Text Probes

Unit Testing of Source Code Analysis

Risberg Alaküla, Anton; Hedin, Görel; Fors, Niklas

Published in:
IEEE International Conference on Source Code Analysis & Manipulation (SCAM), 2026

2026

Document Version:
Peer reviewed version (aka post-print)

Text Probes: Unit Testing of Source Code Analysis

Anton Risberg Alaküla
Department of Computer Science
Lund University
Lund, Sweden
https://orcid.org/0000-0003-0814-3367

Görel Hedin
Department of Computer Science
Lund University
Lund, Sweden
https://orcid.org/0000-0002-3003-2623

Niklas Fors
Department of Computer Science
Lund University
Lund, Sweden
https://orcid.org/0000-0003-2714-9457

***Abstract*—Compilers and other source code analysis tools are typically tested at the system level, running the tool on an input program and checking generated results. However, this does not fit agile test-driven development, where internal results need to be individually unit tested. Such unit tests are seldom written as they typically require cumbersome code navigating the internal representation.**

We suggest a solution to this problem through the introduction of *text probes*: structured comments that embed tests into input source files. Text probes are placed at the position they test, keeping navigation code short. We have implemented support for text probes, both as a standalone test runner and as an integration into an editor, supporting live feedback on test results as well as code completion for internal properties and expected values.

We have evaluated the approach by assessing its applicability for several tools and languages, including a Java 11 compiler, finding that it has good performance and supports a wide range of tests. We have also used it in courses on compilers and program analysis and received overall positive feedback from students.

***Index Terms*—unit testing, compilers, program analysis.**

I. Introduction

Source code analysis is an important part of compilers, bug checkers, and language-based editors. Such analysis can include, e.g., name binding, type checking and inference, control-flow and dataflow analysis, and support various tasks such as code generation, bug detection, bug fix suggestions, code completion, and code navigation.

Often, these analyses build on each other, each with its own API towards downstream analyses and tasks. Nevertheless, they are typically tested at the system level, i.e., running the tool on an input program and checking the generated results. For example, a compiler can be tested by running it on a number of incorrect input programs to check that expected errors and warnings are generated, and by running the generated code for a number of correct input programs, to check that they compute the expected result.

However, testing individual analyses this way is indirect, and if a test fails, it can be difficult to pinpoint the source of the error. Furthermore, system tests do not support agile test-driven development of the analyses, as they require tool output generation to exist before testing can start. Instead, unit testing would be desirable.

Funded by ELLIIT - Excellence Center at Linköping - Lund in Information Technology. Also supported by Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

To support unit testing of source code analysis, the tests need to navigate to particular positions in the internal representation to test internal results at those positions, like the inferred types of expressions (see e.g. Figure 1a), declaration sites of references, etc. The internal representation is typically an abstract syntax tree (AST), and we refer to this kind of tests as *AST-navigated tests*. AST-navigated tests are, however, seldom used in practice. We believe the main reason is that, without special support, the navigation code is tedious to program, it requires detailed knowledge of the AST, and it is fragile to refactorings and other changes to the AST. Donaldson explicitly stated that "*function level unit testing is not that common because it's actually very cumbersome*" in his talk on testing programming language implementations at the PLISS summer school in 2017 [1].

Existing research on testing of compilers primarily addresses system-level testing, focusing on generation of input programs and on differential testing (comparing the output of different compilers for the same language). See, for example, the surveys by Chen et al. [2], [3] and the systematic literature analysis by Tang et al. [4], none of which discuss unit testing of the compiler implementation.

In this paper, we introduce a novel technique, *text probes*, designed to support writing AST-navigated tests. We have implemented this approach both as a test runner to be run from build systems, and as interactive editor support through an integration into CodeProber, an IDE that supports live exploration of source code analysis results [5]. Our integrations work out of the box for language tools developed using the metacompiler JastAdd [6], but can also be adapted to other tools. To demonstrate this, we prototyped text probe support for `rust-analyzer`, a language server for Rust.

In the following, we first give a motivating example of an AST-navigated test and present our design goals (Section II). We then present our main contributions:

- Our novel technique, *text probes*, and how we integrated this technique into a test runner and an IDE (Section III).
- An evaluation of text probes on existing compilers and program analysis tools, assessing both applicability (Section IV) and performance (Section V).
- An assessment of student perceptions of text probes in courses on compilers and program analysis (Section VI).

We end the paper with related work (Section VII), and with some concluding remarks (Section VIII).

Accepted for publication in the *IEEE International Conference on Source Code Analysis & Manipulation (SCAM)*, 2026. DOI: to appear.

```
IC  String code = "import java.util.*;"
        + "public class Test {"
        + "  void m1(List<? extends String> names) {"
        + "    names.stream().mapToInt(name -> name.length());"
        + "  }"
        + "}";
    CompilationUnit cu = parseCompilationUnit(code);
    MethodDecl m1 = (MethodDecl) cu.getTypeDecl(0).getBodyDecl(0);
    Dot dot1 = (Dot) ((ExprStmt) m1.getBlock().getStmt(0)).getExpr();
    Dot dot2 = (Dot) dot1.getRight();
N   MethodAccess mapToInt = (MethodAccess) dot2.getRight();
    LambdaExpr lambda = (LambdaExpr) mapToInt.getArg(0);
    TypeDecl anonType = lambda.toClass().type();
    for (BodyDecl decl : anonType.getBodyDeclList()) {
      if (decl instanceof MethodDecl) {
VE      MethodDecl method = (MethodDecl) decl;
        String typeSignature = method.methodTypeSignature();
        System.out.format("Type signature of %s: %s%n", method.name(),
        typeSignature);
      }
    }
```

(a) Manually programmed AST-navigated test

```
    import java.util.*;
    public class Test {
      void m1(List<? extends String> names) {
        names.stream().mapToInt(name -> name.length());
N       // [[$lambda:=^LambdaExpr]]
      }
IC  }
    /*
N   [[$list:=$lambda.toClass.type.getBodyDeclList]]
VE  [[$list.getNumChild=1]]
    [[$list.getChild(0).methodTypeSignature="(Ljava/lang/String;)I"]]
    */
```

(b) AST-navigated test using text probes.

Fig. 1. Two implementations of the same test case, checking the inferred type of a lambda expression in a Java compiler. Regions of the tests with similar roles across the two versions are labeled with Input Code (IC), Navigation (N), and Value Extraction (VE) respectively.

II. Motivation

In compilers and static analyzers, the main internal representation is typically an Abstract Syntax Tree (AST), constructed from parsing the input program. A source code analysis computes new results based on both the AST and on upstream analysis results. Results are often local to specific AST nodes. For example, a type analysis can compute the type of an expression and store it in association with that expression. An analysis may also compute aggregate information or new internal representations of the program, like a control-flow graph, but often some traceability information is kept to be able to associate the computed information to the original AST and/or to line numbers in the input program.

A. AST-navigated tests

To do test-driven development of an analysis, it is interesting to be able to easily write *AST-navigated tests*, i.e., tests that check some computed value, associated with a particular node in the AST. We define an AST-navigated test as consisting of the following elements.

- *Input Code*: The source code of the input program.
- *Setup*: The part of the test that computes the test fixture. This typically includes parsing the input code, and potentially running a set of analyses.
- *Navigation*: The part of the test that navigates to the relevant position in the internal representation, typically to a node in an AST.
- *Value Extraction*: The part of the test that extracts the value to test, e.g., a type, a set of live variables, etc.
- *Expected Value*: The value that the extracted value should have for the test to succeed.

B. Manually written AST-navigated tests

To illustrate how AST-navigated tests can be written when there is no particular support for them, we will consider the regression test suite for the Java compiler ExtendJ [7], [8]. While most of the tests are system-level tests, there are a few AST-navigated tests that check intricate parts of the static-semantic analysis.

Figure 1a shows an AST-navigated test from this suite that checks the generated method type signature for a lambda expression (`api/jsr335/lambda_01p` from commit `6f265e8`). The Input Code is written as a quoted string in the test. For this compiler, the Setup is very simple, and consists only of parsing the Input Code into an AST. No analysis has to be explicitly triggered because the compiler is implemented using a demand-driven technique, and all analysis happens on demand when results, like types, are accessed. The biggest part of the test is the Navigation which consists of very intricate laborious code that traverses the AST from the root down to the generated method for the lambda expression. Value Extraction is then done simply by calling `methodTypeSignature()`. The test writes the extracted value to standard output, and a test runner compares it to the Expected Value which in this case is stored in a separate file. (The expected value for this test is the string `"(Ljava/lang/String;)I"`, representing a method signature that takes a `java.lang.String` argument and returns a primitive integer.)

While the Navigation part can be somewhat simplified by adding helper methods, see Section VII, its complexity makes it unattractive to write many of these tests. Furthermore, the code is fragile in the sense that even minor changes to the AST design might break the test.

C. AST-navigated tests using text probes

In test-driven development, automated unit tests are written in a tight loop involving writing tests, developing production code, and refactoring both the test code and production code [9]. For this development style to work in practice for AST-navigated tests, much simpler ways are needed than those shown in Figure 1a. In Figure 1b we show how the corresponding test can be written much more easily using our new concept of *text probes*. Text probes are tests written as structured comments directly in the Input Code. For details on syntax and semantics, see Section III. We also comment more on this particular test in Section IV.

Text probes are designed to meet the following overall design goals for AST-navigated tests: Single File, Clear Navigation Positions, Editor Support, Plain Text, and Refactorability.

1) Single File: To make it easy to write, understand, and refactor tests, it is desirable that all information specific to the test can be kept together, in a single file. Text probe tests meet this goal by making Navigation, Value Extraction, and Expected Value explicit in the Input Code. The Setup is normally the same for a complete test suite, and is instead part of the test runner. The manual test does not meet this goal since the Expected Value is in a separate file. It could of course easily have been rewritten to do so, but it follows the pattern for system tests where a separate file makes more sense as they often have large generated results.

2) Clear Navigation Positions: It should be easy to understand what position is navigated to in the Input Code. Text probe tests meet this goal by allowing probes to be placed as comments directly in the input. The manual test does not meet this goal because of the complex navigation code.

3) Editor Support: Support for interactively creating tests and viewing test results encourages the creation of tests. For a manual test, a language server can provide code completions on the available method names. However, the language server has no knowledge of the dynamic AST structure for the particular input program, so the developer has to manually add typecasts, like the casts to `MethodDecl`, `ExprStmt`, etc., in Figure 1a. An IDE with support for text probes can be made aware of the dynamic AST structure and the actual values computed by the test, and give much richer code completion and highlighting support, see Section III-C.

4) Plain Text: Tests should be stored in a plain text format, suitable for version control, and be possible to edit in any editor. This goal is met both by text probes and manual tests. Importantly, it is possible to understand and edit text probes without using the special editor support.

5) Refactorability: If the underlying tool is refactored, e.g., changing the AST structure used, this may call for corresponding refactoring of test cases. The manual tests are brittle to such changes because of the many type casts and hard-coded child accessors. Text probes avoid much of this problem by reducing the amount of navigation code needed. Methods called inside the probes will, however, not be discovered by general refactoring tools, and is an area for future research.

```
text_probe ::= "[[" statement "]]"
statement ::= assertion | var_decl | plain_query
assertion ::= query ("!=" | "=") expected_value
            | query "!"
var_decl ::= "$" ID ":=" query
plain_query ::= query

query ::= type_query | var_query
type_query ::= "^"? ID ("[" NUM "]")? prop_list
var_query ::= "$" ID prop_list

prop_list ::= ("." prop)*
prop ::= ID ("(" (expr ("," expr)*)? ")")?
expected_value ::= expr
expr ::= NUM | STRING | BOOL | query
```

Listing 1. Text probe grammar.

```
1  class Foo {
2    // [[CompilationUnit.problems]]
3    int x = 1 + 2; // [[AddExpr.constant=3]]
4    int y = 4;     // [[$field:=FieldDeclarator]]
5    int z = y;     // [[VarAccess.decl=$field]]
6
7    int bar(int y) { // [[$param:=ParameterDeclaration]]
8      return y + this.y; // [[VarAccess[0].decl=$param]]
9                         // [[^VarAccess[1].decl=$field]]
10   }
11 }
```

Listing 2. Example text probes: a plain query (line 2), assertions (lines 3, 5, 8, 9), and variable declarations (lines 4, 7).

III. Text Probes

We now present the syntax and semantics of text probes, and demonstrate how they can be supported in an IDE. We also demonstrate how the language, while intentionally designed to be very simple, supports the use of frameworks to add new capabilities like fluent testing and group assertions.

A. Syntax

Text probes are structured comments in source code. There are three kinds of text probes: assertions, variable declarations, and plain queries. See Listing 1 for the full syntax. Listing 2 shows simple examples of the three kinds of probes.

1) Assertions: An assertion compares a query result with an expected value. Line 3 in Listing 2 shows an example assertion for testing constant propagation in a compiler. The query is an example of a *type query* that starts with an AST type name (`AddExpr` in this case) that implicitly references an AST node on the same line in the source file. Via this reference, a chain of methods can be called, accessing information from that position in the AST. In this case, the method `constant()` is called which is expected to return the result of the constant propagation algorithm, in this case the value 3.

2) Variable declarations: A variable declaration associates the result of a query with a variable name, for later use in other text probes. Lines 4 and 5 in Listing 2 exemplify this with a test of the name analysis in a compiler. In this case, the `decl` method on the variable access should return a reference to the declaration of the variable. For example, the access to `y` on line 5 should be bound to the AST node representing the field declaration on line 4. This is checked by declaring a variable `$field` on line 4, and using it for the expected value in the assertion on line 5.

3) Plain queries: A plain query is used for exploration in an IDE. An IDE supporting text probes can evaluate the query and show its result. This is analogous to a print statement or breakpoint inside a normal unit test - useful for figuring out the current behavior while creating a new test or debugging a failing test. Line 2 in Listing 2 shows a plain query for the list of compilation problems.

4) Selecting AST nodes: If there are multiple nodes of the same type on the same source line, they can be disambiguated using indices on the type name. The nodes on a source line are ordered according to a depth-first, left to right search in the AST, and "`[N]`" is used to index into the resulting list. The assertions on lines 8 and 9 in Listing 2 exemplify this,

using `VarAccess[0]` and `VarAccess[1]` to select the two variable accesses on line 8. The assertion on line 9 further illustrates that a type query can be applied to the previous source line, by preceding it with "^". This changes the node selection to be on the closest preceding line that does not have a type query with "^". This allows multiple probes targeting a certain line to be written on subsequent lines, so that very wide lines can be avoided, improving readability.

B. Semantics

Text probes are evaluated either as part of a test runner, or within an IDE supporting text probes. Before evaluating the probes, the underlying compiler or analysis tool needs to parse the source code into an AST, and perform any analysis necessary to allow the analysis results to be accessed via the AST nodes. Probes are dynamically evaluated using reflection, they are evaluated in a well-defined order, and certain well-formedness checks are performed before evaluation.

1) Dynamic evaluation: Our implementation of text probes requires the underlying tool to represent the AST as Java objects, and evaluates the methods called in the probes using reflection. The calls within a query are thus dynamically evaluated and do not need to follow Java static-semantic rules, e.g., casting to a static type. If an error is detected during evaluation, e.g., a missing method, wrong number of parameters, etc., the probe is marked as failing. An assertion that fails, or depends on a failing variable declaration probe, is marked as a failed test.

2) Evaluation order: For text probes to be easy to understand, the methods called by the probes should be without observable side effects and possible to call in any order. However, we would like probes to behave deterministically also when this is not the case, to support debugging as well as special cases when intended side-effects are used. Evaluation follows a basic order from top to bottom, left to right. However, a query may use a text probe variable declared anywhere in the file. Therefore, if a variable is used that has not yet been evaluated, it will be recursively evaluated first. As an example, consider the following probes:

```
... // [[$x:=$y.z]] [[$x.n=1]] [[$a:=B.c]] [[$y:=F.g]]
```

Here, evaluation starts with the variable `$x`, but since this probe uses `$y`, `$y` is evaluated before `$x`. Then evaluation proceeds with the assertion `$x.n=1`, and finally the last remaining variable `$a`. The resulting evaluation order is thus `$y:=...`, `$x:=...`, `$x.n=1`, `$a:=...`.

3) Well-formedness checks: Before probes are evaluated, the following well-formedness rules are checked. If they are not fulfilled, no probes are evaluated, and all assertions are considered to have failed.

- Each variable used in a query must be declared.
- There must be no duplicate declarations for variables.
- Cyclic definition of variables is not allowed.
- It is not permitted to mix "^" and non-"^" type queries on the same line. This rule exists purely for readability.

```
class Foo {
    // [[$cu:=CompilationUnit]]
    // [[$cu.problems.isEmpty = true]]
    int x = 1 + 2; // [[AddExpr.constant=3 ✓]]
    int y = x * 14; // [[MulExpr.constant=40 ✗ Actual: 42]]
}
```

Fig. 2. Rendering of text probes in the IDE.

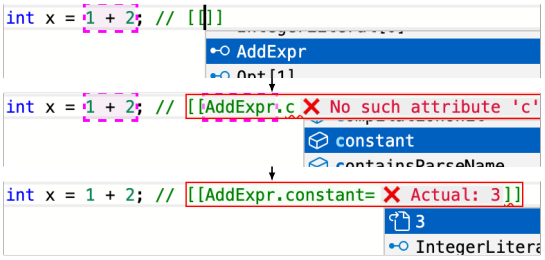

Fig. 3. Text probe interactively created in the IDE, with code completion for AST name, property name, and computed value.

C. IDE Integration

Text probes can be created, edited, and evaluated using any text editor and command-line environment. However, integration into an IDE can significantly improve the experience. We have integrated support for text probes in CODEPROBER, an editor allowing users to interactively select and view intermediate program analysis results [5]. Our integration adds support for text probes, with highlighting, code completion, hovering support, and liveness.

1) Highlighting: Each text probe is highlighted in color, and some additional information is rendered, e.g., to make it easy to distinguish passing and failing assertions. For a passing assertion, a green checkmark is rendered. For a failing assertion, a red × mark is rendered, along with the actual value. For a plain query, the actual value is also rendered. The actual values are synthetic annotations and cannot be edited.

Figure 2 shows examples of text probes in the IDE. The assertions are highlighted in green: the passing one with a green checkmark, the failing one with a red bounding box, a red × mark, and the actual value: 42. The variable declaration is highlighted in yellow. The plain query, highlighted in blue, shows that the actual list of compilation problems is empty.

2) Code Completion: Code completion is provided for all parts of a text probe: AST node names, method names, and actual values. This is illustrated in three steps in Figure 3, showing the addition of an assertion.

In the first step, the user has entered `[[]]` to write a text probe. The IDE then shows all available AST nodes on that line, and the user selects `AddExpr` to start writing a query. Note that the code span corresponding to the `AddExpr`, i.e., `1+2`, is highlighted with a dashed pink box, giving the user feedback on what code part the selection corresponds to.

In the second step, the user writes the character `c` to start writing a method call, and the IDE then shows all methods on the `AddExpr` node that start with `c`. The IDE also tries to

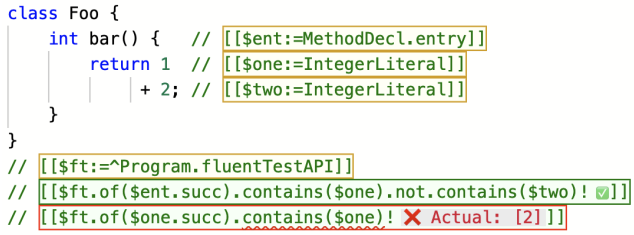

Fig. 4. Fluent-style testing of the successor relation of a control-flow graph. Squiggly-line feedback shows what part of a call chain fails.

evaluate the text probe, but since there is no method named `c`, the probe is rendered with a red bounding box, a red × mark, and an error message "`No such attribute 'c'`".

In the third step, the user has selected the method `constant` and written an = sign to start entering the expected value. The IDE has then reevaluated the probe and marked it as failed, since the current expected value is the empty string, but the actual value is 3. The IDE also shows a code completion menu with the actual value 3 at the top. If the user selects that value, the probe is again evaluated and turns into a passing assertion with a green checkmark.

3) Hovering: All parts of a text probe can be hovered to see their values. This can be useful to see intermediate values in multi-step property chains like "`$cu.problems.isEmpty`" in Figure 2. If the hovered value is an AST node, then the span of the node gets highlighted with a dashed box, like during code completion (Figure 3). If the hovered value is a method call, e.g., `problems`, the current value of that call is shown.

4) Liveness: All results in the IDE are *live*, meaning they immediately respond to input updates. For example, in Figure 2, if the user were to change `1 + 2` to `1 + 3`, the green assertion on the same line would turn red, and the checkmark would be replaced with "`× Actual: 4`".

The IDE also updates the information when the underlying language implementation is updated, e.g., after rebuild of the compiler under test.

D. Fluent-Style Assertions

Some test frameworks, like `AssertJ` [10] and `Chai` [11], have a *fluent interface* where a chain of method calls reads like a sentence [12]. This allows multiple conditions to be tested in the same assertion, like the `AssertJ` snippet below:

```
assertThat(x).hasSize(1).contains("y");
```

Fluent test frameworks can be used in text probes by accessing the framework via a method on an AST node. A fluent-style assertion can cause test failure for any of the called methods in the chain, in which case typically an `AssertionError` is thrown. In evaluating text probes, such errors are caught and treated as test failures.

Figure 4 shows an example of fluent-style testing of the successor relation of a control-flow graph. Here, a variable `$ft` is defined to let all tests access the framework easily.

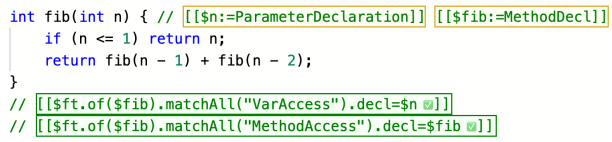

Fig. 5. Text probes using `matchAll` to assert properties on a group of nodes simultaneously.

The second assertion shows an example call chain, testing that the entry node of the control-flow graph has a successor node for `1`, but not for `2`. The third assertion, which fails, asserts that the node for `1` should include itself in its successor list, which it does not. The IDE gives the user feedback by using a squiggly line to highlight the part of the assertion that failed, and uses information in the thrown error to show the actual value of the preceding object in the call chain (the successors list of the control-flow node for `1` in this case).

The text probe grammar supports assertions ending with simply "`!`", which is treated as syntactic sugar for "`!=false`". This was added specifically to help fluent-style testing, as such a test normally doesn't care about the final return value of a method chain. The example in Figure 4 uses a small customized framework, `fluentTestAPI` [13], but any framework can be used, as long as it throws `AssertionError` at failures, and is made accessible via a node in the AST.

E. Group Assertions

A test file may contain multiple identical assertions on different AST nodes. For example, a test may want to assert that all variables inside a certain method have the same type. In this case, being able to assert properties on a group of nodes simultaneously is beneficial. Figure 5 shows text probes that use a method `matchAll` to find all nodes with a certain type in a subtree, and then they make assertions on the nodes as a group. This is implemented within the previously mentioned fluent-testing framework, using CodeProber-specific APIs for dynamically listing and evaluating properties. The implementation added 74 lines of code to the fluent-testing framework.

These examples of fluent testing and group assertions illustrate how the testing language can support new capabilities, by implementing frameworks accessed via AST nodes.

IV. Case Studies

Text probes were specifically designed to support AST-navigated tests and a test-driven development workflow. However, other styles of tests, like system-level tests, are also possible to express using text probes. In this section we present our experiences with converting parts of three existing test suites to use text probes. We do this in order to explore what kinds of tests work well with text probes, and what kinds of tests are better expressed using other testing approaches. We chose the test suites for the Java compiler ExtendJ [7], an intraprocedural analyzer for Java called IntraJ [14], and

a compiler and editor for an automation language called Bloqqi [15]. Additionally, we implemented prototype support for text probes for a Rust language server.

A. Java Compiler (ExtendJ)

The regression test suite for the Java compiler ExtendJ consists of 1811 test cases. Almost all tests are system tests, running the compiler on an input file, and optionally running the generated bytecode. $\sim 80\%$ of the tests perform compile-time checks, and the remaining $\sim 20\%$ check runtime behavior. Only 17 of the tests are AST-navigated tests.

1) Converting System Tests: We added two methods to the AST to be able to convert the system tests in a simple way: "`Program.uiProblems`" which compiles the program and returns a list of compile-time messages, and "`Program.genCodeAndRun`" which compiles the program to bytecode, runs the bytecode, and returns all printed messages in a list. The methods are thin wrappers around existing functionality, requiring 24 and 46 lines of code respectively.

System tests could then be converted by adding a text probe that uses one of these methods. For example, to test that the compilation succeeds, a probe

```
[[Program.uiProblems.isEmpty=true]]
```

can be added. Replacing a system test with a text probe this way does not give any particular advantages, and traditional system tests may be preferable, e.g., to support differential testing. They may also be preferable when the expected value is large, and it is unnatural to write it inside a text probe. An example of this is testing pretty-printing, where it is more natural to keep the output in a separate file that is easy to inspect than to write it as a rather unreadable expression inside a probe.

However, for the compile-time tests, we believe text probes enable simpler, more natural tests, either as a replacement for the current tests, or as additional finer-grained tests. For example, consider a test that checks that a certain input program gives no compile-time errors. Such a test is often aimed at checking a specific part of the compilation, e.g., type checking. By checking the inferred types directly using text probes, the test can better explain what is tested, and give a pinpointed regression failure for a faulty compiler.

2) Converting AST-Navigated Tests: The 17 AST-navigated tests all look similar to the one in Figure 1a. To convert them, we first created a new source file corresponding to the Input Code in the original test. Instead of the navigation code identifying AST nodes, we could place text probes directly in the source file. For some tests we changed the logic slightly to make the test more natural. For example, the test in Figure 1a uses a loop and a conditional statement to access the only element in the list. In our converted test, see Figure 1b, we instead use one text probe to explicitly assert that the list has a single element, and another one to check the type signature of that element. Our test is much shorter than the original one and arguably much easier to understand, at least for someone familiar with text probe syntax. In total, the original AST-navigated tests were 727 lines of code and 106 lines of comments. Our converted tests are 93 lines of code and 96 lines of comments.

Two of the AST-navigated tests had side-effects and were therefore not ideally suited for text probes. Nevertheless, they could both be converted. One of them tests the flushing of a manual cache of types in the compiler and was straightforward to convert because of the deterministic probe evaluation order. The other side-effectful test tested that error messages are printed to standard out when invalid options are supplied to ExtendJ. Text probes cannot access standard out messages directly, so we could not make an AST-navigated conversion of this test. Instead, we converted this test using the `Program.genCodeAndRun` method.

B. Intraprocedural Analyzer for Java (IntraJ)

IntraJ [14] is an intraprocedural program analyzer for Java built on top of ExtendJ. The test suite has two main groups of tests: control-flow graph tests and dataflow analysis tests. We converted a subset of the tests in each group.

1) Converting Control-Flow Graph Tests: IntraJ tests its control-flow graph structure by serializing the entire graph to a Graphviz dot-file, and then comparing the output with an expected value. There are 164 such test cases, covering all the language constructs in Java. In case of a regression failure, the dot files are typically rendered to PDF and manually inspected.

It is possible to convert these tests to text probes in a similar way as for the system tests of the ExtendJ Java compiler: a thin wrapper method `Program.toDot` is added that computes the dot file, and can then be used in a text probe. Rather than comparing the very long string corresponding to the dot file directly in the probe, we used the "$ft" fluent test helper shown in Figure 4, as follows:

```
[[$ft.of(Program.toDot).matchesSnapshot!]]
```

Here, `matchesSnapshot` reads the expected value from a neighboring file, just like the original test case.

Again, just converting system tests like this to text probes does not give any particular advantage. However, a problem with these tests is that they are very large in scope, so test failures become difficult to debug. By using text probes it is possible to complement these tests with more fine-grained tests, precisely checking the successor relation between two nodes in the graph, like the test shown in Figure 4.

2) Converting Dataflow Analysis Tests: IntraJ implements several dataflow-based analyses for Java, on top of the control-flow graph, for example to detect if a variable can have the value `null`. The analysis results lead to methods like "`isNullable`" on AST nodes being `true` or `false`. Then, these methods are used to emit semantic warnings. IntraJ tests its analyses by simply counting the number of warnings that were generated in a given file.

Testing for the number of generated warnings can be expressed in text probes by writing

```
[[CompilationUnit.problems.size=N]]
```

However, this style of testing only indirectly tests the underlying property which is actually the focus of the test case.

```
fn x() -> i32 { 1 } // [[$x:=FN]]

fn y() -> bool {
    let v = x()
    // [[^NAME_REF.decl=$x ☑]] [[$v:=^IDENT_PAT]]
    v > 23
    // [[^NAME_REF.decl=$v ☑]]
}
```

Fig. 6. Name analysis test for `rust-analyzer`.

Therefore, when converting these tests to text probes, we have replaced the existing assertions with more precise assertions, like `[[VarAccess.isNullable=true]]` to check that a given variable reference is computed as nullable.

C. Automation Language (Bloqqi)

Bloqqi [15] is a visual language for building automation systems. Bloqqi code is primarily created and edited with a graphical editor, but when saved it is serialized to a human-readable textual format that can be manually edited too. Bloqqi's test suite includes a large number of tests for semantic analysis and simulated editor actions. Many of the semantic analysis tests are AST-navigated, and could be converted similarly to the AST-navigated tests in ExtendJ, resulting in much more concise and readable tests. However, most of the editor tests mutate the AST, and these tests were in general not possible to express using text probes.

D. Rust-Analyzer

To show that text probes can be used also with tools not developed using the JastAdd metacompiler, we have created a prototype supporting text probes for `rust-analyzer`, a language server for the Rust programming language. `rust-analyzer` computes many of its semantic properties using a framework called `salsa`, which supports on-demand computation. Many of the computations can be connected to nodes in the AST, which fits with text probes. We forked `rust-analyzer` and added basic CodeProber support to it, which includes support for text probes. Figure 6 shows an example of a name analysis test we wrote using text probes.

We added code to the prototype that manually lists what properties are available for certain node types. This code currently supports 11 properties, including name-bindings, type inference, macro expansion, and more. A better implementation would likely involve integrating directly with the `salsa` framework to get automatic listing of all properties inside `rust-analyzer`, similar to how CodeProber can automatically list all properties in tools built by JastAdd. However, we did not spend time exploring this possibility further, as the prototype already fulfills our goal of showing that an integration is technically possible. For similar reasons, we did not spend time converting `rust-analyzer`'s test cases to text probes, like we did with the other tools.

E. Discussion

By converting test cases for three different tools, we gained some experience as to where text probes work well, where they technically work, and where they don't work.

AST-navigated tests are where text probes work best. ExtendJ and Bloqqi both had a number of these tests which could be converted and that became much shorter and clearer by doing so. Additionally, all three test suites had many system-level tests for semantic problems and internal analyses, which could benefit from being converted to more precise assertions on individual nodes in the AST.

Expressing system tests as text probes was possible, but if they are not reformulated to be more precise assertions, there is no particular benefit in doing so. Possibly, there could be benefits for developers in a prototyping or educational setting, since it allows both precise AST-navigated tests and system tests to be expressed within the same testing framework. However, text probes are not suitable for differential testing or for test case generation, so for production tools, traditional system tests are also very useful.

One kind of tests that we could not convert to text probes were editor tests that use side effects to modify the AST. This would be an interesting area to explore for future work, as these tests are often AST-navigated.

It is noteworthy how few AST-navigated tests existed in ExtendJ (17 out of 1811). We believe the verbosity of those tests (like shown in Figure 1a) is the main reason why these kinds of tests were avoided. It remains future work to explore the long-term impact of having access to text probes in a project. We predict that such a study would show more AST-navigated tests being created.

Our replication package [13] includes all our converted tests for ExtendJ, IntraJ, and Bloqqi, as well as our `rust-analyzer` fork.

V. Performance Measurements

To investigate the overhead of text probes, and determine if they are practical to use in larger test suites, we compared the performance of the original ExtendJ test suite with our corresponding suite converted to text probes. The original test suite is executed with `ant` and `JUnit`, and uses some custom test code for loading each test file into a parameterized `JUnit` test. The converted test suite is executed using our test runner via its command line interface.

The original and converted suites perform the same checks, and we have taken care to ensure that any measured performance difference stems from the overhead of text probes rather than from other factors. For instance, both suites run the same number of JVM instances: one for the compilation of all tests and one for running each runtime test.

The original test suite runs the full compiler also for the compile-time tests, even though only running the frontend would have been sufficient. In order to make the comparison more fair, we therefore modified our implementation of `Program.uiProblems` to also run full code generation.

TABLE I
MEAN EXECUTION TIME IN SECONDS WITH 95% CONFIDENCE INTERVALS FOR THE COMPILE-TIME AND RUNTIME TESTS IN THE EXTENDJ REGRESSION TEST SUITE.

Test Suite	Number of tests	Execution Time (s)		
		Original	Converted	Slowdown
Compile-time	1436	5.4 ± 0.03	6.0 ± 0.01	11%
Runtime	359	18.5 ± 0.04	19.6 ± 0.02	6%

TABLE II
COURSE EVALUATION RESULTS FROM THE COMPILER COURSE, 23 RESPONDENTS. *Responses* SHOW THE NUMBER OF STUDENTS THAT TICKED *Fully Disagree*, *Disagree*, *Neutral*, *Agree*, OR *Fully Agree*. *Avg* SHOWS THE AVERAGE CALCULATED BY ASSIGNING A VALUE FROM 1 (FD) TO 5 (FA) TO THE DIFFERENT RESPONSES.

Statement	Responses					
	Avg	#FD	#D	#N	#A	#FA
During lab 4 (name analysis, type analysis, etc.) I worked on my code until it worked, and wrote the tests afterwards.	2.3	8	7	4	1	3
During lab 4 I interleaved coding with writing tests.	4.2	1	1	2	8	11
During lab 4 I primarily wrote CodeProber tests (like "[[A.b=c]]") rather than normal JUnit tests.	3.2	5	2	4	3	7
During lab 4 I would have written fewer tests if I didn't have access to CodeProber tests. (Feel free to expand on these questions in the free text answers.)	3.3	5	0	5	4	6

We ran the measurements on a benchmarking machine with Java 11 on Ubuntu 24.04 LTS, with an Intel i7-11700K CPU and 128GB DDR4 RAM. Each test suite configuration was run 30 times, and we noted the average runtime with a 95% confidence interval. The results are shown in Table I, with separate numbers for the compile-time and runtime tests. The results show that the text probe implementation is 6–11% slower in both test suites. We believe the slowdown is mainly due to text probes being evaluated using reflection, which carries a small performance penalty compared to normal method invocation. Still, we consider an 11% slowdown small enough that text probes remain viable in larger test suites.

There is an opportunity for improving the performance of the text probe suite by rewriting some tests to perform smaller, more precise checks, rather than a full frontend evaluation. Such performance improvements would rely on the compiler being implemented using demand-driven techniques, which is the case for ExtendJ.

VI. COURSE USAGE

We let students use text probes in two university courses: one on compilers, and one on program analysis. The courses were already using the CODEPROBER IDE during labs, and the course leaders now let the students use our variant of the IDE with the text probe integration. We assessed students' perceptions of working with text probes in the labs through brief surveys.

A. Compiler Course

In the compiler course the students build a compiler for a simple C-like language over 6 lab sessions. They were required to write tests for each lab, and all labs include examples of how to test using system-level `JUnit` tests. CODEPROBER is introduced in lab 3, and in lab 4 students spend a majority of their time on analysis-related tasks: name analysis, type analysis, and detecting static-semantic errors in code. These tasks fit well with text probes, and are therefore the focus of our evaluation for this course. The instructions for lab 4 were extended with examples of how to use text probes as an alternative way of testing, and the students were encouraged to experiment with this technique. However, in completing their lab, they were free to use any combination of `JUnit` and text probe tests.

The course survey sent out to students after the course included four questions relating to CODEPROBER and testing. The questions are phrased as statements that students can agree, disagree, or be neutral towards. Table II shows the questions and the results.

The first two questions were created to find out if students interleaved testing and implementation work. The lab instructions encourage the students to do so, and the results confirm that this indeed seems to be how most students work.

The last two questions were created to find out to what degree students used text probes (called "CODEPROBER tests" in the survey), versus traditional `JUnit` tests. From question 3, we see that around half of the students (10 out of 21) mostly used text probes, even though they were more trained in `JUnit`, and using text probes was voluntary. Only a third of the students (7 out of 21) continued to mostly use `JUnit`. From question 4, we see that 10 out of 20 students think that access to text probes made them write more tests.

B. Program Analysis Course

In the program analysis course, the students are given an existing compiler at each lab, and are tasked with implementing an analysis algorithm for it. The handout code also contains some example input files. In previous years, these files contained comments describing the expected values to be computed at various points. For example, in a lab on interval analysis, one line looked like this:

```
return x; // expected: [-4, 10]
```

In the most recent instance of the course, these comments have been replaced with text probe assertions. The line above now looks like this:

```
return x; // [[Access.interval.must.equal(-4, 10)!]]
```

The property `must` returns a fluent-style assertion object, which in turn has methods such as `equal`, `contain`, and `beWithin` for different comparison operations.

Near the end of the course we conducted a small survey to ask students about their experience working with text probes. The survey contained seven statements to agree or disagree with, and three free-text questions. Table III shows

TABLE III
SURVEY RESULTS FROM THE PROGRAM ANALYSIS COURSE, 11 RESPONDENTS. *Responses* SHOW THE NUMBER OF STUDENTS THAT TICKED *Strongly Disagree*, *Disagree*, *Neutral*, *Agree*, OR *Strongly Agree*. *Avg* SHOWS THE AVERAGE CALCULATED BY ASSIGNING A VALUE FROM 1 (SD) TO 5 (SA) TO THE DIFFERENT RESPONSES.

Statement	**Responses**					
	Avg	#SD	#D	#N	#A	#SA
The text probes in the handout code helped me understand what to do in the lab	4.5	0	0	1	4	6
I found it easy to understand what was being tested by a text probe	4.5	0	0	0	5	6
I found the live test results in CodeProber useful (i.e. when I recompiled my compiler, I could directly see new test results)	4.9	0	0	0	1	10
The text probe syntax was unintuitive	2.4	2	4	4	1	0
I mostly checked test results in the command-line (./gradlew cprTest) rather than looking in the CodeProber UI	1.6	7	2	1	1	0
I found it easy to write tests with text probes	4.0	0	0	3	5	3
I prefer working with ordinary JUnit tests rather than text probes	1.5	7	2	2	0	0

the responses to the statements. The free-text questions were "What do you like most about text probes?", "What could be improved about text probes?" and "Any other feedback?".

The survey results paint an overall positive picture of working with text probes in the course. Questions 3 and 5 tell us that students mainly used the live feedback in the IDE to observe test results, rather than manually running tests. Questions 1, 2, and 6 indicate that students were able to read and write the text probes without any significant problems. However, the response to question 4 indicates that the syntax could be improved. In the free text answers, two students mentioned the "`!`" assertion shorthand as being confusing. Finally, question 7 shows an overall strong preference for working with text probes over "ordinary" `JUnit` tests.

The course leader for the program analysis course (not an author of this paper) supplied us with some observations from running this year's course instance. One observation is that students on average wrote fewer test files, but each test file contained more tests. A possible explanation for this is that previously, only high-level system tests were used, so to test a specific feature it was necessary to construct a tiny input program. Now with text probes, many specific/small features can be tested at the same time within a single input program. The course leader also told us text probes make it "easier for me to interpret the intent of tests written by students when looking at them (e.g., during labs)". They also noted some areas for improvement, such as the text probe assertions not being convenient for unordered data structures like sets. Overall they describe text probes as a "highly appealing feature" that will be used in the future.

VII. RELATED WORK

A. Compiler and static analyzer testing

Research on unit testing of compilers and other language implementation tools seems very scarce. For example, consider the 2020 compiler testing survey by Chen et al. [3] and the 2025 Dagstuhl seminar on testing program analyzers and verifiers [16]. Much of the research involves generating input programs, constructing automated oracles, and performing differential testing (comparing multiple implementations). Manual unit testing is only mentioned in passing, as an assumed baseline, and is not itself studied. These techniques are useful and complementary to our work, but provide less support during development. In contrast, our work explicitly supports an agile test-driven approach with manually created tests.

To get some insight into how testing is done in practice, we looked into the test suites of several large open-source compiler and static analyzer projects: Rust [17], Roslyn/C# [18], LLVM [19], Clang, Clang-Tidy, GCC [20], SootUp [21], [22], SonarQube's Java plugin [23], Spoon [24], [25], and ExtendJ [8]. We found a wide range of testing approaches, but also many commonalities. System tests were by far the most common type of tests, although we also saw some examples of AST-navigated tests. Only C#, SootUp and Spoon had a substantial number of AST-navigated tests. This is likely motivated by the fact that these projects are meant to be used as libraries in other projects. Their internal representation is meant to be exposed to others, and therefore needs to be well tested. ExtendJ also has AST-navigated tests, but very few in comparison.

We found several common patterns across the test suites that can be related to text probes.

1) Structured comments for running and checking instructions: In several of the projects, structured comments in the input files are used to simplify the system tests. One common pattern is to use structured comments in the beginning of an input file to specify how to run the tool, and how to check the output. The comments can express what options to use, if a compilation is expected to pass or fail, if the resulting code should be run or not, etc. GCC, LLVM, Clang, Clang-Tidy, and ExtendJ all make use of this approach, using external tools in the testing framework to parse the comments.

2) Structured comments for expected values in output: Another common pattern is to use structured comments to assert that certain substrings are expected in the output of system tests. GCC, LLVM, Clang, Clang-Tidy, Rust, and SonarQube's Java plugin all have a significant number of such tests. For example, the following end-of-line comment from a GCC test states that a particular compile-time error should be present in the output.

```
for (auto g : arr) {
  int g = 42; // { dg-error "'g' redeclared" }
}
```

These approaches have similarities to text probes in the use of structured comments that state expected values. However, they differ in that they only support predefined items to check

in the system output, for example, error and warning messages. The approaches are sometimes used for checking values in the internal representation by running the tool with selectively enabled options, so that the tool prints out debug information rather than the usual output. LLVM uses such system tests. However, they require the tool to support a new debug option for each kind of information that is to be checked, and that it generates suitable output that can be checked in a postprocessing step.

3) Manual AST-navigated tests: SootUp, Spoon and ExtendJ implement AST-navigated tests by manual navigation (see Figure 1 for an example from ExtendJ). They all use ordinary JUnit tests, where the navigation code is implemented programmatically: method calls are used to traverse and search the AST to access properties associated with particular AST nodes.

In some cases helper methods were created to assist in the navigation. For example, we found methods for traversing to nodes with names (e.g. "`getMethod("foo")`"), or to nodes with a certain type (e.g. "`getNode(Bar.class)`"). These helper methods can reduce the number of manual navigation steps that need to be written. However, even with sophisticated navigation helper methods, there is still a disconnect between input code and navigation steps. For example, when navigating to a node with a certain type, a normal IDE is unable to suggest what types are available to navigate to, nor does it give live feedback about whether a chosen type could be found. Conversely, our text probe implementation provides rich IDE support for such navigation. Text probes can also make use of navigation helper methods. `matchAll`, as shown in Figure 5, is an example of such a method.

4) Source code markers: The C# test framework supports marking program elements, e.g., by wrapping them using {| and |} markers. These markers are stripped out before parsing, but their locations are stored, and can later be used to locate program elements in the parse tree. This removes the need for explicit navigation code when writing AST-navigated tests.

These markers are also used for testing IDE functionality, where a location in the input source code is an extra input to a test. For example, to test what text is shown when hovering over a particular method, a marker "`{|caret:...|}`" is placed around the method name `M` as follows:

```
private string {|caret:M|}(int i) { ... }
```

A test can use the tag `caret` to retrieve the corresponding line and column number, and then use that to test that the computed hover string has the expected value.

B. Domain-specific languages for testing language tooling

Iung et al. [26] conducted a systematic mapping study of development tools for domain-specific languages. They identified 59 tools, but only 9 that include language-testing features ("DSL Testing").

Kats et al. [27] introduced a generic testing language that syntactically encloses the language under test, realized in the language workbench Spoofax. They focus on testing a general interface of language features, including parsing, compile-time warnings, name resolution, type checking, transformations, and code generation. In contrast, text probes focus on testing internal properties of a given language implementation, which may not be commonly supported across languages. Kats et al. also support some AST-navigated tests, such as name-resolution tests. For these tests, special markers are used in the input code to navigate to specific AST nodes, for instance, to check that names are correctly resolved. They also support both negative and positive syntax tests. Our implementation is unaware of the underlying language's syntax. Currently, we can only write positive syntax tests with text probes, for example, to check that the AST is correctly built according to the precedence and associativity rules.

MPS is another language workbench with built-in testing support [28], [29], covering aspects similar to those of Kats et al. The language syntax is extended in tests to allow markers for AST navigation and inline assertions. While text probes operate on text, MPS generates projectional editors in which programs are created through AST modifications.

C. Live programming

There are multiple programming systems supporting liveness [30]–[32]. These systems provide immediate feedback on the program's dynamic behavior, and the feedback is automatically updated whenever the programmer changes the program. In contrast, text probes focus on tests that assert internal properties in the underlying language tool for a given input program. Text probes are live, and our IDE integration updates them in response to changes in the input code, the text probe, and the underlying language tool.

Our work has been inspired by Niephaus et al. [32]. Their system supports probes and assertions as structured comments with results rendered in the editor, similar to text probes. However, the values they display are those from the dynamic behavior of the program, whereas we display values from the dynamic behavior of the tool analyzing the program.

VIII. Conclusion

In this paper we presented *text probes*, a new technique for unit testing internal program analysis results. We implemented rich editor support for text probes and integrated this into an IDE, and evaluated its use with case studies, performance benchmarks, and surveys on students that used text probes in course labs. We found that text probes work well with a wide range of tests, in particular for small intermediate properties. We believe that the introduction of text probes will encourage more fine-grained unit testing for language tooling like compilers and program analyzers, and that this will speed up development and reduce maintenance costs for such tools. Our results from the case studies and performance evaluation are available in the replication package [13].

There are opportunities for future work in combining text probes with AI. Text probes express precise low-level requirements, which should prove useful in a test-driven development loop where a human writes tests and an AI generates code.

References

[1] A. Donaldson, "Testing Programming Language Implementations," Video from talk at Programming Language Implementation Summer School (PLISS), May 2017. Accessed: 2026-02-12. [Online]. Available: https://www.youtube.com/watch?v=ZJUk8_k1HbY&t=464s

[2] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 180–190. [Online]. Available: https://doi.org/10.1145/2884781.2884878

[3] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020. [Online]. Available: https://doi.org/10.1145/3363562

[4] Y. Tang, Z. Ren, W. Kong, and H. Jiang, "Compiler testing: a systematic literature analysis," *Frontiers Comput. Sci.*, vol. 14, no. 1, pp. 1–20, 2020. [Online]. Available: https://doi.org/10.1007/s11704-019-8231-0

[5] A. R. Alaküla, G. Hedin, N. Fors, and A. Pop, "Property probes: Live exploration of program analysis results," *Journal of Systems and Software*, vol. 211, p. 111980, 2024. [Online]. Available: https://doi.org/10.1016/j.jss.2024.111980

[6] T. Ekman and G. Hedin, "The jastadd system—modular extensible compiler construction," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 14–26, 2007. [Online]. Available: https://doi.org/10.1016/J.SCICO.2007.02.003

[7] ——, "The jastadd extensible java compiler," *ACM SIGPLAN Notices*, vol. 42, no. 10, pp. 1–18, 2007. [Online]. Available: https://doi.org/10.1145/1297105.1297029

[8] ExtendJ Regression Tests, Git Repository, revision: 6f265e8. [Online]. Available: https://bitbucket.org/extendj/regression-tests

[9] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[10] AssertJ - Fluent Assertions for Java, Software. Accessed: 2026-03-01. [Online]. Available: https://github.com/assertj/assertj

[11] Chai Assertion Library, Software. Accessed: 2026-03-01. [Online]. Available: https://github.com/chaijs/chai

[12] M. Fowler, "Fluent interface," 2005, Blog entry. Accessed: 2026-03-01. [Online]. Available: https://martinfowler.com/bliki/FluentInterface.html

[13] A. R. Alaküla, G. Hedin, and N. Fors, "Replication package for "text probes: Unit testing of source code analysis"," Mar. 2026. [Online]. Available: https://doi.org/10.5281/zenodo.18889871

[14] I. Riouak, C. Reichenbach, G. Hedin, and N. Fors, "A precise framework for source-level control-flow analysis," in *21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Luxembourg, September 27-28, 2021*. IEEE, 2021, pp. 1–11. [Online]. Available: https://doi.org/10.1109/SCAM52516.2021.00009

[15] N. Fors and G. Hedin, "Bloqqi: modular feature-based block diagram programming," in *Proceedings of Onward 2016, the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2016, pp. 57–73. [Online]. Available: https://doi.org/10.1145/2986012.2986026

[16] M. Christakis, A. F. Donaldson, J. Regehr, and T. Sotiropoulos, "Testing program analyzers and verifiers (dagstuhl seminar 25242)," *Dagstuhl Reports*, vol. 15, no. 6, pp. 69–83, 2026. [Online]. Available: https://doi.org/10.4230/DagRep.15.6.69

[17] Rust, Git Repository, revision: b6d74b5e. [Online]. Available: https://github.com/rust-lang/rust

[18] Roslyn .NET compiler, Git Repository, revision: a79a3770. [Online]. Available: https://github.com/dotnet/roslyn

[19] LLVM, Git Repository, revision: f7cfa3a7. [Online]. Available: https://github.com/llvm/llvm-project

[20] GCC, Git Repository, revision: 594dc80c. [Online]. Available: https://github.com/gcc-mirror/gcc

[21] SootUP, Git Repository, revision: c091bbce. [Online]. Available: https://github.com/soot-oss/SootUp

[22] K. Karakaya, S. Schott, J. Klauke, E. Bodden, M. Schmidt, L. Luo, and D. He, "SootUp: A redesign of the soot static analysis framework," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2024, pp. 229–247. [Online]. Available: https://doi.org/10.1007/978-3-031-57246-3_13

[23] SonarQube's Java plugin, Git Repository, revision: 8f3b2014. [Online]. Available: https://github.com/SonarSource/sonar-java

[24] Spoon, Git Repository, revision: c42faab4. [Online]. Available: https://github.com/INRIA/spoon

[25] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. [Online]. Available: https://doi.org/10.1002/spe.2346

[26] A. Iung, J. Carbonell, L. Marchezan, E. Rodrigues, M. Bernardino, F. P. Basso, and B. Medeiros, "Systematic mapping study on domain-specific language development tools," *Empirical Software Engineering*, vol. 25, no. 5, pp. 4205–4249, 2020. [Online]. Available: https://doi.org/10.1007/s10664-020-09872-1

[27] L. C. L. Kats, R. Vermaas, and E. Visser, "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, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, C. V. Lopes and K. Fisher, Eds. ACM, 2011, pp. 139–154. [Online]. Available: https://doi.org/10.1145/2048066.2048080

[28] D. Ratiu and M. Voelter, "Automated testing of dsl implementations: experiences from building mbeddr," in *Proceedings of the 11th International Workshop on Automation of Software Test*, 2016, pp. 15–21. [Online]. Available: https://doi.org/10.1145/2896921.2896922

[29] M. Voelter, B. Kolb, T. Szabó, D. Ratiu, and A. van Deursen, "Lessons learned from developing mbeddr: a case study in language engineering with mps," *Software & Systems Modeling*, vol. 18, no. 1, pp. 585–630, 2019. [Online]. Available: https://doi.org/10.1007/s10270-016-0575-4

[30] S. McDirmid, "Usable live programming," in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, 2013, pp. 53–62. [Online]. Available: https://doi.org/10.1145/2509578.2509585

[31] S. Lerner, "Projection boxes: On-the-fly reconfigurable visualization for live programming," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3313831.3376494

[32] F. Niephaus, P. Rein, J. Edding, J. Hering, B. König, K. Opahle, N. Scordialo, and R. Hirschfeld, "Example-based live programming for everyone: building language-agnostic tools for live programming with lsp and graalvm," in *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–17. [Online]. Available: https://doi.org/10.1145/3426428.3426919