MuGamma: Mutation Analysis of Deployed Software to Increase Confidence and Assist Evolution

Sang-Woon Kim
Division of Computer Science
Department of EECS
KAIST, Korea
swkim@salmosa.kaist.ac.kr

Mary Jean Harrold
College of Computing
Georgia Institute of Technology
Atlanta, Georgia
harrold@cc.gatech.edu

Yong-Rae Kwon
Division of Computer Science
Department of EECS
KAIST, Korea
kwon@cs.kaist.ac.kr

Abstract

This paper presents a novel approach to unit testing that lets users of deployed software assist in performing mutation testing of the software. Our technique, MuGamma, *provisions a software system so that when it executes in the field, it will determine whether users' executions* would *have killed mutants (without actually executing the mutants), and if so, captures the state information about those executions. In the absence of bug reports, knowledge of executions that would have killed mutants provides additional confidence in the system over that gained by the testing performed before deployment. Captured information about the state before and after execution of units (e.g., methods) can be used to construct test cases for use in unit testing when changes are made to the software. The paper also describes our prototype* MuGamma *implementation along with a case study that demonstrates its potential efficacy.*

1 Introduction

Mutation testing is a powerful technique for unit testing of software, whose fundamental premise is that "if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects the fault" [4, 16]. Empirical studies support the effectiveness of mutation testing as a criterion for unit testing (e.g., [3, 13, 15]). Thus, providing unit test suites that are mutation-adequate can increase confidence in the quality of the software under test.

Despite general acceptance of the effectiveness of mutation testing, it has not been adopted in practice. Offutt and Untch discuss three primary reasons for this lack of use in practice [16]. The first reason—the lack of economic incentives for stringent testing—may no longer apply. The pervasiveness of software has led to increased demand for high-quality systems, which will result in increased economic incentives for such stringent testing. The other two reasons—the inability to successfully integrate unit testing into software development processes and the difficulties with providing full and economical automated technology to support mutation analysis and testing—do still apply. Although there is increased interest in improving the quality of software, time-to-market pressures and limited development resources will continue to prohibit integration of effective unit testing into the development process. Furthermore, even with advances in reducing the computation cost of mutation testing, the complex technology required to support mutation analysis and testing, such as automatic test-case generation, identification of equivalent mutants, and processing the large number of mutants, will continue to inhibit development of economical automated support technology. Thus, although mutation testing may help to improve the quality of the software, it may not be used for unit testing of software before deployment.

Lack of effective testing during development results in systems that are deployed with defects in implementation, missing functionality, incompatibilities with complicated running environments, or inferior usability. To address these problems and to continue to improve these systems, there is a need for, and a growing interest in, monitoring the systems after they are deployed. Such monitoring can be used to perform tasks, such as detecting anomalous behavior (e.g., [1, 5]), finding and fixing defects (e.g., [9, 19, 23]), determining the impact of potential changes on users or groups of users (e.g., [18, 22]), and improving the quality of the regression test suite (e.g., [18]).

Such monitoring could also be used to perform mutation analysis as the software is executing in the field to (1) improve confidence in the deployed software and (2) gather test cases for use in regression testing. To improve confidence in the deployed software, monitoring could augment mutation testing (if any) that was performed before deployment by identifying and reporting mutants that *would* be killed by executions of the software in the field. Mutation testing before deployment guides the creation of a test suite that improves confidence in the software. Similarly, in the absence of bug reports from the field, mutation testing after deployment records executions that provide the same "mutant-killing" power, and thus improves confidence in the software. To gather test cases from users' executions, monitoring could also capture state information for those units (e.g., procedures or methods) associated with mutants that would be killed during executions, and use them to create unit test suites. These test suites, possibly mutation-adequate, could then be saved and used to perform regression testing when the software is changed.

1.1 Mutation Testing and Remote Monitoring

Many researchers have developed techniques to reduce the cost of mutation testing. One approach, the Mutant Schemata Generation (MSG) technique [27], encodes all mutants into one source-level program. At each *change point* (a point where a mutation operator can be applied), the program can be executed as a mutant. A *mutant descriptor* indicates which mutation point is to be enabled and which alternative is to be applied for that execution. By incorporating all mutants into one program, the technique reduces the number of mutants that need to be compiled dramatically, and makes mutation-testing tools less complicated and easier to build. Furthermore, once the encoded program is compiled, it is executed at compile speeds. Thus, it achieves its mutation-testing goal faster than previous mutation-testing techniques. Offutt and Untch propose a general mutation testing process using the MSG method that further reduces the cost of mutation testing by generating an encoded mutant program only for selective-mutation operators [17]. Ma and colleagues proposed MUJAVA, an MSG-based mutation-testing technique, that includes operators targeted at object-oriented features of Java programs [11]. Although the MSG approach can reduce the cost of mutation testing, it cannot be used directly for mutation testing of deployed software because it rewrites the program so that it can be run as mutants. Mutation testing in the field must simulate the execution of mutants so that it can report those that would be killed by executions.

Another approach to reducing the cost of mutation testing is weak mutation, which was originally proposed by Howden [7]. Instead of comparing results of entire executions to determine whether the original and mutant programs differ, thus, killing the mutant using *strong mutation*, *weak mutation* inspects the states of the original and mutated programs at some intermediate state and deems the mutant killed if these states differ. Weak mutation has been studied theoretically (e.g., [6]), and studies suggest that it can perform well as an indicator of strong mutation (e.g., [12, 14]). Like the MSG approach, weak mutation actually executes the mutants instead of simulating their execution. However, various forms of weak mutation could be used to reduce the simulation of mutants in the field.

Previous research has developed techniques for monitoring software after deployment. Orso and colleagues [21] present a technique, GAMMA,[1] that uses a process called *software tomography* that divides the monitoring task into subtasks. These subtasks are then assigned to different instances of the deployed system, monitored as the system executes, and combined to get a global view of the executing system. Liblit and colleagues [9] present a different approach that uses statistical sampling of software after deployment to gather information that is then used in fault localization. Although studies suggest the efficacy of these techniques for gathering and using execution data for deployed systems [8, 9, 18], they cannot be used directly for mutation testing of deployed software. These techniques record information about values of variables or coverage of program entities during execution. In contrast, monitoring for mutation testing must determine whether the mutant would have been killed, without actually permitting the execution of the mutant, which is a more complex analysis.

Previous research has also resulted in techniques for creating unit test cases from the execution of system (whole-program) test cases—a form of *capture-replay*. Orso and Kennedy [20] present a selective capture-replay technique that records execution actions that are required to replay designated units in isolation. Saff and colleagues [24] present a similar approach that creates mock objects that record the actions required to replay the designated units. Elbaum and colleagues [2] proposed a capture-replay technique that gathers state information, instead of actions, required to replay the execution. They also present an application of their technique that captures the state before and after execution of each method in a Java program, and then use these states to create unit test cases for regression testing. To date, none of the existing techniques has been applied to deployed software. Furthermore, although these techniques could be used to capture required information that can be saved and reused for regression testing, they capture information about all executions for every method. Thus, the techniques create large test suites that may contain many useless or redundant test cases with respect to a testing criterion, such as the mutation-testing criterion.

1.2 Our Remote Mutation-Testing Technique

To facilitate mutation testing of deployed software and to create unit test cases for use in regression testing, we de-

[1]We use the name Gamma for this type of testing and analysis to represent testing and analysis performed after Alpha and Beta testing and analysis are completed. Whereas Alpha is performed in house and Beta is performed on a subset of real users, Gamma is performed on all users in real execution environments.

veloped, and present in this paper, a novel specialization of the GAMMA technology. Our specialization, which we call MUGAMMA, performs mutation testing in the field and captures only useful test cases for regression testing. Like GAMMA's software-tomography approach, MUGAMMA divides the task of mutation analysis into subtasks—each of which is responsible for a set of mutants—and assigns these subtasks to different instances of the deployed system. Unlike the GAMMA approach, MUGAMMA does not simply monitor for coverage of certain aspects of the execution—users of our instrumented software would not tolerate crashes or incorrect output caused by executing actual mutated statements. Instead, MUGAMMA creates versions of the software that perform analysis and determine whether an execution *would* have killed a particular mutant or set of mutants without actually executing the mutant (s) on the user's site. Using a capture-replay approach, similar to Elbaum and colleagues' [2], MUGAMMA records the states for those executions that kill mutants, and uses them to create test cases for use in regression testing.

There are several benefits of MUGAMMA. First, it lets the developer gain confidence in the system. In the absence of bug reports, knowledge of executions that would have killed mutants provides additional confidence in the system over that gained by testing performed before deployment. Second, in performing mutation testing after deployment, some costs of mutation testing are mitigated. Tasks such as generation of test data and execution of large numbers of mutants are replaced by users' executions and MUGAMMA's ability to use these executions to determine whether they would have killed mutants. Third, MUGAMMA facilitates recording of the state—before and after execution—for those inputs, at the unit level, that would have killed a mutant(s). The test cases created from these states are then sent back to the developer and combined to create a test suite for use in regression testing.

The paper also presents our prototype MUGAMMA System and discusses a case study performed on a Java program. Our prototype system is a specialization of the GAMMA framework to accommodate mutation testing and capture of test cases. The core logic of the mutation-testing specialization is based on MUJAVA, an object-oriented mutation testing system [10, 11] that uses the MSG technique [26, 27] to generate mutants.

The main contributions of the paper are:

- a specialization of the GAMMA System, called MUGAMMA, that adapts a previous mutation testing technique to provide mutation analysis of deployed applications;
- a technique that adapts existing capture-replay techniques to gather test cases for use in regression testing;
- a prototype implementation of MUGAMMA that uses an efficient execution method for mutation testing consisting of wrappers and mutant schemata.

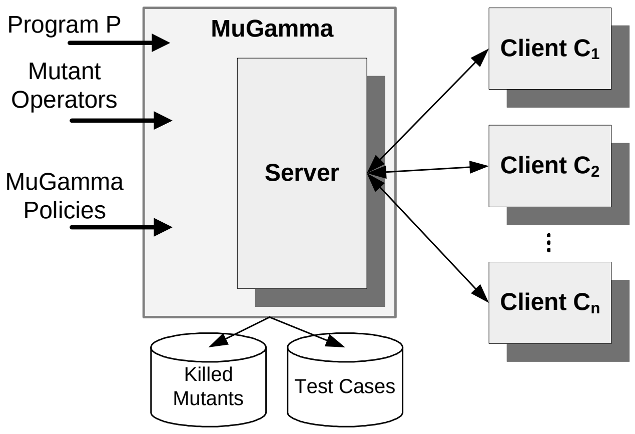

Figure 1. User's interaction with MUGAMMA.

2 Using the MUGAMMA System

Consider a MUGAMMA user who wants to deploy her program so that, during executions of the program in the field, mutation testing will continue to be performed on it and test cases for use in regression testing can be created. The user can employ MUGAMMA to provision the program so that when it is deployed to different clients, those client executions will record and report information about mutants that would have been killed, and create and return test cases from state information about clients' executions.

Figure 1 depicts such a use of MUGAMMA for a program *P* and set of clients $\{C_1, C_2, ..., C_n\}$ who use *P* and will assist in the mutation testing.[2] The user has three inputs to MUGAMMA: a program *P*, a set of Mutant Operators, and a set of MUGAMMA Policies. *P* is the program that the user wants to be provisioned for monitoring and along with P, she can specify those parts of P for which she wants mutants to be monitored. The *Mutant Operators* specify the mutant operators for which she wants *P* to be monitored during execution. The *MuGamma Policies* specify the user's selected customization to create a MUGAMMA server that will interact with the clients to manage activities such as assignment of mutants to the clients, gathering of reports from the clients, and creation of test cases from clients' executions. The results will be the killed mutants and a set of unit test cases created from the executions in the field.

There are four user-designated Policies[3] that guide the customization of the monitoring of *P* when it is deployed on C_i's sites: grouping, assignment, reporting, and completion. The *grouping* policy specifies how the mutants will be grouped for a particular instance of *P*. For example, all mutants for the same method might be grouped so that they will be assigned to and monitored by the same instance. The *assignment* policy specifies how the mutants will be assigned

[2]Of course, the user will need the clients' permissions for participation.
[3]MUGAMMA can also provide a default set of Policies.

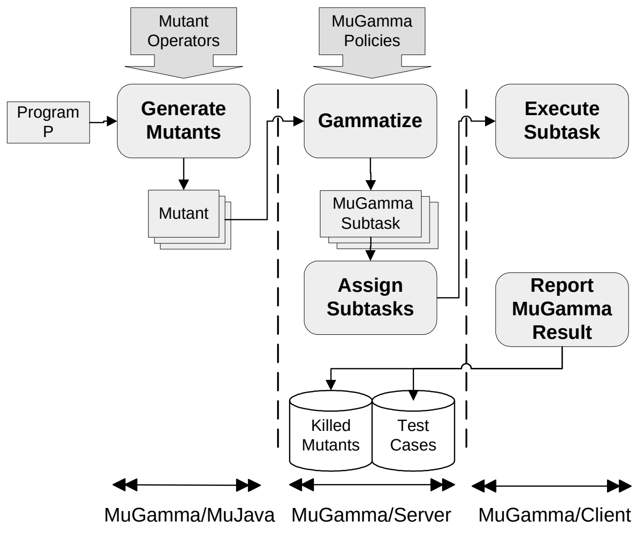

Figure 2. Components of the MUGAMMA system.

as the C_is request *P*. For example, mutants or groups of mutants could be assigned to instances in a round-robin fashion or they could be assigned by the number of mutants that are still live. The *reporting* policy specifies when the clients will report information about the mutation testing back to the user's site. For example, the policy could specify time intervals or occurrences of events, such as killing of mutants. Finally, the *completion* policy specifies the criterion for completion of the monitoring of the group assigned to the client. For example, a group could complete when all mutants in the group are killed or when some specified percentage of mutants are killed.

Section 3.2 provides additional details of the policies including a complete set of current options.

3 Specializing GAMMA for Mutation Testing

There are three main components of the MUGAMMA system, as shown in Figure 2. The first component, Generate Mutants, inputs the program *P*, along with the user's specification of which parts of *P* are to be mutated, and the Mutant Operators specified by the user, and generates a version of the program that encodes the mutants. Section 3.1 provides details of this component.

The next component, Gammatize, inputs the program with the mutants encoded, and groups them into a set of MUGAMMA subtasks according to the Policies specified by the user. This step also creates the customized MUGAMMA Server using these Policies. Section 3.2 provides details of this component.

The third component, Execute Subtask, executes the gammatized *P* in the field, determines which mutants have been executed, and creates test cases for use in regression testing. Section 3.3 provides details of this component.

3.1 Generating Mutants for MuGamma/MuJava

MUGAMMA cannot actually encode mutants in the program for deployment. Thus, it must rewrite the code to create versions wherein the execution of the deployed program can determine whether a mutant would be killed without actually executing the mutant on the clients' sites. On reaching a method m that is to be mutated, the new version of the program will save the state before execution of m, execute m, and save the state after execution of m. Then, it will use the before state of m to set the state for execution of the mutant, m_{mut} of m, execute m_{mut}, compare the after states of m and m_{mut}, and, if the states differ, report that mutant as killed and save the before and after states for use as a test case. This section presents the details of the rewriting of the program to create these versions.

3.1.1 Capturing state for replay

To accommodate the additional behavior that the provisioned program must possess, we rewrite the code to include wrappers to the methods on which mutation is to be performed. These wrappers will receive messages sent to the method and perform the required activities. To perform these activities, we use three wrappers—a controlling wrapper, a generating wrapper, and a comparing wrapper.

The *controlling wrapper* has the same method signature as the target method m and thus, when m is called during execution, this wrapper will be called instead. The main functionality of the controlling wrapper is to call the code to execute mutants, if mutants assigned to that client are in m, and call the original code for m otherwise. To distinguish the original code for m from the wrapper code, m is rewritten to have the same signature and code but is given a new name—*m_original*. Mutants are distinguished by change points, which designate a point in the program where a mutant operator could be applied (and are described in more detail in the next section). Because a method can have many change points, each change point has its own identifier, ID. If the controlling wrapper determines that ID in method m matches some mutant in the group assigned to this client, the generating wrapper is called to continue the activities of the mutation testing.

The *generating wrapper* also has the same method signature as m except that it is given the name *m_gen_wrapper*. *m_gen_wrapper* saves the state, including parameters, before execution of m's code as S_{pre}. The wrapper then calls *m_implemented*, which has the same body as the original method m except that all change points are replaced, using the MSG technique, with method calls that handle the mutation; this replacement is described in Section 3.1.2. After execution of *m_implemented*, the return value and state are saved as S_{post}

```
void m(param1, param2) {
  ...
  a = b + c; --> change point CP1 for AORB
  ...
  a = a + 1; --> change point CP3 for AORB
  ...
}
```

Figure 3. Original method m to be mutated.

```
void m(param1, param2) {
  switch(ID) {
  case CP1 :
  case CP3 :
    m_gen_wrapper(param1, param2);
    break;
  default :
    m_original(param1, param2);
  }
}
void m_gen_wrapper(param1, param2) {
  save S_pre;
  m_implemented(param1, param2);
  save S_post;
}
void m_compare_wrapper() {
  restore S_pre;
  m_mutated(param1, param2);
  compare with S_post;
}
```

Figure 4. The three wrappers generated for m.

The third wrapper, the *comparing wrapper*, is called for each mutant, and facilitates execution of the mutant. This wrapper, *m_compare_wrapper*, sets the state for the execution of the mutant as S_{pre} (i.e., restores the state before m was executed) and calls another method, *m_mutated*, that executes the mutant. *m_compare_wrapper* executes in a separate virtual machine for each mutant. After executing the mutant, the wrapper compares the resulting state with S_{post}. If they differ, the mutant is reported as killed and both the S_{pre} and the S_{post} are saved for use as a test case.

To illustrate the wrappers, consider the original code for a method m that has two change points, CP1 and CP3, shown in Figure 3. Now consider the three wrappers for m, shown in Figure 4. During execution, the client code calls the controlling wrapper (m in Figure 4). The controlling wrapper first checks whether any of the IDs assigned to that client match the change points of m. If the IDs match any change points, the wrapper calls the generating wrapper, *m_gen_wrapper*. Otherwise, the wrapper calls the original code, *m_original*. When *m_gen_wrapper* is called, it saves the state before m executes, S_{pre} and calls *m_implemented*, which executes m's original code and prepares and returns information for the execution of the mutants by the comparing wrapper, *m_comparing_wrapper*. When the simulation of the mutants occurs, *m_compare_wrapper* executes the mutated code with the saved S_{pre} from m's execution as

```
void m_original(param1, param2) {
  ...
  a = b + c;
  ...
  a = a + 1;
  ...
}
void m_implemented(param1, param2) {
  ...
  a =(ID==CP1)?AORBGen(b,c,'+'):b + c;
  ...
  a =(ID==CP3)?AORBGen(a,1,'+'):a + 1;
  ...
}
void m_mutated(param1, param2) {
  ...
  a =(ID==CP1)?AORB(b,c,SM.op):b + c;
  ...
  a =(ID==CP3)?AORB(a,1,SM.op):a + 1;
  ...
}
```

Figure 5. Original, implemented, and mutated versions of m.

input, and compares the resulting state with the saved S_{post} from m's execution.

3.1.2 Encoding the mutants

Our encoding of mutants is based on MUJAVA [11], an existing tool for generating mutants for Java programs, and on the MSG technique [26, 27]. The MSG technique translates the program so that it encodes all selected mutants of a statement into a function call—a *metamutant*—that can act as any of the mutants. The program with all statements changed to metamutants is called a metaprogram. During execution, a mutant descriptor designates which mutant should be executed. Thus, the metaprogram along with a mutant descriptor specifies a particular mutant.

To illustrate, consider one mutant operator—Arithmetic Operator Replacement for Binary or AORB. With AORB, for a statement s containing an arithmetic binary operator, such as addition, subtraction, or multiplication, a mutant is created by replacing the operator in s with each of the other arithmetic binary operators. Thus, a statement "A=B+C" could have mutants "A=B-C" or "A=B*C." The MSG approach creates a metamutant that encodes all these mutants into one function call and contains a parameter, *chpt*, indicating the change point in the program from which AORB is called.[4] Thus, "A=B+C,", would be changed to "A = AORB(B, C, '+', chpt)". An associated mutant descriptor, which specifies the change point and the alternative that is to be executed at that point, instantiates a mutant.

Our approach for generating mutants is an adaptation of the MSG approach. Like the MSG approach, we use

[4]We provide an intuitive description of the MSG method; details of the method can be found in Untch's dissertation [26].

function calls to perform the mutation. Unlike the MSG method, we place these function calls in the code called by the generating and comparing wrappers. Also, unlike the MSG method, we use two function calls to perform the mutation: one is the metamutant that is called to execute a mutant, and is similar to the metamutants generated by the MSG method; the other is the dynamic mutant generator that performs some initial filtering of potential mutants and generates metamutants for those that survive the filtering.

The *dynamic mutant generator* is called by the generating wrapper through *m_implemented*. This function executes the original code for the statement, but it also evaluates each mutant at the point immediately after the statement is executed (i.e., *very weak mutation*). Because the operands have already been evaluated to execute the original statement, this evaluation using the other mutant operands cannot cause side effects. If this very weak mutation at the change point produces differences in the original and mutated code, the results are returned to the client for processing by the simulator.

When the *metamutant* is called by the comparing wrapper through *m_mutated*, the mutants encoded by that metamutant are executed. The function executes the particular mutant until the end of the method. Only those mutants that were very weakly killed during the *m_implemented* call are reconsidered. During execution of *m_mutated*, the mutated method is executed until its end, and the states of the original and mutant are compared to see if the mutant is killed. This additional execution to the end of the method provides more confidence that the mutant would be strongly killed than just considering the very weakly killed results.

As an example, consider again method m of Figure 3, its variations in Figure 5, dynamic mutant generator AORBGen in Figure 6, and the metamutant for AORB in Figure 7. We can now see how the original statements in method m (Figure 3) are changed to create *m_implemented* and *m_mutated*. *m_original* contains the original version of m, which shows two statements that will be mutated. *m_implemented* contains the statements in m but the MSG technique has been applied to them to create a function call to AORBGen(). Function AORBGen() performs as indicated in Figure 6. When AORBGen() is called (i.e., by *m_implemented*), the original code is executed and all the mutants of that statement are evaluated. For any of these mutants that differ after this evaluation (i.e., are very weakly killed), those mutants are reported to the client for later execution by the simulator. When AORB() is called by *m_mutated*, the mutated code is executed and a mutant of that statement, one of reported mutants to the client, is evaluated. If the state after execution of the mutated code differs from that of original code, the execution information and mutant is reported to the Client.

Algorithm 3.1: AORBGen(lEx, rEx, op)

```
$Ops = \{Plus, Minus, Multi, Div, Mod\}$
$original \leftarrow \begin{cases} lEx + rEx & \textbf{if } op = Plus \\ lEx - rEx & \textbf{if } op = Minus \\ lEx * rEx & \textbf{if } op = Multi \\ lEx \,/\, rEx & \textbf{if } op = Div \\ lEx \,\%\, rEx & \textbf{if } op = Mod \end{cases}$
for each $t \in Ops$
    if $op \neq t$
        then
            SubMutant M = new SubMutant()
            $result$ = evaluate $t(lEx, rEx)$
            M.setResult($result$)
            if $result \neq original$
                then report M to the client
return ($original$)
```

Figure 6. Pseudo code for AORB Gen.

Algorithm 3.2: AORB(lEx, rEx, op)

```
$Ops = \{Plus, Minus, Multi, Div, Mod\}$
$mutant \leftarrow \begin{cases} lEx + rEx & \textbf{if } op = Plus \\ lEx - rEx & \textbf{if } op = Minus \\ lEx * rEx & \textbf{if } op = Multi \\ lEx \,/\, rEx & \textbf{if } op = Div \\ lEx \,\%\, rEx & \textbf{if } op = Mod \end{cases}$
return ($mutant$)
```

Figure 7. Pseudo code for AORB metamutant.

3.1.3 Generating mutants

Generation of mutants is similar to that used by the MSG method. For the generation, the program to be mutated is input to MuGamma, which produces the parse tree. Two supporting steps–the Change Point Finder and the Static Analyzer—use the parse tree to rewrite the code. The Change Point Finder searches for change points to which given Mutant Operators are applied. For each of mutant operators, it traverses the parse tree and, if a change point for a mutant operator is found, it reports to the code generator.

The Static Analyzer performs its analysis to identify which variables in the state before a method call need to be saved. Thus, instead of saving, restoring, and comparing the entire state space, only that part of the space required for the execution of the mutant is saved. The Code Generator generates mutant code using the information from these two supporting steps. For that part of the source not containing any change points, the Code Generator writes the same code as original one. For that part of the source that has change points, the Wrapper Generator generates new wrappers and functions.

3.2 Gammatizing Mutants and Mutation Testing

Because MuGamma is a specialization of the Gamma framework, users can customize the monitoring activities that will be performed on their deployed programs. Figures 1 and 2 show these as *Policies* provided by the user to MuGamma. Using these policies, MuGamma configures the MuGamma Server so that it can perform activities automatically and make decisions about the testing and monitoring. There are currently four types of policies for which the user provides input to configure MuGamma; other policies can easily be added.

Grouping Policy. The *grouping* policy specifies how MuGamma elements—the smallest subtasks that are to be monitored—are grouped together. For the MuGamma specialization of Gamma, each generated mutant is a MuGamma element. Thus, the *grouping* policy specified by the user, designates how these mutants will be grouped into MuGamma groups. Currently in MuGamma, there are four grouping policies:

- *One-to-One*. Each mutant is transformed into a MuGamma Element and that Element itself is put into a MuGamma Group.
- *Same-operator*. All mutants generated by a mutant operator are transformed and grouped into a MuGamma Group.
- *Same-target*. All mutants for a target class are transformed and grouped into a MuGamma Group.
- *Random*. N mutants are selected randomly, without any relationship to mutant operators or target classes, and transformed into MuGamma Elements and grouped into a MuGamma Group.

Assignment Policy. When a user of the deployed software executes a MuGamma Client, the client attempts to connect to the MuGamma Server. If the connection is successful, the MuGamma Server assigns one of the MuGamma Groups, specified by the grouping policy. The *assignment policy* determines which MuGamma Group is assigned to the connecting client. Currently, there are three assignment policies that can be specified:

- *Round-robin*. Groups are assigned in order of the connecting clients in a usual round-robin manner.
- *Highest-mutant-score*. Groups are given priority in assignment depending on the current mutant score (i.e., percentage of mutants reported as killed so far); the group with the highest mutant score is assigned first.
- *Lowest-mutant-score*. Groups are given priority in assignment depending on the current mutant score (i.e., percentage of mutants reported as killed so far); the group with the lowest mutant score is assigned first.

Reporting Policy. At one time, in the MuGamma network, a MuGamma Server can be connected to many MuGamma Clients. For efficient communication between the Server and the Clients, the *reporting policy* determines when the Clients report their results to the Server. Currently, there are two reporting policies that the MuGamma user can specify:

- *Time-based*. Reporting is done in a time-based manner. The user specifies the unit of time—in seconds, minutes, hours, or days—in which the reporting will be done.
- *Event-based*. Reporting is done after specified events occur. Examples of events are the killing of a mutant and the killing of all mutants assigned in that group.

Completion Policy. Without a stopping or completion criterion, a MuGamma Client could continue monitoring a MuGamma Group indefinitely. To avoid this, and to facilitate reassignment of groups for better testing, a MuGamma user can specify a *completion policy*, which indicates when the MuGamma Elements, Groups, and Projects are considered to be completed.

- *MuGamma Elements*. The user can specify the level of killing desired: not killed, very weakly killed (difference after the mutated statement), weakly killed (difference after the mutated method), or strongly killed (difference after complete execution).
- *MuGamma Groups*. The user can specify the least condition, to set the minimum number of complete MuGamma Elements in this MuGamma Group and the number of duplicated executions desired.
- *MuGamma Projects*. The user can specify the least condition for completing the project.

3.3 Executing a mutant

The provisioned program generated by MuGamma for a program P consists of the three wrappers—controlling, generating, and comparing—and the three method calls—original, implemented, and mutated. These wrappers and methods were described in detail in Section 3.1. This section provides a detailed description of the execution of the provisioned program in the field.

Figure 8 shows the process of executing mutants. As the figure shows, the provisioned program, along with an ID indicating a change point that is to be mutated, is launched by what we call the "Software Executor," the virtual machine or processor on which the program will execute. When a method that is to be mutated is called during execution, its Controlling Wrapper is called instead. If the change point and the ID do not match, the Original code is called, and the program continues without evaluating mutants.

If the change point and the ID match, the Generating Wrapper is called. The Generating Wrapper saves the state, S_{pre} before the implemented code is executed, and then saves post state, S_{post} after the code is executed. On executing the implemented code, the software will reach the pre-defined MSG method call. The MSG method evaluates a given expression and applies small changes, and then compares the results of them; this is the very weak mutation analysis. For every mutant that shows a difference (i.e., the mutant is very weakly killed), the mutant is returned as a *submutant*, which is one of the number of mutants at that change point. The set of submutants, shown in the figure as $SM_1, SM_2, ..., SM_N$, is reported to the Client, along with S_{pre} and S_{post}, and are candidates for further mutation analysis by the simulator. Note that filtering out those mutants that are not very weakly killed results in a savings because fewer mutants are executed by the simulator.

The Software Simulator now invokes a new virtual machine for each of the submutants, and launches a Comparing Wrapper to execute it. For each of the submutants, S_i, the Comparing Wrapper restores the state of the original method, S_{pre} at that point, executes the mutant, and compares the resulting state and S_{post} to see if the submutant is killed. Because the virtual machine has its own memory space, the mutant execution does not interfere with the execution of the program.

After comparing the states, the Comparing Wrapper reports the result to the Client. If the submutant is killed, the Comparing Wrapper returns the result and that mutant is marked as killed. For any killed mutant, the S_{pre} and S_{post} are also used to create a test case that is also returned. If the submutant is not killed, it is still live, and will be processed, when possible, by other executions of the program by that Client in the field. The user-input MUGAMMA Policies determine the duration of the Clients testing of particular mutants.

4 MUGAMMA Prototype

We have implemented a prototype of the MUGAMMA system and have begun to test it on several programs. This section first describes the implementation of MUGAMMA, and then presents a case study on two small subjects.

4.1 Implementation

MUGAMMA consists of four main subsystems: MUJAVA for MUGAMMA, SERVER, CLIENT, and MUGAMMA PLUG-IN.

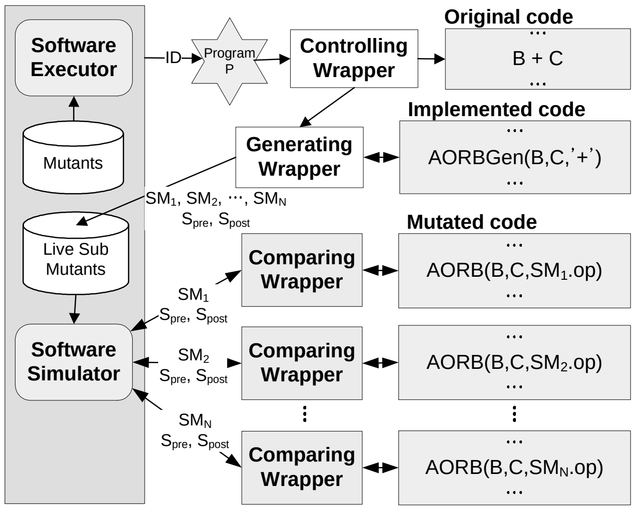

Figure 8. The process of executing mutants.

The MUJAVA for MUGAMMA subsystem is an adaptation of Ma and colleagues' mutation-testing tool for Java programs [11]. Our adaptation of MUJAVA is an Eclipse plug-in that integrates into the Eclipse IDE for Java program development. This subsystem parses the Java program, designated by the MUGAMMA user. Like the original MUJAVA tool, ours uses the OpenJava library for rewriting Java code. Unlike the original tool, ours has been adapted as an Eclipse plug-in and our generation of the mutant code follows the approach presented in Section 3. Currently, our subsystem generates mutant code for the traditional mutant operators for the Java language [11]; future work includes extending our subsystem to include the class mutant operators [11].

The next two subsystems—the SERVER and the CLIENT—are implemented using Eclipse Rich Client Platform (RCP), which is a minimal set of plug-ins needed to build an Eclipse application [25]. These subsystems provide the general GAMMA functionality, and are configured according to the Policies specified by the MUGAMMA user. The SERVER manages the grouping and assignment according to the user-specified policies, the CLIENT executes the mutants that it has been assigned, and reports the results according to the user-specified policies. Currently, our prototype does not return the pre- and post-state information required to create the regression test suites. The SERVER then makes the information available to the MUGAMMA user.

The fourth subsystem is the main MUGAMMA PLUG-IN, which contains the detailed specialized functionalities of the system. This subsystem implements the Server customization and transformations from the mutants generated by the MUJAVA subsystem to the MUGAMMA elements, required for the SERVER. Additionally, this subsystem also implements all steps required for executing the mutation testing in the field. The MUGAMMA PLUG-IN also contains the library of MSG methods.

Table 1 shows an overview of the metrics about the MUGAMMA system. In the table, for each subsystem, the second column gives the number of classes, the third column gives the number of methods, and the last column gives the number of lines of code.

Table 1. Overview of the MUGAMMA System

Subsystem	Classes	Methods	Lines of Code
MUJAVA for MUGAMMA	90	731	11091
SERVER	104	566	6725
CLIENT	111	595	6625
MUGAMMA PLUG-IN	82	452	8020
Total	387	2344	32461

4.2 Case Study

Because the MUGAMMA system is still under development, we have not yet performed extensive empirical studies on it to determine its efficiency and effectiveness. However, as a first study, to begin to evaluate performance, we have compared the mutant-generation phase of MUGAMMA with the mutant-generation phase of MUJAVA [11].

For the study, we used two programs, Sudoku, a puzzle board game, and Polynomial Solver. The first program contains nine classes and 3363 lines of code, and the second program contains eight classes and 454 lines of code. For each program, we used both MUJAVA and MUGAMMA to generate mutants. We generated mutants only for the traditional mutant operators because this is the set of mutant operators that MUGAMMA currently implements. We used MUJAVA to generate traditional mutant operators. For each application, we measured the time required to generate the set of mutants using each of MUJAVA and MUGAMMA. Table 2 gives the results for the programs. In the table, the second and third columns show the results for MUJAVA and the fourth and fifth columns show the results for MUGAMMA. In each pair of columns, the Time in seconds and the number of Mutants generated is is shown for each application. The table indicates that the number of mutants generated by each technique differs for both applications. This difference occurs because, in its current implementation, MUJAVA fails to generate all mutants for some categories of mutants.

Even with the difference in the number of mutants generated, MUGAMMA performs better than MUJAVA for the time to generate the mutants. This performance is partially due to the use of the MSG method for the implementation of MUGAMMA. In previous research [11], it was shown that speed-up is achieved when the MSG technique is used

Table 2. A summary of generated mutants

	MuJava		MuGamma	
	Time(sec)	Mutants	Time(sec)	Mutants
Sudoku	1138	2084	491	4826
Polynomial Solver	109	224	56	620

to generate mutants. By using MSG method, the number of generated files are reduced, so the total generation time to write codes and compile them also are reduced. Because of the differences in the current implementations of MUJAVA and MUGAMMA, we cannot make direct comparisons. However, the results show that, for these two applications, MUGAMMA is comparable to MUJAVA for generating mutants.

5 Conclusions and Future Work

In this paper, we have presented a novel technique that facilitates performing mutation testing on software in the field after deployment and gathering of state information about those field executions that can kill particular mutants for use in regression testing of the software when it is changed. The system, MUGAMMA, is a specialization of the GAMMA framework—it creates the instrumented versions of a program, and manages the deployment of the system and reporting of the results.

This paper also describes our prototype implementation of MUGAMMA. Our prototype currently implements only the selective set of traditional mutants for Java [10]. We are currently extending MUGAMMA to include class mutants [11]. In addition, our prototype version of MUGAMMA currently does not implement the capture-replay and formation of a set of regression test cases. Our future work includes implementing this gathering of regression test suites.

This paper also presented a case study that showed, for the two subjects studied, that MUGAMMA can be efficient in generating mutants for remote monitoring. That study gives an indication of the potential efficiency of MUGAMMA for generating mutants. However, it does not provide evidence of the efficiency of MUGAMMA in general or for larger subjects. Additional empirical studies will need to be performed on larger and varied subjects to assess this efficiency of MUGAMMA in generating mutants.

Mutant generation is not the only time-consuming part of mutation testing, and to date, we have not performed experiments to determine the overhead imposed on clients. Future work will include deploying a subject to clients, and experimenting with different allocations of mutants to assess the overhead and identify efficient configurations.

Acknowledgments

After his participation in Mutation 2000, Richard Lipton suggested performing mutation testing in the field as a way to let users of a system help with its validation. This work was supported in part by National Science Foundation awards under CCR-0096321, CCR-0205422, and SBE-0123532 to Georgia Tech, by the State of Georgia to Georgia Tech under the Yamacraw Mission, and by Brain Korea 21 Project.

References

[1] J. Bowring, J. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 195–205, July 2004.

[2] S. Elbaum, H. Chin, M. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. Technical Report TR-UNL-CSE-2006-0008, University of Nebraska-Lincoln, March 2006.

[3] P. Frankl, S. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *The Journal of Systems and Software*, 38(3), September 1997.

[4] R. Geist, A. J. Offutt, and F. Harris. Estimation and enhancement of rea-ltime software reliability through mutation analysis. *IEEE Trans. Comput.*, 41:550–558, May 1992.

[5] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying Classification Techniques to Remotely-Collected Program Execution Data. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, pages 146–155, Lisbon, Portugal, september 2005.

[6] J. R. Horgan and A. P. Mathur. Weak mutation is probably strong mutation. Technical Report SERC-TR-83-P, Purdue University, December 1990.

[7] W. E. Howden. Weak mutation testing and completeness of test sets. 8(4):371–379.

[8] J. A. Jones, A. Orso, and M. Harrold. Gammatella: Visualizing program-execution data for deployed software. *Palgrave Macmillan Information Visualization*, 3(3):173–188, 2004.

[9] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, June 2003.

[10] Y. S. Ma, M. J. Harrold, and Y. Kwon. Evaluation of mutation testing for object-oriented programs (poster). In *Proceedings of the IEEE/ACM International Conference on Software Engineering*, pages 78–81, May 2006.

[11] Y. S. Ma, A. J. Offutt, and Y. R. Kwon. Mujava: An automated class mutation system. *Journal Software Testing, Verification and Reliability*, pages 97–133, June 2005.

[12] B. Marick. The weak mutation hypothesis. pages 190–199, October 1991.

[13] A. P. Mathur and W. E. Wong. A theoretical comparison between mutation and data flow based test adequacy criteria. In *Proceedings of the 22nd Annual ACM Computer Science Conference*, pages 38–45, March 1994.

[14] A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. 20(5):337–344, May 1993.

[15] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Software—Practice and Experience*, 26(2):165–176, February 1996.

[16] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Proceedings of Mutation 2000: Mutation Testing in the Twentheth and the Twenty First Centuries*, pages 45–55, October 2000.

[17] J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An expreimental determination of sufficient mutant operators. In *ACM Transaction on Software Engineering Methodlogy*, April 1996.

[18] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Sept. 2003.

[19] A. Orso, J. Jones, and M. J. Harrold. Visualization of program-execution data for deployed software. In *Proc. of the ACM Symposium on Software Visualization*, pages 67–76, Jun 2003.

[20] A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions. In *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 29–35, St. Louis, MO, USA, May 2005.

[21] A. Orso, D. Liang, M. Harrold, and R. Lipton. Gamma System: Continuous Evolution of Software after Deployment. In *Proc. of the Int'l Symposium on Software. Testing and Analysis*, July 2002.

[22] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the International Conference on Software Engineering*, pages 277–284, 1999.

[23] M. Rinard, C. Cadar, D. Dumitran, D. M. Troy, T. Leo, and J. William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proc. of 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec 2004.

[24] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *ASE 2005: Proceedings of the 20th Annual International Conference on Automated Software Engineering*, pages 114–123, Long Beach, CA, USA, November 9–11, 2005.

[25] TheEclipseFoundation. Rich client platform at eclipsepedia. http://wiki.eclipse.org/index.php/Rich_Client_Platform.

[26] R. Untch. *Schema-Based Mutation Analysis: A New Test Data Adequacy Assessment Method*. Ph.D. dissertation, Clemson University, 1995.

[27] R. Untch, M. J. Harrold, and J. Offutt. Mutation analysis using program schemata. In *Proc. of International Symposium on Software Testing and Analysis*, Cambridge, Massachusetts, June 1993.