Integrated Language Definition Testing

Enabling Test-Driven Language Development

Lennart C. L. Kats
Delft University of Technology
l.c.l.kats@tudelft.nl

Rob Vermaas
LogicBlox
rob.vermaas@logicblox.com

Eelco Visser
Delft University of Technology
visser@acm.org

Abstract

The reliability of compilers, interpreters, and development environments for programming languages is essential for effective software development and maintenance. They are often tested only as an afterthought. Languages with a smaller scope, such as domain-specific languages, often remain untested. General-purpose testing techniques and test case generation methods fall short in providing a low-threshold solution for test-driven language development. In this paper we introduce the notion of a *language-parametric testing language (LPTL)* that provides a reusable, generic basis for declaratively specifying language definition tests. We integrate the syntax, semantics, and editor services of a language under test into the LPTL for writing test inputs. This paper describes the design of an LPTL and the tool support provided for it, shows use cases using examples, and describes our implementation in the form of the Spoofax testing language.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Testing Tools; D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.6 [*Software Engineering*]: Interactive Environments

General Terms Languages, Reliability

Keywords Testing, Test-Driven Development, Language Engineering, Grammarware, Language Workbench, Domain-Specific Language, Language Embedding, Compilers, Parsers

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

1. Introduction

Software languages provide linguistic abstractions for a domain of computation. Tool support provided by compilers, interpreters, and integrated development environments (IDEs), allows developers to reason at a certain level of abstraction, reducing the accidental complexity involved in software development (e.g., machine-specific calling conventions and explicit memory management). *Domain-specific* languages (DSLs) further increase expressivity by restricting the scope to a particular application domain. They increase developer productivity by providing domain-specific notation, analysis, verification, and optimization.

With their key role in software development, the correct implementation of languages is fundamental to the reliability of software developed with a language. Errors in compilers, interpreters, and IDEs for a language can lead to incorrect execution of correct programs, error messages about correct programs, or a lack of error messages for incorrect programs. Erroneous or incomplete language implementations can also hinder understanding and maintenance of software.

Testing is one of the most important tools for software quality control and inspires confidence in software [1]. Tests can be used as a basis for an agile, iterative development process by applying test-driven development (TDD) [1], they unambiguously communicate requirements, and they avoid regressions that may occur when new features are introduced or as an application is refactored [2, 31].

Scripts for automated testing and general-purpose testing tools such as the xUnit family of frameworks [19] have been successfully applied to implementations of general-purpose languages [16, 38] and DSLs [18, 33]. With the successes and challenges of creating such test suites by hand, there has been considerable research into *automatic generation* of compiler test suites [3, 27]. These techniques provide an effective solution for thorough black-box testing of complete compilers, by using annotated grammars to generate input programs.

Despite extensive practical and research experience in testing and test generation for languages, rather less attention has been paid to supporting language engineers in writing tests, and to applying TDD with tools specific to the do-

main of language engineering. General-purpose testing techniques, as supported with xUnit and testing scripts, require significant investment in infrastructure to cover test cases related to syntax, static semantics, and editor services, specific for the tested language. They also use isolated test programs to test particular language aspects, requiring a certain amount of boilerplate code with each test program (e.g., import headers), and require users to manually code how to execute the test (parse/compile/run/etc.) and how to evaluate the result and compare it to an expectation. Tool support for writing test cases and specifying test conditions is lacking, particularly for negative test cases where errors are expected. Test generation techniques are an effective complementary technique for stress testing complete compiler implementations, but are less effective during the development of a new language definition.

In this paper, we present a novel approach to language definition testing by introducing the notion of a *language-parametric testing language (LPTL)*. This language provides a reusable, generic basis for declaratively specifying language definition tests. It can be *instantiated* for a specific *language under test* through language embedding: we integrate the syntax, semantics, and editor services of a language under test into the LPTL for writing test inputs.

For the generic basis of the LPTL, we provide general constructs to configure test modules and to declaratively specify test conditions. Based on the observable behavior of languages implementations, we selected an open-ended set of test condition specification constructs. These form the heart of the testing language, and support writing tests for language syntax, static semantics, editor services, generated code, and dynamic semantics.

To support language engineers in writing and understanding tests, we show how full language-specific IDE support can be provided for writing test cases. The instantiated LPTL provides editor services such as syntax highlighting, syntax and semantic error marking, and content completion, based on the definition of the language under test.

Using an LPTL significantly reduces the threshold for language testing, which is important because such a threshold is often a reason for developers to forgo testing [15]. First, by providing a reusable infrastructure for language test specifications that facilitates test execution, analysis, maintenance, and understanding. Second, by providing full language-specific IDE support for writing tests.

The contributions of this paper are as follows.

- The design of a generic, declarative test specification language for language definition testing.
- A fully language-agnostic approach to language embedding that incorporates syntactic, semantic, and editor service aspects of a language under test.
- The implementation of such a testing language as the *Spoofax testing language*[1] and a description of its implementation architecture.

Outline We begin this paper with background on language definitions. Next, we discuss the design of a language-parametric testing language from three angles: first from a purely linguistic perspective in Section 3, then from a tool support perspective in Section 4, and finally by illustrating use cases with examples in Section 5. Our implementation architecture is described in Section 6. We conclude with related work on language testing approaches and directions for future work.

2. Background: Language Definitions

The development of a compiler for a DSL for a domain comprises many tasks, ranging from construction of a parser to a semantic analyzer and code generator. In addition to a compiler, the construction of an integrated development environment (IDE) is essential, as developers increasingly rely on IDEs to be productive. Traditionally, a lot of effort was required for each of these tasks. Parsers, data structures for abstract syntax trees, traversals, transformations, and so on would be coded by hand for each language. The implementation of editor services expected from modern IDEs, such as syntax highlighting, an outline view, reference resolving for code navigation, content completion, and refactoring, added to this already heavy burden. This meant that a significant investment in time and effort was required for the development of a new language.

Language engineering tools Nowadays there are many tools that support the various aspects of language engineering, allowing language engineers to write high-level *language definitions* rather than handwrite every compiler, interpreter and IDE component. Particularly successful are parser generators, which can generate efficient parsers from declarative syntax definitions. For semantic aspects of languages, there are numerous meta-programming languages and frameworks. For the development of IDE support there are also various tools and frameworks that significantly decrease the implementation effort.

Language workbenches are a new breed of language development tools [13] that integrate tools for most aspects of language engineering into a single environment. Language workbenches make the development of new languages and their IDEs much more efficient, by *a)* providing full IDE support for language development tasks and *b)* integrating the development of the language compiler/interpreter and its IDE. Examples of language workbenches include MPS [37], MontiCore [28], Xtext [11], and our own Spoofax [25].

Language workbenches allow for an agile development model, as they allow developers to use an IDE and to "play" with the language while it is still under development. Figure 1 shows a screenshot illustrating how they can combine the development of a language with the use of generated ed-

[1] Distributed as part of the Spoofax language workbench [25], available with nightly builds of version 0.6.1 via `http://www.spoofax.org/`.

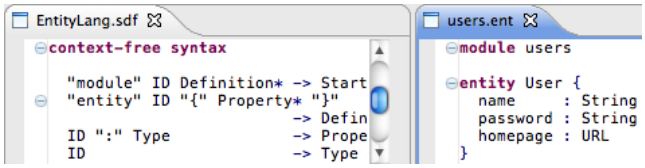


Figure 1. Language workbenches can combine language definition (left) and language use (right).

itors for that language. Once a syntax definition has been developed (at the left), they can generate an editor with basic editor services such as syntax highlighting and syntax error marking. From there, language engineers can incrementally and iteratively implement new language components and editor services.

Examples and tests Many new languages start with sketches. Sketches of example programs or code snippets that solve a particular problem in the domain of the language. With language workbenches, it is common practice to maintain a "scratch pad" with some example program that focuses on new features that are under development. Language engineers can interact with it in various ways. For example, they can introduce type errors ("does the type checker catch this?"), control-click on an identifier ("does this hyperlink point to the right place?") or generate and run code for the example.

Example programs quickly grow unwieldy, and, as commonly seen with interactive systems [31], they are often thrown away once they show satisfactory results. This is a major problem as these test cases are a valuable investment that simply disappears after testing is completed. The problem results from the high threshold of setting up tests for languages and their IDE, running tests in an automated fashion, ensuring that the observed behavior complies with the expectations, and so on. The effort does not compare to the ease of testing it interactively. Without better tool support, proper testing remains an afterthought, even in an integrated language engineering environment such as a language workbench.

3. Test Specification Language Design

In this section we describe the design of a language-parametric testing language and show how it can be used to test different aspects of language definitions. The design of the language is highly intertwined with the tool support that is provided for it and how users can interact with it. We discuss those aspects of the language in the next section.

The central goal set out for design of an LPTL is to provide a low-threshold test specification language that forms the basis for a reusable infrastructure for testing different languages. The design principles of this language are as follows:

P1 Provide a language-agnostic framework. The language should provide a generic, language-agnostic basis that caters for a wide spectrum of different types of tests.

P2 Maintain implementation independence. The language should emphasize black-box testing [31], allowing tests to be written early in the design process, and abstracting over implementation specifics.

P3 Support language-specific instantiation. It should be possible to instantiate the language for a specific *language under test*, thereby integrating the two languages and the tool support provided for the two.

P4 Facilitate series of tests with test fixtures. The language should support test fixtures to specify series of tests with common boilerplate code such as import headers.

In the remainder of this section we show how these principles can be realized, and show the design of the Spoofax testing language, our implementation of an LPTL.

A language-agnostic framework (P1) Key to providing a reusable, language agnostic framework is providing a generic language that can quote test fragments and can specify conditions to validate for those tests. We realize this using the following syntax to specify tests:

```
test description [[
  fragment
]] condition*
```

where *description* is a string that describes the current test case in human readable form and *fragment* is an embedded program or program fragment in the language under test. The *condition** elements specify the expectations of the test case, and control what test is performed for the input fragment.

Figure 2 shows an example test case where we test the mobl [20] language, a domain-specific language for mobile applications. In this example we declare a local variable `s` of type `String` and assign an integer literal value to it. This is a *negative test case*: a value of type `String` would be expected here. The conditions clause of this test case indicates that exactly one error was *expected* here, which means that the test case *passes*.

To ensure the test specification syntax is language agnostic, it cannot have any specific elements for a particular language. The set of different possible tests that can be specified must be based on a generic interface for observable behavior commonly supported by languages. Furthermore, the quotation mechanism cannot be limited to only a single, fixed sequence of characters such as the double square brackets above, since those may also be in use by the language under test. In the Spoofax testing language we address this issue by supporting additional quotation markers such as `[[[`, `[[[[`, and variations with series of `"` quotes.

```
test Cannot assign an integer to a string [[
  module Example

  function f() {
    var s : String = 1;
  }
]] 1 error
```

Figure 2. A basic mobl test case.

```
test basic completion [[
  module Example
  function get(argument : String) : String {
    return [[arg]];
  }
]] complete to "argument"
```

Figure 3. A test to verify that the selected identifier `arg` completes to `argument`.

Implementation independence via black-box testing (P2) Black-box tests [31] test the functionality rather than the internal workings of a software artifact. They are independent of the internal workings of a tested artifact or unit and focus only on its observable behavior (output) given some input. The example of Figure 2 illustrates this principle, as it reveals nor assumes anything about the implementation of the language under test.

As inputs of a black-box language test we use *1)* the quoted fragment of code, *2)* the conditions clause, and *3)* selections of code within the quoted fragment. The first two were illustrated in the example of Figure 2. The test input fragment indicates the input to feed to the language implementation, and the conditions clause indicates what action to trigger and what check to perform. In our example, the `1 error` clause indicates that the implementation should perform semantic checking and that only one error is expected. Other clauses can specify other actions and checks such as syntactic checks, name resolution, refactoring, and execution. In some cases, they specify user input such as a name for a rename refactoring or command-line arguments for an execution test. We give an overview of these facilities at the end of this section.

For many forms of tests it is useful to specify some form of selection in the input fragment. For instance, consider Figure 3, which shows a content completion test. The double brackets inside the quotation indicate a selected part of the program where content completion would be applied. Selections can indicate identifiers or larger program fragments for tests of features such as reference resolving, content completion, and refactorings. Some tests use multiple selections, in particular for reference resolving, where both a name reference and its declaration can be selected.

Language-specific instantiation (P3) Instantiation of the testing language for a specific language under test requires that the test suite specifies which language to use for its test cases. Optionally tests suites can also specify which

```
language mobl

setup [[
  module Example

  function f() {
    var s : String = "";
    [[...]]
  }
]]

test Cannot assign an integer to a string [[
  s = 1;
]] 1 error

test Can assign a string to a string [[
  s = "a string";
]]
```

Figure 4. A testing module with a shared setup block.

syntactic start symbol they use, e.g. a module, statement, or expression. Based on this information, it becomes possible to evaluate the test cases by invoking or interpreting the language implementation. To fully realize language-specific instantiation, the IDE that supports the testing language can also be adapted to incorporate the syntax and semantics of the tested language, as we illustrate in the next section.

We organize suites into one or more modules (i.e., files), where each module has a series of test cases and its own configuration. For each module we use headers that indicate their name, what language to use, and what start symbol to use:

```
module test-assignments
language Mobl
start symbol Expression
```

Of these headers, only the **`language`** header is compulsory. Once the language under test is specified, the LPTL and the language under test are composed together, and quoted test fragments are no longer treated as mere strings but as structured part of test specifications.

Test fixtures (P4) A common technique in testing frameworks such as the xUnit [19] family of frameworks, is to use `setup()` and `tearDown()` methods[2] to create test fixtures. These methods respectively initialize and de-initialize common state for a series of tests. For language testing, the same principle can be applied. We use *setup blocks* to specify common elements for tests.

Consider again our basic example of Figure 2. Suppose we want to write multiple, similar tests for assignments in the same context. This would require writing the same boilerplate in for each test case: a module, function, and variable declaration. These commonalities can be factored out using setup blocks, as shown in Figure 4. Instead of writing the same boilerplate for every test case, it suffices to write it only once using the shared setup block. The contents

[2]Note that more recent implementations such as JUnit 4 often use annotations for this functionality.

Configuration	
module *name*	Module name *name*.
language *language*	Use *language* as the language under test.
start symbol *symbol*	Use syntactic start symbol *symbol*.
Test cases	
test *description f c**	A test case where *f* must satisfy conditions *c**.
test *description expression*	A white-box test case where freeform test condition *expression* must be satisfied.
setup *description f*	A setup block for test cases in this module. Can use `[[...]]`[3] in *f* for placeholders.
Tested fragments (*f*)	
`[[` (*code* | `[[`*code*`]]`)* `]]`[3]	Partial code fragments in the language under test.
`[[[` (*code* | `[[[`*code*`]]]`)* `]]]`	Alternative fragment quoting style with three brackets.
`[[[[` (*code* | `[[[[`*code*`]]]]`)* `]]]]`	Alternative fragment quoting style with four brackets.
Test conditions (*c*)	
succeeds	Fragment succeeds parsing and has no semantic errors/warnings (default cond.).
fails	Fragment has semantic errors or warnings.
parse *pattern*	Fragment parses according to *pattern*.
n **error** | *n* **errors**	Fragment has exactly *n* semantic error(s).
n **warning** | *n* **warnings**	Fragment has exactly *n* semantic warning(s).
/regex/	Fragment has an error or warning matching regular expression *regex*.
resolve (#*n*)?	Successfully resolves the identifier at the (n^{th}) selection.
resolve #*n* **to** #*m*	Resolves the identifier in the n^{th} selection to a declaration at the m^{th} selection.
complete (#*n*)? **to** *x*	Content completion proposals for the (n^{th}) selection include a name *x*.
refactor (#*n*)? *r* ((*arg*))? *p*	Applies refactoring *r* with argument string *arg* according to pattern *p*.
build *builder* ((*arg*))? *p*	Builds the fragment using builder *builder* with argument *arg* according to *p*.
run *runner* ((*arg*))? *p*	Executes the fragment using runner *runner* with argument *arg* according to *p*.
e	Freeform expression *e*, specified in the language definition language, is satisfied.
Patterns in test conditions (*p*)	
	Empty pattern: same as **succeeds**.
succeeds	Operation (i.e., refactoring, builder, execution) is expected to be successful.
fails	Operation is expected to fail.
to *term*	The result should match a term pattern such as `PropAccess("a","b")`.
to *fragment*	The result should match a code fragment *fragment*.
to file *file*	The result should match the contents of file *file*.

Figure 5. Summary of the test specification syntax.

of the setup block serves as a template for the test cases, where the `[[...]]` placeholder is filled in with the contents of each test block.[4] The placeholder is optional: if none is specified, we assume it occurs at the end of the setup block.

Setup blocks are essentially a purely syntactic, language-agnostic feature, but they are highly flexible. They can be used to factor out boilerplate code from individual tests, such as module and import declarations. They can also be used to declare types, functions and values used in test cases. Much like with the `setup()` and `tearDown()` methods of xUnit, they can also be used to perform tasks such as database initialization for test cases that execute tested programs.

[3] Alternatively, `[[[...]]]` or `[[[[...]]]]` can be used.

[4] Note that we overload the quotation brackets to specify anti-quotations for selections and for placeholders in setup blocks. This design ensures minimal syntactic interference with the language under test, as language engineers can pick which quotation markers to use (e.g., `[[`, `[[[`, and so on).

Overview We conclude this section with an overview of our test specification syntax, shown in Figure 5. So far we already discussed test configuration, test cases, and tested fragments. The table also shows the possible condition clauses for syntactic, static semantic, and dynamic semantics tests, and the patterns that can be used with some condition clauses. We further illustrate those elements with a series of examples in Section 5.

4. Test Specification Interaction Design

Tool support is an important factor for productivity with programming languages. For the domain of testing in particular, good tool support is important to lower the threshold of testing [15]. In this section we show how tool support in the form of IDE components can be applied to lower the threshold to language definition testing and to increase the productivity of language engineers when editing and running tests.

We propose a combination of four forms of tool support for language definition testing. For editing tests, we propose to aid language engineers by providing editor services for *1)* the generic test specification language, and *2)* editor services of the language under test in test fragments. For running tests, we propose a combination of *3)* live evaluation of test cases as they are edited, and *4)* a batch test runner for testing larger test suites. In the remainder of this section we show how these forms of tool support are realized in the Spoofax testing language and how they can be used. The underlying implementation aspects are discussed in Section 6.

4.1 Editor Services for Test Specification

Integrated development environments are a crucial factor in programmer productivity [32]. Modern IDEs incorporate many different kinds of *editor services*, assisting developers in code understanding and navigation, directing developers to inconsistent or incomplete areas of code, and even helping them with editing code by providing automatic indentation, bracket insertion, and content completion.

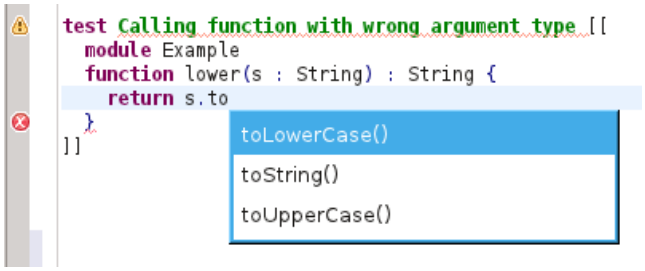
Most editor services provided in modern IDEs are *language specific*, and can be defined as part of the language definition. The challenge in providing effective IDE support for language definition testing is in providing language-specific support for both the testing language and for the embedded language under test.

Editor services for the generic testing language Editor services for the generic testing host language are the meat and potatoes for making language engineers more productive with testing. Our implementation provides the full range of syntactic and semantic editor services for working with the testing language, ranging from syntax highlighting to error marking and content completion for all elements of Figure 5.

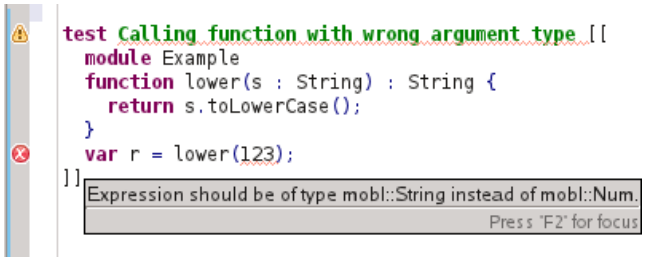
Editor services for language under test Rather than treat tested fragments as an opaque input string, we use editor services of the language under test to support them as first-class parts of a test specification. Our implementation provides services such as syntax highlighting, syntax error marking, semantic error marking, and content completion, as shown in the screenshots of Figure 6 (a) and (b). Note that error markers are only shown for failing test cases, not for negative test cases where errors are expected (Figure 6 (c)).

4.2 Running Language Definition Tests

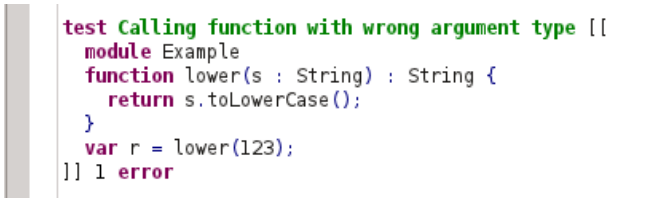
Live evaluation of test cases Live evaluation of test cases as they are edited ensures that language engineers get the same rapid feedback and editing experience as they get with "throwaway" programs used to test language definitions. To achieve this effect, our implementation evaluates tests in the background and shows which tests fail through error and warning markers in the editor. With this feedback, developers can quickly determine the status of tests in a testing module. Since some operations may be long-running, we exclude test cases that depend on building or executing the test from background execution, and instead focus on tests of the syntax, static semantics, and transformations defined for a language.



(a) Content completion for the language under test.



(b) Online evaluation of tests and error markers.



(c) A passing test case specifying negative test condition.

Figure 6. IDE support for test specifications.

Batch execution To support long-running test cases and larger test suites, we also provide a batch test runner as described in [15]. Such a test runner is particularly important as a language project evolves and the number of tests grows substantially and tests are divided across multiple test modules. Figure 7 shows a screenshot of our graphical test runner. The test runner gives a quick overview of passing and failing tests in different modules and allows developers to navigate to tests in a language project. Tests can also be evaluated outside the IDE, for example as part of a continuous integration setup.

4.3 Using Integrated Language Definition Testing

Rather than designing a complete new language "on paper," before its implementation, it is good practice to incrementally introduce new features and abstractions through a process of evolutionary, inductive design [13, 36]. The LPTL approach makes it possible to start a language design with examples that (eventually) form test cases.

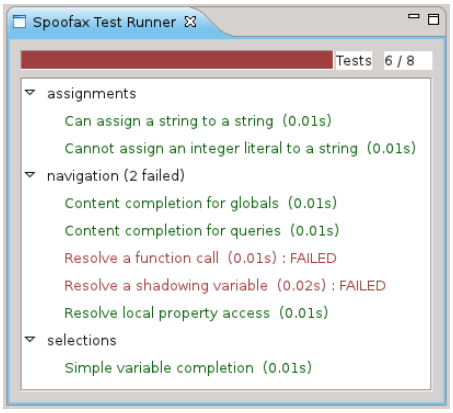


Figure 7. The batch test runner.

Testing from the point of inception of a language requires that the tested language implementation artifact is in such a state that it can produce some form of output for a given input program. Language workbenches such as Spoofax can generate an executable – and thus testable – language plugin from only a (partial) syntax definition [25]. Additional features, in the form of editor services and static and dynamic semantics, can then be iteratively and incrementally added. With an LPTL, each new feature can be tested at at any stage of the development process. This makes it possible to develop languages in a test-driven fashion, following the rhythm described in [1]:

1. Write a test case.
2. Watch it fail.
3. Implement the tested feature.
4. Watch all tests succeed.
5. Refactor when necessary and repeat.

Our approach facilitates this process for language engineering by providing a specialized language testing infrastructure that gives direct feedback at any stage in this development cycle.

5. Language Definition Testing by Example

In this section we show how different language aspects can be tested through examples using the mobl language.

Mobl Mobl is a statically typed language and compiles to a combination of HTML, Javascript, and CSS. Mobl integrates sub-languages for user interface design, data modeling and querying, scripting, and web services into a single language. In this paper we focus on the data modeling language.

An excerpt of the syntax of mobl is shown in Figure 8. In mobl, most files starts with a **`module`** header, followed by a list of entity type definitions, functions, and possibly statements. An example of a mobl module that defines a single entity type is shown in Figure 9. Entities are persistent data types that are stored in a database and can be retrieved using

```
Start ::= "module" QId Def*

Def ::= "entity" ID "{" EBD* "}"
      | Function
      | Stm

EBD ::= ID ":" Type
      | Function

Function ::= "function" ID
             "(" (FArg ("," FArg)*)? ")"
             ":" Type "{" Stm* "}"

Stm ::= "var" ID ":" Type "=" Exp ";"
      | "var" ID "=" Exp ";"
      | Exp "=" Exp ";"
      | Exp ";"
      | "{" Stm* "}"
      | "if" "(" Exp ")" Stm ("else" Stm)?
      | "foreach" "(" ID "in" Exp ")" Stm
      | "foreach" "(" ID ":" Type "in"
        Exp ")" Stm

Exp ::= STRING
      | NUMBER
      | "null"
      | "this"
      | Exp "." ID
      | ID "(" (NameExp ("," NameExp)*)? ")"

NameExp ::= ID "=" Exp | Exp

FArg ::= ID ":" Type

Type ::= ID | "Collection" "<" Type ">"

QId ::= ID | QId "::" ID
```

Figure 8. A subset of mobl's syntax, from [20].

```
module tasks::datamodel

entity Task {
  name : String
  date : DateTime
}
```

Figure 9. A mobl definition of a `Task` data type.

mobl's querying API. We import the `tasks::datamodel` example module in tests throughout this section.

5.1 Syntax

The syntax definition forms the heart of the definition of any textual language. It incorporates the concrete syntax (keywords etc.) and the abstract syntax (data structure for analysis and transformations) of a language, and is generally the first artifact developed with a new language definition. The syntax can be considered separately from the remainder of a language definition, and can be used to generate a parser and editor with basic syntactic services. It can also be tested separately from any semantic aspects of the language.

Syntax tests can be used to test newly added language constructs. They can include various non-trivial tests such as

```
language mobl
start symbol Stm

test Named parameters [[
  var e = Task(name="Buy milk");
]] parse succeeds

test 'true' is a reserved keyword [[
  var true = 1;
]] parse fails

test Test dangling else [[
  if (true)
    if (true) {}
    else {}
]] parse to
    IfNoElse(True, If(True, _, _))

test Nested property access [[
  v = a.b.c;
]] parse to
    Assign("v",
    FieldAccess(FieldAccess("a", "b"), "c"))
```

Figure 10. Syntactic tests.

```
language mobl

setup [[
  module tasks
  import tasks::datamodel

  var todo = Task(name="Create task list");
]]

test Entity types have an all() built-in [[
  var all : Collection<Task> = Task.all();
]] succeeds

test Assigning a property to a Num [[
  var name : Num = todo.name;
]] 1 error /type/

test Local variable shadowing [[
  function f() {
    var a : A;
    {
      var a : A;
    }
  }
]] 1 error /already defined/
```

Figure 11. Tests for static semantic checks.

tests for operator precedence, reserved keywords, language embeddings, or complex lexical syntax such as the quotation construct of Figure 5.

We distinguish two forms of syntax tests. First, there are pure black-box tests, which test if a code fragment can be parsed yes or no. The first two examples of Figure 10 show positive and negative black-box tests. Next, we also support syntactic tests that use tree patterns to match against the abstract syntax produced by the parser for a given fragment. The third and fourth tests in the figure show examples of such tests.[5] These tests are not pure black-box tests as they expose something about the implementation of the parser. They may not rely directly on the internals of the parser, but they still depend on the technology used. Many parser generators rely on restricted grammar classes [26], placing restrictions on the syntax definition, making it difficult to produce certain trees such as the left-recursive trees for field access in Figure 10. In Spoofax, these restrictions are not an issue since we use a generalized-LR parser.

5.2 Static semantic checks

Static semantic checks in languages play an important role in the reliability of programs written in that language. With DSLs such as mobl, these checks are often specific to the domain of the language, and not supported by the target platform. In mobl's case, a dynamic language is targeted that performs no static checks at all.

With tests we can have better confidence in the static checks defined for a language. Examples are shown in Figure 11. We use a setup block[6] in this figure to import the `tasks::datamodel` module, and to initialize a single `Task` for testing. The first test case is a positive test, checking that the built-in `all()` accessor returns a collection of tasks. The other tests are negative tests. For such test cases, it is generally wise to test for a specific error message. We use regular expressions such as `/type/` to catch specific error messages that are expected.

5.3 Navigation

Modern IDEs provide editor services for navigation and code understanding, such as reference resolving and content completion. These services are a manifestation of the *name analysis* that is highly important for the dependability of a language implementation. Tests for reference resolving and content completion test not only the user experience in an editor, but also the underlying name analysis and any other analyses it may depend on.

Figure 12 shows examples of tests for reference resolving and content completion. Note how reference resolving tests can use multiple selected areas. Our first test case tests variable shadowing, while the second one tests reference resolving for function calls. For content completion we test completion for normal local variables, and for built-ins such as the `all()` accessor.

5.4 Transformations and Refactorings

Transformations can be used to create views and for compilation. To test transformations, we use the notion of a

[5] We use prefix constructor terms to match against tree patterns, matching against the name of a tree node and its children. Wildcards are indicated with an underscore.

[6] Recall that the `[[...]]` placeholder notation is optional: by default, we assume that the placeholder is at the end. This default elegantly allows most mobl tests to avoid explicit placeholders.

```
language mobl

setup [[
  module navigation
  import tasks::datamodel

  var example = "Magellan";
]]

test Resolve a shadowing variable [[
  function getExample() : String {
    var [[example]] = "Columbus";
    return [[example]];
  }
]] resolve #2 to #1

test Resolve a function call [[
  function [[loop]](count : Num) {
    [[loop]](count + 1);
  }
]] resolve #2 to #1

test Content completion for globals [[
  var example2 = [[e]];
]] complete to "example"

test Content completion for queries [[
  var example2 = Task.[[a]];
]] complete to "all()"
```

Figure 12. Reference resolving and content completion tests.

```
language mobl

setup [[
  module Example
  import tasks::datamodel
]]

test Desugaring adds a type to foreach [[
  foreach (t in Task.all()) {
    // ...
  }
]] build desugar to [[
  foreach (t : Task in Task.all()) {
    // ...
  }
]]

test Rename refactoring [[
  var x = 1;
  function x(x : Num) : Num {
    return [[x]];
  }
]] refactor rename("y") to [[
  var x = 1;
  function x(y : Num) : Num {
    return y;
  }
]]
```

Figure 13. Tests for transformations and refactorings.

builder. Builders are transformations that can be triggered by the user, displaying a view or generating code [25]. Mobl implements several, for use by both end-programmers and meta-programmers. The first test case in Figure 13 shows an example of a test for the `desugar` builder, one of the builders used by the designers of mobl to inspect the desugared version of a module.

Refactorings are a form of transformations that rely on pre-conditions and post-conditions to perform behavior-preserving transformations [12]. With tests, language engineers can gain more confidence about the transformation performed for a refactoring and its pre- and post-conditions. The second test case in Figure 13 is an example of a refactoring test. The example tests the `rename` refactoring with the input string `"y"` which determines the new name of the selected identifier. The test condition compares the output to the expectation, where behavior is only preserved if just the local variable `x` is replaced and not the other `x`'s.

5.5 Code Generation and Execution

The ultimate goal of most language definitions is to generate or interpret code for execution. Sometimes, languages also generate artifacts for inspection by the user, such as a graphical view of all entities in a mobl application. Testing can be used to confirm that exactly the right output is generated for a particular input, but those tests are often rather fragile: one small change in a compiler can break a test case even if the program still compiles correctly. It is more practical to use an external oracle for those tests, such as a compiler or lint-type checker. Another strategy is to ensure that the program is executable and to simply run it: execution tests can indirectly serve as tests of generated code correctness. For execution tests we use the notion of a *runner*. Similar to a builder, runners are operations that execute code, through interpretation or by running a generated program.

Figure 14 shows tests for code generation and execution. We use a setup block to initialize the database by adding new `Task` instances that can be used in the tests. In the first test case we have a test that only triggers code generation, using mobl's `generate-artifacts` builder. In this case the builder is only required to succeed, we do not explicitly check its output. The other test cases use a runner `run-test`, which invokes the `getResult()` function and returns the result as a string for comparison.

5.6 Testing for end-programmers

So far we have considered testing for meta-programmers. End-programmers that use a language are generally not interested in testing the syntax or static semantics of a language. They are, however, interested in the dynamic semantics; writing unit tests for programs written in the language. An LPTL can be used as a basis for maintaining and running such tests. End-programmers then get the same language-specific feedback and tooling for writing tests as meta-programmers, and can use the same testing language for testing multiple DSLs that may be employed in a project.

The LPTL as we designed it is aimed at meta-programmers, and provides a general platform for testing. For end programmers it can be specialized for one partic-

```
language mobl

setup [[
  module runtimetests
  import tasks::datamodel

  foreach (n in range(0, 10)) {
    add(Task(name="Task "+n));
  }

  function getResult() : Num {
    return [[...]];
  }
]]

test Compile to HTML/JavaScript/CSS [[
  1 + 1
]] build generate-artifacts

test String API [[
  "string".indexOf("r")
]] run run-test("getResult") to "2"

test Counting query [[
  Task.all().count()
]] run run-test("getResult") to "10"
```

Figure 14. Code generation and execution tests.

ular language (eliminating the **language** header) and for the purpose of execution tests (simplifying the **run** clause). Providing specialized instances of the test specification language is considered future work.

5.7 Freeform tests

The test specification language is open-ended: if there are aspects of a language definition that need testing but are not covered by the fixed conditions in the table of Figure 5, *freeform* test expressions can be used. In the Spoofax testing language, we use the Stratego language [5] to specify them, as it is also the language used to define semantic aspects of language definitions in Spoofax [25]. Freeform expressions can directly interact with the language implementation to express white-box test cases. For example, they can test whether an internal function that retrieves all the ancestors in the inheritance chain of a class works, or they can test that `generate-artifacts` correctly writes a `.js` file to disk.

5.8 Self application

An interesting capability of the testing language is that it can be applied to itself. In our implementation of the Spoofax testing language, it can be applied to any language designed in the language workbench, including instantiations of the testing language. Figure 15 shows an example. Note how we use the triple-bracket quotation form (i.e., `[[[ ... ]]]`) in this example, as the testing language itself uses the normal double brackets. For the outer test specification, any selections or setup placeholders should then also be specified using triple brackets. The inner test specification is free to use double brackets.

```
language Spoofax-Testing

test Testing a mobl test specification [[[
  module test-mobl
   language mobl
   test Testing mobl [[
     module erroneous
     // ...
   ]] 1 error
]]] succeeds
```

Figure 15. Testing the test specification language.

6. Implementation

In this section we describe our implementation of an LPTL and the infrastructure that makes its implementation possible. We implemented the Spoofax testing language as a language definition plugin for the Spoofax language workbench [25]. Spoofax itself is, in turn, implemented as a collection of plugins for the extensible Eclipse IDE platform. Most Spoofax language definitions consist of a combination of a declarative SDF [35] syntax definition and Stratego [5] transformation rules for the semantic aspects of languages, but for this language we also wrote parts of the testing infrastructure in Java.

6.1 Infrastructure

The Spoofax language workbench provides an environment for developing and using language definitions [25]. It provides a number of key features that are essential for the implementation of an LPTL.

A central language registry Spoofax is implemented as an extension of the IDE Meta-tooling Platform (IMP) [7] which provides the notions of languages and a language registry. The language registry is a component that maintains a list of all languages that exist in the environment. It also allows for runtime reflection over the services they provide and any meta-data that is available for each language, and can be used to instantiate editor services for them.

Dynamic loading of editor services Spoofax supports dynamic, headless loading of separate language and editor services of the language under test. This is required for instantiation of these services in the same program instance (Eclipse environment) but without opening an actual editor for them.

Functional interfaces for editor services Instantiated editor services have a functional interface. This decouples them from APIs that control an editor, and allows the LPTL to inspect editor service results and filter the list of syntactic and semantic error markers shown for negative test cases.

Support for a customized parsing stage Most Spoofax plugins use a generated parser from an SDF definition, but it is also possible to customize the parser used. This allows the LPTL to dynamically embed a language under test.[7]

[7] Note that even though Spoofax supports generalized parsing and syntax embedding techniques, a different approach is required in this case as

These features are not trivially supported in language workbenches, but there are other workbenches that support a subset. For instance, where many language workbench implementations generate relatively opaque, autonomous Eclipse plugins, MPS [37] is an example of a workbench with first-class languages and a language registry. Many workbenches support some level of dynamic loading of services, although their implementation may be tied to IDE interfaces that may make it hard to instantiate them in a headless fashion. Functional interfaces for editor services are rare, but could be implemented for workbenches that generate the service implementations. MontiCore [28] is an example of a workbench that applies similar techniques with regard to combining host and embedded language parsers.

6.2 Syntax and Parsing

Language engineers can instantiate the testing language for any Spoofax language that is loaded in the Eclipse environment, either as an Eclipse plugin, or as a language project in source form. Once the developer specifies which language to test, the syntax of the testing language is instantly specialized by integrating the syntax of the language under test. This makes it possible to provide syntactic editor services such as syntax highlighting, and to parse the file to a single abstract syntax tree that is used in the implementation of semantic editor services and tests evaluation.

In previous work, we have shown how generalized parsers can be used to syntactically compose languages [6]. Generalized parsers support the full class of context-free grammars, which is closed under embedding. That is, any two context-free grammars can be composed to form again a context-free grammar. This makes it possible to support modular syntax definitions and allow for language composition scenarios such as embedding and extension.

Unfortunately, the embedding as we have defined it for the LPTL is not context-free. First, because test fragments can only be parsed when their setup block (context) is taken into consideration. Second, because test fragments are allowed to contain syntax errors, such as spurious closing brackets. Even when considering syntax error recovery techniques, which use a local or global search space or token skipping techniques [10], the test fragments must be considered in isolation to ensure correct parsing of the test specification.

As we cannot compose the LPTL parser with that of the language under test at the level of the syntax definition, we have chosen for a more ad hoc approach, parsing the LPTL and the language under test in separate stages. Figure 16 illustrates these stages. First, we parse the LPTL using a skeletal test specification syntax, where every setup and test fragment is parsed as a lexical string. Second, we parse each setup and test fragment again using the parser of the language under test. For this, the setup fragment is instantiated for each test, and also parsed separately. As a third step, we merge the abstract syntax trees and token streams[8] of the skeletal LPTL and of the test cases. The merged tree and token stream are then used for editor services and to evaluate the tests in the module.

Our approach makes it possible to directly instantiate the language without generating a new parser for the instantiated LPTL. Our Java-based scannerless generalized-LR (JSGLR) parser is relatively efficient, ensuring good runtime performance and allowing for interactive use of the Spoofax testing language. With support for error recovery techniques [9, 23], JSGLR also ensures that a valid abstract syntax tree is produced for providing editor services in case a test module is syntax incorrect (as seen in Figure 6 (a)). There are still opportunities for performance optimizations; e.g. the three stages could be more tightly integrated and caching could be added for parsing the test fragments, but so far we have not found the need for this.

6.3 Tool Support

The language registry provided by Spoofax and IMP maintains a collection of all languages supported in the environment, and provides access to factory classes to instantiate language-specific editor services (e.g. a syntax highlighter, content completion service, or code generator). Using the language registry, and the dynamic editor service loading facilities of Spoofax, it is possible to access the parser, syntactic start symbols, and a list of editor services that can be instantiated for a language, given its name. We use the registry to instantiate these services for editor support in the language under tests and for evaluating tests.

Editor service support in test fragments is provided by *delegation* to services of the language under test. An example is content completion support to help write test cases. The editor services for the testing language simply detect that they are invoked inside a test fragment, and then delegate the work to a service of the language under test. The only special cases are the syntax error marking and semantic error marking service. These produce lists of errors that must be filtered according to the test expectations (e.g., if an error is *expected*, the IDE should not add a red marker for it).

The parser and editor services are transparently loaded on demand once they are used for a test case. As a result, the editor for a test module is instantly specialized by simply specifying the name of the language under test. Further configuration is not required. Test cases are also automatically re-evaluated in the editor if a language definition is changed.

Test evaluation Tests are evaluated by instantiating the appropriate editor services for the language under test and

the embedding cannot be expressed as a context-free grammar, as we discuss in Section 6.2.

[8] Note that Spoofax uses scannerless parsing, but still constructs a token stream *after* parsing based on the lexical structure of parsed sentences, used for editor services such as syntax highlighting.

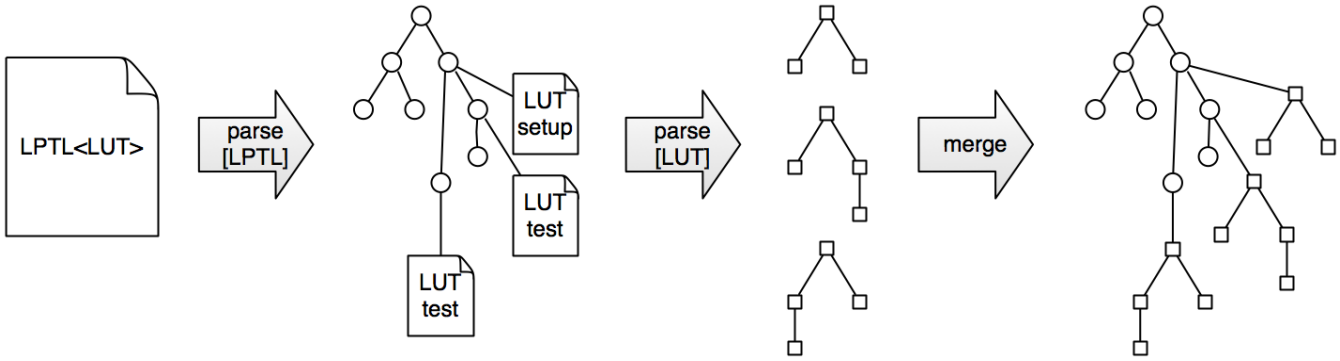


Figure 16. Parsing the LPTL with the embedded language under test (LUT).

applying them to the abstract syntax tree that corresponds to the test input fragment. For example, consider a reference resolving test such as that of Figure 12. To evaluate such a test, the language registry is used to instantiate services for semantic analysis and reference resolving. Like all editor services, the reference resolver has a functional interface, which essentially gets an analyzed program and an abstract syntax tree node of a reference as its input, and returns the declaration abstract syntax tree node. To test it, we give it the analyzed program, obtained from the analysis service, and the tree node that corresponds to the reference selected in the test fragment. The result is then compared to the expected result of the test case.

Execution tests often depend on some external executable that runs outside the IDE and that may even be deployed on another machine. Our implementation is not specific for a particular runtime system or compiler backend. Instead, language engineers can define a custom "runner" function that controls how to execute a program in the language. For a language such as mobl, a JavaScript engine such as Rhino or a browser emulator such as WebDriver can be used. At the time of writing, we have not yet completed that binding for mobl yet.

Test evaluation performance Editor services in Spoofax are cached with instantiation, and run in a background thread, ensuring low overhead and near-instant responsiveness for live test evaluation. Most editor services are very fast, but long-running tests such as builds or runs are better executed in a non-interactive fashion. We only run those through the batch test runner of Section 4.2 and display information markers in the editor if the test was changed after it last ran.

7. Discussion and Related Work

Related work on testing of language implementations can be divided into a number of categories: testing with general-purpose tools, testing with homogeneous and heterogeneous language embeddings, and test case generation.

Testing with general-purpose tools Considerable experience exists in the use of general-purpose testing tools and scripts for tests of language definitions [14, 16, 18, 22, 30]. Sometimes, they take the form of a simple shell script that builds all files in a directory. In other cases they use JUnit or a related xUnit-family [19] testing framework.

The use of these tools introduces a number of challenges when applied to language definitions. First, a major issue is that a significant investment in language-specific testing infrastructure must be made to support tests for different aspects of languages, ranging from syntax to semantics and editor services. We support a generic, declarative language for testing these aspects. A second issue is that to make sure tests are considered in isolation, each test case is generally put in a separate file. Using separate files for test cases introduces boilerplate code such as import headers. It also makes it harder to organize tests, requiring conventions for file and directory names. Using test files specified purely in the tested language also separates test conditions and expectations from the test program. With an additional investment in effort, some of these issues can be solved, but only on a per-language basis. We provide a language-generic solution.

Another limitation of these general testing tools is that they do not provide specialized IDE support for writing tests. Standard IDEs are only effective for writing valid programs and report spurious errors for negative test cases where errors are expected. Batch test runners such as JUnit also do not have the capability to directly direct users to the failing line of code in a test input in case a test fails.

Testing with homogeneous embeddings Language embedding is a language composition technique where separate languages are integrated. An example is the embedding of a database querying language into a general-purpose host language. These embeddings can be *heterogeneous*, where an embedded language can be developed in a different language than the host language, or *homogeneous*, where the host language is used to define the embedded language [21, 34]. Homogeneous embeddings of DSLs are sometimes also called internal DSLs. The embedding technique applied in this paper is a heterogeneous embedding.

Homogeneously embedded languages must always target the same host language. This can be a weakness when execution on a different platform is desired (e.g., JavaScript in the case of mobile development with mobl [20]). It can also be a strength in terms of tool support. As embedded languages all target the same platform, they can be tested in the same way. General-purpose testing frameworks can then be applied more effectively, since they can directly use the embedded language. In MPS [37], tests based on JUnit can be evaluated as they are typed, much like with our testing language. A restriction of the approach used in MPS is that it can only be used test the dynamic semantics of a language, and only allow for *positive* test cases. With MPS, a projectional editor is used that even restricts the tests so they can only follow the existing syntax of the language, ruling out sketches of new syntax and test-driven development of new syntactic constructs. Our test specification language is more flexible, supporting a much wider spectrum of test conditions (Figure 5), including negative test cases for syntax or static semantics and refactoring tests.

Testing with heterogeneous embeddings In previous work, we developed *parse-unit*, a grammar testing tool developed as part of the Stratego/XT program transformation system and tool [5]. This tool formed a precursor to the present work. Parse-unit would embed quoted program fragments in a test module, much like in the Spoofax testing language, but only supported syntactic test cases. There was also no IDE support for parse-unit, and all quoted program fragments were treated as strings rather than forming a first-class part of the language.

Other testing tools that support embedding have similar limitations as parse-unit, supporting only syntactic tests, and lacking IDE support. A notable example is gUnit [17], a testing language for ANTLR grammars. Facilities as those provided by gUnit have been lacking in current language workbenches and other interactive tools for building and debugging parsers such as ANTLRWorks [4].

Test case generation techniques There is a long history of research on test case generation techniques for language implementations. An overview is given in surveys by Boujarwah and Saleh [3], and Kossatchev and Posypkin [27]. These techniques use grammars to generate test programs. To control the theoretically infinite set of programs that can be generated using most grammars, they use annotations in grammars, external control mechanisms, or even imperative generators [8], to constrain this set. In some cases, sophisticated, hand-tailored program generators are used, such as Csmith [40], a successful 40,000-line C++ generator program for randomly generating C programs.

The set of test programs generated with these approaches can be used to stress-test compiler implementations. For example, they can be used to compare a compiler to its reference implementation, or to check for validity of generated code for the subset of programs that type-check. As such, they provide an excellent complementary approach to our test specifications, possibly catching corner cases that a language engineer did not think of. However, as they only test complete compilation chains and rely on a test oracle such as a reference compiler, they are less effective for testing languages while they are still under development. In contrast, our approach can be used from the point of inception of a language and even in the design process. By applying test-driven development, test specifications can be used to guide the development process. Our test specifications also provide a more varied array of tests by providing an extensive, open-ended set of test condition specification constructs for observable behavior of language implementations.

Unit tests for domain-specific languages As a side-effect of providing a language for testing language implementations, our test specification language can also be used to test programs written in that language (see Section 5.6). This makes it particularly useful in the domain of testing DSL programs, where testing tools and frameworks are scarce. Traditionally, language engineers would have to build such tools and frameworks by hand, but recently Wu et al. [39] provided a reusable framework for language testing. They require language engineers to extend their language with scripting constructs that generate JUnit test cases. The combined scripting and DSL language can then be used to specify tests. Their framework ensures that the mapping between the DSL test line numbers and the generated JUnit tests is maintained for reporting failing tests. They also provide a graphical batch test runner. While our approach does not provide the same flexibility and requires tests to be specified with a quotation marks and `language` and `run` clauses, it is interesting to note how our approach relieves language engineers from much of the heavy lifting required for implementing a DSL testing solution. We only require language engineers to specify a binding to an execution engine (Section 6.3), and we provide a generic test specification host language that is combined with the DSL to test.

8. Concluding Remarks

In this paper we proposed an approach to language definition testing by introducing the notion of a language-parametric testing language. The LPTL provides a zero-threshold, domain-specific testing infrastructure based on a declarative test specification language and extensive tool support for writing and executing tests. Our implementation in the form of the Spoofax testing language shows the practical feasibility of the approach.

Tests inspire confidence in language implementations, and can be used to guide an agile, test-driven language development process. Unfortunately, in current software language engineering practice, tests are still too often an afterthought. Especially DSLs often remain untested, as they are developed in a short timespan with limited resources. We believe that declarative language test suites should become

standard components of language definitions, just as BNF-style grammars are. Supported by an LPTL, tests are concise, implementation-independent, and require little to no effort to setup.

Future work In this paper we emphasized testing of observable behavior of languages, such as reported errors and name analysis as manifested by reference resolving in an IDE. Other analyses such as type or flow analysis are not manifested that way, but it can be useful to write test cases for them. Right now, these aspects are either indirectly tested, or tested using the generic "builders" interface for custom transformations. Direct support for testing such language definition aspects could be a worthwhile addition. Alternatively, rather than seeking to support all possible compiler and IDE aspects in a testing language, perhaps a better test abstraction mechanism is needed to specify multiple tests that interact with a language definition in the same way. Similarly, an abstraction mechanism for setup blocks could be introduced for improved modularization of test suites, e.g. by allowing of setup blocks to be put in libraries, to support multiple arguments, and to support composition.

For the interaction design of the LPTL, previous work on interactive disambiguation [24] could be applied for handling ambiguities of quoted test programs. Test understandability can also be improved using further visual aids, for example to emphasize differences between test inputs and outputs for refactoring tests.

The declarative basis provided by the LPTL can be used to integrate generally applicable supportive techniques for testing, such as test case prioritization, coverage metrics, coverage visualization, mutation testing, and mutation-based analyses for untested code. In particular, the integration of such techniques specialized for the domain of parser and compiler testing [3, 27, 29] is an important area of future work.

Acknowledgements This research was supported by NWO project 612.063.512, *TFA: Transformations for Abstractions* and the NIRICT LaQuSo Build Farm project. We would like to thank Martin Bravenboer for his work on the parse-unit project, which provided a basis for the Spoofax testing language; Danny Groenewegen for his inspiring tests scripts and test collection for WebDSL; Zef Hemel for his work on mobl that was used in examples in this paper; and Maartje de Jonge, Sander Vermolen, and the anonymous referees for suggestions for this paper.

References

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

[2] B. Beizer. *Software testing techniques*. Dreamtech Press, 2002.

[3] A. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617–625, 1997.

[4] J. Bovet and T. Parr. ANTLRWorks: an ANTLR grammar development environment. *Software: Practice and Experience*, 38(12):1305–1332, 2008.

[5] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[6] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In J. M. Vlissides and D. C. Schmidt, editors, *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004*, pages 365–383. ACM, 2004.

[7] P. Charles, R. M. Fuhrer, S. M. Sutton, Jr., E. Duesterwald, and J. Vinju. Accelerating the creation of customized, language-specific IDEs in Eclipse. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 191–206. ACM, 2009.

[8] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In I. Crnkovic and A. Bertolino, editors, *6th joint meeting of the European Software Engineering Conference and the Int. Symposium on Foundations of Software Engineering*, pages 185–194. ACM, 2007.

[9] M. de Jonge, E. Nilsson-Nyman, L. C. L. Kats, and E. Visser. Natural and flexible error recovery for generated parsers. In M. van den Brand, D. Gasevic, and J. Gray, editors, *Software Language Engineering, SLE 2009*, volume 5969 of *LNCS*, pages 204–223. Springer, 2009.

[10] P. Degano and C. Priami. Comparison of syntactic error handling in LR parsers. *Software – Practice and Experience*, 25(6):657–679, 1995.

[11] S. Efftinge and M. Völter. oAW xText - a framework for textual DSLs. In *Modeling Symposium, Eclipse Summit*, 2006.

[12] M. Fowler. Refactoring: Improving the design of existing code. In D. Wells and L. A. Williams, editors, *Extreme Programming and Agile Methods - XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference*, volume 2418 of *LNCS*, page 256. Springer, 2002.

[13] M. Fowler. Language workbenches: The killer-app for domain specific languages? `http://martinfowler.com/articles/languageWorkbench.html`, 2005.

[14] J. D. Frens and A. Meneely. Fifteen compilers in fifteen days. In D. Baldwin, P. T. Tymann, S. M. Haller, and I. Russell, editors, *39th Technical Symposium on Computer Science Education, 2006*, pages 92–96. ACM, 2006.

[15] E. Gamma and K. Beck. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[16] J. B. Goodenough. The Ada compiler validation capability. In *Proceedings of the ACM-SIGPLAN symposium on Ada programming language*, SIGPLAN '80, pages 1–8. ACM, 1980.

[17] gUnit - grammar unit testing. http://www.antlr.org/wiki/display/ANTLR3/gUnit+-+Grammar+Unit+Testing.

[18] R. Gómez, J. C. Augusto, and A. Galton. Testing an event specification language. In *Software Engineering & Knowledge Engineering, SEKE 2001*, pages 341–345, 2001.

[19] P. Hamill. *Unit Test Frameworks, chapter. Chapter 3: The xUnit Family of Unit Test Frameworks*. O'Reilly, 2004.

[20] Z. Hemel and E. Visser. Declaratively programming the mobile web with mobl. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011*, Portland, Oregon, USA, 2011. ACM.

[21] P. Hudak. Modular domain specific languages and tools. In *Software Reuse, ICSR '98*. Computer Society, jun 1998.

[22] Jacks (Jacks is an Automated Compiler Killing Suite). http://sources.redhat.com/mauve/jacks.html.

[23] L. C. L. Kats, M. de Jonge, E. Nilsson-Nyman, and E. Visser. Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-LR parsing. In S. Arora and G. T. Leavens, editors, *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009*, pages 445–464. ACM, 2009.

[24] L. C. L. Kats, K. T. Kalleberg, and E. Visser. Interactive disambiguation of meta programs with concrete object syntax. In M. van den Brand, B. Malloy, and S. Staab, editors, *Software Language Engineering, SLE 2010*, LNCS. Springer, 2011.

[25] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463. ACM, 2010.

[26] L. C. L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 918–932. ACM, 2010.

[27] A. Kossatchev and M. Posypkin. Survey of compiler testing methods. *Programming and Computer Software*, 31(1):10–19, 2005.

[28] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In R. F. Paige and B. Meyer, editors, *Objects, Components, Models and Patterns, TOOLS EUROPE 2008*, volume 11 of *Lecture Notes in Business Information Processing*, pages 297–315. Springer, 2008.

[29] R. Lämmel. Grammar testing. In H. Hußmann, editor, *Fundamental Approaches to Software Engineering, FASE 2001*, volume 2029 of *LNCS*, pages 201–216. Springer, 2001.

[30] B. A. Malloy, J. F. Power, and J. T. Waldron. Applying software engineering techniques to parser design: the development of a C# parser. In *SAICSIT '02: 2002 research conference of the South African institute of computer scientists and information technologists on Enablement through technology*. South African Institute for Computer Scientists and Information Technologists, 2002.

[31] G. Myers. *The art of software testing, 2nd edition*. Wiley-India, 2008.

[32] R. W. Selby, editor. *Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research*. Wiley-Computer Society Press, 2007.

[33] M. Strembeck and U. Zdun. An approach for the systematic development of domain-specific languages. *Software: Practice and Experience*, 39(15):1253–1292, 2009.

[34] L. Tratt. Domain specific language implementation via compile-time meta-programming. *Trans. Program. Lang. Syst.*, 30(6), 2008.

[35] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[36] E. Visser. WebDSL: A case study in domain-specific language engineering. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II, Int. Summer School, GTTSE 2007*, volume 5235 of *LNCS*, pages 291–373. Springer, 2007.

[37] M. Voelter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. In M. van den Brand, B. Malloy, and S. Staab, editors, *Software Language Engineering, SLE 2010*, LNCS. Springer, 2010.

[38] B. Wichmann and Z. Ciechanowicz. *Pascal compiler validation*. John Wiley & Sons, Inc. New York, NY, USA, 1983.

[39] H. Wu, J. Gray, and M. Mernik. Grammar-driven generation of domain-specific language debuggers. *Software: Practice and Experience*, 38(10):1073–1103, 2008.

[40] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Conference on Programming Language Design and Implementation, PLDI 2011*. Press, June 2011.