mutation coverage against branch coverage in an industrial setting. Int. J. Softw. Tools Technol. Transfer **22**, 365–388 (2020)
Pech, V., Shatalin, A., Voelter, M.: JetBrains MPS as a tool for extending Java. In: Proceedings of the 11th International Conference on the Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '13), pp. 165–168. ACM, New York, NY, USA (2013)
Petrović, G., Ivanković, M.: State of mutation testing at Google. In: Proceedings of Software Engineering in Practice. ICSE-SEIP, pp. 163–171. ACM, USA (2018)
Prakash, M.: Software build automation tools a comparative study between Maven, Gradle, Bazel and Ant. Int J Softw Eng Appl **13**, 1–20 (2022)
Prasanna, D.R.: Dependency Injection: Design Patterns Using Spring and Guice, 1st edn. Manning, USA (2009)
Ratiu, D., Voelter, M., Pavletic, D.: Automated testing of DSL implementations–experiences from building mbeddr. Softw. Qual. J. **26**(4), 1483–1518 (2018)
Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C., Poulding, S.: Epsilon flock: a model migration language. Softw. Syst. Model. **13**(2), 735–755 (2012)
Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. Computer **39**(2), 25–31 (2006)
Seifermann, S., Krach, S.: Catching up with state of the art continuous integration pipelines in palladio-an experience report. Softwaretechnik-Trends **40**(3), 52–54 (2020)
Shahin, M., Babar, M.A., Zahedi, M., Zhu, L.: Beyond continuous delivery: an empirical investigation of continuous deployment challenges. In: ESEM, pp. 111–120 (2017). IEEE
Sheridan, F.: Practical testing of a C99 compiler using output comparison. Softw. Pract. Exper. **37**(14), 1475–1488 (2007)
Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston, MA, USA (2008)
Taentzer, G., Mantz, F., Arendt, T., Lamo, Y.: Customizable model migration schemes for meta-model evolutions with multiplicity changes. In: MODELS. LNCS, vol. 8107, pp. 254–270. Springer, Berlin, Heidelberg (2013)
Tang, Y., Ren, Z., Kong, W., Jiang, H.: Compiler testing: A systematic literature analysis. Front. Comp. Sci. **14**(1), 1–20 (2020)
Tom, E., Aurum, A., Vidgen, R.: An exploration of technical debt. J. Syst. Softw. **86**(6), 1498–1516 (2013)
Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL engineering: Designing, implementing and using domain-specific languages, (2013). Available online at dslbook.org
Voelter, M., Ratiu, D., Kolb, B., Schätz, B., et al.: mbeddr: instantiating a language workbench in the embedded software domain. Autom. Softw. Eng. **20**, 339–390 (2013)
Voelter, M., Kolb, B., Szabó, T., Ratiu, D., Deursen, A.: Lessons learned from developing mbeddr: a case study in language engineering with MPS. Softw. Syst. Model. **18**, 585–630 (2019)
Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: ECOOP. LNCS, vol. 4609, pp. 600–624. Springer, Berlin, Heidelberg (2007)

Warmer, J.B., Kleppe, A.G.: The Object Constraint Language: Getting Your Models Ready for MDA, 2nd edn. Addison-Wesley Professional, Boston, MA, USA (2003)
Weyuker, E.J.: On testing non-testable programs. Comput. J. **25**(4), 465–470 (1982)
Williams, L., Maximilien, E.M., Vouk, M.: Test-driven development as a defect-reduction practice. In: Proc. International Symposium on Software Reliability Engineering, pp. 34–45. IEEE, Denver, Colorado, USA (2003)
Wong, W.E., Mathur, A.P.: Reducing the cost of mutation testing: An empirical study. J. Syst. Softw. **31**(3), 185–196 (1995)
Woodward, M.R.: Mutation testing–its origin and evolution. Inf. Softw. Technol. **35**(3), 163–169 (1993)
Wu, H., Gray, J.: Automated generation of testing tools for domain-specific languages. In: Proc. ASE, pp. 436–439. ACM, New York, NY, USA (2005)
Wu, H., Monahan, R., Power, J.F.: Metamodel instance generation: A systematic literature review. arXiv preprint arXiv:1211.6322 (2012)
Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011), pp. 283–294. ACM, San Jose, CA, USA (2011)
Zarnekow, S.: Xtext corner #8 - libraries are key (2012). https://zarnekow.blogspot.com/2012/11/xtext-corner-8-libraries-are-key.html. Accessed: 15 Jan 2026
Zhou, Z.Q., Towey, D., Poon, P.-L., Tse, T.H.: Introduction to the special issue on test oracles. J. Syst. Softw. **136**, 187 (2018)