An Analysis of Rule Coverage as a Criterion in Generating Minimal Test Suites for Grammar-Based Software

Mark Hennessy
Computer Science Dept.
National University of Ireland, Maynooth,
Co. Kildare, Ireland
markh@cs.nuim.ie

James F. Power
Computer Science Dept.
National University of Ireland, Maynooth,
Co. Kildare, Ireland
jpower@cs.nuim.ie

ABSTRACT

The term *grammar-based software* describes software whose input can be specified by a context-free grammar. This grammar may occur explicitly in the software, in the form of an input specification to a parser generator, or implicitly, in the form of a hand-written parser, or other input-verification routines. Grammar-based software includes not only programming language compilers, but also tools for program analysis, reverse engineering, software metrics and documentation generation. Such tools often play a crucial role in automated software development, and ensuring their completeness and correctness is a vital prerequisite for their use.

In this paper we propose a strategy for the construction of test suites for grammar based software, and illustrate this strategy using the ISO C++ grammar. We use the concept of rule coverage as a pivot for the reduction of implementation-based and specification-based test suites, and demonstrate a significant decrease in the size of these suites. To demonstrate the validity of the approach, we use the reduced test suite to analyze three grammar-based tools for C++. We compare the effectiveness of the reduced test suite with the original suite in terms of code coverage and fault detection.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging; D.3.4 [**Programming Languages**]: Processors

General Terms
Measurement, Experimentation, Languages

Keywords
Software testing, grammar-based software, test suite reduction, rule coverage

ASE'05, November 7–11, 2005, Long Beach, California, USA.

1. INTRODUCTION

The term *grammar-based software* describes software whose input can be specified by a context-free grammar. Grammar-based software includes not only programming language compilers, but also tools for program analysis, reverse engineering, software metrics and documentation generation. Grammar-based applications are an important category of software in their own right, but also form the core of many tools that are fundamental to software engineering. Such tools often play a crucial role in automated software development, and ensuring their completeness and correctness is a vital prerequisite for their use.

The need for a structured and disciplined approach to the engineering of grammars and grammar-based software has previously been noted, and ensuring the quality of such software has been identified as a priority [14]. There have been some preliminary approaches to the integration of software engineering techniques, such as testing and metrics, into grammar-based software development [15, 19, 28]. However, there is still a considerable gap between theoretical developments in grammar engineering and their application to practical, large-scale grammars for modern programming languages.

A grammar may occur either explicitly or implicitly in grammar-based software. An *explicit* occurrence typically takes the form of input to a parser-generation tool such as *yacc* and, in this case, a direct correlation can often be achieved with the rules of the programming language grammar. An *implicit* occurrence may be in the form of a hand-written parser, where it is not easy to distinguish parsing code from the remainder of the tool. Further, many tools that require only partial information from the input make use of a *fuzzy* parser, where irrelevant parts of the input are ignored by the parsing routines. However, whether the grammar is explicitly defined or not, we expect to find a commonality that pervades all grammar-based systems: the acceptable input can be defined by a context-free grammar.

Since a grammar constitutes a formal specification of the input to grammar-based software, it is possible to utilize formal approaches to verifying such software. However, in the case of implicit grammar occurrences, less formal techniques such as testing become important. Even in the case of soft-

Accepted for the 20th IEEE/ACM International Conference on Automated Software Engineering
Long Beach, California, USA, November 7-11, 2005

ware based on explicit grammars, the scale and complexity of modern programming languages can cause considerable difficulties for theoretical approaches, such as those based on attribute grammars. Thus, in such situations issues associated with software testing, such as coverage, fault detection capability and test suite size come to the fore.

Test suites typically evolve in tandem with the software they test: as new features are added to the software, and new bugs are uncovered and fixed, relevant test cases are added to the suite. Since large test suites can impose a considerable overhead on regression testing, it is desirable to reduce the test suite size if overlaps or redundancies exist. The reduction is typically based on a *code* coverage criterion within the system under test. For grammar-based software however, we choose to use the *rule* coverage of the inputs to the system as the reduction criterion.

In this paper we describe an approach to the testing of grammar-based software, using the ISO C++ grammar as a case study. In Section 2 we outline some of the background relating to grammars, rule coverage and ISO C++. In Section 3 we describe the generation of two reduced test suites for ISO C++, and examine some of their code coverage properties. In Sections 4 and 5 we investigate the coverage and fault-detection capabilities of the reduced suites for three examples of C++ grammar based software. Section 6 discusses some of the threats to the validity of our experiment, and Section 7 reviews some of the related work in the area of test suite reduction. Section 8 concludes the paper.

2. BACKGROUND

In this section we briefly review the main concepts underlying grammars and grammar-based software, and discuss approaches to testing software based on the ISO C++ grammar.

2.1 Grammars and rule coverage

Formally, a grammar is a four-tuple (N, T, S, P) where N and T are disjoint sets of symbols known as non-terminals and terminals respectively, S is a distinguished element of N known as the start symbol, and P is a relation between elements of N and the union and concatenation of symbols from $(N \cup T)$, known as the production rules. The grammar rules may be read as rewrite rules, thus specifying alternative ways of re-writing the start symbol to a sequence of terminal symbols, known as the sentences of the language. In programming language terms, these sentences are programs that confirm to the grammar of the language.

The use of *rule coverage* as a criterion for testing grammars was introduced by Purdom [23]. A test case is said to cover a grammar rule if that rule is used at least once in deriving that test case. Since a non-terminal may have many alternative rules, rule coverage is similar to decision coverage at the code level in a traditional software testing context [24]. Purdom described an algorithm that systematically uses the grammar rules to generate valid sentences, such that each grammar rule is used at least once. Thus, the output of Purdom's algorithm is a test suite of grammatically correct programs that achieves 100% rule coverage. Purdom applied the technique to several small grammars, as well as a grammar for ALGOL, and it has since been applied to other languages including PL/1 and Pascal [2, 3].

However, there are at least three main difficulties in applying this technique to grammars for modern programming languages. First, many grammars over-specify the language, in that they admit constructs that are not syntactically valid. This approach can often make the grammar easier to understand, but means that extra constraints must be applied on the generation algorithm to weed out spurious programs. Second, context-sensitive information, such as the scope and type of variables, is not represented in the grammar, and thus has to be added to the programs using some other technique. While it is possible to define these extra constraints using multi-level grammars [3] or attribute grammars [7], it would be extremely difficult to apply this in full to a programming language like C++. Finally, if the grammar is ambiguous, such as the C++ grammar, there is no guarantee that the rules used in generating a sentence will be the same as those used in parsing that sentence.

2.2 The ISO C++ grammar

The C++ programming language was standardized by the International Standards Organization (ISO) in 1998, and further updated in 2003 [12]. Appendix A of the ISO standard contains a grammar for the language, with 123 non-terminals, 184 terminals and explicitly specifying 399 grammar rules. The notation used for the rules permits optional symbols in the productions; when these are replaced systematically[1] this rises to 479 rules using plain context-free notation. This grammar is significantly more complex than that for similar languages, such as C, Java and C# [22] and, unlike these languages, does not readily admit to standard parser construction tools based on LL or $LALR$ parsing algorithms.

Given the popularity of the C++ programming language, and its inherent complexity, it is vital that automated tools for processing the language be robust and accurate. Since the language is standardized, its syntax may be considered as fixed, at least in the short term, and thus a standardized set of test cases should be usable across all applications accepting ISO C++.

Each of the difficulties with Purdom's generation algorithm described in the previous section applies to the ISO C++ grammar. When applied to the C++ grammar, Purdom's algorithm generates 81 test cases, only 7 of which are valid C++ programs [18]. Thus, it is necessary to consider alternative techniques to achieving the same result, i.e. a small test suite that gives full rule coverage. In the remainder of this paper we consider reduction rather than generation techniques, and investigate the effect of reducing an existing test suite to a size comparable to that produced by Purdom's algorithm

[1]We replace optional occurrences of 40 grammar symbols with a pair of rules of the form $A\_opt \rightarrow A \mid \epsilon$

Accepted for the 20th IEEE/ACM International Conference on Automated Software Engineering
Long Beach, California, USA, November 7-11, 2005

2.3 Existing test suites for ISO C++

The popular, open-source GNU compiler collection *gcc* includes a robust C++ compiler that implements ISO C++. It also includes a large test suite for the various languages accepted by the compiler. The C++-specific part of the test suite distributed with *gcc* version 4.0.0 contains 5067 C++ programs. This is an *implementation-based* test suite, in that it was assembled to test various compiler features, and augmented as bugs were discovered or new features were added. Indeed, the four most recent versions of *gcc*, 3.2, 3.3, 3.4.0 and 4.0.0, released roughly at annual intervals, show an increase in the size of the C++ test suite of 8%, 10%, 18% and 12% respectively on the previous version.

Since the *gcc* test suite was developed to test the *gcc* C++ compiler, a number of its test cases relate to compiler internals or to back-end issues such as code generation. This, along with its size and continued expansion, makes the test suite unsuitable for use in its entirety with other grammar-based tools, and a suitable candidate for test suite reduction.

An alternative approach to gathering a test suite is to consult the language specification, and to attempt to create test cases that cover all aspects of the language. This *specification-based* approach is commonly used to test for compliance with the standard, to ensure that a compiler implements all features of the language. Examples for C++ include the CppETS suite developed as a benchmark suite for reverse engineering tools [26], and the *DDJ* suite, developed to test compliance of different compilers to the ISO standard [6, 17].

2.4 Goals of our study

Our study involves taking two existing test suites for ISO C++ and analyzing test suite reduction techniques based solely on grammar coverage. In the remainder of this paper we refer to these test suites as:

- T_{gcc}, the C++ programs from the test suite distributed as part of *gcc* version 4.0.0
- T_{ddj}, the DDJ test suite derived from the ISO standard, described in reference [17]

We examine whether reduced version of these test suites will be as effective as their larger counterparts; specifically, we investigate the following hypotheses:

Hypothesis 1: Reducing test suites based on rule coverage will not adversely affect code coverage when used to test grammar-based software.

Hypothesis 2: Reducing test suites based on rule coverage will not adversely affect the fault detection capability when used to test grammar-based software.

3. A REDUCED TEST SUITE FOR ISO C++

In this section we describe the construction of two reduced test suites for ISO C++. We discuss the implementation of rule coverage measurement using *gcc*, and we present the results of applying test suite reduction to the T_{gcc} and T_{ddj} test suites.

Test suite	Test Cases	LOC	Rule Coverage
T_{gcc}	5067	83919	95.3 %
T_{ddj}	440	4019	89.6 %

Table 1: Results of profiling the two test suites. *For each of the original test suites we show the size in terms of the number of test cases and lines of executable code, along with the percentage grammar rule coverage achieved by each.*

There are two main phases in reducing a test suite based on rule coverage. First a system must be constructed, capable of determining which grammar rules from the grammar were used during a parse. Second, a test suite reduction algorithm must be implemented and applied to the test suite.

3.1 Measuring Rule coverage

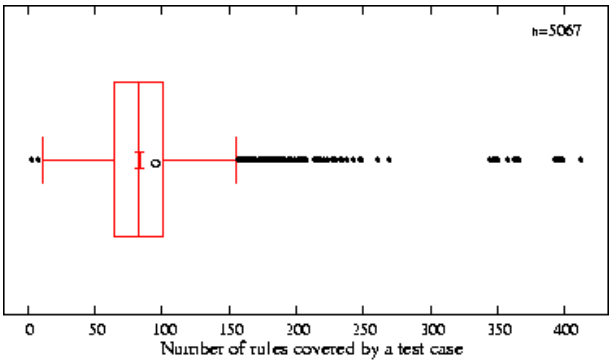
Since our reduction strategy is based on grammar rule coverage, it is essential to be able to determine which rules are used by each test case. Given that the C++ grammar is heavily context-sensitive, it is essential to use a fully-functional parser and front-end in order to correctly determine the rules that are used. In previous work, we had developed an instrumented version of GNU bison, and had used this with the parser in the version 3.0 of the *gcc* C++ compiler to produce an XML trace of the grammar rules used [11, 21]. However, while harnessing this explicit grammar facilitated profiling, the grammar in question had undergone considerable evolution, and it proved difficult to reconcile its rules directly with the ISO standard.

Fortunately, the C++ parser in *gcc* has been completely re-written as a hand-coded recursive descent parser, which corresponds closely to the grammar in the ISO standard. To track rule coverage, the parsing code in *gcc* version 4.0.0 was identified and profiling code was added to generate a log of grammar rules that were used as each input program was processed. Each test case in our two test suites was then profiled in this way using our modified *gcc*.

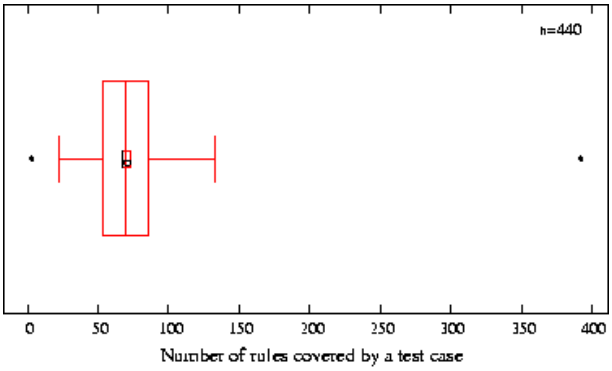
The results of profiling the two test suites is given in Table 1. As can be seen from column 4 of this table, neither test suite achieves 100% rule coverage, though both come close. Based on this analysis, both suites were augmented with extra test cases in order to achieve 100% rule coverage. These test cases were generated by slightly modifying Purdom's sentence generation algorithm so that it produced sentences guaranteeing coverage of just a single rule at a time. These generated test cases were simple enough so that they could then be modified by hand to ensure that they were correct C++ programs. In the remainder of this paper, we use T^+_{gcc} and T^+_{ddj} to denote the original test suites augmented with the test cases necessary to bring them to 100% rule coverage.

Since we are not aware of any existing work analyzing rule coverage for test suites, we present a summary of the rule coverage data for the T^+_{gcc} and T^+_{ddj} test suites in Figure 1. Figure 1 is a box plot showing the distribution of rule

Accepted for the 20th IEEE/ACM International Conference on Automated Software Engineering
Long Beach, California, USA, November 7-11, 2005



(a) T^+_{gcc} test suite



(b) T^+_{ddj} test suite

Figure 1: Distribution of rule coverage among the T^+_{gcc} and T^+_{ddj} test suites. *For each graph, the horizontal axis represents a count of the number of rules, and the box plot shows the distribution of rules covered.*

coverage among the 5067 test cases in T^+_{gcc} and the 440 test cases in T^+_{ddj}. As can be seen from this figure, the T^+_{gcc} has a slightly higher mean rule coverage, but also a larger spread of coverage. In fact, the mean coverage for T^+_{gcc} is 92 rules, against 70 rules for T^+_{ddj}, but the standard deviation is 55.6 for T^+_{gcc}, against 27.4 for T^+_{ddj}. The 25th and 75th quartile are 65 and 101 for T^+_{gcc}, and 53 and 86 for T^+_{ddj}.

3.2 Test suite reduction

The test suite reduction algorithm follows that of Jones *et al.* [13], and operates as follows:

1. For each test case in turn we compile a vector of length 479, with one entry corresponding to each C++ grammar rule, holding a 1 or 0, depending on whether or not that rule was used as the test case was parsed.
2. The vectors for all the test cases are placed together in a 2D array whose rows are indexed by the test cases and whose columns are indexed by grammar rule number.
3. If any *column* sums to one, then only one test-case covers the corresponding rule, and these test-cases are deemed essential and added to the reduced test suite. Whenever a test-case is added to the reduced suite, all of the vector entries corresponding to rules that are covered by this test-case are set to zero.
4. The *rows* are then summed to identify the test-case that contributes the most to rule coverage. This is added to the reduced set, the vector entries corresponding to the rules it covers are set to zero, and the process is repeated.

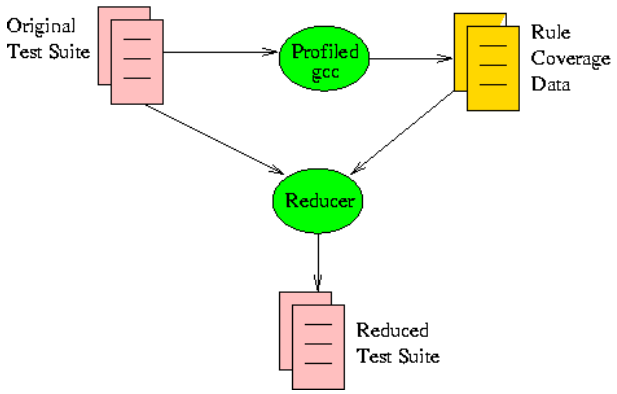


Figure 2: Overview of the test-suite reduction process. *An instrumented version of* gcc *is used to gather rule coverage data for each program in the test suite, and this data then used to reduce the test suite.*

Test Suite	Test Cases	Reduction	LOC	Reduction
R_{gcc}	26	99.6%	779	99.1%
R_{ddj}	22	95.0%	239	94.1%

Table 2: Percentage reduction achieved by the test suite reduction algorithm. *For each of the reduced test suites we show its size, in terms the number of test cases and total lines of executable code (LOC), and the percentage reduction compared to the corresponding original test suites.*

The test-suite reduction process is summarized in Figure 2.

It is worth noting that once all the essential test-cases have been removed, the problem of choosing the minimum test-set that covers the remaining rules is equivalent to the minimum cardinality hitting set, which is an intractable problem [4]. Hence the process will always be heuristic and in our case we choose to always use the test-case that contributes the most coverage even though it can be proved that this will not guarantee the smallest test suite.

The test suite reduction algorithm was applied to both of the existing test suites, generating two new suites which we refer to as R_{gcc} and R_{ddj}. By design, each of these suites achieves 100% rule coverage. The results of applying this algorithm are summarized in Table 2. For R_{gcc} there are 5040 *less* test-cases, a reduction of 99.6%. For R_{ddj}, there are

Accepted for the 20th IEEE/ACM International Conference on Automated Software Engineering
Long Beach, California, USA, November 7-11, 2005

System	Version	Source Files	LOC (≈)
Doc++	3.4.10	17	6,300
Keystone	0.0.6.9	52	6,500
Puma	0.9.3	141	18,200

Table 3: Systems Under Test. *For each of the three grammar-based applications used in our case study we show the version number used, the number of C++ source files, and the number of executable lines of code (LOC).*

418 less test-cases, a reduction of 95%. Both of these suites represent a dramatic reduction in size from the originals, and are comparable to the size of the test suites generated for C++ using Purdom's algorithm. However, all the test cases in these reduced suites are semantically correct C++ programs, unlike the test cases generated using Purdom's algorithm.

4. EMPIRICAL STUDY: CODE COVERAGE

In this section we investigate our first hypothesis, that reduction under rule coverage does not adversely affect code coverage. In order to do this we use three examples of grammar-based software that accept C++ programs as input. We refer to these as the systems under test (SUT), and they contain a mixture of implicit and explicit grammars.

Doc++ is an automatic documentation generator for C++ files [1]. There is no explicit grammar file and it must rely on code landmarks within an input C++ program to complete a fuzzy parse.

Keystone is a complete front-end to aid in the static analysis of ISO C++ programs [5]. It has an explicit grammar, modeled on the grammar in the ISO standard, which is used as input for the *btyacc* parser generator.

Puma is a library for parsing C++ that is used as the front-end for Aspect C++, the Aspect Oriented extension to C++ [27]. The parser code is hand written and thus has no explicit grammar.

Table 3 gives the version numbers and some basic size measures of these programs. In this and subsequent sections, all measurements in terms of lines of code (LOC) refer to *executable* lines of code, as reported by version 4.0.0 of the *gcov* utility.

4.1 Calculating code coverage

The first experiment was conducted in a highly structured manner and where possible automated scripts were used. The steps involved are outlined below.

1. Each of the three SUTs was built with compiler flags set to profile using *gcov*, a profiling tool that is part of the *gcc*.

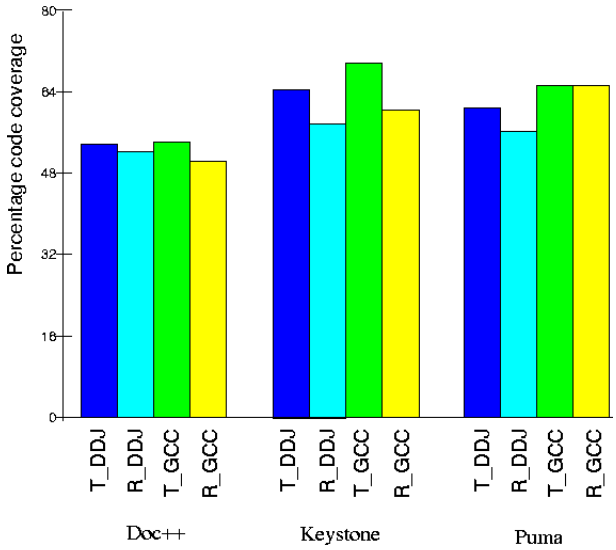


Figure 3: Code coverage results for each of the SUTs. *For each SUT we show the code coverage percentages for the total and reduced* DDJ *and* gcc *test suites.*

2. Each of the three SUTs were run using the full and reduced test suites as input. The output from each SUT for each test case was stored for use later in the mutation testing phase.

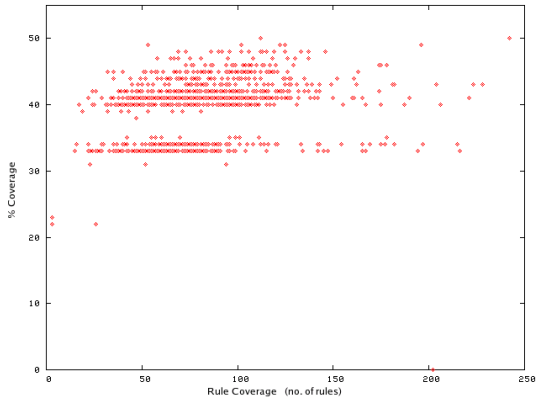
3. The coverage figures for each test suite were measured twice. For the first run, all test cases were passed through to give cumulative coverage figures for each of the test suites. For the second run, all individual test cases had their code coverage figures measured to determine if there was a correlation between rule coverage and code coverage.

4.2 Results

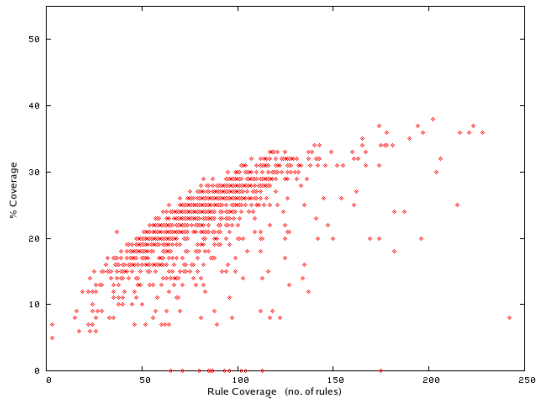
Figure 3 displays the summarized code coverage figures for each of the test suites. Each SUT is represented by four bars representing the code coverage percentage for the T^+_{ddj}, R_{ddj}, T^+_{gcc} and R_{gcc}. The first point to note in Figure 3 is the relatively low overall code coverage, even for the two larger suites, T^+_{ddj} and T^+_{gcc}. This is due to neither test suite being developed specifically for the SUT in question, and thus some of the back-end functions of each SUT remain untested.

However, the main result shown in Figure 3 is the relatively low decrease in the degree of code coverage between the total and reduced versions of the test suite, despite the considerable reduction in test suite size. The largest reduction in coverage for any SUT is that shown for *keystone*, where moving from T^+_{gcc} to R_{gcc} reduces the code coverage from 69.6% to 60.4%. Further investigation of the actual lines covered in *keystone* determined that the larger test suites contain *negative* test-cases not present in the reduced

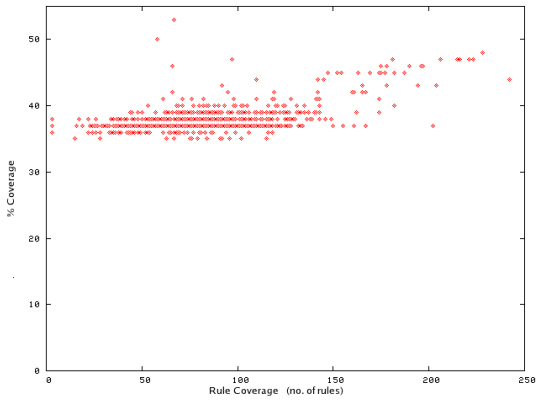
Accepted for the 20th IEEE/ACM International Conference on Automated Software Engineering
Long Beach, California, USA, November 7-11, 2005



(a) Doc++



(b) Keystone



(c) Puma

Figure 4: Rule coverage versus percentage line coverage for the three SUTs. *In each graph the horizontal axis measures the number of grammar rules covered, and the vertical axis represents percentage line coverage. Each point on the graph represents a single test case, with outliers covering more than 250 rules removed for clarity.*

test suites, and the difference is due to the proportion of error-recovery code being executed.

Figure 4 contains a scatter plot for each SUT, showing the relationship between rule coverage and code coverage. Here, each point on the scatter plot represents a single test case from the T^{+}_{gcc} test suite. As might be expected from the visual data in Figure 4, no strong linear correlation exists between rule coverage and code coverage. For *doc++* and *puma*, the graphs show that code coverage is largely invariant in the range 30%-50% for many of the test cases. *keystone* exhibits a very weak linear relationship, but again code coverage for individual test cases lie predominantly in the range 15%-35% These result demonstrate that the results shown in Figure 3 are not simply due to rule coverage acting as a surrogate measure for code coverage.

5. EMPIRICAL STUDY: FAULT DETECTION

In this section we investigate the usefulness of the reduced test suites in terms of detecting faults within a grammar-based system. Fault detection is the central focus of the testing process, and provides an external measure of the effectiveness of that process. Our second hypothesis under investigation aims to determine whether the reduced test suites can detect as many faults as their larger counterparts.

5.1 Mutation Testing

To investigate the fault detection capability of the reduced test suites, we seed the three SUTs from Section 4 with faults, and compare the effectiveness of the full and reduced test suites in detecting these faults. This approach is broadly similar to mutation testing, except that our goal here is to compare test suites, rather than to ensure full fault detection capability. In mutation testing, the source code of the SUT is mutated to introduce an error, and a test suite is evaluated on its ability to detect this error. If the test suite produces different output or behavior for the mutated version of the SUT then it has detected the error, and the mutant is said to be *killed*. Failure to kill all mutants indicates a deficiency in the test suite and, typically, new test cases are added to address this.

There are numerous ways of mutating a SUT but a study by Offutt *et al.* analyzed 22 different types of mutation, and identified a core set of five mutation types that were almost as effective as the entire set [20]. These five kinds of mutation are listed in Table 4. We applied these five kinds of mutation to our three SUTs automatically, using the following process:

1. Each SUT is run with each test case in the test suite as input, and the output for that test case is recorded.

2. The C++ code for each SUT is analyzed, and relevant expressions in the code are identified automatically as candidates for mutation.

3. The mutation operators are applied to each expression in turn, and the mutated expression, along with the position where it occurs in the program is output.

Accepted for the 20th IEEE/ACM International Conference on Automated Software Engineering
Long Beach, California, USA, November 7-11, 2005

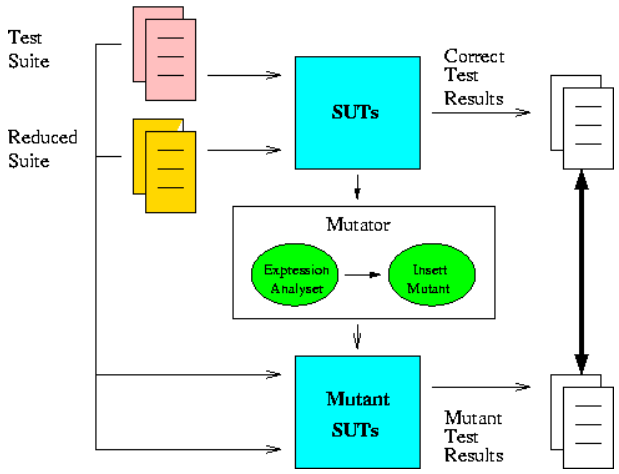


Figure 5: Overview of the fault-insertion process. *The mutator parses the source code of the SUT, identifies expressions, and outputs the relevant mutation operations. These are then applied one at a time, and the results are compared with the original test results.*

4. A simple script then applies each mutation to the relevant source file in the SUT, which is then re-built. The mutant SUT is first tested using the reduced test suite and, if the mutant is not killed, it is tested using the full version of the test suite.

The mutation generator consists of a scanner and fuzzy parser for C++ and is written in just under 1,000 lines of Python. For simplicity, the parser does not use a symbol table, and thus over-recognizes expressions in the code. While this does not impact the findings of the experiment, it does result in a high number of mutant programs being invalid, since they fail to compile. The effect of over-recognition of expressions resulted in between 60% and 90% of the generated mutations being invalid. Since the process of discarding an invalid mutant is simpler that writing an accurate parser for C++, this was deemed to be an acceptable level of invalidity.

Applying a single mutant involves changing a single source file, rebuilding the SUT, and, if the SUT compiles, running the SUT with each program from the test suite as input. If the output for any one of the test cases is different from the non-mutated version of the SUT, then that mutant is killed. The process to apply the mutations is not complex, but is quite time consuming. The total number of mutations generated ranged from about 50,000 for *keystone* to about 300,000 for *doc++*. Hence, not all possible mutations were applied to the SUTs. Instead, a random selection was made of the possible mutations, in quantities roughly approximating the size of the file to be mutated.

5.2 Results

Table 5 summarizes the results of the mutation process. The first data column shows the total number of valid mutant programs generated by the mutation process. The second data column shows the number of mutants killed by the reduced test suite, and the third data column shows the number of mutants missed by the reduced test suite, but killed by the total test suite. The final column of Table 5 gives the reduction in fault detection effectiveness, expressed as a percentage of the number of faults detected by the whole suite; that is:

$$Reduction = \frac{Missed}{(Killed + Missed)} * \frac{100}{1}$$

As can be seen from this column, the reduction in effectiveness is quite severe for all three applications, ranging from 59% for *keystone* to 100% for *puma*. While some diminution in the fault detection capability might have been expected due to the slightly lower code coverage results, the level shown in Table 5 cannot be considered acceptable for the programs involved. Thus we reject the second of our hypotheses, and conclude that reducing test suites based on rule coverage *does* adversely affect their fault detection capability.

Operator	Description
ABS	Absolute value insertion Replace an expression by 0, a positive value and a negative value
AOR	Arithmetic operator replacement Replace one of the binary arithmetic operators by each of the others
LCR	Logical connector replacement Replace one of the binary logical operators by each of the others
ROR	Relational operator replacement Replace one of the binary relational operators by each of the others
UOI	Unary operator insertion Insert a unary operator before the expression

Table 4: The five kinds of mutation operator applied to the SUTs. *All mutation types apply to expressions, and, when applied recursively to a single expression, can give rise to many mutated versions.*

	Mutants			
SUT	Total	Killed	Missed	Reduction
Doc++	523	18	490	96.5%
Keystone	310	123	177	59%
Puma	305	0	57	100%

Table 5: Results for fault detection within the SUTs. *For each SUT we show the total number of mutant programs generated, the number of mutants killed and missed by the reduced suite, and the percentage reduction in fault-detection effectiveness.*

Accepted for the 20th IEEE/ACM International Conference on Automated Software Engineering
Long Beach, California, USA, November 7-11, 2005

6. THREATS TO VALIDITY

In this section we discuss the threats to the internal and external validity of our study.

Threats to internal validity

The two test suites used, T_{gcc} and T_{ddj} may not be representative of test suites for C++ programs. While T_{gcc} is certainly among the most comprehensive implementation-based test suites available, it should be noted that commercial test suites for ISO compliance, such as those produced by Perennial or Plum Hall, can be an order of magnitude larger than either T_{gcc} or T_{ddj}.

Our reduction strategy was based solely on rule coverage, and it is possible that a combination of rule coverage with other kinds of coverage might yield better results. For example, one stronger form of rule coverage is *context-dependent* rule coverage [15], although our preliminary analysis suggests that there is little practical benefit to be gained from context-dependent coverage, at least in the context of the ISO C++ grammar [10].

The mutations applied to each SUT were selected randomly from the possible mutations, and it is possible that a more representative selection would produce different results. Further, it should be noted that these mutations are only a selection of those possible. For example, mutation operators can be defined that test object-oriented features (such as those defined by Ma *et al.* for Java [16]), that could yield different results.

Threats to external validity

Threats to external validity center on the choice of grammar used and the choice of SUTs. ISO C++ was chosen for our study as it represents a particularly challenging grammar for analysis purposes. It is thus possible that grammars for less complex languages may yield better results, although in the absence of a formal quantification of the link between grammars and the back-end code this is difficult to judge.

The three SUTs used in our experiments in Sections 4 and 5 were chosen as examples of medium-sized applications that took C++ code as input. As discussed in the introduction, grammar-based software includes many other kinds of application, and it would be useful to add examples of these to our study. Using larger applications as SUTs might also be useful, but it seems unlikely that they would yield better back-end coverage results than the ones presented here.

7. RELATED WORK

In this section we review some of the related work in the area of test suite reduction.

There are two central issues when performing test suite reduction. First, some criteria must be used to decide if a test case is redundant with respect to others in the suite. Typically, the criteria used are coverage based, although there are many different types of coverage criteria. Second, it is desirable that the reduced test suite have the same fault detection capability as the original.

Harrold *et al.* use coverage of definition-use pairs as their reduction criterion, and apply it to a set of seventeen C programs each containing less than 100 lines of source code [8]. The test suites for these programs range in size from 4 to 80 test cases, and a reduction of up to 60% in the size of the test suite is reported. They do not report on the fault detection capability of the reduced suites.

Wong et *al.* investigate the fault-detection effectiveness of reduced test suites for ten C programs, ranging in size from 90 to 842 executable lines of code [29]. They use block, decision and all-uses coverage as the reduction criterion, and with test suites for each application ranging in size from 156 to 997 test cases they achieve reductions in size in excess of 94%. To measure the effectiveness of the reduced test suite, between 12 and 30 faults were manually injected into each program, with an average reduction in effectiveness ranging from 4.44% to 9.20%.

In contrast, a more recent study by Rothermel *et al.* finds a significant decrease in the fault-detection capability of reduced test suites [25]. This study uses seven C programs, ranging in size from 138 to 516 lines of code, with substantial test suites, ranging in size from 1052 to 5542 test cases. Using edge-coverage as their criterion for reduction, and starting with randomly selected subsets of the test suites, they achieve a reduction in test suite size of between 87% and 95%. However, after manually injecting between 7 and 41 faults into the programs, they report a significant decrease in the fault detection capability of the reduced suites, in many cases by up to 100%.

Jones *et al.* use modified condition/decision coverage as the reduction criterion, and apply it to two software systems written in C [13]. The first system, TCAS consists of 138 executable lines of C code, and its test suite is reduced in size from 1608 test cases to 10 test cases. The second system, Space, consists of 6,218 executable lines of C code and its test suite of 13,585 test cases is reduced to 11 test cases. To evaluate the fault detection capability of the reduced suites, 41 faulty versions of TCAS and 35 faulty versions of Space were employed, with the average loss in fault detection being 44.4% and 10.2% respectively. The figures for coverage and fault detection are average figures, since 1,000 randomly sized selections of test cases were used as the starting point for the reduction process.

Heimdahl *et al.* apply test suite reduction to specification-based tests for a flight system consisting of 2564 lines of code in RSML^{-e} [9]. They generate and then reduce test suites using six different coverage criteria. With the original test suite sizes ranging in size from 115 to 537 test cases, they report an average reduction in test suite size of 80%. Using a random fault seeder they create 100 faulty versions of the program, and report a decrease of between 7% and 16% in fault detection capability for the reduced suites, which they deem unacceptable for their domain of interest.

The work presented in this paper has a number of sim-

Accepted for the 20th IEEE/ACM International Conference on Automated Software Engineering
Long Beach, California, USA, November 7-11, 2005

ilarities to the above work. First, we note a considerable decrease in test suite size as a result of the test suite reduction process, in line with the related work. Second, our work echoes the results of Rothermel in particular, in finding that the reduced test suites are comparatively poor at fault detection.

However, the work presented in this paper also differs from the above related work in a number of ways. First, we are not aware of any other test suite analysis or reduction based on rule coverage, as opposed to code coverage measures. Second, our test suites are considerably larger than those of Harrold or Wong, comparable in size to those of Rothermel, and slightly smaller than those used by Jones. Third, the three C++ programs used as our SUTs are considerably larger than those C programs used by the above approaches, except for the Space system used by Jones. Finally, all the above approaches that use mutation testing with manually generated mutants use considerably fewer mutants than those reported here in Table 5.

8. CONCLUSION

In this paper we have tested the practical use of rule coverage as a criterion in the reduction of test suites for grammar-based software. We have taken two existing test suites for ISO C++, and applied a reduction strategy based on rule coverage. To estimate the effect of this reduction we have studied three grammar-based applications, and investigated the code coverage and fault detection capabilities of the reduced test suite.

The main findings of our work are:

1. Test suite reduction based on rule coverage provides a significant reduction in the number of test-cases, and thus in the testing overhead. The size of the reduction is comparable to strategies that use other coverage criteria, and produces a test suite that is comparable in size to that generated by Purdom's algorithm, with the added advantage of semantic correctness.

2. We have demonstrated for three grammar-based applications that, while there is no formal correlation between rule coverage and code coverage, the reduced test suites do not significantly reduce the level of code coverage. While this was to be expected for the code purely relating to parsing, it is notable that it also holds for other parts of the applications that were tested.

3. However, our mutation testing results indicate that the reduced test suite does not adequately preserve fault detection capability. In this, the results are consistent with those of Rothermel *et al.* [25], although the reduction in fault detection capability is more severe for rule based coverage.

Despite the encouraging results in relation to the preservation of code coverage for the reduced suites, the failure in the rate of fault detection must be considered a significantly negative finding.

We identify the novel contributions of this paper as:

- The use of standardized test suited for grammar-based applications. This differs from standard testing techniques where, typically, test suites are designed anew for each individual application.

- A rule coverage analysis of two significant test suites for ISO C++, based on results from profiling the parser from the *gcc* C++ compiler.

- The implementation and analysis of automated test suite reduction using rule coverage as the criterion.

- An analysis of the reduced test suite in terms of code coverage and fault detection, and its application to three instances of grammar-based software.

In future work we intend to build on the positive aspects of this study; we aim to extend the reduced test-suites to incorporate negative test-cases that fully exercise of the code within a grammar-based system. Furthermore we plan to investigate the additional criteria that need to be used in order that test suite reduction will not cause such a significant loss in fault detection capability.

9. ACKNOWLEDGMENTS

This work is supported in part by the Embark Initiative, operated by the Irish Research Council for Science, Engineering and Technology (IRCSET).

10. REFERENCES

[1] D. Acostachioaie. Doc++: Open source - open science - open systems. *Circles Electronic Magazine*, (36), 2000.

[2] F. Bazzichi and I. Spadafora. An automatic generator for compiler testing. *IEEE Trans. Softw. Eng.*, 8(4):343–353, 1982.

[3] A. Celentano, S. Crespi-Reghizzi, P. D. Vigna, C. Ghezzi, G. Granata, and F. Savoretti. Compiler testing using a sentence generator. *Software: Practice and Experience*, 10(11):897–918, November 1980.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[5] T. H. Gibbs, B. A. Malloy, and J. F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software: Practice and Experience*, 33(1):19–39, January 2003.

[6] T. H. Gibbs, B. A. Malloy, and J. F. Power. Progression toward conformance of C++ language compilers. *Dr. Dobbs Journal*, 28(11):54–60, September 2003.

[7] J. Harm and R. Lämmel. Two-dimensional Approximation Coverage. *Informatica*, 24(3), 2000.

[8] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

Accepted for the 20th IEEE/ACM International Conference on Automated Software Engineering
Long Beach, California, USA, November 7-11, 2005

[9] M. P. E. Heimdahl and D. George. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *19th IEEE International Conference on Automated Software Engineering*, pages 176–185, Linz, Austria, 20-25 September 2004.

[10] M. Hennessy and J. F. Power. Generation strategies for test-suites of grammar-based software. Technical Report NUIM-CS-TR-2005-02, Department of Computer Science, National Univesity of Ireland, Maynooth, April 13 2005.

[11] M. Hennessy, J. F. Power, and B. A. Malloy. gccxfront:exploiting gcc as a front end for program comprehension via XML/XSL. *11th International Workshop on Program Comprehension*, May 9 - 11 2003.

[12] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:2003(E). American National Standards Institute, second edition, October 15 2003.

[13] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Trans. Softw. Eng.*, 29(3):195–210, 2003.

[14] P. Klint, R. Lämmel, and C. Verhoef. Towards an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology*, May 2005. To appear; Online since July 2003, 47 pages.

[15] R. Lämmel. Grammar Testing. In *Fundamental Approaches to Software Engineering*, volume 2029 of *LNCS*, pages 201–216. Springer Verlag, 2001.

[16] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.

[17] B. A. Malloy, S. A. Linde, E. B. Duffy, and J. F. Power. Testing C++ compilers for ISO language conformance. *Dr. Dobbs Journal*, 27(6):71–78, June 2002.

[18] B. A. Malloy and J. F. Power. An interpretation of Purdom's algorithm for automatic generation of test cases. In *1st Annual International Conference on Computer and Information Science*, Orlando, FL., 2001.

[19] B. A. Malloy, J. F. Power, and J. T. Waldron. Applying software engineering techniques to parser design. In *Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 75–82, Port Elizabeth, South Africa, September 16-18 2002.

[20] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, April 1996.

[21] J. F. Power and B. A. Malloy. Program annotation in XML: a parser-based approach. In *Working Conference on Reverse Engineering*, pages 190–198, Virginia, USA, October 28 - November 1 2002.

[22] J. F. Power and B. A. Malloy. A metrics suite for grammar-based software. *Software Maintenance and Evolution: Research and Practice*, 16(6):405–426, Nov/Dec 2004.

[23] P. Purdom. A sentance generator for testing parsers. *BIT*, 12(3):366–375, 1972.

[24] M. Roper. *Software Testing*. McGraw-Hill, 1994.

[25] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *International Conference on Software Maintenance*, pages 34–43, Maryland, USA, November 16-19 1998.

[26] S. E. Sim, R. C. Holt, and S. Easterbrook. On using a benchmark to evaluate C++ extractors. In *10th International Workshop on Program Comprehension*, Paris, France, June 2002.

[27] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *40th International Conference on Technology of Object-Oriented Languages and Systems*, pages 53–60, Sydney, Australia, February 18-21 2002.

[28] M. van den Brand, A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *International Workshop on Program Comprehension*, Ischia, Italy, June 24-26 1998.

[29] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. *Software: Practice and Experience*, 28(4):347–369, April 10 1998.

Accepted for the 20th IEEE/ACM International Conference on Automated Software Engineering
Long Beach, California, USA, November 7-11, 2005