Mutation Testing `implements` Grammar-Based Testing

Jeff Offutt and Paul Ammann
Information and Software Engineering
George Mason University
Fairfax, VA 22030, USA
{offutt,pammann}@gmu.edu

Lisa (Ling) Liu
Chair of Software Engineering
ETH (Swiss Federal Institute of Technology)
Ch-8092 Zurich, Switzerland
ling.liu@inf.ethz.ch

Abstract

This paper presents an abstract view of mutation analysis. Mutation was originally thought of as making changes to program source, but similar kinds of changes have been applied to other artifacts, including program specifications, XML, and input languages. This paper argues that mutation analysis is actually a way to modify any software artifact based on its syntactic description, and is in the same family of test generation methods that create inputs from syntactic descriptions. The essential characteristic of mutation is that a syntactic description such as a grammar is used to create tests. We call this abstract view grammar-based testing, and view it as an interface, *which mutation analysis* implements. *This shift in view allows mutation to be defined in a general way, yielding three benefits. First, it provides a simpler way to understand mutation. Second, it makes it easier to develop future applications of mutation analysis, such as finite state machines and use case collaboration diagrams. The third benefit, which due to space limitations is not explored in this paper, is ensuring that existing techniques are complete according to the criteria defined here.*

1 Introduction

The traditional view of mutation is that we apply it to software by creating mutant versions of the source. In recent years, researchers have also applied the concepts behind mutation to other artifacts, including formal software specifications, XML, design models, and input languages. Thus, mutation analysis for programs is merely one of a family of test criteria, all of which operate on grammar-based syntactic objects. In object-oriented terms, mutation "`implements`" the general "`interface`" of grammar-based testing.

Legend has it that the first ideas of mutation analysis were postulated in 1971 in a class term paper by Richard Lipton. The first research papers were published by Budd and Sayward, Hamlet, and DeMillo, Lipton, and Sayward in the late '70s [11, 16]; DeMillo, Lipton, and Sayward's paper [11] is usually cited as the seminal reference.

A key to applying mutation to any artifact is the design of suitable mutation operators. Mutation operators have been designed for various programming languages, including Fortran IV [7], COBOL [17], Fortran 77 [12, 22], C [10], integration testing of C [9], Lisp [6], Ada [4, 29], Java [21], and Java class relationships [24]. They have also been designed for the formal specification languages SMV [2], and for XML messages [23, 30, 31].

This paper places what we call "traditional mutation testing," based on program source, in the larger context of testing based on grammar-based software artifacts. This is called *grammar-based mutation analysis*. In this view, mutation is generally about making modification to syntactic objects. Given a syntactic description of a software artifact, we can design mutation operators to generate artifacts that are valid (correct syntax), or artifacts that are invalid (incorrect syntax). They can be created directly from the grammar or by modifying a ground string (such as a program). Sometimes the structures we generate are test cases themselves and sometimes they are used to help us find test cases.

The term *mutation analysis* is used to refer to the process of modifying syntactic software artifacts, and *mutation testing* is used to refer to test criteria that are based on mutation analysis.

The remainder of this paper explores this concept. Section 2 presents a general view of grammar-based mutation analysis, and introduces generic test criteria. The new formulation and definitions makes it easier to understand mutation, and to apply mutation to new context areas. Sections 3, 4, 5, and 6 define how grammar-based testing is or can be implemented by mutation analysis for specific kinds of artifacts. The material in Section 5 is very recent and most of Section 6 is new to this paper, demonstrating the ability to apply mutation to new areas.

```
bank    ::= action*
action  ::= dep | deb
dep     ::= "deposit" account amount
deb     ::= "debit" account amount
account ::= digit^3
amount  ::= "$" digit^5 "." digit^2
digit   ::= "0" | "1" | "2" | "3" | "4"
            "5" | "6" | "7" |"8" | "9"
```

Figure 1. Grammar for bank transactions.

2 Grammar-Based Mutation Analysis

Software engineers often use structures from automata theory such as grammars, usually expressed in BNF, to describe the syntax of software artifacts (programs, inputs, specifications, models, etc.). Programming languages are described in grammar notation, program behavior is described in finite state machines, and allowable inputs to programs are defined by grammars. With grammar-based testing, tests are created from the grammar.

Before proceeding, we provide a short review of the BNF syntax with an example that will be used subsequently in the paper. In the example in Figure 1, `bank` is the *start symbol*, the strings on the left (`bank`, `action`, etc.) are nonterminal symbols and the strings on the right in quotes ("deposit," "debit," etc.) are terminal symbols. The *production rules* are separated by '|' or start with new nonterminals.

Grammatical descriptions can be used in two ways. A *recognizer*, such as an automaton or parser, decides if a given string (or test case) is in the set of strings represented by the grammar[1]. Grammars are also used to build *generators*, which derive strings from the grammar. An example derivation is:

`bank` $\rightarrow$ `action`*
$\rightarrow$ **action action***
$\rightarrow$ **dep** `action`*
$\rightarrow$ **deposit account amount** `action`*
$\rightarrow$ `deposit` **digit**^{3} `amount action`*
$\rightarrow$ `deposit` **digit digit**^{2} `amount action^*`
$\rightarrow$ `deposit` **7** `digit`^{2} `amount action`*
$\rightarrow$ `deposit 7` **digit digit** `amount action`*
$\rightarrow$ `deposit 7`**3** `digit amount action`*
$\rightarrow$ `deposit 73`**9** `amount action`*
$\rightarrow$ `deposit 739` **\$ digit**^{5} **. digit**^{2} `action`*
$\rightarrow$ `deposit 739 $`**digit**^{2}`.digit`^{2} `action`*
$\rightarrow$ `deposit 739 $`**digit digit**`.digit`^{2} `action`*
$\rightarrow$ `deposit 739 $`**1** `digit.digit`^{2} `action`*

[1]This paper abbreviates the phrase "set of strings represented by the grammar" as "in the grammar" for brevity and clarity. Note that in old 1960s-era terminology, "grammar" was used synonymously with "generator," but the testing area uses the term slightly differently.

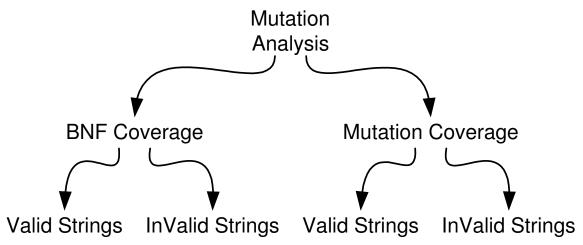

Figure 2. Mutation as grammar-based testing.

$\rightarrow$ `deposit 739 $12.  digit`^{2} `action`*
$\rightarrow$ `deposit 739 $12.` **digit digit** `action`*
$\rightarrow$ `deposit 739 $12.`**3 digit** `action`*
$\rightarrow$ `deposit 739 $12.3`**5** `action`*
$\vdots$

Derivation continues until all nonterminals have been rewritten and only terminal symbols remain. This paper uses the term *ground string* for a string that is in the grammar.

Figure 2 illustrates this concept. Grammar-based mutation analysis can be used to derive un-mutated or mutated strings. The mutation operators can be applied directly to the grammar (grammar mutation) or to strings that are derived from grammars (ground string mutation). The intent is to create invalid strings in grammar mutation, but valid strings are sometimes created accidentally. Finally, the strings can be in the language defined by the original grammar (valid) or not (invalid). In ground string mutation, the grammar is used to design the mutation operators and also to differentiate valid strings from invalid. All categories have uses in software testing.

The rest of this section describes a general view of mutation analysis and defines test criteria in terms of grammars. Criteria are defined for mutations to grammars and mutations to strings that are derived from grammars.

2.1 UnMutated Derivation Test Criteria

Testing is usually based on *test criteria*, the rules or collection of rules that impose requirements on test cases. Criteria for creating unmutated derivations are given in this subsection; criteria for mutated derivations are given in the next. Although many test criteria could be defined, the most straightforward are *terminal symbol coverage* and *production coverage*. These definitions use TR to refer to the set of test requirements imposed by a criterion.

CRITERION **1 Terminal Symbol Coverage (TSC):** *The set of test requirements, TR, contains each terminal symbol t in the grammar G.*

A test is simply a string that is derived by using the grammar. The number of tests generated by TSC is bounded by the number of terminal symbols; 14 in the `bank` example of Figure 1. Because most tests will include more than one terminal symbol, the actual number of tests is usually less than the bound.

CRITERION **2 Production Coverage (PC):** *TR contains each production p in the grammar G.*

The number of tests generated by PC is bounded by the number of production rules; 17 in the `bank` example of Figure 1.

Another possible criterion could be that of of deriving all possible strings from a grammar, but this is not defined because many grammars have infinite derivations. For example, the first production in the `bank` grammar is "action*", which theoretically can be used to create an infinite number of strings. If we ignore the first production in `bank`, the number of derivable strings is finite but still very large. There are two possible actions ("deposit" and "debit"), and "account" has a maximum of three digits with 10 choices, or 1000. The production for "amount" has two occurrences of "digit," one that has length up to two (100 choices) and the other with length up to five (100,000 choices). Altogether, the `bank` grammar can generate $2 * 1000 * 100,000$ = $200,000,000$ strings.

The criteria in the next subsection directly mutate ground strings rather than the grammar that defines them.

2.2 Mutated Derivation Test Criteria

One of the interesting things that grammars do is describe what an input is *not*. For example, it is quite common to require a program to reject malformed inputs, and this property should clearly be tested, since it is easy for programmers to forget it or get it wrong.

Thus, when a grammar defines a program's inputs, it is often useful to produce invalid strings from the grammar. It is also helpful to test with strings that are *valid* but that follow a different derivation from a pre-existing string. Both of these strings are called *mutants*. This can be done by mutating the grammar, then generating strings, or by mutating values during a production.

Mutation is always based on a set of *mutation operators*, which are usually expressed with respect to a *ground* string. A *mutation operator* is a rule that specifies syntactic variations of strings generated from a grammar. A *mutant* is the result of one application of a mutation operator.

Mutation operators can be applied to ground strings, grammars, or dynamically during derivations. In this formulation, the traditional notion of mutating program source (*traditional mutation*) uses the program as a ground string and applies mutation operators to create valid strings. The notion of a mutation operator is extremely general, and so a very important part of applying mutation to any artifact is the design of suitable mutation operators. A well designed collection of operators can result in very powerful testing, but a poorly designed collection is almost useless. For example, a tool that "implements mutation" but that changes only predicates to *true* and *false* would simply provide an expensive way to satisfy branch coverage.

Sometimes an explicit ground string exists, but sometimes it exists only implicitly as the (possible) result of not applying any mutation operators. For example, we care about the ground string when applying mutation to program statements. The ground string is the sequence of program statements in the program under test, and the mutants are slight syntactic variations of that program. We do not use the ground string during invalid input testing when the goal is to see if a program correctly responds to invalid inputs. The ground strings are valid inputs, and variants are the invalid inputs. For example, a valid input might be a transaction request from a correctly logged-in user. The invalid version might be the same transaction request from a user who is not logged in.

Two issues often come up when applying mutation operators. First, should more than one mutation operator be applied at the same time to create one mutant? Common sense indicates no, and strong experimental and theoretical evidence has been found [5, 7, 12, 17, 29] for mutating only one element at a time in traditional (program-based) mutation. Another question is should every possible application of a mutation operator to a ground string be considered? This is usually done in traditional mutation for the theoretical reason that traditional mutation subsumes a number of other test criteria, but only if operators are applied comprehensively. However, this is not always done when the ground string does not matter, for example, in the case of invalid input testing.

When a grammar defines a program's input space, mutants can be created directly from the grammar by modifying productions during a derivation, using a generator approach as introduced in the previous section. These mutants are *tests*. When a given grammar defines the tested program, a derivation is mutated to produce valid strings as mutants. Tests are then designed to "kill" the mutants by causing the mutants to produce different output from original program. This leads a different formulation for the traditional idea of killing mutants.

Given a mutant $m \in M$ for a derivation D and a test t, t is said to *kill* m if and only if the output of t on D is different from the output of t on m.

The derivation D may be represented by the complete list of productions followed, or it may simply be represented by the final string. For example, in Section 3.1, the strings are programs or program components.

Table 1. Structure of grammar-based mutation analysis.

	Program-Based	Integration	Model-Based	Input-Based
Grammar	Section 3		Section 5	Section 6.1
Grammar	Programming languages	No known applications	Algebraic specifications	Input languages (XML)
Summary	Compiler testing			Input space testing
Valid?	Valid and invalid strings			Valid strings
String Mut	Section 3.1	Section 4	Section 5.1	Section 6.2
Grammar	Programming languages	Programming languages	FSMs	Input languages (XML)
Summary	Mutates programs	Tests integration	Uses model-checking	Error checking
Ground?	Yes	Yes	Yes	No
Valid?	Yes, must compile	Yes, must compile	Yes	No
Tests?	Mutants are not tests	Mutants are not tests	Traces are tests	Mutants are tests
Killing?	Yes	Yes	Yes	No notion of killing
Notes	Subsumes many other techniques.	Includes object-oriented testing		Can mutate grammar, then produce strings

If a grammar defines a tested program or a model of the program, coverage is defined in terms of killing mutants.

CRITERION **3 Mutation Coverage (MC):** *For each mutant $m \in M$, TR contains exactly one requirement, to kill m.*

That is, coverage in this kind of mutation testing equates to killing the mutants. The amount of coverage is usually written as a percent of mutants killed and called the "*mutation score*."

When a grammar is mutated to produce invalid strings, the testing goal is to run the mutants to see if the behavior is correct. The coverage criterion is therefore simpler, as the mutation operators are the test requirements.

CRITERION **4 Mutation Operator Coverage (MOC):** *For each mutation operator, TR contains exactly one requirement, to create a mutated string m that is derived using the mutation operator.*

CRITERION **5 Mutation Production Coverage (MPC):** *For each mutation operator, and each production that the operator can be applied to, TR contains the requirement to create a mutated string from that production.*

If the mutation operator replaces a variable name with another, MOC requires only one mutated program with a single replacement. MPC requires every variable reference to be replaced by every variable that is of compatible type.

The number of test requirements for mutation on ground strings cannot be generally quantified because the number depends on the syntax of the artifact as well as the mutation operators. In most situations, mutation yields more test requirements than any other test criterion.

The rest of this paper explores various forms of mutation testing as a form of grammar-based testing. Table 1 summarizes the characteristics of the various flavors of syntax testing. The section in the paper where each application appears is listed. Whether valid or invalid tests are created is noted. For mutation testing, we also note whether a ground string is used, whether the mutants are tests or not, and whether there is a notion of killing mutants.

3 Grammar-Based Testing of Programs

The most common application of mutation is to programs. At the grammar level, grammar derivations are used to create programs to test compilers (sometimes attributed to Hamlet [16] and sometimes unattributed). This is a specialized application with no recent literature, so this paper does not dwell on it.

3.1 Program-Based Mutation

Program-based mutation uses operators that are defined in terms of the grammar of a particular programming language. We start with a **ground string**, which is the program component that is being tested. We then apply mutation operators to create mutants. These mutants must be compilable, so program-based mutation creates **valid** strings. The mutants are not tests, but are used to help us find tests.

A key to successful use of mutation is the mutation operators, which must be separately designed for each language. In program-based mutation, invalid strings are syntactically illegal and would be caught by a compiler. These are called *stillborn* mutants and should not be generated. A *trivial* mutant can be killed by almost any test case. Some mutants are functionally *equivalent* to the original program. That is, they always produce the same output as the original program, so no test case can kill them. Equivalent mutants represent infeasible test requirements, just like unreachable statements, infeasible paths, and definition-use pairs for which no definition-free path exists [15, 18, 20, 28].

Program-based mutation has traditionally been applied to individual statements for unit level testing. The mutation operators are defined to satisfy one of two goals. One goal is to mimic typical programmer mistakes, thus trying to ensure that the tests can detect those mistakes. The other goal is to force the tester to create tests that have been found to effectively test software. Statement level mutation operators have been designed for various programming languages, including Fortran IV [7], COBOL [17], Fortran 77 [12, 22], C [10, 9], Lisp [6], Ada [4, 29], Java [21] and Java class relationships [24] (discussed in Section 4).

The number of program-based mutants is roughly proportional to the product of the number of data references times the number of data objects ($O(Refs * Vars)$) [5]. The term *selective mutation* describes the strategy of using only mutation operators that are particularly effective [27, 32]. Effectiveness has been evaluated as follows: If tests that are created specifically to kill mutants created by mutation operator o_i also kill mutants created by mutation operator o_j with very high probability, then mutation operator o_i is more *effective* than o_j. The selective mutation approach eliminates the number of data objects so that the number of mutants is proportional to the number of variable references ($O(Refs)$) [27].

Many program-level mutation operators could be defined, as in other systems [4, 6, 7, 12, 17, 21, 22, 29]. Researchers have found that a collection of mutation operators that insert unary operators and that modify unary and binary operators will be **effective** [32, 27]. This research was done with Fortran-77 (the Mothra system), resulting in five operators, and applied to Java by Ma, Offutt and Kwon, resulting in 11 operators for muJava [26]. These operators are summarized here and defined in detail in the muJava papers [24, 25].

1. **ABS**—Absolute Value Insertion
2. **AOR**—Arithmetic Operator Replacement
3. **ROR**—Relational Operator Replacement
4. **COR**—Conditional Operator Replacement
5. **SOR**—Shift Operator Replacement
6. **LOR**—Logical Operator Replacement
7. **ASR**—Assignment Operator Replacement
8. **UOI**—Unary Operator Insertion
9. **UOD**—Unary Operator Deletion
10. **SVR**—Scalar Variable Replacement
11. **BSR**—Bomb Statement Replacement

4 Grammar-based Integration Mutation Testing

Beizer defined *integration testing* to be assessing whether the interfaces between modules (defined below) in a given subsystem have consistent assumptions and communicate correctly [3]. This section first discusses how mutation can be used for testing at the integration level without regard to object-oriented relationships, then how mutation can be used to test for problems involving inheritance, polymorphism and dynamic binding.

Integration mutation (also called "interface mutation") works by creating mutants based on the grammar of connections between components [9]. These mutants must be compilable, thus they are also **valid** strings. Most mutations are around method calls, and both the calling (caller) and called (callee) method must be considered. Interface mutation operators do the following:

- Change a calling method by modifying the values that are sent to a called method.
- Change a calling method by modifying the call.
- Change a called method by modifying the values that enter and leave a method. This should include parameters as well as variables from a higher scope (class level, package, public, etc.).
- Change a called method by modifying statements that return from the method.

Delamaro, Maldonado and Mathur defined the following five interface mutation operators for C as part of the Proteum system [9, 10]. They are summarized here and defined in detail in the Proteum papers.

1. **IPVR**—Integration Parameter Variable Replacement
2. **IUOI**—Integration Unary Operator Insertion
3. **IPEX**—Integration Parameter Exchange
4. **IMCD**—Integration Method Call Deletion
5. **IREM**—Integration Return Expression Modification

Object-Oriented Mutation Operators. Languages that include features for inheritance and polymorphism also often include features for information hiding and overloading. Thus, mutation operators to test those features are usually included with the OO operators, even though these are not usually considered to be essential to call a language "object-oriented." The following operators are used in the muJava system [26] and defined in detail in the muJava papers [24, 25].

1. **AMC**—Access Modifier Change
2. **HVD**—Hiding Variable Deletion
3. **HVI**—Hiding Variable Insertion
4. **OMD**—Overriding Method Deletion

5. **OMM**—Overridden Method Moving
6. **OMR**—Overridden Method Rename
7. **SKD**—*Super Keyword Deletion*
8. **PCD**—Parent Constructor Deletion
9. **ATC**—Actual Type Change
10. **DTC**—Declared Type Change
11. **PTC**—Parameter Type Change
12. **RTC**—Reference Type Change
13. **OMC**—Overloading Method Change
14. **OMD**—Overloading Method Deletion
15. **AOC**—Argument Order Change
16. **ANC**—Argument Number Change
17. **TKD**—*this Keyword Deletion*
18. **SMC**—*Static Modifier Change*
19. **VID**—Variable Initialization Deletion
20. **DCD**—Default Constructor Delete

5 Grammar-based Testing Using Models

The general term "model-based" is applied to languages that describe software in abstract terms. This includes formal specification languages such as Z, SMV, OCL, etc., and informal specification languages and design notations such as statecharts, FSMs, and other UML diagram notations. Such languages are becoming more widely used, partly because of increased emphasis on software quality and partly because of the widespread knowledge of the UML.

To our knowledge, terminal symbol coverage and production coverage (grammar coverage) have been only applied to one type of specification language: algebraic specifications [1, 13, 14, 19]. Algebraic specifications are not widely used, so this paper does not cover this topic.

5.1 Model & Spec-Based Mutation

Mutation testing can also be a valuable method at the specification level [2, 8]. In fact, for certain types of specifications, mutation testing is actually easier at the specification level. We address one such type of specification in this section, namely specifications expressed as finite state machines. Model-based mutants should be **valid** with respect to the grammar.

A finite state machine is a graph G, with a set of states (nodes), a set of initial states (initial nodes), and a transition relation (the set of edges). When finite state machines are used, sometimes the edges and nodes are explicitly identified, as in the typical bubble and arrow diagram. Sometimes the finite state machine is more compactly described in the following way.

1. States are implicitly defined by declaring variables with limited ranges. The state space is then the Cartesian product of the ranges of the variables.

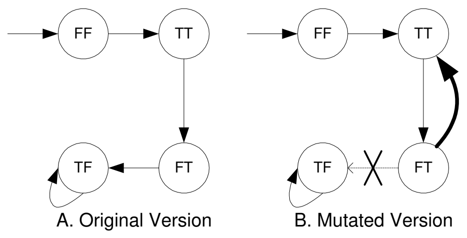

Figure 3. Finite State Machine for SMV Specification.

2. Initial states are defined by limiting the ranges of some or all of the variables.

3. Transitions are defined by rules that characterize the source and target of each transition.

The following example in the language SMV from Ammann and Black [2] clarifies these ideas. We describe a machine with a simple syntax, and show the same machine with explicit enumerations of the states and transitions.

```
MODULE main
#define false 0
#define true 1
VAR
   x, y : boolean;
ASSIGN
   init (x) := false;
   init (y) := false;
   next (x) := case     next (y) := case
      !x & y : true;       x & !y : false;
      !y     : true;       x & y  : y;
      x      : false;      !x & y : false;
      true   : x;          true   : true;
   esac;                esac;
```

There are two variables, each of which can have only two values (boolean), so the state space is of size $2 * 2 = 4$. There is one initial state, as defined in the two `init` statements under `ASSIGN`. The transition diagram is shown in Figure 3 (A). Transition diagrams for SMV can be derived by mechanically following the specifications. For example, assume the above specification is in the state $(true,\ true)$. The next value for x will be determined by the "`x : false`" statement. x is $true$, so its next value will be $false$. Likewise, $x\ \&\ y$ is true, so the next value of y will be $true$. Thus, the state following $(true,\ true)$ is $(false,\ true)$.

In our context, there are two particularly important aspects of such a structure.

1. Finite state descriptions can capture system behavior at a very high level – suitable for communicating with the

end user. Finite state machines are incredibly useful for the hardest part of testing, namely system testing.

2. The verification community has built powerful analysis tools for finite state machines. These tools are highly automated (in contrast to typical theorem provers). Further, these tools produce explicit evidence, in the form of counterexamples, for properties that do not hold in the finite state machine. These counterexamples can be interpreted as test cases. Thus, it is easier to automate test case generation from finite state machines than from program source.

5.2 Mutations and Test Cases

Mutating the syntax of state machine descriptions is very much like mutating program source. Mutation operators must be defined, and then they are applied to the description.

One example is the *Constant Replacement* operator, which replaces each constant with other constants. Given the phrase `!x & y :  false` in the `next` statement for `y`, replace it with `!x & y :  true`. The machine for this mutant is shown in Figure 3 (B). The new transition is drawn as an extra thick arrow and the replaced transition is shown as a crossed out dotted arrow.

Generating a test case to kill this mutant is a little different from with program-based mutation. We need a sequence of states that is allowed by the transition relation of the original state machine, but not by the mutated state machine. Such a sequence is precisely a test case that kills the mutant.

Finding a test to kill a mutant of a finite state machine expressed in SMV can be automated using a *model checker*. A model checker takes two inputs. The first is a finite state machine, described in a formal language such as SMV. The second is a statement of some property, expressed in a *temporal logic*. Temporal logics can be used to express only properties that are true “now,” and also properties that will (or might) be true in the future. The following is a simple temporal logic statement for the mutant in Figure 3:

> The original expression, `!x & y :  false` in this case, is **always** the same as the mutated expression, `!x & y :  true`.

For the given example, this statement is false with respect to a sequence of states allowed by the original machine if and only if that sequence of states is rejected by the mutant machine. In other words, such a sequence in question is a test case that kills the mutant. If we add the following SMV statement to the (unmutated) machine in Figure 3:

```
SPEC AG (!x & y) ⟶ (y = false)
```

The model checker will obligingly produce the desired test sequence:

```
/* state 1 */ { x  = 0, y  = 0 }
/* state 2 */ { x  = 1, y  = 1 }
/* state 3 */ { x  = 0, y  = 1 }
/* state 4 */ { x  = 1, y  = 0 }
```

Some mutated state machines are equivalent to the original machine. Model checkers are exceptionally well adapted to deal with this. The key theoretical reason is that a model checker has a finite domain to work in, and hence the equivalent mutant problem is decidable (unlike with program code). In other words, if the model checker does not produce a counterexample, we *know* that the mutant is equivalent.

6 Grammar-Based Input Testing

The last type of mutation testing that is included in this paper is based on grammars that formally define the syntax of the inputs to a program, method, or software component. For example, a language’s grammar defines the inputs of its compiler, and the XML schema defines the inputs of a XML parser [23, 30, 31]. This section explains how to apply mutation to grammars that define the input space of a program or program component to generate tests.

6.1 Grammar Input-Based Testing

Consider a program that processes a sequence of deposits and debits, where each deposit is of the form `deposit` *account amount* and each debit is of the form `debit` *account amount*. The input structure of this program can be described with the regular expression:

$(\texttt{deposit}\ \mathit{account}\ \mathit{amount} \mid \texttt{debit}\ \mathit{account}\ \mathit{amount})^*$

This regular expression describes any sequence of deposits and debits. The regular expression input description is still fairly abstract, in that it does not say anything about what an *account* or an *amount* looks like. One input that can be derived from this grammar is:

```
deposit 739 $12.35
deposit 644 $12.35
debit 739 $19.22
```

Undergraduate CS courses teach how to build finite automata (graphs) that capture the effects of regular expressions. These can be used with graph coverage criteria. Although regular expressions suffice for some programs, others require more expressive grammars. The prior example was specified in grammar form in Figure 1 in Section 2.

```
<books>
 <book>
    <ISBN>0471043281</ISBN>
    <title>The Art of Software Testing</title>
    <author>Glen Myers</author>
    <publ>Wiley</publ>
    <price>50.00</price>
    <year>1979</year>
 </book>
</books>
```

Figure 4. Simple XML message for books.

The graph that would represent even this simple example is quite large with all the details; it has nine states and eleven edges. Tests are derived from grammars by systematically replacing nonterminals with productions, as shown in the example in Section 2.

Of course, it often happens that an informal description of the input syntax is available, but a formal grammar is not. This means that the tester is left with the engineering task of formally describing the input syntax. This process is **extremely** valuable, and will often expose ambiguities and omissions in the requirements and software. Thus, this step should be carried out early in development, definitely before implementation and preferably before design. Once defined, it is sometimes helpful to use the grammar directly in the program for execution-time input validation.

6.1.1 XML example

The *eXtensible Markup Language (XML)* is used to describe program inputs (among other things). XML describes and encodes data for transmission. XML uses *tags*, which are textual descriptions of data enclosed in angle brackets (‘<’ and ‘>’). A simple example XML message for books is shown in Figure 4. This example is used to illustrate the use of grammar testing on software that uses XML messages. The example lists a book, with tag names “books,” “book,” “ISBN,” etc.

XML documents can be constrained by grammar definitions written in *XML Schemas*. For example, a schema for the books example can specify that each message has an unbounded number of “book” tags and put restrictions on some of the tags. First, the “title,” “author,” “publ,” “price” and “year” fields may be mandatory but the “ISBN” is optional. Additionally, the “price” data element can be of type decimal (numeric) with two digits after the decimal point. Two data elements, “ISBN” and “year,” can be defined in a more structured way using type definitions, for example, four numeric digits for the year and 10 numeric digits for the ISBN.

Given an XML schema, the criteria defined in Section 2.1 can be used to derive XML messages that serve as test inputs. Following the production coverage criteria would result in two XML messages for this simple schema, one that includes the optional ISBN and one that does not.

6.2 String Mutation Input-Based Testing

Programs should reject invalid inputs, and this functionality needs to be tested. Invalid inputs often cause the software to behave in surprising ways, which malicious parties can use to their advantage. This is how the classic “buffer overflow attack” works. Similarly, a key step in certain web-based browser attacks is to provide a string input that contains malicious HTML, Javascript, or SQL. Software should behave “reasonably” with invalid inputs. “Reasonable” behavior may not always be defined, which is a significant reason why it should be tested.

Invalid inputs can be created by mutating input grammars. When mutating grammars, the mutants are the tests and we create **valid** and **invalid** strings. There is **no ground string**, so the notion of killing mutants does not apply to mutating grammars. Several mutation operators for grammars are defined below. Unlike previous operators, these have not appeared in the literature before and so more details are given.

1. ***Nonterminal Replacement***: *Every nonterminal symbol in a production is replaced by other nonterminal symbols.*

This is a very broad mutation operator that could result in many strings that are not only invalid, they are so far away from valid strings that they are useless for testing. If the grammar provides specific rules or syntactic restrictions, some nonterminal replacements can be avoided. This is analogous to avoiding compiler errors in program-based mutation. For example, some strings represent type structures and only nonterminals of the same or compatible type should be replaced.

Consider the example in section 2. The production `dep ::= "deposit" account amount` can be mutated to create the following three productions:

```
dep ::= "debit" account amount
dep ::= "deposit" amount amount
dep ::= "deposit" account digit
```

Which can result in the corresponding tests:

```
debit 739 $12.35
deposit $19.22 $12.35
deposit 739 1
```

2. ***Terminal Replacement***: *Every terminal symbol in a production is replaced by other terminal symbols.*

Just as with terminal replacement, some terminal replacements may lead to strings that are not in the original grammar. Recognizing replacements that yield invalid strings depends on the grammar that is being mutated. For example, the production `amount ::= "$" digit`$^+$ `"." digit`^{2} can be mutated to create the following three productions:

```
amount ::= "." digit+ "." digit2
amount ::= "$" digit+ "$" digit2
amount ::= "$" digit+ "1" digit2
```

Which can result in the corresponding tests:

```
deposit 739 .12.35
deposit 739 $12$35
deposit 739 $12135
```

3. ***Terminal and Nonterminal Deletion***: *Every terminal and nonterminal symbol in a production is deleted.*

For example, the production `dep ::= "deposit" account amount` can be mutated to create the following three productions:

```
dep ::= account amount
dep ::= "deposit" amount
dep ::= "deposit" account
```

Which can result in the corresponding tests:

```
739 $12.35
deposit $12.35
deposit 739
```

4. ***Terminal and Nonterminal Duplication***: *Every terminal and nonterminal symbol in a production is duplicated.*

This is sometimes called the "stutter" operator. For example, the production `dep ::= "deposit" account amount` can be mutated to create the following three mutated productions:

```
dep ::= "deposit" "deposit" account amount
dep ::= "deposit" account account amount
dep ::= "deposit" account amount amount
```

Which can result in the corresponding tests:

```
deposit deposit 739 $12.35
deposit 739 739 $12.35
deposit 739 $12.35 $12.35
```

Just as with program-based mutation, some inputs from a mutated grammar rule are still in the grammar. The example above of changing the rule

```
   dep ::= "deposit" account amount
```

to be

```
   dep ::= "debit" account amount
```

yields an "equivalent" mutant. The resulting input, `debit 739 $12.35`, is valid, although the effects are (sadly) quite different for the customer. If the idea is to generate invalid inputs exclusively, some way must be found to screen out mutant inputs that are valid. Although this sounds much like the equivalence problem for programs, there is a small but significant difference. Here the problem is solvable and can be solved by creating a recognizer from the grammar, and checking each string as it is produced.

Many programs are supposed to accept some, but not all, inputs from some larger language. For example, a web-based program might restrict its inputs to a subset of HTML. In this case, we have two grammars: the full grammar, and a grammar for the subset. In this case, the most useful invalid tests to generate are those that are in the first grammar, but not in the second.

7 Conclusions

The traditional view of mutation analysis is that it is a method for modifying programs according to specific rules to help create high quality tests. This paper points out that mutation is one instantiation of a very general form of testing, which we call **grammar-based testing**. Given a grammar description of a software artifact, mutation operators can be defined to create alternate versions of artifacts. These alternate versions (mutants) can either be valid according to the grammar or invalid. They can be created directly from the grammar or by modifying a ground string (such as a program).

This re-definition of mutation has three benefits. First, it allows mutation to be described in a more simple way and understood more readily. Second, it is easier to develop new applications of mutation analysis. it makes it easier to apply mutation analysis to new contexts. Essentially, we can instantiate the concept onto new grammar-based artifacts in a seamless way. The third benefit is left for future work, that of ensuring that existsing techniques are complete according to the generic criteria in Section 2.

References

[1] P. Ammann and J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS 94)*, pages 69–80, Gaithersburg MD, June 1994. IEEE Computer Society Press.

[2] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54, Brisbane, Australia, December 1998.

[3] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.

[4] J. H. Bowser. Reference manual for Ada mutant operators. Technical report GIT-SERC-88/02, Georgia Institute of Technology, February 1988.

[5] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven CT, 1980.

[6] T. A. Budd and R. J. Lipton. Proving Lisp programs using test data. In *Digest for the Workshop on Software Testing and Test Documentation*, pages 374–403, Ft. Lauderdale FL, December 1978. IEEE Computer Society Press.

[7] T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward. Mutation analysis. Technical report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, April 1979.

[8] H. Chockler and O. Kupferman. Coverage of implementations by simulating specifications. In *TCS '02: Proceedings of the IFIP 17th World Computer Congress - TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science*, pages 409–421. Kluwer, 2002.

[9] M. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, March 2001.

[10] M. E. Delamaro and J. C. Maldonado. Proteum-A tool for the assessment of test adequacy for C programs. In *Proceedings of the Conference on Performability in Computing Systems (PCS 96)*, pages 79–95, New Brunswick, NJ, July 1996.

[11] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[12] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[13] R. K. Doong and P. G. Frankl. Case studies on testing object-oriented programs. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 165–177, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

[14] I. R. Forman. An algebra for data flow anomaly detection. In *Proceedings of the Seventh International Conference on Software Engineering*, pages 278–286. IEEE Computer Society Press, March 1984.

[15] A. Goldberg, T. C. Wang, and D. Zimmerman. Applications of feasible path analysis to program testing. In *Proceedings of the 1994 International Symposium on Software Testing, and Analysis*, pages 80–94, Seattle WA, August 1994.

[16] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[17] J. M. Hanks. Testing COBOL programs by mutation: Volume I – introduction to the CMS.1 system, volume II - CMS.1 system documentation. Technical report GIT-ICS-80/04, Georgia Institute of Technology, February 1980.

[18] D. Hedley and M. A. Hennell. The causes and effects of infeasible paths in computer programs. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 259–266, London UK, August 1985. IEEE Computer Society Press.

[19] W. E. Howden. Algebraic program testing. *Acta Informatica*, 10(1):53–66, 1978.

[20] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In *Proceedings of the 1994 International Symposium on Software Testing, and Analysis*, pages 95–107, Seattle WA, August 1994.

[21] S.-W. Kim, J. A. Clark, and J. A. McDermid. Investigating the effectiveness of object-oriented testing strategies using the mutation method. *Software Testing, Verification and Reliability*, 11(3):207–225, December 2001.

[22] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software-Practice and Experience*, 21(7):685–718, July 1991.

[23] S. C. Lee and J. Offutt. Generating test cases for XML-based web component interactions using mutation analysis. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 200–209, Hong Kong China, November 2001. IEEE Computer Society Press.

[24] Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for Java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, pages 352–363, Annapolis MD, November 2002. IEEE Computer Society Press.

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

[26] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. muJava home page. online, 2005. http://ise.gmu.edu/∼offutt/mujava/, http://salmosa.kaist.ac.kr/LAB/MuJava/, last access April 2006.

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

[28] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. *Software Testing, Verification, and Reliability*, 7(3):165–192, September 1997.

[29] A. J. Offutt, J. Payne, and J. M. Voas. Mutation operators for Ada. Technical report ISSE-TR-96-09, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, March 1996. http://www.ise.gmu.edu/techrep/.

[30] J. Offutt and W. Xu. Generating test cases for web services using data perturbation. In *Workshop on Testing, Analysis and Verification of Web Services*, Boston, MA, July 2004. ACM SIGSoft.

[31] J. Offutt and W. Xu. Testing web services by XML perturbation. In *Proceedings of the 16th International Symposium on Software Reliability Engineering*, Chicago, IL, November 2005. IEEE Computer Society Press.

[32] W. E. Wong and A. P. Mathur. Fault detection effectiveness of mutation and data flow testing. *Software Quality Journal*, 4(1):69–83, March 1995.