

Delft University of Technology
Software Engineering Research Group
Technical Report Series

Automated Evaluation of Syntax Error Recovery

Maartje de Jonge and Eelco Visser

Report TUD-SERG-2012-009

TUD-SERG-2012-009

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:

<http://www.se.ewi.tudelft.nl/techreports/>

For more information about the Software Engineering Research Group:

<http://www.se.ewi.tudelft.nl/>

Note: Short Paper at ASE 2012

© copyright 2012, by the authors of this report. Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the authors.

Automated Evaluation of Syntax Error Recovery

Maartje de Jonge
Delft University of Technology
Netherlands

Eelco Visser
Delft University of Technology
Netherlands

ABSTRACT

Evaluation of parse error recovery techniques is an open problem. The community lacks objective standards and methods to measure the quality of recovery results. This paper proposes an automated technique for recovery evaluation that offers a solution for two main problems in this area. First, a representative testset is generated by a mutation based fuzzing technique that applies knowledge about common syntax errors. Secondly, the quality of the recovery results is automatically measured using an oracle-based evaluation technique. We evaluate the accuracy of different oracle metrics by comparing results obtained by automated evaluation with results obtained by manual inspection. The evaluation shows a clear correspondence between the evaluated oracle metrics and human judgement.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—*Parsing*

General Terms

Languages, Measurement

Keywords

Error Recovery, Parsing, IDE, Evaluation, Test Generation

1. INTRODUCTION

Integrated development environments (IDEs) increase programmer productivity by combining generic language development tools with services tailored for a specific language. Language specific services require as input a structured representation of the source code in the form of an abstract syntax tree (AST) constructed by the parser.

To provide rapid syntactic and semantic feedback, IDEs interactively parse programs as they are edited. The parser runs in the background with each key press or after a small delay passes. As the user edits a program, it is often in a syntactically invalid state. Parse error recovery techniques can diagnose and report parse errors, and can construct ASTs for syntactically invalid programs [8].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASE '12, September 3–7, 2012, Essen, Germany

Copyright 12 ACM 978-1-4503-1204-2/12/09 ...\$10.00.

Thus, to successfully apply a parser in an interactive setting, proper parse error recovery is essential.

Evaluation of error recovery techniques is an open problem in the area of parsing technology. An objective and automated evaluation method is essential to do benchmark comparisons between existing techniques, and to detect regression in recovery quality due to adaptations of the parser implementation. Currently, the community lacks objective standards and methods for performing thorough evaluations. We identified two challenges: first, the recovery technique must be evaluated against a representative set of test inputs, secondly, the recovery outputs must be automatically evaluated against a quality metric. The aim of this paper is to provide an automated method for evaluating error-recovery techniques.

Test data.

The first challenge for recovery evaluation is to obtain a representative test suite. Evaluations in the literature often use manually constructed test suites based on assumptions about which kind of errors are the most common [11, 13, 18]. The lack of empirical evidence for these assumptions raises the question how representative the test cases are, and how well the technique works in general. Furthermore, manually constructed test suites tend to be biased because in many cases the same assumptions about edit behavior are used in the recovery algorithm as well as in the test set that measures its quality. Many test inputs need to be constructed to obtain a test set that is statistically significant. Thus, manual construction of test inputs is a tedious task that easily introduces a selection bias.

A better option for obtaining representative test data is to construct a test suite based on collected data, an approach which is taken in [20] and [21]. However, collecting practical data requires administration effort and may be impossible for new languages that are not used in practice yet. Furthermore, practical data does not easily provide insight into what kind of syntax errors are evaluated, nor does it offer the possibility to evaluate specific types of syntax errors specified by the compiler tester. An additional problem is automated evaluation of the parser output, which is not easily implemented on collected data.

As an alternative to collecting edit scenarios in practice, this paper investigates the idea of generating edit scenarios. The core of our technique is a general framework for iterative generation of syntactically incorrect files. To ensure that the generated files are realistic program files, the generator uses a mutation based fuzzing technique that generates a large set of erroneous files from a small set of correct input files taken from real projects. To ensure that the generated errors are realistic, the generator implements knowledge about editing behavior of real users, which was retrieved from an empirical study. The edit behavior is implemented by *error generation rules* that specify how to construct an erroneous fragment

from a syntactically correct fragment. The implicit assumption is that these rules implement edit behavior that is generic for different languages. The generation framework provides extension points where compiler testers can hook in custom error generation rules to test the recovery of syntax errors that are specific for a given language.

Quality measurement.

The second challenge for recovery evaluation is to provide a systematic method to measure the quality of the parser output, e.g., the recovered AST which represents a speculative interpretation of the program.

Human judgement is decisive with respect to the quality of recovery results. For this reason, Pennello and DeRemer [20] introduce human criteria to categorize recovery results. A recovery is rated *excellent* if it is the one a human reader would make, *good* if it results in a reasonable program without spurious or missed errors, and *poor* if it introduces spurious errors or if excessive token deletion occurs. Though human criteria most accurately measure recovery quality, application of these criteria requires manual inspection of the parse results which makes the evaluation subjective and inapplicable in an automated setting. Another disadvantage is the lack of precision, only three quality criteria are distinguished.

Oracle-based approaches form an alternative to manual inspection. First, the intended program is constructed manually. Then, the recovered program is compared to the intended program using a diff based metric on either the ASTs or the textual representations obtained after pretty printing. An oracle-based evaluation method is applied in [6] and [18]. The former uses textual diffs on pretty-printed ASTs, while the latter uses tree alignment distance [12] as a metric of how close a recovered interpretation is to the intended interpretation of a program.

A problem with these approaches is the limited automation. Differential oracle approaches allow automated evaluation, but the intended files must be specified manually which requires considerable effort for large test suites. Furthermore, the intended recovery may be specified after inspecting the recovery suggestion in the editor, which causes a bias towards the technique implemented in the editor.

To address the concern of automation, we extend the error generator so that it generates erroneous files together with their oracle interpretations. The oracle interpretations follow the interpretation of the base file, except for the affected code structures; for these structures, an *oracle generation rule* is implemented that specifies how the intended interpretation is constructed from the original interpretation. Oracle generation rules are complementary to error generation rules, e.g., the constructed oracle interpretation must be in line with the textual modifications applied to the code structure by the error generation rule.

The remaining problem is to define a suitable metric between recovered programs and their intended oracle programs. We compared four differential oracle metrics based on their accuracy and on qualitative aspects such as applicability and comprehensibility. We concluded that all evaluated metrics accurately reflect recovery quality. A token based diff metric seems the most attractive since 1) it is transparent to humans, 2) it does not depend on the particular abstract interpretation constructed by the parser, and 3) it does not depend on a particular formatting function. A practical disadvantage however is the additional language tooling that is needed to construct token sequences from abstract syntax trees.

Contributions.

This paper provides the following contributions:

- A preliminary statistical study on syntax errors that occur during interactive editing.
- A test input generation technique that generates realistic edit scenarios specified by error generation rules.
- A full automatic technique to measure the quality of recovery results for generated edit scenarios.
- A comparison of different metrics for recover quality measurement.

We start this paper with a statistical analysis of syntax error that occur during interactive editing (Section 2). The input generation technique is discussed in Section 3, while automatic quality measurement is the topic of Section 4.

2. UNDERSTANDING EDIT BEHAVIOR

We did an empirical study to gain insight into edit scenarios that occur during interactive editing. The final objective was to implement these scenarios in an error generation tool used for the generation of test inputs for error recovery. With this objective in mind, we focus on the following research questions:

- How are syntax errors distributed in the file? Do multiple errors occur in clusters or in isolation? What is the size of the regions that contain syntax errors?
- What kind of errors occur during interactive editing? Can we classify these errors in general categories?

2.1 Experimental Design

For the analysis of syntax errors we examined edit data for three different languages: Stratego [3], a transformation language used for code generation and program analyses; SDF [16], a declarative syntax definition language; and WebDSL [9], a domain-specific language for web development. We choose these languages since they are considerably different from each other, covering important characteristics of respectively functional, declarative and imperative languages.

We did a large study on edit behavior in Stratego, collecting edit scenarios from two groups; a group of students that took part in a graduate course on compiler construction, and a group of researchers that work in this field. Contrary to the researchers, the students had only limited experience with the Stratego language. In total, we collected 43.984 Stratego files (program snapshots) from 22 students with limited experience in Stratego programming and 13.427 Stratego files from 5 experienced researchers. From both groups we randomly took a subset of +-2000 scenarios. To evaluate the impact of familiarity with a language on the syntactic errors being made, the data was examined from the two groups separately. However, it turned out that the unfamiliarity with the language had only a minor effect on the edit behavior we observed. For WebDSL and SDF we did not have a large user group, we analysed respectively 936 WebDSL files from 2 programmers and 103 SDF files from 3 programmers.

The edit data was collected during interactive editing in an Eclipse based IDE. The support offered by the IDE included main editor services such as syntax highlighting, code completion, automatic bracket completion, and syntax error recovery and reporting. The edit scenarios were created by copying the file being edited, effectively recording all key touches applied by the programmer. We filtered structurally equivalent scenarios that only differ at the character level.

2.2 Distribution of Syntax Errors

We analysed the distribution of errors in the file by counting the number of erroneous constructs that reside in a program file being edited. We only looked at the main constructs of a language such

	Stratego S.	Stratego K.	SDF	WebDSL
Incomplete Construct	24	18	22	28
Comments	2	4	2	4
String Literal	3	3	18	3
Scope	0	0	0	1
Random Edits	17	22	8	11
Large Region	2	3	0	3
Lang. Spec.	2	0	0	0

Figure 1: Classification of edit scenarios for different languages. The numbers indicate the percentage of errors that fall in a given category for a given test group.

as method declarations in Java. The results show that programmers typically edit code constructs one by one, solving all syntax errors before editing the next construct. It follows that syntax errors are typically clustered in a single construct, which was the case in about 80% of the analyzed snapshots.

Syntax errors are typically located in the region that is edited inbetween two syntactically correct program states. We measured the size of the edited region by taking the minimum textual diff between the erroneous files and their surrounding correct files. The results show that in most cases (70%) all syntax errors are located on a single line. The cases where the affected region was larger than five lines (+10%) were mostly the cases where multiple unrelated errors were involved.

2.3 Classification of Syntax Errors

We manually inspected erroneous files to gain insight into the kind of errors that occur in the region being edited. We inspected 50 randomly selected files for each test group (Stratego students, Stratego researchers, SDF and WebDSL). The results are summarized in Figure 1. The categories are explained below.

- *Incomplete constructs*, language constructs that miss one or more symbols at the suffix, e.g. an incomplete `for` loop as in `for (x = 1; x`.
- *Random errors*, constructs that contain one or more token errors, e.g., *missing*, *incorrect* or *superfluous* symbols.
- *Scope errors*, constructs with missing or superfluous scope opening or closing symbols.
- *String or comment errors*, block comments or string literals that are not properly closed, e.g., `/* . . . *`
- *Large erroneous regions*, severely incorrect code fragments that cover multiple lines.
- *Language specific errors*, errors that are specific for a particular language.

3. GENERATION OF SYNTAX ERRORS

To test the quality of an error recovery technique, parser developers can manually write erroneous input programs containing different kind of syntax errors. However, to draw a conclusion that is statistically significant, the developer must extend the test set so that it becomes sufficiently large and diversified. Variation points are: the error type, the construct that is broken, the syntactic context of the broken construct, and the layout of the input file nearby the syntax error.

It is quite tedious to manually write a large number of invalid input programs that cover various instances of a specific error type. As an alternative, our error generation framework allows the tester to write a generator that automates the creation of test files that contain syntax errors of the given type. Error generators are composed from error generation rules and error seeding strategies. The error

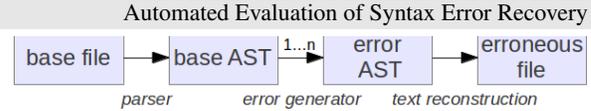


Figure 2: Error generators specify how to generate multiple erroneous files from a single base file.

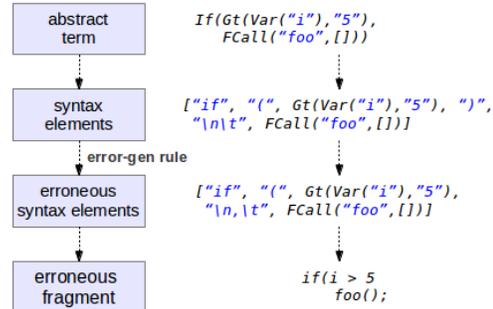


Figure 3: Error generation rules create erroneous constructs by modifying the syntax elements of correct constructs.

generation rules specify how to construct an erroneous fragment from a syntactically correct fragment; the error seeding strategies control the application of the error generation rules, e.g., they select the code constructs that will be broken in the generated test files. A generator for files with multiple errors can be defined by combining generators for single error files.

Figure 2 shows the work flow for test case generation implemented by the generation framework. First, the parser constructs the AST of the base file. Then the generator is applied which constructs a large set of ASTs that represent syntactically erroneous variations of the base program. Finally, the texts of the test files are reconstructed from these ASTs by a text reconstruction algorithm that preserves the original layout [7]. Error generation rules may by accident generate modified code fragments that are in fact syntactically correct. As a sanity check, all generated files that are accidentally correct are filtered out by parsing them with error recovery turned off. The framework is implemented in Stratego [3], a language based on the paradigm of strategic rewriting.

3.1 Error Generation Rules

Error generation rules are applied to transform abstract syntax terms into string terms that represent syntactically erroneous constructs. The error generation rules operate on the concrete syntax elements that are associated to the abstract term, e.g., its child terms plus the associated literals and layout tokens. Applying modifications to concrete syntax elements rather than token or character streams offers refined control over the effect of the rule. Typically, error generation rules create syntax errors on a code construct, while leaving its child constructs intact. As an example, Figure 3 illustrates the application of an error generation rule on an `if` construct. The given rule removes the closing bracket of the `if` condition from the list of syntax elements.

The generation framework predefines a set of primitive generation rules that form the building blocks for more complex rules. Primitive error generation rules introduce simple errors by applying insertion, deletion, or replacement operations on the syntax elements associated with a code structure. For example, the following generation rule drops the last syntax element of a code construct.

```

if(i > 5) {           if(i > 5) {           if(i
  foo(i);             if(i > 5)           if(i)
}                     if(i > 5)           if(
if(i > 5) {           if(i >           if()
  foo(i);             if(i >           if

```

Figure 4: Incomplete construct: prefixes of an if statement

```

drop-last-element:
  syntax-elements -> <init> syntax-elements

```

By composing primitive generation rules, complex clustered errors can be generated. For example, iterative application of the rule `drop-last-element` generates incomplete constructs that miss n symbols at the suffix.

```

generate-incompletion(|n) =
  repeat(drop-last-element, n)

```

A second example is provided by nested incomplete construct errors. By applying the `generate-incompletion` rule twice, first to the construct and then to the last child construct in the resulting list, an incomplete construct is created that resides in an incomplete context, for example “if(i >”.

```

generate-nested-incompletion(|n,m) =
  generate-incompletion(|n);
  at-last-elem(generate-incompletion(|m))

```

A final example is provided by the (optional) support for automated bracket completion offered by most commonly used IDEs. We simulate this edit scenario by generating test cases with added closing brackets for all unclosed opening brackets in the incomplete prefix of a construct, for example “if(i <)”. The test cases are generated by composing the incompletion rule with a rule that repairs the bracket structure.

```

generate-incompletion-matching-brackets(|n) =
  generate-incompletion(|n);
  repair-bracket-structure

```

Figure 4 shows the prefixes of an if construct that are constructed from the incompletion rules described in this section.

3.2 Error Seeding Strategies

In principle, error generation rules could be applied iteratively to all terms in the abstract syntax tree, generating test files each time that the rule application succeeds. However, the resulting explosion of test files increases evaluation time substantially without yielding significant new information about the quality of the recovery technique. As an alternative to exhaustive application, we let the tester specify an error seeding strategy that determines to which terms an error generation rule is applied. Typically, constraints are specified on the sort and/or the size of the selected terms, and, to put a limit on the size of the test suite, a maximum is set or a coverage criterion is implemented. For example, we predefined a coverage criterion that states that all types of constructs that appear in the abstract syntax tree must occur exactly once in the list of selected terms.

3.3 Predefined Generators

The remaining challenge for test input generation is to implement error generators that cover common syntax errors. From the preliminary study covered in Section 2 we conclude that most syntax errors are editing related and generic for different languages. We implemented reusable generators for these scenarios. In addition, we implemented a generator that generates erroneous files containing a combination of errors of the identified scenarios. Only

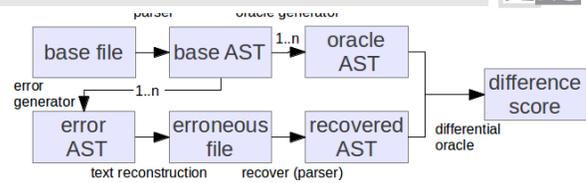


Figure 5: Automated evaluation of test outputs.

a few errors were related to error prone constructs in a particular language. We leave it to an expert of a language to implement custom error generators for language specific errors.

4. QUALITY MEASUREMENT

An important problem in automated generation of test inputs is automated checking of the outputs, also known as the oracle problem. We extend the test case generation framework with a differential oracle technique that automates quality measurement of recovery outputs. Figure 5 shows the work flow for the evaluation framework that includes an oracle generation technique and a differential oracle metric which are discussed below.

4.1 Oracle Construction

The quality of the recovered AST is given by its closeness to the AST that represents the intended program, also called the oracle AST. We automate the construction of oracle ASTs for generated error files. First, we assume that all unaffected code constructs keep their original interpretation. Thus, the constructed oracle AST follows the base AST except for the affected terms. Secondly, we let the compiler tester implement an *oracle generation rule* that complements the error generation rule and specifies how the oracle interpretation of the affected term is constructed from the original term. In the given example (Figure 6) the original term itself represents the optimal repair which is the common situation for errors that are constructed by deletion, insertion or replacement of literal tokens. In some exceptional cases, error generation rules may generate errors for which the intended recovery is too speculative to be specified by an automated oracle generation rule. For these cases, manual inspection is required to determine the intended recovery.

The interpretation of an erroneous construct can be ambiguous. For example, the broken arithmetic expression $1\ 2$ has many reasonable recover interpretations, such as: $1 + 2$, $1 * 2$, 1 , and 2 . Selecting only one of these interpretations as the intended interpretation can cause a small inaccuracy in the measured quality. A possible solution is to allow the specification of multiple ambiguous interpretations or the specification of a placeholder interpretation that serves as a wildcard. The disadvantage of this approach is that it violates the well-formedness of the oracle AST which complicates further processing by language tools such as pretty-printers.

4.2 Quality Metrics

The remaining problem is to define a suitable metric between recovered programs and their intended oracle programs. Below we discuss different metrics schematically shown in Figure 7. In addition, we discuss human criteria introduced by Pennello and DeRemer [20] as a non-automated quality metric.

Human criteria.

Human judgement is decisive with respect to the quality of recovery results. For this reason, Pennello and DeRemer [20] introduce human criteria to rate recovery outputs. A recovery is rated

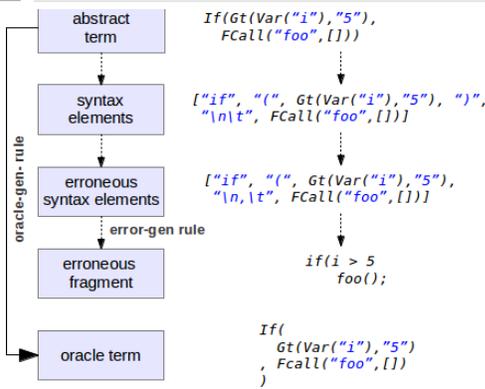


Figure 6: Oracle generation rules construct oracle interpretations for terms affected by their complementary error generation rules.

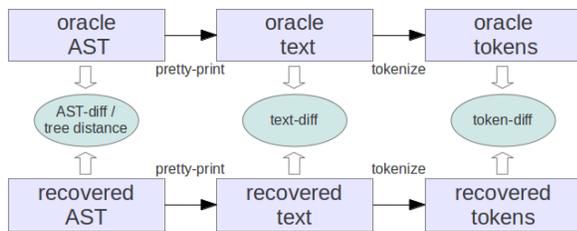


Figure 7: Possible metrics on different program representations.

excellent if it is the one a human reader would make, *good* if it results in a reasonable program without spurious or missed errors, and *poor* if it introduces spurious errors or if excessive token deletion occurs. Human criteria represent the state of the art evaluation method for syntactic error recovery, applied in, amongst others, [20, 19, 8, 5].

AST diff.

A simple AST-based metric can be defined by taking the size of the textual AST diff. First, the ASTs of both the recovered program and the intended program are printed to text, formatted so that nested structures appear on separate lines. Then, the printed ASTs are compared by counting the number of lines that are different in the recovered AST compared to the intended AST (the “diff”). The diff size obviously depends on the AST formatter and the settings of the textual diff function. For the results in this section we use the ATerm formatter [2], and the Unix diff utility with the settings: “-w, -y, -suppress-common-lines, -width=100”.

Tree edit distance.

An alternative AST-based metric is given by the tree edit distance. Given the edit operations term insertion, term deletion, term move and label update, and given a cost function defined on these operations, the tree edit distance is defined as follows. An edit script between T1 and T2 is a sequence of edit operations that turns T1 into T2. The cost of an edit script is given as the weighted sum over the edit operations contained in the script. The tree edit distance between T1 and T2 is defined as the cost of the minimum-cost edit script. An algorithm to calculate tree edit distances is described

in [4]. The tree edit distances in this chapter are calculated based on a cost function that assigns cost 1 to each edit operation.

Token diff.

A metric can be based on concrete syntax by taking the diff on the token sequences that represent the recovered, respectively the intended program. The token diff counts the number of (non-layout) token insertions and deletions required to turn the recovered token sequence into the intended token sequence. The token sequences can be obtained via pretty-printing and tokenization (Figure 7).

Textual diff.

Assumed that we have a human-friendly pretty printer, a concrete syntax metric can also be based on the textual representations obtained after pretty printing (Figure 7). The textual diff counts the number of lines that are different between the recovered and intended program text. These lines typically correspond to reasonably fine-grained constructs such as statements. The size of the textual diff depends on the pretty-print function and the settings of the textual diff utility. For the results in this section we pretty-print Java programs using the `pp-java` function that is part of the java frontend [22] defined in the Stratego/XT framework [1]. Furthermore, we use the Unix diff utility with the settings: “-w, -y, -suppress-common-lines, -width=100”.

4.3 Comparison of Metrics

In this section we compare the different metrics based on their accuracy and on qualitative criteria such as applicability and comprehensibility.

4.3.1 Accuracy

The Pearson product-moment correlation coefficient [23] is widely used to measure the linear dependence between two variables X and Y . The correlation coefficient ranges from -1 to 1 . A correlation equal to zero indicates that no relationship exists between the variables. A correlation of $+1.00$ or -1.00 indicates that the relationship between the variables is perfectly described by a linear equation; all data points lying on a line for which Y increases, respectively decreases, as X increases. More general, a positive correlation coefficient means that X_i and Y_i tend to be simultaneously greater than, or simultaneously less than, their respective means; while a negative correlation coefficient means that X_i and Y_i tend to lie on opposite sides of their respective means.

We measured the correlation between the different metrics based on the Pearson correlation coefficient. For this, we applied the quality metrics to a set of 150 randomly generated Java test cases that cover the different error categories identified in Section 2. The correlation matrix of Figure 8 shows the results. The results indicate that there is a strong correlation between the tree edit distance, ast-diff and token-diff metrics. The correlations that involve human criteria and/or the pp-diff metric are somewhat less pronounced. A likely reason is that these metrics have a lower resolution, i.e. they provide a more coarse grained categorization. To test this presumption, we also calculated the correlation coefficients after calibrating all metrics to a scale *no diff*, *small diff* and *large diff* so that these categories roughly correspond to the human criteria *excellent*, *good* and *poor*. The results shown in Figure 9 indicate a strong correlation between all compared metrics.

The accuracy of a measurement technique indicates the proximity of measured values to ‘true values’. A strong correlation between measured values and true values indicates that the measured values are an accurate predictor for true values. Since human cri-

	tree distance	ast-diff	token-diff	pp-diff	human criteria
tree-distance	1.00	0.95	0.94	0.54	0.71
ast-diff	0.95	1.00	0.94	0.60	0.72
token-diff	0.94	0.94	1.00	0.51	0.69
pp-diff	0.54	0.60	0.51	1.00	0.76
human criteria	0.71	0.72	0.69	0.76	1.00

Figure 8: Correlation matrix for different metrics on programs.

	tree distance	ast-diff	token-diff	pp-diff	human criteria
tree-distance	1.00	0.98	0.96	0.93	0.96
ast-diff	0.98	1.00	0.99	0.91	0.97
token-diff	0.96	0.99	1.00	0.93	0.98
pp-diff	0.93	0.91	0.93	1.00	0.92
human criteria	0.96	0.97	0.98	0.92	1.00

Figure 9: Correlation matrix for different metrics on programs. All diff metrics are calibrated to a scale *no diff*, *small diff* and *large diff* so that these categories roughly correspond to the human criteria *excellent*, *good* and *poor*.

teria are decisive with respect to recovery quality, we use the Pennello and DeRemer criteria to determine the ‘true values’. From the results in Figure 9 for human criteria we conclude that all metrics have a high prediction accuracy, that is, a high diff is a good predictor of a poor recovery and the same for the other categories.

4.3.2 Qualitative Criteria

Although human criteria are decisive with respect to recover quality, application of these criteria requires manual inspection of the parse results which makes the evaluation subjective and inapplicable in an automated setting. Another disadvantage is the low resolution; only three quality criteria are distinguished. A more refined classification is possible but at the risk of increasing the subjectivity.

AST-based metrics (AST-diff and tree edit distance) reflect recovery quality in the sense that they assign a larger penalty to recoveries that for a larger part of the program provide an inferior interpretation. Empirical evaluation shows a strong correspondence with human judgement (Figure 9). A disadvantage of AST based metrics however is that these metrics are not transparent to humans who are typically more familiar with concrete syntax than with abstract syntax. A related problem is that AST-based metrics heavily dependent on the particular abstract interpretation defined for a language. That is, a verbose abstract syntax will lead to higher difference scores than a more concise abstract syntax. The dependence on the abstract interpretation makes it harder to compare parsers for the same language that construct a different abstract representation. To avoid systematic errors, the parser outputs should preferably be translated into the same abstract interpretation; for example via pretty-printing and reparsing. Unfortunately, this may require effort in case the recovered ASTs are not well-formed or in case a pretty-print function is not yet implemented.

Concrete syntax better reflects human intuition and is independent from the particular abstract interpretation defined for a language. Empirical evaluation shows that metrics based on concrete

syntax accurately reflect human judgement (Figure 9). The diff metric based on the token stream has a higher measurement resolution since it can assign a larger penalty to recoveries for which multiple deviations appear on what would be a single line of pretty-printed code. An additional advantage is that the token diff does not depend on the particular layouting that is provided by the pretty-printer. A possible limitation of text-based metrics is that they require additional compiler tooling in the form of a human-friendly pretty-printer for textual diffs, and a pretty-printer and tokenizer for token based diffs.

From this discussion we conclude that all discussed differential oracle metrics are suitable to measure recovery quality. The token based metric has the advantage that it is understandable for humans, that it does not depend on the particular abstract interpretation constructed by the parser, and that it does not depend on a particular formatting function. A practical disadvantage however is the additional language tooling that is needed to construct token sequences from abstract syntax trees. From the AST-based metrics, the tree edit distance seems preferable over the AST-diff count since it does not depend on a particular formatter and diff function.

5. RELATED WORK

Error recovery evaluation.

Recovery techniques in literature use testsets that are either manually constructed [6, 18], or composed from practical data [20, 21]. According to our knowledge, test generation techniques have not yet been applied to recovery evaluation.

Human criteria [20] and differential oracles [6, 18] form the state of the art methods to measure the quality of recovery results. We accomplished to apply a differential oracle technique in a full automatic setting.

Analysis of syntax errors.

To gain insight into syntax errors that are actually made during editing, we investigated edit behavior by applying a statistical analysis of collected edit data. Previous studies on syntax errors [15, 21] focus on on-demand compilation. These studies report that most syntax errors consisted of a single missing or erroneous token. In contrast, we studied syntax errors that occurred during background compilation as applied in modern IDEs. Comparison of the results show that syntax errors that occur during background compilation are more complex and often involve multiple errors clustered in a small fragment.

Fuzzing.

Fuzzing is a popular method used in software testing. The technique of fuzzing consists of sending a large number of different inputs to a program to test its robustness against unpredictable or invalid data. The inputs are normally constructed in one of two fashions, generation or mutation. Generation approaches produce testdata based on a specification of the set of input data. Alternatively, mutation-based approaches modify valid input data collected from practical data. The modifications may be random or heuristic and are possibly guided by stochastic parameters. Mutation and generation-based fuzzing techniques are compared in [17]. We use a mutation based fuzzing technique to generate testcases for syntactic error recovery.

Compiler testing.

Papers on compiler testing generally consider the parser as a language recognizer that outputs `true` in case a sequence of characters

belongs to a given formal language and false otherwise. Other features of the parser implementation such as the constructed abstract syntax tree and the reported error messages are ignored.

Previous work on automated compiler testing [14] primarily focuses on the generation of valid input programs. Based on a language specification, valid sentences are generated that form positive test cases for the parser implementation. A parser for a language must not only accept all valid sentences but also reject all invalid inputs. Only a few papers address the concern of negative test cases, a generation based approach is discussed in [24]. Generation based techniques are good at constructing small input fragments in a controlled manner. The problem these techniques address is to meet a suitable coverage criterion by a (preferable small) set of generated testcases [10, 24].

In contrast, the evaluation of error recovery techniques exclusively targets negative test cases. Furthermore, the generated negative test cases must be realistic error scenarios instead of small input fragments that are ‘likely to reveal implementation errors’. Generation-based techniques construct testcases starting from a formal specification. It hardly seems possible to formally specify what a realistic input fragment is that contains realistic syntax errors. For this reason, we consider a mutation based fuzzing technique more appropriate for the generation of error recovery test inputs.

6. CONCLUSION AND FUTURE WORK

This chapter introduces a method for fully automated recovery evaluation; the method combines a mutation-based fuzzing technique that generates realistic test inputs, with an oracle-based evaluation technique that measures the quality of the outputs. Automated evaluation makes it feasible to do a benchmark comparison between different techniques. As future work we intend to do a benchmark comparison between different parsers used in common IDEs. Furthermore, we plan to extend the empirical foundation of our recovery evaluation method.

7. REFERENCES

- [1] www.stratego-language.org.
- [2] M. G. J. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.
- [3] 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, June 2008.
- [4] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25:493–504, June 1996.
- [5] R. Corchuelo, J. A. Pérez, A. R. Cortés, and M. Toro. Repairing syntax errors in LR parsers. *ACM Trans. Program. Lang. Syst.*, 24(6):698–710, 2002.
- [6] 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, *SLE*, volume 5969 of *LNCS*, pages 204–223. Springer, 2009.
- [7] M. de Jonge and E. Visser. An algorithm for layout preservation in refactoring transformations. In U. Assmann and T. Sloane, editors, *SLE*, volume 6940 of *LNCS*, pages 40–59. Springer, 2012.
- [8] P. Degano and C. Priami. Comparison of syntactic error handling in LR parsers. *Software – Practice and Experience*, 25(6):657–679, 1995.
- [9] D. M. Groenewegen, Z. Hemel, L. C. L. Kats, and E. Visser. WebDSL: A domain-specific language for dynamic web applications. In N. Mielke and O. Zimmermann, editors, *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*, pages 779–780, New York, NY, USA, October 2008. ACM. (poster).
- [10] J. Harm and R. Lämmel. Two-dimensional approximation coverage. *Informatica (Slovenia)*, 24(3), 2000.
- [11] J. J. Horning. Structuring compiler development. In F. L. Bauer and J. Eickel, editors, *Compiler Construction, An Advanced Course*, 2nd ed, volume 21 of *Lecture Notes in Computer Science*, pages 498–513. Springer, 1976.
- [12] T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. In *CPM '94*, volume 807 of *LNCS*, pages 75–86, London, 1994. Springer-Verlag.
- [13] 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 G. T. Leavens, editor, *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, volume 44 of *ACM SIGPLAN Notices*, pages 445–464, New York, NY, USA, October 2009. ACM Press.
- [14] A. Kossatchev and M. Posypkin. Survey of compiler testing methods. *Programming and Computer Software*, 31(1):10–19, 2005.
- [15] C. R. Litecky and G. B. Davis. A study of errors, error-proneness, and error diagnosis in cobol. *Commun. ACM*, 19:33–38, January 1976.
- [16] B. Luttkik and E. Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *ASF+SDF 1997*, Electronic Workshops in Computing, Berlin, 1997. Springer-Verlag.
- [17] C. Miller and Z. N. J. Peterson. Analysis of mutation and generation-based fuzzing. Technical report, Independent Security Evaluators, 2007.
- [18] E. Nilsson-Nyman, T. Ekman, and G. Hedin. Practical scope recovery using bridge parsing. In D. Gasevic, R. Lämmel, and E. V. Wyk, editors, *SLE*, volume 5452 of *LNCS*, pages 95–113. Springer, 2009.
- [19] A. Pai and R. Kieburtz. Global Context Recovery: A New Strategy for Syntactic Error Recovery by Table-Drive Parsers. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 2(1):18–41, 1980.
- [20] T. J. Pennello and F. DeRemer. A forward move algorithm for LR error recovery. In *POPL*, pages 241–254. ACM, 1978.
- [21] G. D. Ripley and F. C. Druseikis. A statistical analysis of syntax errors. *Computer Languages, Systems & Structures*, 3(4):227–240, 1978.
- [22] <http://strategox.org/Stratego/JavaFront/>.
- [23] http://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient/.
- [24] S. V. Zelenov and S. A. Zelenova. Generation of positive and negative tests for parsers. *Programming and Computer Software*, 31(6):310–320, 2005.

TUD-SERG-2012-009
ISSN 1872-5392

