Compiler-Integrated Program Mutation*

R. A. DeMillo
E. W. Krauser
A. P. Mathur

Software Engineering Research Center
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-2004

Abstract

Software testing is of critical importance to the software development life cycle. Consequently, tools that aid the developer/tester are of great interest. Traditionally, available compilers have supported only two simple testing techniques, namely, statement and branch coverage. However, during compilation, sufficient syntactic and semantic information is available to provide support for more sophisticated testing techniques. This paper presents a novel method to efficiently support program mutation via information and code generated at compile-time.

1 Introduction

This paper is concerned with compiler-integrated support for program mutation [1], a well-known software testing technique. Mutation requires information attained by modifying the internal execution behavior of a program P under test. Faults are injected into P that are intended to model simple errors possibly introduced by programmers using a specific programming language L. Each fault, or mutation, is obtained by a single point, syntactically correct change to the original program P. Each program resulting from such a change to P is called a mutant program. The goal of a mutation tester is to select test data such that the output of the original program is distinguished from the output of all mutant programs. Such test data is deemed to be 'mutation adequate' and regarded as offering very strong evidence that a well chosen set of simple errors do not exist in P. More importantly, an established theoretical framework and large body of empirical evidence suggest that mutation adequate test data also distinguishes a much larger set of complex errors in P [4]. Figure 1 depicts several examples of mutations applied to programs written in the C programming language [2].

Tools currently exist that support program mutation via an interpreter-based execution environment [7]. These tools offer flexibility and ease of implementation, however, they execute mutant programs much slower than if the mutants were to consist of native machine code. Therefore, testing tools based on current approach are not attractive as a means to test most commercial software systems. Below we present a novel method that permits efficient support for program mutation via information and code generated at compile-time, hence, we call this a compiler-integrated approach.

2 A simple compiler

The set of mutations appropriate for a given program is defined in terms of its source code representation. Therefore, one approach to program mutation is to generate mutant programs via source level edits to the original program, compile those mutant programs, and execute them on the appropriate test data. However, compiling each mutant individually suffers from redundancy because, by definition, each mutant is very similar to the program under test. This observation leads directly to a compiler-integrated approach, as illustrated in Figure 2. The program P under test is input to the compiler which generates two outputs: an executable object code image for P, P_x, and a set of patches, P_pat. Each patch consists of one or more code sequences and a corresponding set of editing operations directing how the code sequences are to be
applied to $P_{exec}$. When a patch is applied to $P_{exec}$, the result is an executable object code image for a mutant of $P$. As described later in section 3, the application of patches to $P_{exec}$ is an efficient process with little overhead. Therefore, a compiler-integrated approach avoids the overhead incurred by individually compiling mutant programs while retaining the advantage of execution at machine speeds on the native processor. It is believed that this approach will afford a significant increase in the execution speed and cost-effectiveness of a mutation-based software test.

The method presented here is applicable to any Algol-like block-structured programming language. The target processor is assumed to be based on a classical Von-Neuman style architecture supporting a segmented memory organization. Memory is assumed to be addressable via offsets and the target machine code relocatable.

For simplicity, it is assumed that compilation occurs in the distinct phases illustrated in Figure 3. The source code of the program under test is denoted by $P$, while $P_{tok}$, $P_{int}$, $P_{asm}$, $P_{obj}$, and $P_{exec}$ represent $P$ at the token, intermediate code, assembly code, object code, and executable code levels, respectively. The symbol table for $P$ is denoted by $P_{sym}$. In a production compiler, the organization may vary, phases may be interleaved, optimization phases may be present, and separate compilation may be supported [3]. However, this simple organization is appropriate to illustrate the ideas presented here.

We assume that one or more source code files, each containing one or more functions, are submitted to the compiler. The result of compilation is a relocatable machine executable version of the program to be compiled. Figure 4 presents an algorithm for compilation of source files within this framework. The token stream, parse tree, intermediate code, assembly code, object code, and symbol table for the files and functions comprising $P$ are analogous to those for $P$ (i.e.: $s_{tok}$, $f_{tok}$, $s_{tree}$, $f_{tree}$, etc.).

Functions are compiled to assembly code on a per function basis. A function $f$ is lexically, structurally, and semantically analyzed in one interleaved phase by the function $Parse(f)$. This first phase results in the construction of a parse tree, $f_{tree}$, and augmented symbol table, $s_{sym} \cup f_{sym}$. The intermediate code, $f_{int}$, is generated by a traversal of $f_{tree}$ in $IntCodeGen(f_{tree}, s_{sym})$, while the assembly code, $f_{asm}$, is generated by a subsequent traversal of $f_{int}$ in $AsmCodeGen(f_{int}, s_{sym})$. Once $f_{asm} = \cup_{f \in s_{sym}} f_{sym}$ is available, $s_{obj}$ is generated by $Assemble(f_{asm})$. Finally, the collective results of assembly over all source files $s$, $P_{obj} = \cup_{s \in s_{obj}}$, and the global symbol table, $P_{sym} = \cup_{s \in s_{sym}}$, are link/edited by $LinkEdit(\cup_{s \in s_{obj}}, \cup_{s \in s_{sym}})$ to produce the end result, $P_{exec}$. Although we assume no optimizations occur during this process, both local and global register allocation is permitted.

<table>
<thead>
<tr>
<th>Mutation</th>
<th>Statement in $P$</th>
<th>Statement in Mutant of $P$</th>
<th>Change to $P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrong Constant Fault</td>
<td>$a = b + 5$</td>
<td>$a = b + 0$</td>
<td>$5 \rightarrow 0$</td>
</tr>
<tr>
<td>Wrong Operand Fault</td>
<td>$a = b + c$</td>
<td>$b = b + c$</td>
<td>$a \rightarrow b$</td>
</tr>
<tr>
<td>Wrong Operator Fault</td>
<td>$a = b + c$</td>
<td>$a = b + c$</td>
<td>$+ \rightarrow *$</td>
</tr>
<tr>
<td>Ineffective Computation Fault</td>
<td>$a = b + c$</td>
<td>$a = b + c$</td>
<td>$- \rightarrow +$</td>
</tr>
</tbody>
</table>

Figure 1: Examples of mutations applied to programs written in the C programming language.

$$f_{sym} := \emptyset$$

for each source file $s$ do

for each function $f$ in source file $s$ do

\{ $f_{tree}, s_{sym} \cup f_{sym}$ \} := $Parse(f)$

\begin{align*}
&f_{int} := IntCodeGen(f_{tree}, s_{sym}) \\
&s_{asm} \cup f_{asm} := AsmCodeGen(f_{int}, s_{sym})
\end{align*}

end

$s_{obj} := Assemble(s_{asm})$

end

\{ $P_{exec} := LinkEdit(\cup_{s \in s_{obj}}, \cup_{s \in s_{sym}})$ \}

Figure 4: An algorithm for compilation of a multi-file source program.

$\tau(f, \tilde{P}) = (\sigma_f, \tilde{P})$
where \( \sigma_i \) and \( \sigma_f \) are the initial and final states in the translation, respectively. At each stage in the compilation from the source to target representation, assume the translation is performed in steps. Let \( \sigma_{i-1} \) represent the state of the compilation prior to translation at step \( s_i \).

For example, one may view the translation of \( P \) from source to intermediate code as occurring on a statement-by-statement basis. The source and target languages of the translation are not important, nor is the granularity of each translation step. However, capturing the state of the translation at each step is important. In terms of a compiler about to translate the \( i^{th} \) statement in \( P \), \( \sigma_{i-1} \) is intended to capture all facets of the internal state of the compiler upon which subsequent generation of correct target instructions is dependent. If this internal state is consistently maintained and target instructions correctly generated at each stage in the compilation, the semantics of the resulting target program must be correct.

At any stage, let the translation of a program \( \hat{P} \) consist of \#s steps. Then \( \sigma_0 \) and \( \sigma_{\#s} \) represent the initial and final states in one stage of the compilation process. Let

\[
\tau_i = \tau(\sigma_{i-1}, \hat{w}_i) = (\sigma_i, \hat{w}_i), \quad 1 \leq i \leq \#s,
\]

where \( \hat{w}_i \) is the input string to be scanned at step \( i \) and \( \hat{w}_i \) is the sequence of target instructions generated. Then the translation of \( \hat{P} \) to \( \hat{P} \) consists of the sequence

\[
(\sigma_f, \hat{P}) = \tau_{\#s} \circ \ldots \circ \tau_2 \circ \tau_1 \circ \tau_0 (\sigma_0, \hat{P}).
\]

Each program mutation is defined in terms of a syntactically correct transformation on the constructs of a source language \( L \). Therefore, each patch corresponding to a mutant of program \( P \) can be represented by edits on representation \( \hat{P} \) that result in corresponding changes to the \( L \) instructions of \( \hat{P} \). Each edit ripples through all stages of the translation process and are manifested in the final target representation. For each mutant program, the input \( \hat{w}'_i \) at step \( i \) will be transformed to \( \hat{w}'_i \), resulting in a new translation state \( \sigma'_i \), where \( \sigma'_i = \sigma_i \) may not be true:

\[
\tau'_i = \tau(\sigma_{i-1}, \hat{w}'_i) = (\sigma'_i, \hat{w}'_i), \quad 1 \leq i \leq \#s,
\]

<table>
<thead>
<tr>
<th>( P )</th>
<th>Lexical ( P_{tok} )</th>
<th>Structural/Intermediate ( P_{intr} )</th>
<th>Semantic ( P_{int} )</th>
<th>Code</th>
<th>Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow )</td>
<td>Analysis</td>
<td>( \rightarrow ) Semantic</td>
<td>( \rightarrow ) Code</td>
<td>( \rightarrow ) Generation</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>( P_{int} ) Assembly ( P_{asm} ) Object ( P_{obj} ) Link/Editing ( P_{exec} )</td>
<td>...</td>
<td>( \rightarrow ) Code</td>
<td>( \rightarrow ) Code</td>
<td>( \rightarrow ) Editing</td>
</tr>
</tbody>
</table>

Figure 2: A compiler-integrated approach to program mutation.

Figure 3: Phases in a simple compiler.
We want to integrate the compilation of all mutant programs and avoid the inefficient alternative of individual mutant compilation. However, how can the simultaneous compilation of mutants occur if program mutations result in new states of compilation?

Denote the difference between the two states \( \sigma_i \) and \( \sigma'_i \) by \( \delta (\sigma_i, \sigma'_i) \) and define \( \tilde{\tau}_{iv} : \Sigma \rightarrow \hat{L} \times \Sigma \) such that

\[
\tilde{\tau}_{iv} = \tilde{\tau} (\sigma_i, \sigma'_i) = (\sigma_i, \tilde{\omega}_i (\sigma_i, \sigma'_i)).
\]

The function \( \tilde{\tau}_{iv} \) is a state transition function that, given the states \( \sigma_i \) and \( \sigma'_i \), generates target language instructions for \( \hat{L} \) such that the internal state of the compilation is converted from state \( \sigma'_i \) to state \( \sigma_i \).

Let \( \text{Mut}(\hat{P}) \) represent all possible mutations for a program \( \hat{P} \) and let \( \text{Pat}(\hat{P}) \) represent the collection of all mutant patches corresponding to mutations in \( \text{Mut}(\hat{P}) \). We utilize \( \tilde{\tau}_{iv} \) to integrate the generation of mutant program patches into a single compilation of program \( \hat{P} \) as presented in the algorithm in Figure 5.

\[
\text{Pat}(\hat{P}) := \emptyset \\
\text{for each step } s_i, 1 \leq i \leq \#s \text{ in the translation of } \hat{P} \text{ do} \\
\quad \text{Save the state } \sigma_{i-1} \\
\quad \text{Compute } \tilde{\tau}_i \text{ for input } \tilde{\omega}_i \text{ in } \hat{P} \\
\quad \text{Save the state } \sigma_i \\
\quad \text{for each mutant } \tilde{\omega}'_i \text{ of } \tilde{\omega}_i \text{ in } \text{Mut}(\hat{P}) \text{ do} \\
\quad \quad \text{Restore the state } \sigma_{i-1} \\
\quad \quad \text{Compute } \tilde{\tau}'_i \text{ for } \tilde{\omega}'_i \text{ in mutant program } \hat{P}' \\
\quad \quad \text{Compute the state transition } \tilde{\tau}_{iv} \\
\quad \text{end} \\
\quad \hat{P}_\text{mut} \cup \tilde{\tau}_i \| \tilde{\tau}_{iv} \text{ Restore the state } \sigma_i \\
\text{end}
\]

Figure 5: An algorithm for generation of mutant program patches.

The translation function \( \tilde{\tau}_i \) and transition function \( \tilde{\tau}_{iv} \) perform the real work here. The compiler cannot efficiently manage all states in all compilations of all mutant programs (e.g., this is equivalent to separately compiling each mutant). Our approach is to modify the input stream \( \tilde{\omega}_i \) at step \( i \) to reflect a mutation \( \tilde{\omega}'_i \) in some mutant program \( \hat{P}' \). For any mutation, all steps in the compilation of an original and mutant program are identical, up until the point at which the mutation occurs in the source representation of the original program. At this point, on input \( \tilde{\omega}'_i \) the function \( \tilde{\tau}'_i \) results in the generation of target instructions \( \tilde{\omega}'_i \) that reflect this mutation within the target representation of the mutant program and a new state \( \sigma'_i \). However, the remaining steps in the compilation of the mutant program are abandoned. Instead, the function \( \tilde{\tau}_{iv} \) is utilized to generate target instructions that augment those generated by \( \tilde{\tau}'_i \). These instructions are designed to restore the state of the compilation from \( \sigma'_i \) to \( \sigma_i \).

This permits a mutant program to utilize the target instructions subsequently generated during the compilation of the original program because the mutation (1) only differs in a change to statement \( s_i \), and (2) the states of the compilation of the mutant and original programs, prior to subsequent code generation, are equivalent.

An example

Assume the program \( P \) being mutated contains the variables \( a, b, c, x, y, z \) and the assignment statement \( a = b + c \). To simplify this example, let the compiler translate \( P \) from source code to assembly code (i.e.: \( \hat{P} = P \) and \( \bar{P} = P_{\text{asm}} \)) and let the level of granularity of each translation step be one source-level statement. Then, for any statement \( s_j \) in \( P \), the compiler is in predecessor state \( \sigma_{i-1} \) immediately prior to the translation of \( s_j \), and successor state \( \sigma_i \) immediately following translation of \( s_j \). Let any state be composed of only two types of information \([3]\):

\[
V: \text{ A mapping of variables to locations (e.g., the current value of the variable } a \text{ is located in memory and register } R0); \\
R: \text{ A mapping of registers to variables (e.g., register } R0 \text{ holds the current value of the variables } a \text{ and } b). \\
\]

For example, the predecessor state \( \sigma_{i-1} \) prior to the translation of \( a = b + c \) at statement \( s_i \) may be represented by:

<table>
<thead>
<tr>
<th>( V )</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( x )</th>
<th>( M )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>M</td>
</tr>
<tr>
<td>( b )</td>
<td>c</td>
<td>y</td>
<td>z</td>
<td>M</td>
<td></td>
</tr>
</tbody>
</table>

Here, only the mappings for variables \( a, b, c, x \) and registers \( R0,R1,R2,R3 \) have been depicted. Given the state \( \sigma_{i-1} \) above and the input \( \tilde{\omega}_i \) as \( a = b + c \), one possible instruction sequence \( \tilde{\omega}_i \) that may be generated by \( \tilde{\tau}_i \): 1

\[
\text{ADD } R0, R1
\]

along with the successor state \( \sigma_i \):

\[1\text{The simple assembly-level instructions depicted here have the format "op src, dest," where dest = src + dest.} \]
In order to comply with \( \sigma_i \), the state transition function \( T_{il} \) is invoked to generate the instructions \( \omega_i \):

\[
\begin{align*}
\text{MOV} & \quad a, R1 \\
\text{MOV} & \quad R0, b
\end{align*}
\]

This results in a final mutant patch of

\[
\begin{align*}
\text{ADD} & \quad R1, R0 \\
\text{MOV} & \quad a, R1 \\
\text{MOV} & \quad R0, b
\end{align*}
\]

while leaving the internal state of the compiler as \( \sigma_i \), as desired.

A mutant program that has been obtained by patch application is dependent upon the instructions generated for statements \( s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_{p2} \) in \( P \). Only the sequence of instructions for statement \( s_i \) in \( P' \) are generated for the mutant, and only these instructions are applied to the executable image of \( P \). Of the instructions generated during the compilation of the remainder of the program \( P \), many may depend on the fact that the current value of \( a \) is in \( R1 \) and the current value of \( b \) is in \( R0 \). Because the instructions generated by \( T_{il} \) leave the state of the mutant program's compilation equivalent to \( \sigma_i \), these instructions will be correct in a patched mutant program.

This example shows how the instruction sequences for mutant patches consist of the target instructions generated by both \( T_{il} \) and \( \tau_{il} \). These functions allow integrated compilation of all program mutations while correctly preserving the state of the compilation and generating instructions that, when applied to the program \( P \), are consistent with the semantics of the source-level representation of a mutant program; had it been accepted as input and compiled directly. Because mutations comprise very localized changes to a program, we can expect that the code generation functions \( T_{il} \) and \( \tau_{il} \) exist and can be incorporated into a compiler for programs of given source language \( L \).

### 3 Applying patches

Assume a program \( P \) has been compiled and its patches generated. During subsequent mutation testing, these code patches are threaded into the instruction stream of a program \( P \) in order to obtain its mutants. This is the process of patch application and is the function of the patch applicator. Each patch consists of a sequence of editing operations and their associated instruction sequences. The patch applicator interprets the editing operators described by a patch and edits the machine executable image of \( P \) by installing new instruction sequences to obtain a specific mutant program. This process is depicted in Figure 6.

At each location code replacement is to occur, instruction sequences are threaded into the program \( P \) via the insertion of jump instructions. The address at which any instruction sequence is to be inserted is known. Applying an instruction sequence at this address requires overwriting the existing instruction with a jump to an alternative location where a patch's instruction sequence(s) have been installed. Each instruction sequence is suffixed by an unconditional jump returning control to the appropriate location in program \( P \).

### 4 Conclusions and future work

In this paper we have presented a method to integrate support for program mutation into a compiler. The method is both efficient and sufficiently powerful to support program mutation software testing. Moreover,
existing research suggests that this approach appears to be essential for the cost-effective application of program mutation to testing large commercial software systems [6].

We are currently building a test environment based on a compiler-integrated approach. At the heart of this environment is the GNU C compiler which has been modified to generate patches as described in this paper. In addition to the compiler, the environment consists of a patch applicator, test case editor and manager, test display manager, and a data base for storing the status of the software test. The modified compiler is also being interfaced with PMothra [5], a tool for scheduling mutants on the Neube/2 hypercube.

It is believed that a compiler-integrated approach will provide a significant increase in the efficiency of several existing testing tools and allow program mutation to be effectively employed to test commercial software systems. This hypothesis has yet to be empirically substantiated. However the approach seems to be innately appealing to the software tester because it also enhances the reliability of a software test. The program under test retains much of its original operational behavior (e.g., timing characteristics, while executing in its intended operational environment.

References