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 validity of our approach by comparing results obtained by automated evaluation with results obtained by manual inspection. The evaluation shows a clear correspondence between our quality metric 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 offering language specific editor services. These 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. 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.Thus, to successfully apply a parser in an interactive setting, proper parse error recovery is essential.

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

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 [3, 8]. 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 [9] and [10]. 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.

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. 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 generation framework provides extension points were 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. Human judgement is decisive with respect to the quality of recovery results. For this reason, Pennello and DeRemer [9] 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.

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 [3] and [8]. The former uses textual diffs on pretty-printed ASTs, while the latter uses tree alignment distance [6] as a metric.

An intrinsic problem with these approaches is that in a significant number of cases the optimal recovery is questionable, or arguable not unique. For example, the broken arithmetic expression `1 2` has many reasonable recover interpretations, such as: `1 + 2`, `1 * 2`, `1`, and `2`. The oracle program specifies only one of these solutions as the optimal solution. As a consequence, the alternative equivalent solutions will be rated lower than the specified solution.

A second concern is the lack of 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.

To address the concern of multiple equivalent recovery alternatives, a placeholder oracle can be specified that serves as a wildcard for the interpretation of an erroneous structure. Correct structures that are substructures of the affected structure are checked against oracle interpretations contained in the placeholder. By default, the generator constructs placeholder oracles for the erroneous code structures it creates. The compiler tester can overwrite this behavior by complementing error generation rules with custom oracle generation rules.

We evaluate the accuracy of our technique by comparing results obtained by automated evaluation with results obtained by manual inspection. The evaluation shows a clear correspondence between our quality metric and human judgement.

Contributions. This paper provides the following contributions:

- A test input generation technique that generates edit scenarios specified by error generation rules.
- A full automatic technique to measure the quality of recovery results for generated edit scenarios.

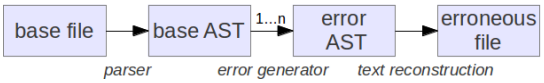

Figure 1: Error generators specify how to generate multiple erroneous files from a single base file.

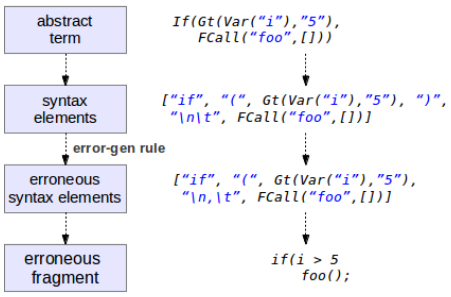

Figure 2: Error generation rules create erroneous constructs by modifying the syntax elements of correct constructs.

The input generation technique is discussed in Section 2, while automatic quality measurement is the topic of Section 3.

2. TEST GENERATION

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 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 1 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 [4]. 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 [1], a language based on the paradigm of strategic rewriting.

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

then 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 2 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.

```
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))
```

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.

Generators for Common Syntax Errors. The remaining challenge for test input generation is to implement error generators that cover common syntax errors. To gain insight into the kind of syntax errors that occur during interactive editing, we did an empirical research on collected edit data. We inspected 50 randomly selected files for three different languages, namely: Stratego [1], a transformation language; SDF [7], a declarative syntax definition language; and WebDSL [11], 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.

From this preliminary research we conclude that most syntax errors are editing related and generic for different languages. Only a few errors were related to error prone constructs in a particular language. We implemented reusable generators for the language generic edit scenarios. We leave it to an expert of a language to implement custom error generators for language specific errors. Below, we list the error types that we identified during our empirical research.

- *Incomplete constructs*, language constructs that miss one or more symbols at the suffix, e.g. an incomplete `for` loop `for (x = 1; x`.

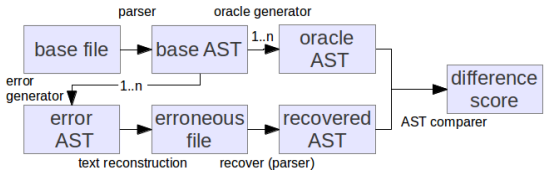

Figure 3: Automated evaluation of test outputs.

Figure 4: Oracle interpretations for erroneous constructs are given by placeholder terms (left) or constructed by custom oracle generation rules (right).

- *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 given language.
- *Combined errors*, two or more errors from the above mentioned categories, randomly distributed over the source file.

3. AUTOMATED QUALITY EVALUATION

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 3 shows the work flow for the evaluation framework that includes an oracle generation technique and an AST compare algorithm which are discussed below.

Oracle AST. 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 observe that the interpretation of an affected (erroneous) construct is often highly speculative and ambiguous. Therefore, by default, a placeholder oracle is constructed that serves as a wildcard for the interpretation of the affected term. Unaffected subterms of the affected term are checked against oracle interpretations contained in the placeholder. Figure 4 (left) provides an example. The placeholder term `PLACEHOLDER([Gt("i",5), FCall("foo", [])])` matches each term that has `Gt("i",5)` and `FCall("foo", [])` as subterms.

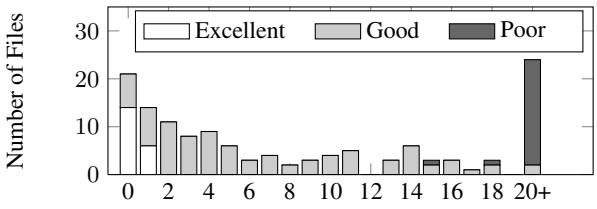

Figure 5: Correspondence of diff scores with human criteria.

In the given example, a more precise oracle interpretation is given by `If(Gt("i",5), FCall("foo", []))`. Compiler testers can choose to overwrite the default placeholder oracle by implementing an *oracle generation rule* that specifies how the oracle term is constructed from the original term. In the given example (Figure 4, right) the original term itself represents the optimal repair which is the common situation for errors that are constructed by deletion, insertion or replacement of a single literal token.

Difference Score. We calculate a difference score between the recovered AST and the oracle AST as a quantitative measurement of recovery quality. The difference score is calculated by an AST comparer that extends a change detection algorithm for trees [2] with support for traversing placeholder terms which do not contribute to the difference score. Oracle terms that do not appear in the recovered AST contribute the number of associated tokens, while oracle terms that show up in a wrong syntactic context increase the difference score by one. We base our quantification on concrete syntax (token count) rather than abstract syntax (term size), since concrete syntax better reflects human intuition, and because concrete syntax is independent from the particular abstract interpretation defined for a language.

Accuracy. The accuracy of a measurement technique indicates the proximity of measured values to 'true values'. We evaluated the accuracy of our quality measurement technique, using the Pennello and DeRemer criteria to determine the 'true values'. We generated 135 erroneous Java programs for which we plotted the measured difference scores against human determined values. Figure 5 shows the results. The figure clearly shows that low difference scores (smaller or equal to one) correlate to excellent recoveries, while high difference scores (larger than 20) correlate to poor recoveries.

4. RELATED WORK

Compiler Testing. Previous work on automated compiler testing primarily focuses on the generation of valid input programs that form positive test cases for the parser implementation [5, 12]. 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 [12]. 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.

Error Recovery Evaluation. Recovery techniques in literature use testsets that are either manually constructed [3, 8], or composed from practical data [9, 10]. According to our knowledge, test generation techniques have not yet been applied to recovery evaluation. Human criteria [9] and differential oracles [3, 8] 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.

5. CONCLUSION AND FUTURE WORK

This paper introduces a technique for fully automated recovery evaluation; the technique 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 and the evaluation of our techniques.

6. REFERENCES

[1] 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.

[2] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. SIGMOD Rec., 25:493–504, June 1996.

[3] 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.

[4] 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.

[5] J. Harm and R. Lämmel. Two-dimensional approximation coverage. Informatica (Slovenia), 24(3), 2000.

[6] 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.

[7] B. Luttik and E. Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, ASF+SDF 1997, Electronic Workshops in Computing, Berlin, 1997. Springer-Verlag.

[8] 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.

[9] T. J. Pennello and F. DeRemer. A forward move algorithm for LR error recovery. In POPL, pages 241–254. ACM, 1978.

[10] G. D. Ripley and F. C. Druseikis. A statistical analysis of syntax errors. Computer Languages, Systems & Structures, 3(4):227–240, 1978.

[11] E. Visser. WebDSL: A case study in domain-specific language engineering. In R. Lämmel, J. Visser, and J. Saraiva, editors, GTTSE, volume 5235 of LNCS, pages 291–373, Heidelberg, 2008. Springer.

[12] S. V. Zelenov and S. A. Zelenova. Generation of positive and negative tests for parsers. Programming and Computer Software, 31(6):310–320, 2005.