Empir Software Eng (2008) 13:343–368
DOI 10.1007/s10664-008-9067-7

Analysing the effectiveness of rule-coverage as a reduction criterion for test suites of grammar-based software

Mark Hennessy · James F. Power

Published online: 7 May 2008
© Springer Science + Business Media, LLC 2008
Editor: Gregg Rothermel

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. Grammar-based software includes not only programming language compilers, but also tools for program analysis, reverse engineering, software metrics and documentation generation. Hence, 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 grammar-rule coverage as a pivot for the reduction of an implementation-based test suite, and demonstrate a significant decrease in the size of this suite. The effectiveness of this reduced test suite is compared to the original test suite with respect to code coverage and more importantly, fault detection. This work greatly expands upon previous work in this area and utilises large scale mutation testing to compare the effectiveness of grammar-rule coverage to that of statement coverage as a reduction criterion for test suites of grammar-based software. This work finds that when grammar rule coverage is used as the sole criterion for reducing test suites of grammar based software, the fault detection capability of that reduced test suite is greatly diminished when compared to other coverage criteria such as statement coverage.

Keywords Software testing · Grammar-based software · Test suite reduction · Rule coverage · Mutation testing

M. Hennessy (✉) · J. F. Power
Computer Science Department, National University of Ireland,
Maynooth, Co. Kildare, Ireland
e-mail: markh@cs.nuim.ie

J. F. Power
e-mail: jpower@cs.nuim.ie

1 Introduction

Grammar-based software in the form of source code analysis tools, reverse-engineering tools, refactoring tools and documentation generation tools provide a crucial level of automated support during the software development lifecycle. Given their important role, it is therefore crucial that these systems are robust and reliable, hence the need for test suites that expose and test all facets of their underlying source code. The underlying commonality that pervades all grammar-based systems is the fact that their inputs can be specified using a context-free grammar. Given this fact, our work investigates the effectiveness of grammar-rule coverage as an adequacy criterion for test suites of grammar-based software.

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 (Koppler 1997). However, whether the grammar is explicitly defined or not, we expect 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 utilise formal approaches to verifying such software (Lämmel 2001). However, in the case of implicit grammar occurrences, less formal techniques such as testing become important. Even in the case of software 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 (Harrold et al. 1993). For grammar-based software however, we choose to use *rule* coverage based upon the test suite's coverage of the underlying grammar's rules 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 the ISO C++ grammar. The main hypothesis under investigation are outlined in Section 3. In Section 4 we describe the generation of a reduced test suite for ISO C++, and examine some of its code coverage properties. In Sections 5 and 6 we investigate the coverage and fault-detection capabilities of the reduced suites for three examples of C++ grammar-based software. Section 7 discusses some of the threats to the validity of our experiment, and Section 8 reviews some of the related work in the area of test suite reduction. Section 9 concludes the paper.

2 Background

In this section we give an overview of the main concepts underlying grammars and rule coverage. We also describe the ISO C++ grammar and the current test suites that are available for ISO C++.

2.1 Grammars

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. Some grammars further define another special symbol, ε, to represent the empty string. The ε symbol can only occur on the right-hand side of a rule.

The grammar's production 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 conform to the grammar of the language.

2.2 Grammar-rule Coverage

The use of *rule coverage* as a criterion for testing grammars was introduced by Purdom (Purdom 1972). 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 (Roper 1994). Purdom described an algorithm that systematically uses the grammar rules to generate valid sentences, so 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 (Bazzichi and Spadafora 1982; Celentano et al. 1980).

However, there are at least three main difficulties in applying this technique to grammars for modern programming languages (Malloy and Power 2001). 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 to 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 (Celentano et al. 1980) or attribute grammars (Harm and Lämmel 2000), it would be extremely difficult to apply this in full to a programming language like C++. Finally, if the grammar contains ambiguities, 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.

Figure 1 gives an example of grammar production rules and some corresponding sentences. The upper-half of Fig. 1 shows seven grammar rules taken from Appendix A of the ISO C++ standard which define three non-terminals, `class-name`, `class-specifier` and `class-head`. The lower-half of Fig. 1 shows three sentences corresponding to the non-terminal `class-specifier`. These three pieces of C++ code have been automatically generated by traversing the grammar rules, and achieve 100% rule coverage of the seven grammar rules. However, as can be seen from the third code fragment in Fig. 1, it is not always possible to generate semantically correct test-cases directly from the grammar specification.

2.3 The ISO C++ Grammar

The C++ programming language was standardised by the International Standards Organization (ISO) in 1998 (ISO/IEC JTC 1 1998), and further updated in 2003 (ISO/IEC JTC 1 2003). 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 by expanding optional grammar rules, this rises to 479 rules using plain context-free notation. This grammar is significantly more complex than that for other popular programming languages such as C and Java (Power and Malloy 2004) and constructing a parser for this grammar using existing parsing algorithms is quite difficult.

Given the popularity of the C++ programming language, and its inherent complexity, it is vital that automated tools such as program analysis tools, reverse engineering tools and metrics tools that process the language be robust and accurate. The commonality that pervades these tools is that their input can be specified as a context-free grammar and collectively these tools can be described as *grammarware* (Paul Klint et al. 2005). Since the language is standardized, its syntax may be considered as fixed in the short term, thus a standardized set of test-cases should be usable across all applications accepting ISO C++.

1	class-specifier:	class-head { member-specification$_{opt}$ }
2	class-head:	class-key identifier$_{opt}$ base-clause$_{opt}$
3		class-key nested-name-specifier identifier base-clause$_{opt}$
4		class-key nested-name-specifier$_{opt}$ template-id base-clause$_{opt}$
5	class-key:	`class`
6		`struct`
7		`union`

	Test-Cases	
1	`class` A { }	Rules: 5, 2, 1
2	`struct` ::B : A { }	Rules: 6, 3, 1
3	`union` W::X< Y> : Z { }	Rules: 7, 4, 1

Fig. 1 A fragment of the C++ grammar and three resulting test-cases. *The upper half of this figure shows seven grammar rules from the ISO C++ standard. The lower-half shows three test-cases, pieces of C++ code that achieve* 100% *coverage of the seven grammar rules*

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 (Malloy and Power 2001). 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

2.4 Test Suites for ISO C++

The popular, open-source GNU compiler collection *gcc* includes a large test suite utilised by the DejaGnu testing framework to test 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.

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 (Sim et al. 2002), the *DDJ* suite, developed to test compliance of different compilers to the ISO standard (Gibbs et al. 2003b; Malloy et al. 2002) and the commercial test suites from Plum-Hall and Perennial (Plum Hall test suites; Perennial test suite for ISO C++).

3 Goals of this Work

The goal of this work is to investigate the effectiveness of grammar-rule coverage as a criterion when reducing test suites for grammar-based software. As all grammar-based systems must perform some analysis and processing on the syntactic structure of their inputs, this work determines the effectiveness of front-end coverage.

Our study involves taking the test suite distributed with version 4.0.0 of the GNU compiler collection, *gcc*, and analysing test suite reduction techniques based solely on grammar-rule coverage. We refer to this test suite throughout the remainder of this paper as T_{gcc}. The code coverage offered by T_{gcc} is also used to define the front-end for each system we test in this paper, hence all of the experiments within this paper are concerned only with the areas of code covered by each of the test-cases within T_{gcc}.

In this study, we examine whether a reduced version of T_{gcc} will be as effective as its larger counterpart; 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.

Hypothesis 3: The positive properties exhibited by the first two hypotheses are a direct result of maintaining grammar-rule coverage while reducing the test suite.

4 A Reduced Test Suite for ISO C++

In this section we describe the construction of a reduced test suite 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} test suite.

There are two main phases in reducing a test suite based on rule coverage. First a system capable of determining which grammar-rules are present within each of the test-cases in T_{gcc} must be constructed. Second, a test suite reduction algorithm must be implemented and applied to the test suite.

4.1 Measuring Rule Coverage

Since our reduction strategy is based on grammar-rule coverage, it is necessary 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. Previous work 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 (Hennessy et al. 2003; Power and Malloy 2002). 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 test suite was then profiled in this way using our modified *gcc*.

The results of profiling T_{gcc} are given in Table 1. As can be seen from column five of this table, the test suite does not achieve 100% rule coverage, though it comes close. The second column in Table 1 lists the number of *positive* test-cases in the test suite. The T_{gcc} test suite also includes negative test cases which should be rejected by the compiler, but we made the decision to exclude these from our experiment since they make little contribution to rule coverage, and thus do not impact the results in the rest of the section.

Based on the rule-coverage analysis, the suite was 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

Table 1 Results of profiling the normal and reduced test suites

Test suite	Test cases	Positive test-cases	LOC	Rule coverage (%)
T_{gcc}	5067	4195	95946	95.3
T^{+}_{gcc}	5068	4196	95979	100
R_{gcc}	40	40	1210	100

For each of three test suites we show the size in terms of the number of test-cases and lines of executable C++ code (LOC), along with the percentage grammar rule coverage achieved by each. The positive test-cases are the number of test-cases that are positive with respect to keystone. T^{+}_{gcc} is the T_{gcc} suite augmented with an extra test-case to ensure 100% coverage of the grammar rules

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} to denote the set of positive test-cases augmented to bring it to 100% rule coverage.

The T^{+}_{gcc} test suite consists of over 4,000 test-cases that show a huge variation in their level of rule coverage. The box-plot in Fig. 2 illustrates the variation in rule coverage between the individual test-cases in the T^{+}_{gcc} suite. The box-plot (also called a “box-and-whiskers plot”) summarises the distribution of rule coverage for the set of test-cases. The box in the middle represents the quartiles, and is width is thus the inter-quartile range. The leftmost whisker is 1.5 times the inter-quartile range below the first quartile, and the rightmost whisker is 1.5 times the inter-quartile range above the third quartile. The dots outside these whiskers indicate outliers. As can be seen from this figure, the mean coverage for a test-case in T^{+}_{gcc} is 92 rules, the standard deviation is 55.6 while the 25th and 75th quartile are 65 and 101 rules covered respectively. The wide coverage range for reflects on the broad spread of test-cases designed to test a myriad of compiler features not explicitly related to the front-end of *gcc*.

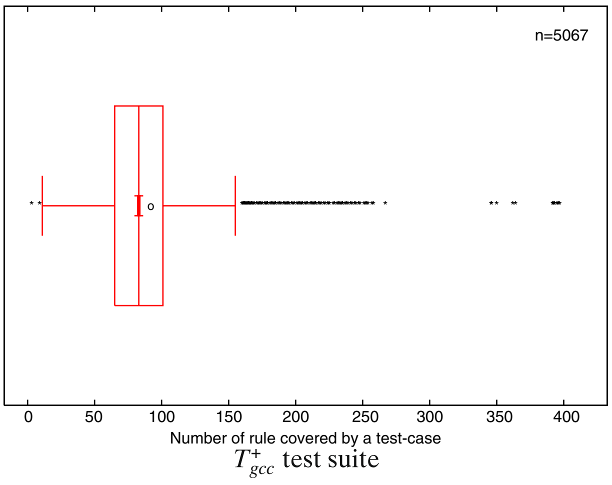


Fig. 2 Distribution of rule coverage among the T^{+}_{gcc} test suite. *This box plot shows the distribution of the rules covered across each of the* 5067 *test cases in the* T^{+}_{gcc} *test suite. As can be seen, the majority of test cases are covering between* 50 *and* 100 *of the grammar rules*

4.2 Test Suite Reduction

The test suite reduction algorithm (Harrold et al. 1993) operates as follows:

1. Taking each positive 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.

The test suite reduction process is illustrated in Fig. 3.

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

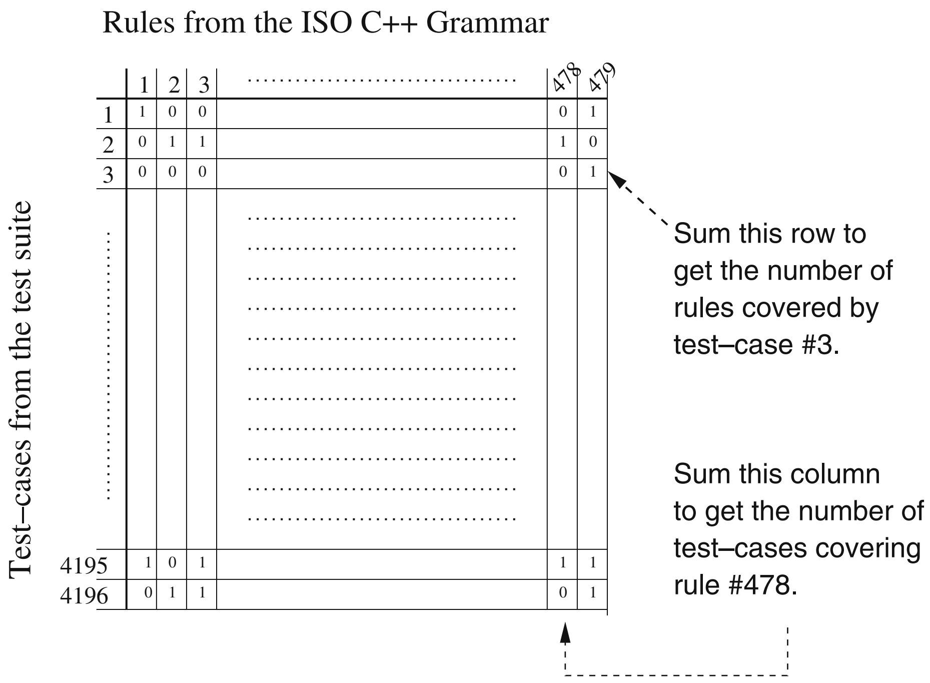

Rules from the ISO C++ Grammar
1
2
3
478
479
Test–cases from the test suite
1
2
3
4195
4196
1 0 0 0 1
0 1 1 1 0
0 0 0 0 1
1 0 1 1 1
0 1 1 0 1
Sum this row to get the number of rules covered by test–case #3.
Sum this column to get the number of test–cases covering rule #478.

Fig. 3 An illustration of the test-suite reduction process. *In this 2-D array, each row represents rule coverage for a test case and each column represents the test cases covering a particular rule. Each cell contains a* 1 *or* 0 *representing a yes/no for coverage*

(Garey and Johnson 1979). 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 T^{+}_{gcc} generating a new suite which we refer to as R_{gcc}. By design, this suite achieves 100% rule coverage. The results of applying this algorithm are summarised in Table 1. For R_{gcc} there are 4155 *fewer* test-cases, a reduction of 99%. This represents a dramatic reduction in size from the original, and is comparable to the size of the test suites generated for C++ using Purdom's algorithm. While our approach is based on test suite *reduction* rather than *generation*, it does have the advantage over test suites generated using Purdom's algorithm that all of the resulting test-cases are semantically correct. The caveat, of course, is that we must start with a larger suite of semantically correct programs.

5 Empirical Study: Code Coverage

In this section we investigate our first hypothesis, that reduction under grammar-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 (Acostachioaie 2000). 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 (Gibbs et al. 2003a). It has an explicit grammar, modelled 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 AspectC, an Aspect Oriented extension for C++ (Spinczyk et al. 2002). The parser code is hand written and thus has no explicit grammar.

Table 2 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 C++ code, as reported by version 4.0.0 of the gcov utility, and is the maximum number of LOC in the system that could be covered by any test suite. In what follows, all code coverage figures are expressed as a percentage of the number of LOC.

Table 2 Systems under test

System	Version	Source files	LOC (≈)
Doc++	3.4.10	17	5,561
Keystone	0.2.3	52	6,879
Puma	1.0pre3	136	14,438

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 LOC

5.1 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*.
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. Two sets of coverage figures were recorded for each test suite. The code coverage for each individual test-case was measured to determine if there was a correlation between rule coverage and code coverage, and the cumulative coverage was measured to evaluate the whole test suite.

The code coverage results obtained from these steps are summarised in Figs. 4, 5 and 6, and are discussed in the following subsection.

5.2 Results

Figure 4 displays the summarised code coverage figures for each of the test suites. Each of the three SUTs is represented by a pair of bars, where the first bar represents coverage for the T^{+}_{gcc} suite and the second bar represents coverage for the R_{gcc} suite. The green (lower) area in each bar represents the number of LOC covered by the test suite, and the red (upper) area in each bar represents the number of executable LOC not covered. Thus, the whole bar represents the total number of executable LOC in the SUT.

The low code coverage for *doc++* can be attributed to the fact that it is designed to process C, Java and IDL as well as C++. In addition it has a number of different output formats, all triggered by various command line flags, only one of which was exercised in our study.

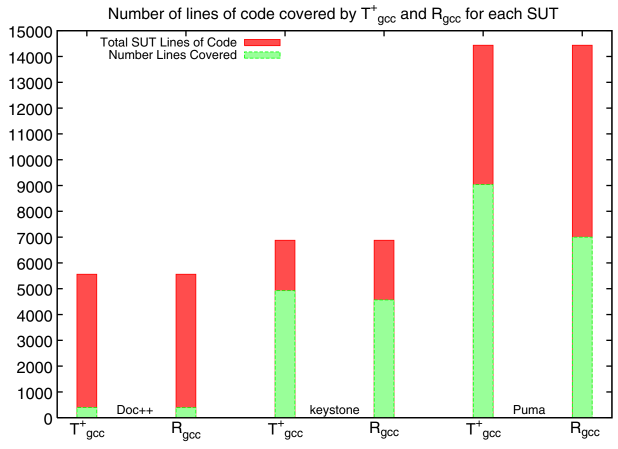


Fig. 4 Code coverage results for each of the SUTs. *For each SUT we show the code covered by the large and reduced test suite alongside the total amount of code in the SUT*

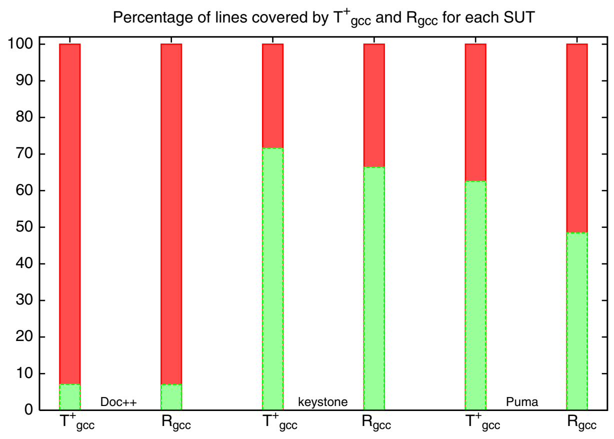


Fig. 5 Code coverage results for each of the SUTs. *For each SUT we show the percentage of code covered for each test suite where* 100% *represents all of the executable LOC*

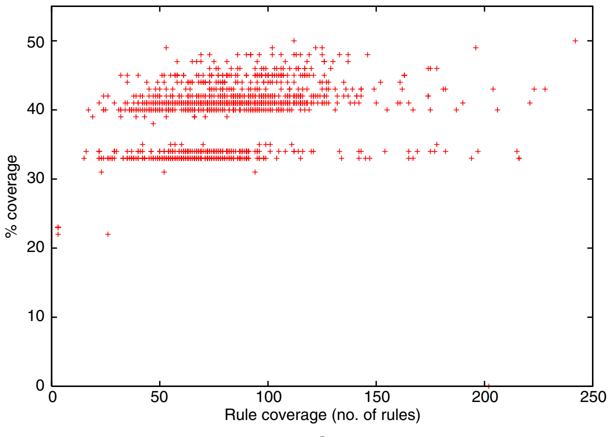
Figure 5 is based on the same data as Fig. 4, but is expressed in percentage terms, thus normalising the difference in size between the three SUTs. Here , 100% coverage represents all of the LOC shown in the rightmost column of Table 2. The first point to note in Fig. 5 is the lack of complete code coverage, even for the larger suite, T^{+}_{gcc}. This is due to the fact that not all of the various command line options were exercised for each SUT such as ASG generation in *keystone* and advanced symbol table output in Puma. In addition to these options not being applied, the omission of negative test-cases results in error handling and recovery code not being executed.

However, the main result shown in Fig. 5 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. This small decrease in code coverage is also observed with another test suite that was used in a similar study (Hennessy and Power 2005a). This appears to be an initial success for the technique; however, it should be noted that test suite reduction based on low initial coverage results (particularly for *doc*++) may not be generalisable to test suites with better coverage results.

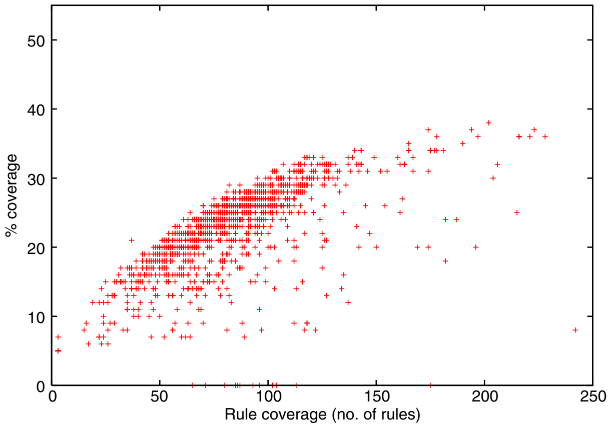
In the results displayed in Fig. 5 the largest reduction in coverage for any SUT is that shown for *puma*, where moving from T^{+}_{gcc} to R_{gcc} reduces the code coverage from 63% to just under 50%. This large reduction is not visible in either *doc*++ or *keystone* and given the 99% reduction in the size of the test suite is deemed acceptable for the purposes of our study.

Figure 6 contains a scatter plot for each SUT, showing the relationship between rule coverage and statement coverage. In these figures 100% coverage relates to the maximum number of LOC reported by gcov that could be covered by *any* test suite. 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 Fig. 6, no strong linear correlation exists between rule coverage and code coverage.

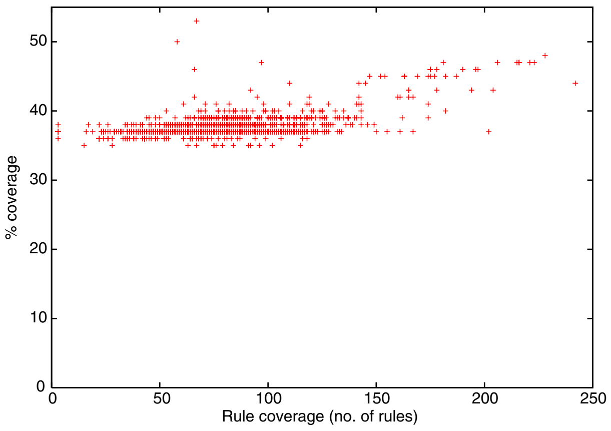
For *doc*++ and *puma* displayed in Fig. 6a and c respectively, the graphs show that code coverage is largely invariant with coverage having a threshold effect in the range of 30%–50% for many of the test-cases. That is, for both *doc*++ and *puma*,



a



b



c

Fig. 6 Rule coverage versus percentage statement 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.* ***a*** *Doc++.* ***b*** *Keystone.* **c** *Puma*

a significant proportion of the code is executed, irrespective of the level of rule coverage. Figure 6b shows that *keystone* exhibits a very weak linear relationship, but again code coverage for individual test-cases lie predominantly in the range 15%–35%.

A linear correlation would have implied that rule coverage increased in proportion to code coverage. These results demonstrate that the results shown in Fig. 5 are not simply due to rule coverage acting as a surrogate measure for code coverage.

6 Empirical Study: Fault Detection

In this section we investigate the usefulness of the reduced test suite 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 (Fig. 7).

6.1 Mutation Testing

To investigate the fault detection capability of the reduced test suite, we seed the three SUTs from Section 5 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 behaviour for the mutated

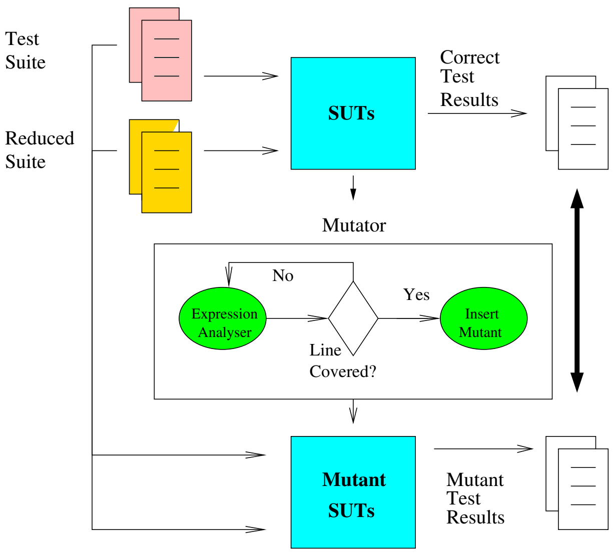


Fig. 7 Overview of the fault-insertion process. *The mutator parses the source code of the SUT, identifies expressions, and outputs the relevant mutation operations. These are then applied one at a time, and the results are compared with the original test results*

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. It should be noted that our earlier decision to exclude negative test cases is not typical for mutation testing, where both positive and negative test cases can be used to kill mutants. Including negative test cases at this point may yield different results to those presented in this section, but this is not explored further in this paper.

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 (Offutt et al. 1996). These five kinds of mutation are listed in Table 3. 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 correct output for that test-case is recorded to be utilised in step 5.
2. The code coverage of the R_{gcc} suite was recorded using the gcov utility for each SUT.
3. The C++ code for each SUT is analysed, and relevant expressions in the code are identified automatically as candidates for mutation.
4. The mutation operators are applied to each expression in turn if and only if gcov reports that the R_{gcc} suite has covered the current line, and the mutated expression, along with the position where it occurs in the program is output.
5. 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. Since the reduced suite is a subset of the full suite, any mutant that is killed by the reduced suite is guaranteed to also be killed by the full 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-recognises 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

Table 3 The five kinds of mutation operator applied to the SUTs

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

All mutation types apply to expressions, and, when applied recursively to a single expression, can give rise to many mutated versions 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 checking to see if the current line in the current source file is covered by R_{gcc} and then changing that single source file by inserting the mutant, 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 varies in any way from the original, then the mutant is killed. The comparison of the outputs was quite fine grained, with the output alongside the file size of the output being compared to the original non-mutated output. The file size comparison was a simple short-circuit evaluation, as some of the mutants caused outputs in the hundreds of MBs, and the comparison was sped up by ignoring a textual comparison if the file sizes differed.

Mutants were only seeded in the areas of code that were covered to ensure that principles of *reachability*, *infection* and *propagation* were adhered to (Offutt et al. 2001). Reachability means that the line with the mutant must be capable of being executed by the test-case in question. Infection and propagation are related in that if a mutated line gets infected then the error must exhibit itself all throughout the lifetime of the program execution and propagate to the output stage where the fault can be detected externally. Thus it is quite possible that an error could be reached and infected but may not be detected. In this case strong mutation testing would be required to give a more definitive measure of the kill rate but given the automated nature of our testing and the large size of our SUTs, the automatic addition of code to the SUT to track the internal state of a mutated variable would add unnecessary complexity to the experiment.

The seeding of mutants in this experiment in lines covered solely by R_{gcc} was to ensure that even allowing for the poorer coverage, the kill rate is as good as the larger suite in the areas it does cover. According to a previous study, the application of mutants throughout the whole SUT irrespective of coverage resulted in a very weak kill rate (Hennessy and Power 2005a). Thus the purpose of this study was to factor in the coverage of the reduced suite to gain a clearer view of its kill rate.

6.2 Results

Table 4 summarises the results of the mutation process for both test suites. The first data column shows the total number of programs containing mutants generated by

Table 4 Results for fault detection within the SUTs

SUT	Mutants			
	Total applied	Killed *gcc*	Missed *gcc*	Reduction *gcc*
Doc++	893	509	0	0
Keystone	5434	3376	0	0
Puma	15308	4575	0	0

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 the mutation process for LOC covered by R_{gcc}. 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 4 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 Table 4, the minimum test suite appears to be just as effective as its larger counterpart. The R_{gcc} suite does not miss any seeded mutants and the fifth column of the table shows that the total suite does not catch anything *extra* that R_{gcc} misses.

The results in Table 4 indicate that the reduced test suite is just as effective as the larger suite at catching mutants applied with respect to the coverage of the minimum suite. These results seem to indicate that suites reduced by rule coverage can maintain the fault detection capabilities of their larger counterparts, hence we might provisionally accept that Hypothesis 2 is true.

6.3 Comparison with Randomly Created Reduced Suites

The results of the previous subsection might indicate that effective test suite reduction could be obtained using rule coverage as the main criterion. However, given the thresholding behaviour shown in Fig. 6a, we must eliminate the possibility that the size of the test suite, rather than its rule coverage properties, is determining the level of code coverage. Hence, in this subsection we examine Hypothesis 3, that the positive results thus far for reduced suites are genuinely attributable to the maintenance of grammar-rule coverage.

To investigate Hypothesis 3 we created 100 new test suites of the same size as the reduced suite, without considering rule coverage criteria. These 100 new test suites were randomly created by sampling *with replacement* the T^{+}_{gcc} suite. Sampling the suite *with replacement* meant that each new sample suite was drawn from the full suite, and thus could share test-cases with other suites. Each new test suite contained 40 test-cases, exactly the same amount of test-cases as the R_{gcc} suite but were selected randomly with no regard to their rule coverage.

The box plot in Fig. 8 shows the distribution of the rules throughout the randomly created suites. The number of rules covered ranges from 247 to 413 with more than 50% of the random suites having a coverage figure of between 380 and 400 rules. As can be seen from Fig. 9 all of the randomly created suites cover less rules than R_{gcc}. Thus *if* Hypothesis 3 is true and grammar-rule coverage is an effective reduction criterion, we expect the randomly created test suites to kill *less* mutants than the specially created R_{gcc} suite.

To investigate the third hypothesis the main mutation experiment was repeated under the same conditions as the original. This meant that the total number of mutations applied, seen in the first column of Table 5, was quite large. Since mutation testing can be quite time-consuming, with each applied mutant requiring a rebuild of the SUT followed by testing with each test-case, this can appear a daunting prospect. However, it should be noted that since the test suites shared test-cases, the total effort

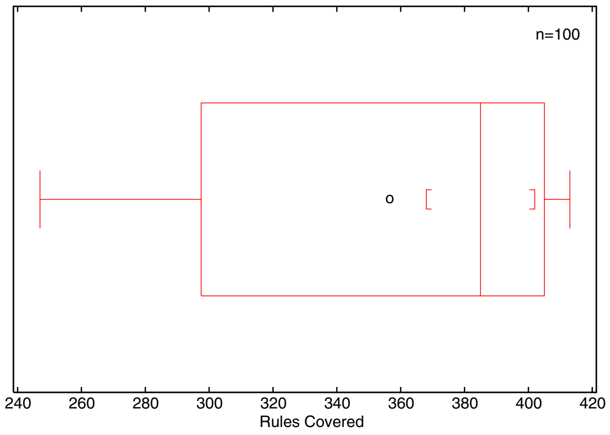


Fig. 8 A box plot showing the distribution of the rule coverage of randomly created test suites. *The number of rules covered range from* 247 *to* 413 *with* 356 *being the mean. All of the random suites cover less rules than the* R_{gcc} *and* R_{ddj} *suites*

involved was equivalent to performing mutation testing for each relevant case from the whole test suite, and then combining the individual results to calculate the impact for each random suite.

If Hypothesis 3 is correct, then the randomly created test suites should have a lower mutant kill rate than the reduced suite, due to the lower level of grammar-rule coverage. However, should the random suites find more inserted faults than the minimum suites then we can question the effectiveness of grammar-rule coverage as a criterion in reducing test suites of grammar-based software.

Table 5 gives a summary of the total number of mutants applied and caught during the investigation of Hypothesis 3. As the third column in Table 5 shows, the *average* number of mutants caught is roughly similar to the *total* number of mutants caught by R_{gcc} for *doc*++ and *keystone* and *twice* that of the mutants caught for *puma*. The fourth column in Table 5 shows the maximum amount of mutants caught by a single random test suite during the experiment. The summarised results of Table 5 are expanded in Figs. 10, 11, 12 respectively. In each of these figures, the number of

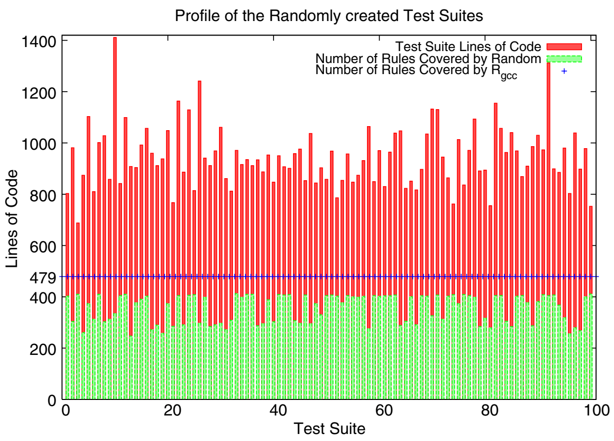


Fig. 9 A profile of the randomly created test suites used to test Hypothesis 3. *Each* line *in the bar chart represents a randomly created test suite. There are two distinct scales in this figure, the* red line *refers to the total LOC in the test suite while the* green line *refers to the number of rules covered. Each random suite clearly covers less rules than* R_{gcc}*, illustrated here by the* blue horizontal line

Table 5 The total number of mutants applied and caught during the investigation of Hypothesis 3

SUT	Total # of mutants applied	Average # of mutants caught	Max # of mutants caught
Doc++	55681	499.19	518
Keystone	9194	3223.75	5077
Puma	44474	8193.85	9739

mutants killed by each of the random suites is plotted. The kill rate of R_{gcc} for the SUT in question is included in each graph for comparative purposes.

Our study in Subsection 6.1 showed that the minimised test suite was able to maintain the fault detection capability of its larger counterpart. However, the results in this section show that similarly-sized suites that cover less grammar rules are equally good, if not better than the minimised suite. This would indicate that grammar-rule coverage is not as effective as code coverage when creating a minimised test suite. To further investigate the third hypothesis, a statistical analysis was performed on the results of Section 6.3.

6.4 Identifying the Effectiveness of Grammar-rule Coverage

The third hypothesis claims that the positive features of the reduced test suite is solely a result of grammar-rule coverage. The negative results obtained in Section 6.3 showed that the random test suites were more effective than the specially created grammar-rule reduced test suite. In this section we aim to show how effective grammar-rule coverage is as a reduction criterion when compared to simple code coverage.

The percentage of mutants killed, grammar-rules covered and code covered by each random test suite for each SUT was calculated. Based on this, it is possible to apply statistical correlation techniques to determine which factor has a greater influence the mutant kill-rate. The technique we used to determine the correlation is the Kendal τ rank coefficient. This technique is used to measure the degree of

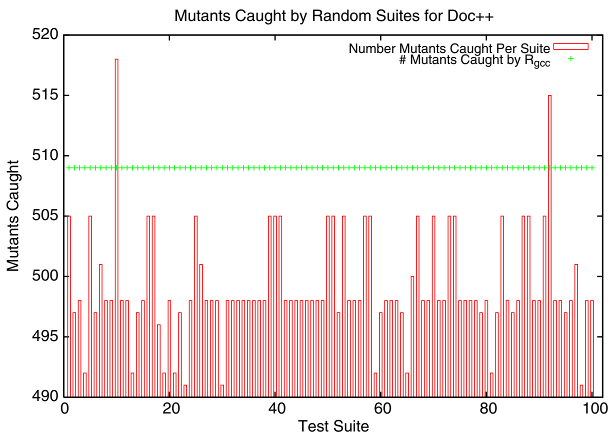


Fig. 10 The results of fault detection with the random suites for *doc++. This graph shows that R_{gcc} was more effective than* 98% *of the random suites. It is worth noting that all of the suites only caught a small fraction of the* 55,000 *mutants applied*

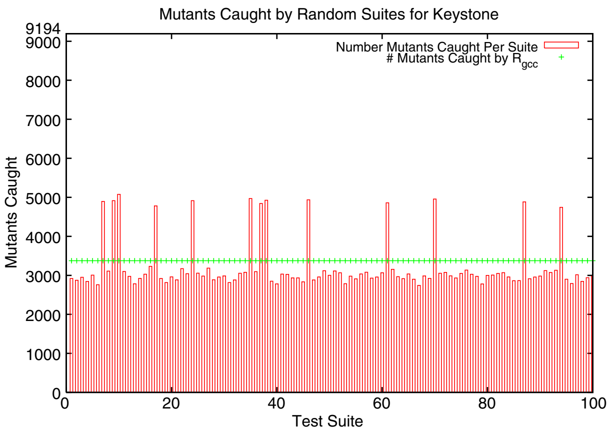


Fig. 11 The results of fault detection with the random suites for *keystone. It can be seen that for each of the* 9,194 *mutants applied, R_{gcc} outperforms* 86% *of the random suites but falls well behind the other* 14%

correspondence between two values. The result of applying this technique is a value between 0 and 1, with 1 representing a perfect correlation between the two values and 0 indicating that the values do not correlate at all.

Kendall's τ technique was applied to the values obtained from the work in Subsection 6.3 to produce the table given in Table 6. This table clearly shows that for each SUT, there is a greater correlation between code coverage of that SUT and the mutant kill rate. The graphs shown in Figs. 13 and 14 respectively illustrate the results for *puma* graphically.

The results for both *doc*++ and *keystone* shows that code coverage has a marginally greater influence on the mutant kill rate over grammar-rule coverage. This, in conjunction with the negative results from Subsection 6.3, must cause us to reject Hypothesis 3. With the rejection of the third hypothesis, it is clear that the positive properties shown by R_{gcc} are also shared by other test suites of similar size.

The rejection of third hypothesis indicates that the level of fault detection is a factor of the number of statements covered rather than the level of rule coverage. This is particularly interesting, since it seems to contradict the more positive results

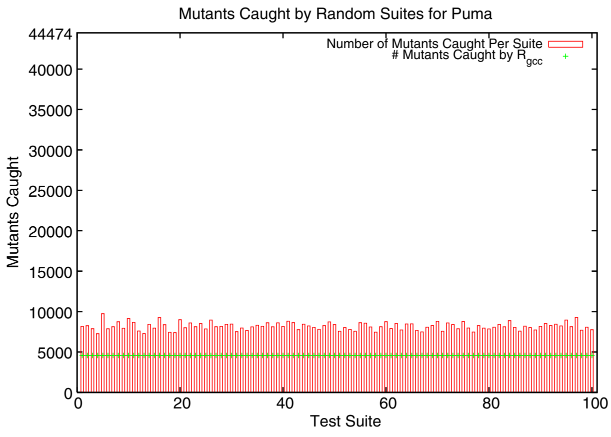


Fig. 12 The results of fault detection with the random suites for *puma. For each of the* 44,474 *mutants applied, no random suite was able to kill more than* 10,000 *mutants. However, every random suite caught more mutants than R_{gcc}*

Table 6 The Kendall τ coefficients for each of the random test suites for each of the SUTs

SUT	Mutants/rules	Mutants/code
Doc++	0.233	0.271
Keystone	0.265	0.342
Puma	0.004	0.738

For each SUT, we show the Kendall τ correlation between mutants killed and grammar-rule coverage alongside the τ correlation between mutants killed and code coverage

of the first two hypotheses. What it demonstrates is that these apparently positive results were due to secondary factors, not directly included in the initial evaluation. In particular, while rule coverage was used as a central factor in asserting the first two hypotheses, the results of investigating the third hypothesis indicate that it was not the *determining* factor in the experiments. In total, our experiments allow us to conclude that rule coverage *alone* is an ineffective criterion when reducing test suites of grammar-based software.

7 Threats to Validity

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

7.1 Threats to Internal Validity

The test suite used, T_{gcc}, 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 (Perennial test suite for ISO C++) or Plum Hall

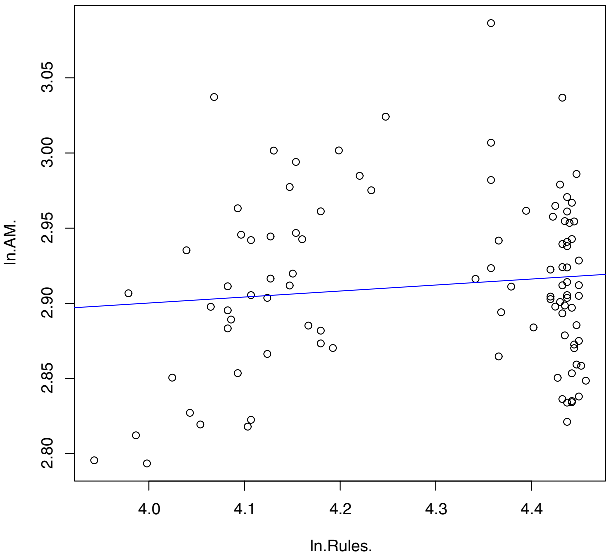


Fig. 13 The results of regressing the percentage of mutants caught versus rule coverage for Puma. *This graph plots the percentage code covered versus mutants killed for each of the random test suites for Puma. The* blue regression line *indicates that there is no correlation between rule coverage and mutant kill rate*

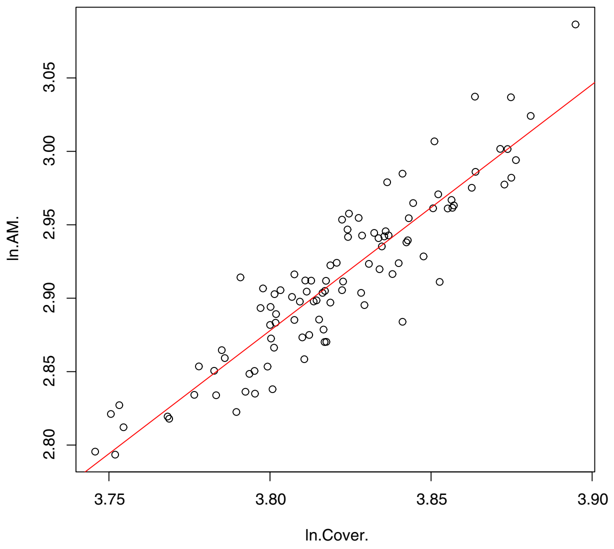


Fig. 14 The results of regressing the percentage of Mutants caught versus Code Coverage for Puma. *This graph plots the percentage code covered versus mutants killed for each of the random test suites for Puma. The* red regression line *indicates that there is a strong correlation between code coverage and mutant kill rate*

(Plum Hall test suites), can be an order of magnitude larger than T_{gcc}. However this work dealt only with the evaluation of test suites in the public domain.

One particular threat offered by the choice of this test suite is that low code coverage from a test suite may yield results which are not generalisable when the test suite is reduced. For example, the T_{gcc} test suite only exercises the C++ specific features of *doc++*, giving relatively low code coverage, with the effect that the percentage *reduction* in code coverage from R_{gcc} is also quite small. This could give the false impression that the code coverage was preserved via a function of the reduction criteria instead of this coverage coming from some initialisation code within *doc++*. The much higher code coverage figures for *keystone* and *puma* alongside the broad preservation of code coverage by R_{gcc} offer us some assurance against this threat for the purposes of this study.

The 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 (Lämmel 2001), although our analysis of context-dependent rule coverage (Hennessy and Power 2005b) suggests that there is little practical benefit to be gained from context-dependent coverage, at least for the ISO C++ grammar.

The mutation operators applied to each SUT are only a selection of those that can be applied. For example, mutation operators can be defined that test object-oriented features (such as those defined by Ma et al. for Java (Ma et al. 2005)), that could yield different results. However constructing a similar system for mutating C++ source is a massive undertaking in itself, and as such, there are only a limited number of commercial products available for this task.

7.2 Threats to External Validity

Threats to external validity centre 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 5 and 6 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.

8 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 (Harrold et al. 1993). 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 LOC (Wong et al. 1998). 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 (Rothermel et al. 1998). This study uses seven C programs, ranging in size from 138 to 516 LOC, 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 (Jones and Harrold 2003). 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 LOC in RSML^{-e} (Heimdahl and George 2004). 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.

Andrews et al. present a study to investigate the effectiveness of established test coverage criteria via mutation testing (Andrews et al. 2006). They focus solely on the Space program, used in a number of the other studies outlined above, to measure the effectiveness of test suites created with respect to four well known coverage criteria, namely block coverage, decision coverage, C-use and P-use. The number of mutants applied in this work is a sample of 10% of the total possible distributed evenly across the entire system. The authors find that the use of mutation testing is indeed a positive when verifying the effectiveness of a test suite. Another finding of this study is that test suite size is a major factor in determining the effectiveness of a test suite. Our results in Section 6.4 would back up this finding.

In our previous work we created reduced test suites for grammar-based software and compared their code-coverage to those test suites generated by Purdom's algorithm (Hennessy and Power 2005b). This work was extended to measure the fault detection capabilities of the reduced test suites (Hennessy and Power 2005a) through the selective use of mutation testing. This work only applied a sample of all possible mutants for each of the SUTs, showed that the reduced test suites were poor at detecting injected faults within the SUTs.

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. In addition to this, all the above approaches that use mutation testing with manually generated mutants use considerably fewer mutants than those reported here in Table 4. Finally, this work extends our own previous work by conducting a large scale empirical investigation involving mutation testing to assess the relative effectiveness of rule-coverage as a reduction criterion when compared to statement coverage. In this work, each SUT has *every* possible mutant applied to each line of code covered by a test suite. Thus, the number of mutants applied is a significantly higher number than in either our earlier work or in any other study we are aware of involving mutation testing.

9 Synopsis

In this paper the feasibility of using rule coverage as a criterion in the reduction of test suites for grammar-based software has been tested. We have taken an existing test suite 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 the 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 when compared to a suite selected with respect to code coverage. This is further illustrated by the rejection of the third Hypothesis which shows that rule coverage is a subset of code coverage with respect to reduction criteria.

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 standardised test suites 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 real-world grammar-based software.

There have been many studies involved in the theoretical investigation of grammar testing but there are few example of studies of this kind. We firmly believe that there is considerable scope for more empirical software-engineering research in the area of adequate test suite for grammar-based software.

Acknowledgements This work has been funded under the Embark scheme administered by the Irish Research Council for Science, Engineering and Technology.

References

Acostachioaie D (2000) DOC++: Open Source - Open Science - Open systems. Circles Electronic Magazine (36)

Andrews JH, Briand LC, Labiche Y, Namin AS (2006) Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans Softw Eng 32(8):608–624

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

Celentano A, Crespi-Reghizzi S, Vigna PD, Ghezzi C, Granata G, Savoretti F (1980) Compiler testing using a sentence generator. Softw Pract Exp 10(11):897–918

Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness. W.H. Freeman, San Francisco

Gibbs TH, Malloy BA, Power JF (2003a) Decorating tokens to facilitate recognition of ambiguous language constructs. Softw Pract Exp 33(1):19–39

Gibbs TH, Malloy BA, Power JF (2003b) Progression toward conformance of C++ language compilers. Dr Dobbs J 28(11):54–60

Harm J, Lämmel R (2000) Two-dimensional approximation coverage. Informatica 24(3):355–369

Harrold MJ, Gupta R, Soffa ML (1993) A methodology for controlling the size of a test Suite. ACM Trans Softw Eng Methodol 2(3):270–285

Heimdahl MPE, George D (2004) Test suite reduction for model based tests: effects on test quality and implications for testing. In: 19th IEEE international conference on automated software engineering. IEEE, Linz, pp 176–185

Hennessy M, Power JF (2005a) An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In: 20th IEEE international conference on automated software engineering. IEEE, Long Beach, pp 104–113

Hennessy M, Power JF (2005b) 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

Hennessy M, Power JF, Malloy BA (2003) gccXfront : Exploiting gcc as a Front-end for Program Comprehension via XML/XSL. In: 11th international workshop on program comprehension, Portland, OR, 10–11 May 2003

ISO/IEC JTC 1 (1998) International standard: programming languages - C++. No. 14882:1998(E), 1st edn. American National Standards Institute, New York

ISO/IEC JTC 1 (2003) International standard: programming languages - C++. No. 14882:2003(E), 2nd edn. American National Standards Institute, New York

Jones JA, Harrold MJ (2003) Test suite reduction and prioritization for modified condition/decision coverage. IEEE Trans Softw Eng Methodol 29(3):195–210

Klint P, Lämmel R, Verhoef C (2005) Toward an engineering discipline for grammarware. ACM Trans Softw Eng Methodol 14(3):331–380

Koppler R (1997) A systematic approach to fuzzy parsing. Softw Pract Exp 27(6):637–649

Lämmel R (2001) Grammar testing. In: Fundamental approaches to software engineering. LNCS, vol 2029. Springer, Berlin Heidelberg New York, pp 201–216

Ma Y-S, Offutt J, Kwon YR (2005) MuJava : an automated class mutation system. J Softw Test Verif Reliab 15(2):97–133

Malloy BA, Linde SA, Duffy EB, Power JF (2002) Testing C++ compilers for ISO language conformance. Dr Dobbs J 27(6):71–78

Malloy BA, Power JF (2001) An interpretation of Purdom's algorithm for automatic generation of test-cases. In: 1st annual international conference on computer and information science, Orlando, FL, 3–5 October 2001

Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C (1996) An experimental determination of sufficient mutant operators. ACM Trans Softw Eng Methodol 5(2):99–118

Offutt AJ, Alexander RT, Wu Y, Xiao Q, Hutchinson C (2001) A fault model for subtype inheritance and polymorphism. In: 12th international symposium on software reliability engineering, Hong Kong, 27–30 November 2001, pp 84–95

Perennial (2007) Perennial test suite for ISO C++. http://www.peren.com/pages/cppvs.htm. Accessed March 2008

Plum Hall (2008) Plum Hall test suites. http://www.plumhall.com/. Accessed March 2008

Power JF, Malloy BA (2002) Program annotation in XML: a parser-based approach. In: Working conference on reverse engineering, Richmond, 28 October–1 November 2002, pp 190–198
Power JF, Malloy BA (2004) A metrics suite for grammar-based software. Softw Maint Evol Res Pract 16(6):405–426
Purdom P (1972) A sentence generator for testing parsers. BIT 12(3):366–375
Roper M (1994) Software testing. McGraw-Hill, New York
Rothermel G, Harrold MJ, Ostrin J, Hong C (1998) An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: International conference on software maintenance, Bethesda, 16–19 November 1998, pp 34–43
Sim SE, Holt RC, Easterbrook S (2002) On using a benchmark to evaluate C++ extractors. In: 10th international workshop on program comprehension, Paris, 27–29 June 2002, pp 114–126
Spinczyk O, Gal A, Schröder-Preikschat W (2002) AspectC++: an aspect-oriented extension to C++. In: 40th international conference on technology of object-oriented languages and systems, Sydney, February 2002, pp 53–60
Wong WE, Horgan JR, London S, Mathur AP (1998) Effect of test-set minimization on fault detection effectiveness. Softw Pract Exp 28(4):347–369

Mark Hennessy works as an independent test consultant in the IT industry. He received his BSc in Computer Science and his PhD in Computer Science from the National University of Ireland, Maynooth. His research interests include programming language design, program analysis and testing methodologies.

James F. Power is a lecturer at the Department of Computer Science, National University of Ireland, Maynooth. He received his BSc in Computer Science from University College Dublin and an MSc and PhD in Computer Science from Dublin City University. His research interests include compiler design, program analysis and formal methods.