Carving Differential Unit Test Cases from System Test Cases

Sebastian Elbaum  
*University of Nebraska-Lincoln*, selbaum@virginia.edu

Hui Nee Chin  
*University of Nebraska-Lincoln*, hchin@cse.unl.edu

Matthew B. Dwyer  
*University of Nebraska-Lincoln*, dwyer@cse.unl.edu

Jonathan Dokulil  
*University of Nebraska-Lincoln*, jdokulil@cse.unl.edu

Follow this and additional works at: [http://digitalcommons.unl.edu/csetechreports](http://digitalcommons.unl.edu/csetechreports)
Carving Differential Unit Test Cases from System Test Cases

Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, Jonathan Dokulil

Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, Nebraska
{elbaum,hchin,dwyer,jdokulil}@cse.unl.edu

ABSTRACT

Unit test cases are focused and efficient. System tests are effective at exercising complex usage patterns. Differential unit tests (DUT) are a hybrid of unit and system tests. They are generated by carving the system components, while executing a system test case, that influence the behavior of the target unit, and then re-assembling those components so that the unit can be exercised as it was by the system test. We conjecture that DUTs retain some of the advantages of unit tests, can be automatically and inexpensively generated, and have the potential for revealing faults related to intricate system executions. In this paper we present a framework for automatically carving and replaying DUTs that accounts for a wide-variety of strategies, we implement an instance of the framework with several techniques to mitigate test cost and enhance flexibility, and we empirically assess the efficacy of carving and replaying DUTs.

1. INTRODUCTION

Software engineers develop unit test cases to validate individual program units (e.g., methods, classes, packages) before they are integrated into the whole system. By focusing on an isolated unit, unit tests are not constrained by other parts of the system in exercising the target unit. This smaller scope for testing usually results in significantly more efficient test execution and fault isolation relative to whole system testing and debugging [1, 18]. Unit test cases are also used as a component of several popular development methods, such as extreme programming (XP) [2], test driven development (TDD) practices [3], continuous testing [35], and efficient test prioritization and selection techniques [32].

Developing effective suites of unit test cases presents a number of challenges. Specifications of unit behavior are usually informal and are often incomplete or ambiguous, leading to the development of overly general or incorrect unit tests. Furthermore, such specifications may evolve independently of implementations requiring additional maintenance of unit tests even if implementations remain unchanged. Testers may find it difficult to imagine sets of unit input values that exercise the full-range of unit behavior and thereby fail to exercise the different ways in which the unit will be used as a part of a system. An alternative approach to unit test development, that does not rely on specifications, is based on the analysis of a unit’s implementation. Testers developing unit tests in this way may focus, for example, on achieving a coverage-adequacy criteria of the target unit’s code. Such tests, however, are inherently susceptible to errors of omission with respect to specified unit behavior and may thereby miss certain faults. Finally, unit testing requires the development of test harnesses or the setup of a testing framework (e.g., junit [17]) to make the units executable in isolation.

System tests are usually developed based on documents that are commonly available for most software systems that describe the system’s functionality from the user’s perspective, for example, requirement documents and user’s manuals. This makes system tests appropriate for determining the readiness of a system for release, or to grant or refuse acceptance by customers. Additional benefits accrue from testing system-level behaviors directly. First, system tests can be developed without an intimate knowledge of the system internals, which reduces the level of expertise required by test developers and which makes tests less-sensitive to implementation-level changes that are behavior preserving. Second, system tests may expose faults that unit tests do not, for example, faults that emerge only when multiple units are integrated and jointly utilized. Finally, since they involve executing the entire system no test harnesses need be constructed.

While system tests are an essential component of all practical software validation methods, they do have several disadvantages. They can be expensive to execute; for large systems days or weeks, and considerable human effort may be needed for running a thorough suite of system tests [23]. In addition, even very thorough system testing may fail to exercise the full-range of behavior implemented by system’s units; thus, system testing cannot be viewed as an effective replacement for unit testing. Finally, fault isolation and repair during system testing can be significantly more expensive than during unit testing.

The preceding characterization of unit and system tests, although not comprehensive, illustrates that system and unit tests have complementary strengths and that they offer a rich set of tradeoffs. In this paper, we present a general framework for carving and replaying of what we call differential unit tests (DUT) which aim at exploiting those tradeoffs. We termed them differential because their primary function is detecting differences between multiple versions of a unit’s implementation. DUTs are meant to be focused and efficient, like traditional unit tests, yet they are automatically generated along with a custom test-harness, making them inexpensive to develop and easy to evolve. In addition, since they indirectly capture
the notion of correctness encoded in the system tests from which they are carved, they have the potential for revealing faults related to complex patterns of unit usage.

In our approach, DUTs are created from system tests by capturing components of the exercised system that influence the behavior of the targeted unit, and that reflect the results of executing the unit; we term this carving. Those components are automatically assembled into a test harness that establishes the pre-state of the unit that was encountered during system test execution. From that state, the unit is replayed and the resulting state is queried to determine if there are differences with the recorded unit post-state.

Ideally DUTs will (a) retain the fault detection effectiveness of system tests on the target unit, (b) only report small numbers of differences that are not indicative of differing system test results, (c) be executed faster than system tests, and (d) be applicable across multiple system versions. We empirically investigate DUT carving and replay techniques with respect to these criteria through a controlled study within the context of regression testing where we compare the performance of system tests and carved unit tests. The results indicate that carved test cases can be as effective as system test cases in terms of fault detection, but much more efficient.

When compared again existing work aimed at providing automated extraction of powerful unit tests from system executions, [25, 28, 33], the contributions of this paper are: (i) a framework for automatically carving and replaying DUTs that accounts for a wide-variety of implementation strategies with different tradeoffs; (ii) a new state-based strategy for carving and replay at a method level that offers a range of costs, flexibility, and scalability; and (iii) an evaluation criteria and an empirical assessment of the efficiency and effectiveness of carving and replay of DUTs on multiple versions of a Java application. We believe these contributions lay a solid and general foundation for further study of carving and replay of DUTs and we outline several directions for future work in Section 6. In the next Section, we present our framework for carving and replay testing. Section 3 details the implementation of one of those instantiations. Section 4 describes our study and results.

2. A FRAMEWORK FOR TEST CARVING AND REPLAY

Java programs can have millions of allocated heap instances [14] and hundreds of thousands of live instances at any time. Consequently, carving the raw state of real programs is impractical. We believe that cost-effective carving and replay (CR) based testing will require the application of multiple strategies that select information in raw program states and use that information to trade a measure of effectiveness to achieve practical cost. Strategies might include, for example, carving a single representative of each equivalence class of program states or pruning information from a carved state that a method under test is guaranteed to not be dependent on. The space of possible strategies is vast and we believe that a general framework for CR testing will aid in exploring cost-effectiveness trade-offs possible in the space of CR testing techniques.

Regardless of how one develops, or generates, a unit test, there are four essential steps: (1) identify a program state from which to initiate testing, (2) establish that program state, (3) execute the unit from that state, and (4) judge the resulting state as to its correctness. In the rest of this Section, we define a general framework that allows different strategies to be applied in each of these steps.

2.1 Program States and Program Executions

For the purposes of explaining our framework, we consider a Java program to be a kind of state machine. At any point during the execution of a program the program state, $S$, can be defined, conceptually, as all of the values in memory. As needed, we will define notation for accessing specific portions of a state, for example, the parameters in the current active frame of the call stack.

A program execution can be formalized either as a sequence of program states or as a sequence of program actions that cause state changes. A sequence of program states is written as $s_0, s_1, \ldots$ where $s_i \in S$ and $s_0$ is the initial program state as defined by Java. A state $s_{i+1}$ is reached from $s_i$ by executing a single action (e.g., bytecode). A sequence of program actions is written as $\sigma$. We denote the final state of an action sequence $s(\sigma)$.

2.2 Basic Carving and Replaying

Figure 1 illustrates the CR process. Given a system test case $st_x$, carving a unit test case $cl_{x,m}$ for target unit $m$ during the execution of $st_x$ consists of capturing $s_{pre}$, the program state immediately before the first instruction of an activation of method $m$, and $s_{post}$, the program state immediately after the final instruction of the activation of $m$ has executed. The captured pair of states $(s_{pre}, s_{post})$, defines a differential unit test case for a method, $cl_{x,m}$. States in this pair can be defined by capturing the appropriate states in $\sigma$, or through the cumulative effects of a sequence of program actions, by capturing $s(\sigma)$ at the appropriate points in $\sigma$. A CR testing approach is said to be state-based if it records pairs $(s_{pre}, s_{post})$ and action-based if it records pairs $(\sigma_{pre}, \sigma_{post})$ where $s_{pre} = s(\sigma_{pre})$.

In practice, it is common for a method, $m$, to undergo some modification, e.g., to $m'$, over the program lifetime. To efficiently validate the effects of a modification, we replay $cl_{x,m}$ on $m'$. Replaying a differential unit test method for a test case $m'$ requires the ability to either load state $s_{pre}$ into memory or execute $s_{pre}$ depending on how the state was carved. From this state, execution of $m'$ is initiated and it continues until it reaches the point corresponding to the carved state $s_{post}$. At that point, the current execution state, $s_{post}', \sigma_{post}$ is compared to $s_{post}$. If the resulting states are the same, we can attest that the change did not affect the behavior of the target unit. However, if the change altered the behavior of $m$, then further processing will be required to determine whether the alteration matches the developer’s expectations.

There are multiple techniques for diagnosing the root cause of de-
ected differences. For example, a difference could trigger the execution of system test $st_k$ to determine whether a difference manifests at higher levels of abstractions, the results of $ct_{xm}$ could be compared with the results of manually developed unit tests for $m$, or intermediate states within the execution of $m$ and $m'$ (e.g., after every statement) could be compared to identify the earliest point at which states differ. We discuss support for some of these diagnostics in Section 2.4 and leave the others for future work.

Several fundamental challenges must be addressed in order to make CR cost-effective. First, the proposed basic carving procedure is at best inefficient and likely impractical. Inefficient because a method may only depend on a small portion of the program state, thus storing the complete state is wasted effort. Furthermore, two distinct complete program states may be identical from the point of view of a given method, thus carving complete states would yield redundant unit tests. Impractical because storing the complete state of a program may be prohibitively expensive in terms of time and space. Second, changes to $m$ may render $ct_{xm}$ unexecutable in $m'$. Reducing the cost of CR testing is important, but we must produce DUTs that are robust to changes so that they can be executed across a series of system modifications in order recover the overhead of carving. Finally, the use of complete post-states to detect behavioral differences is not only inefficient but may also be too sensitive to behavior differences caused by reasons other than faults (e.g., fault fixes, improvements, internal refactoring) leading to the generation of brittle tests. The following sections address these challenges.

2.3 Improving CR with Projections

We focus CR testing on a single unit by defining projections on carved pre-states that preserve information related to the unit under test and provide significant reduction in pre-state size.

2.3.1 State-based Projections

A state projection function $\pi : S \rightarrow S$ preserves selected program state components. For example, a state projection function may preserve only the values of reference fields, thereby eliminating all scalar fields, which would maintain the heap shape of a program state. Many useful state projections are based on the notion of heap reachability. A reference $r'$ is reachable in one dereference from $r$ if the value of some field of $r$ holds $r'$; let $reach^h(r)$ be the set of all objects reachable from $r$ in one or more dereferences up to length $h$ from $r$ are defined by using the iterated composition of this binary relation, $\bigcup_{1 \leq i \leq h} reach^i(r)$; as a notational convenience we will refer to this as $reach^h(r)$. The positive-transitive closure of the relation, $reach^h(r)$, defines the set of all reachable references from $r$ in one or more dereferences.

State-based CR testing approaches should use projections that retain at most the interface reachable projection which is defined to preserve the set of heap objects reachable from a calling context, $\{r \mid \exists_{p \in Parameters} \text{reach}^1(p)\}$. This includes the local frame of the method, all reachable objects from parameters $Parameters$ to the method (including $this$), and all fields of those objects. Robustness to change under this projection is identical to that of the complete program pre-state since all data that the method could possibly reference is captured. It is possible to trade robustness for reduction in carving cost by defining projections that eliminate more state information. Section 3 presents two projections that exercise this trade-off.

2.3.2 Action-based Projections and Transformations

Projections on sequences of program actions, $\pi : \bar{\sigma} \rightarrow \bar{\sigma}$, can be used to distill the portion of a program run that affects the pre-state of a unit method. Unfortunately, a purely projection-based approach to state-capture will not work for all Java programs. For example, a program that calls native methods does not, in general, have access to the native methods instructions. To accommodate this, we can allow for transformation of actions during carving, i.e., replace one sequence of instructions with another. Transformation could be used, for example, to replace a call to a native method with an instruction sequence that implements the side-effects of the native method. More generally, one could design an instance of $\pi$ that would replace any portion of a trace with a summarizing action sequence.

2.3.3 Applying Projections

Figure 2 illustrates two potential applications of the projections: test case reduction and test cases filtering.

Reduction aims at thinning a single carved test case by retaining only the projected pre-state (in Figure 2 the projection of $s_{pre}$ carved from $ct_{xm}$ leads to a smaller $s_{pre}$). Reducing a DUT’s pre-state results in reduced space requirements and, more importantly, in quicker replay since loading time is a function of the pre-state size. As we shall see, depending on the type of projection, these gains may be achieved at the expense of reduced fault detection power (e.g., a projection may discard an object that was necessary to expose the fault). Furthermore, test executability may be sacrificed as well. State-based projections may become unexecutable if the data structures used by the target unit changes, for example, shifting from an array to a heap-based structure, even if behavior is preserved. Action-based projections may become unexecutable if the behavior of a unit method changes so that a different number or sequence of methods is needed in the modified program to produce the desired pre-state. Still, reduction can be a valuable mechanism to improve the efficiency of CR by keeping just the portions of the pre-state that are most likely to be relevant to the targeted method.

Filtering aims at removing redundant DUTs from the suite. Consider a method that is invoked during the program initialization and is independent of the program parameters. Such method would be exercised by all the system tests in the same way and result in multiple identical DUTs for that particular methods. A simple filter would remove such duplicates tests, keeping just the unique DUTs. Now consider a simple accessor method with no parameters that just returns the value of a scalar field. If this method is invoked by the tests from different pre-states, then multiple DUTs will be carved, and a simple lossless filter will not discard any DUT even though they exercise similar behavior. In this case, applying a projection that preserves the pre-state components directly reachable from this would result in many DUTs that are redundant (in Figure 2 $\pi(s_{pre})$ for $ct_{xm}$, and for $ct_{xm}$ are identical so one of them can be removed). Clearly, in some cases, this kind of lossy filtering may result in a lower fault detection capability since we may discard a DUT that is indeed different and hence potentially valuable. Note that, contrary to test case reduction, filtering only uses projections to judge test equivalence, consequently, test executability is preserved since the DUTs that are kept are complete. In practice, however, reduction and filtering are likely to be applied in tandem such that reduced tests are then filtered, or filtered tests are then reduced (without necessarily using the same projection for reduction and filtering).
2.4 Adjusting Sensitivity through Difference Functions

The basic CR testing approach described earlier compares a carved complete post-state to a post-state produced during replay to detect behavioral differences in a unit. The use of complete post-states is both inefficient and unnecessary for the same reasons as outlined above for pre-states. While we could use comparison of post-state projections to address these issues, we believe that there is a more flexible solution.

Method unit test are typically structured so that, after a sequence of method calls that establish a desired pre-state the method under test is executed. When it returns additional method calls and comparisons are executed to implement a pseudo-oracle. For example, unit tests for a red-black tree might execute a series of insert and delete calls and then query the tree-height and compare it to an expected result to judge partial correctness. We allow a similar kind of pseudo-oracle in CR testing by defining difference functions on post-states that preserve selected information about the results of executing the unit under test. These difference functions can take the form of post-state projections or can be more aggressive, capturing simple properties of post-states, such as tree height, and consequently may greatly reduce the size of post-states while preserving information that is important for detecting behavioral differences.

We define difference functions that map states to a selected differencing domain, \( \delta : S \rightarrow D \). Difference in CR testing is achieved by evaluating \( \delta(s_{post}) = \delta(s_{post'}) \). State projection functions are simply differencing functions where \( D = S \). In addition to the reachability projections defined in the previous subsection, projections on unit method return values, called return differencing, and on fields of the unit instance, t.h.i.s, called instance differencing, are useful since they correspond to techniques used widely in hand-built unit tests.

A central issue in differential testing is the degree to which differencing functions are able to detect changes that correspond to faults while masking implementation changes. We refer to this as the sensitivity of a differencing function. Clearly, comparing complete post-states will be highly sensitive, detecting both faults and implementation changes. A projection function that only records the return value of the method under test will be insensitive to implementation changes while preserving some fault-sensitivity. Note also that these differencing functions provide different incomplete views on program state. Their incompleteness reduces cost and provides a measure of implementation change insensitivity, but it is problematic since it may reduce their fault detection effectiveness.

We address this by allowing for multiple differencing functions to be applied in CR testing which has the potential to increase fault-sensitivity, without necessarily increasing implementation change-sensitivity. For example, using a pair of return and instance differencing functions allows one to detect faults in both instance field updates and method results, but will not expose differences related to deeper structural changes in the heap. Fault isolation efficiency could also be enhanced by the availability of multiple differencing functions, since each could focus on a specific property or set of program state components that will help developers restrict their attention on a potentially small portion of program state that may reflect the fault.

There is another differencing dimension that can improve fault isolation. It consists of generalizing the definition of DUTs to capture a sequence of post-states, \( (s_{pre}, s_{post}) \), that capture intermediate points during the execution of the method under test. Figure 3 illustrates a scenario in which a generalized DUT begins execution of \( m \) at \( s_{pre} \). Conceptually, during replay a sequence of post-states is differentiated with corresponding states at intermediate states of the method under test. For example, at point 1, the test compares the current state to the captured \( s_{post1} \); similarly at points 2 and 3 the pre and post-states of the call out of the unit are compared. Using a sequence of post-states requires that a correspondence be defined between locations in \( m \) and \( m' \). Correspondences could be defined using a variety of approaches, for example, one could use the calls out of \( m \) and \( m' \) to define points for post-state comparison (as is illustrated in Figure 3) or common points in the text of \( m \) and \( m' \) could be detected via textual differencing. Fault isolation is enhanced using multiple post-states, since if the first detected difference is at location \( i \) then that difference was introduced in the region of execution between location \( i - 1 \) and \( i \). Of course, storing multiple post-states may be expensive so we advocate the use of \( s_{post} \) to narrow the scope of code that must be considered for fault isolation once a behavioral difference is attributed to a fault.

3. Instantiating the Framework

In this section we describe the architecture and some implementation details of a state-based instantiation of the framework. (Section 5 discusses existing carving and replay implementations which are action-based).

3.1 System Architecture

Figure 4 shows the architecture of the CR tools, with the shaded rectangles being the primary components. The carving activity starts with the Carver class which takes four inputs: the program name, the target method \( m \) within the program, the system test case \( stx \) inputs, and options to bound the carving process.
Carver utilizes a custom class loader CustomLoader (that utilizes BCEL [13]) to incorporate into the program: a singleton ContextFactory class configured to store pre and post states, and invocations of the ContextFactory at the entry and exit(s) of m. Then, every execution of m will cause two invocations of ContextFactory: one to store s_pre, and one to store s_post. ContextFactory utilizes the ContextBounding class to assist with the determination of what part of the state should be stored when test case reduction is utilized. By default, ContextBounding performs the most conservative projection: an interface reachability projection (as described in Section 2.3). More restrictive projections can be performed through the BoundingAnalysis class; we have implemented two such projections and describe them in the next section. Finally, the open source package XStream, described in more detail in the next section, performs state serialization and temporary storage. Finally, the data is compressed with the off-the-shelf compression utility bzip.

The Replay component shares many of the classes with Carver. As in Carver, Replay instruments the class of the target unit, in this case m', and utilizes the ContextFactory, but only to store s_post. The ContextLoader class obtains and loads s_pre, using XStream to unmarshal the stored program state, and then invokes the target unit for execution.

Two set of scripts, represented with double-side rectangles in Figure 4, are utilized to provide the filtering and differencing mechanisms. While a test suite is being generated, only the DUTs that capture a unique s_pre (not captured by others DUTs), and that can be replayed successfully in the same version where they were carved, are retained. Once a test suite of DUTs is generated, test case filtering can be performed to remove redundant test cases based on the same set of projections available through BoundingAnalysis. Dif scripts compare two s_post according to a specified differencing function to determine whether the changes from m to m' generate a behavioral difference. Currently, differencing functions on return values, on instance fields, on full program state (the default) are fully automated. To facilitate experimentation with different Dif functions our tools currently store the full s_post, but we plan to implement options to store only dif(s_post) which has the potential to significantly reduce the cost of carving, replay and differencing.

3.2 Interesting Implementation Aspects

In this section we briefly describe the most interesting aspects of the implementation.

Limitations of the java.io.Serializable interface. Our approach requires the ability to save and restore object data representing the program state. However, the Java java.io.Serializable interface limits the type of objects that can be serialized. For example, Java designates file handler objects as transient (non-serializable) because it reasonably assumes that a handler’s value is unlikely to be persistent, and restoring it could enable illegal accesses. The same limitations apply to other objects, such as database connections and network streams. In addition, the Java serialization interface may impose additional constraints on serialization. For example, it may not serialize classes, methods, or fields declared as private or final in order to avoid potential security threats.

Fortunately, we are not the first to face these challenges. We found multiple serialization libraries that offer more advanced and flexible serialization capabilities with various degrees of customization. We ended up choosing the XStream library [39] because it comes bundled with many converters for non-serializable types and a default converter that uses reflection to automatically capture all object fields, it serializes to XML which is more compact and easier to read than native Java serialization, and it has built-in mechanisms to traverse and manage the storage of the heap which was essential in implementing the following projections.

Interface k-bounded reachable projection. The interface k-bounded reachable projection defines the set of preserved references to include only those reachable via reference chains of length k, i.e., \( \{ r \mid \exists p \in \text{ParamsReach}^k(p) \} \). Using small values of k can greatly reduce the size of the recorded pre-state and for many methods it will have no impact on unit-test robustness. For example, a value of 1 would suffice for a method whose only dereferences are accesses to fields of this. In the implementation, when traversing the program using XStream to store the program state, we keep track of the length of dereference chains to halt traversal when k is reached.
If the unit accesses data along a reference chain of length greater than $k$, then a $k$-bounded projection will retain insufficient data about the pre-state to allow replay. Our implementation dynamically detects this situation and issues a `SentinelAccessException` to distinguish replay failure from an `application` exception. This is achieved by extending Xstream with a custom converter that automatically transforms objects that lie at a depth of $k + 1$ to contain an additional boolean field that marks it as a `sentinel` instance. The unit under test is then instrumented to insert a test of this boolean field and raise the exception if true.

**May-reference reachable projection.** The may-reference reachable projection uses a static analysis that calculates a characterization of the heap instances that may be referenced by a method activation either directly or through method calls. This characterization is expressed as a set of regular expressions of the form: $pf_1 \ldots pf_n (F^*)$. This captures an access path that is rooted at a parameter $p$ and consists of $n$ dereferences by the named fields $f_i$. If the analysis calculates that the method may reference an object through a dereference chain of length greater than $n$, the optional final term is included to capture objects that are reachable from the end of the chain through dereference of fields in the set $F$. Let $reach_{f_i}(r) = \{ r^\prime \mid \exists f \in field(r, f) = r^\prime \}$. This projection can significantly reduce the size of carved pre-states while retaining arbitrary large heap structures that are relevant to the method under test.

We implemented a $k$-bounded access path based may-reference analysis that used the flow-insensitive context-sensitive equivalence-class based read-write analysis implemented in Indus [27]. This analysis partitions parameter and variable names into equivalence classes. The two distinct features of the analysis are: 1) for each equivalence class, an abstract heap structure based on the names involved in read/write access is maintained, and 2) distinct equivalence classes are maintained for each method scope except in the case of static fields and variable names occurring in methods involved in recursive call chains. We generate regular expressions that capture the set of all possible referenced access paths up to a given fixed length, $k$, with a default of $k = 2$. When traversing the program using Xstream, we simultaneously keep track of all regular expressions and mark only those objects that lie on a defined access path for storage in XML. This analysis is also capable of detecting when a method is side-effect free and in such cases the storage of post-states is skipped since method return values completely define the effect of such method.

**4. EMPIRICAL STUDY**

The goal of the study is to assess execution efficiency, fault detection effectiveness, and robustness of the DUTs. We will perform such assessment through the comparison of system tests and their corresponding carved unit test cases in the context of regression testing. Within this context, we are interested in the following research questions:

- **RQ1:** Can carving techniques save regression test execution costs?
  
  We would like to compare the cost of reusing carved unit test cases versus the costs of utilizing regression test selection techniques that work on system test cases.

- **RQ2:** What is the fault detection effectiveness of the carved test cases? This is important because saving testing costs while reducing fault detection is rarely an enticing trade-off.

- **RQ3:** How robust are the carved tests in the presence of software evolution? We would like to assess the reusability of the carved unit test cases under a real evolving system, and examine how different types of change can affect the carved tests sensitivity.

**4.1 Testing Techniques**

Let $P$ be a program, let $P'$ be a modified version of $P$, and let $T$ be a test suite developed initially for $P$. Regression testing seeks to test $P'$. To facilitate regression testing, test engineers may re-use $T$ to the extent possible. In this study we considered four types of test regression techniques, two that work with system tests (S) and two that worked with carved tests (C):

- **S-retest-All**
  
  When $P$ is modified, creating $P'$, we simply reuse all non-obsolete test cases in $T$ to test $P'$; this is known as the retest-all technique [21] and it has been said to represent current industrial practices [23].

- **S-selection**
  
  The retest all technique can be expensive: rerunning all test cases may require an unacceptable amount of time or human effort. Regression test selection techniques [6, 10, 22, 30] use information about $P$, $P'$, and $T$ to select a subset of $T'$, with which to test $P'$. We utilize the modified entity technique [10], which selects test cases that exercise methods, in $P$, that (1) have been deleted or changed in producing $P'$, or (2) use variables or structures that have been deleted or changed in producing $P'$.

- **C-selection-k**
  
  Similar in concept to $S$-selection, this technique executes all DUTs, carved with a $k$-bounded reachable projection, that exercise methods that were changed in $P'$. This technique follows the conjecture that deeper references are less likely to be required for replay, so bounding the carving depth may improve the CR efficiency while maintaining a DUT’s strengths. Within this technique we explore depth bounding levels of 1, 5, and $\infty$ (unlimited depth which corresponds to the interface reachable projection).

- **C-selection-mayref**
  
  Similar to $C$-selection-$k$ except that it carves DUTs utilizing a may-reference reachable projection. This technique is based on the notion that program changes are more likely to affect reachable methods, so it concentrates in carving just those.

**4.2 Measures**

Regression test selection techniques achieve savings by reducing the number of test cases that need to be executed on $P'$, thereby reducing the effort required to retest $P'$. We conjecture that CR techniques achieve additional savings by focusing on units of $P'$.

To evaluate these effects, we measure the time to execute and the time to check the outputs of the test cases in the original test suite, the selected test suite, and the carved selected test suites. For a carved test suite we also measure the time and space to carve the original DUT test suite.
Methods

One potential cost of regression test selection is the cost of missing faults that would have been exposed by the system tests prior to test selection. Similarly, DUTs may miss faults due to the use of projections aimed at improving carving efficiency. We will measure fault detection effectiveness by computing the percentage of faults found by each test suite. We will also qualify our findings by analyzing instances where the outcomes of a carved test case is different from its corresponding system test case.

To evaluate the robustness of the carved test cases in the presence of program changes, we are interested in considering three potential outcomes of replaying a ct\_x\_m on unit m. For 1) fault is detected, ct\_x\_m causes m\_f to reveal a behavioral differences due to a fault; 2) false difference is detected, ct\_x\_m causes m\_f to reveal a behavioral change from m to m\_f that is not a fault (not captured by s\_x); and test is unexecutable, ct\_x\_m is ill-formed with respect to m\_f. Tests may be ill-formed for a variety of reasons, e.g., object protocol changes, internal structure of object changes, invariants change, and we refer to the degree to which a test set becomes ill-formed under a change its sensitivity to change. We assess robustness by computing the percentage of carved tests and program units falling into each one of the outcomes. Since the robustness of a test case depends on the change, we qualify robustness by analyzing the relationship between the type of change and the lifespan and sensitivity of the DUT.

4.3 Artifact

The artifact we will use to perform this experiment study is Siena [9]. Siena is an event notification middleware implemented in Java. This artifact is available for download in the Subject Infrastructure Repository (SIR) [15, 31]. SIR provides Siena’s source code, a system level test suite with 567 test cases, multiple versions corresponding to product releases, and a set of seeded faults in each version (the authors were not involved in this latest activity).

For this study we consider Siena’s core components (not on application included in the package that is built with those components). We utilize the five versions of Siena that have seeded faults that did not generate compilation errors (faults that generated compilation errors cannot be tested) and that were exposed by at least one system test case (faults that were not found by system tests would not affect our assessment). For brevity, we summarize the most relevant information to our study in Table 1 and point the reader to SIR [31] to obtain more details about the process employed to prepare the Siena artifact for the empirical study. Table 1 provides the number of methods, methods changed between versions and covered by the system test suite, system tests covering the changed methods, and faults included in each version.

4.4 Study Setup and Design

The overall process consisted of the following steps. First, we prepared the test suites generated by S-retest-all, S-selection, C-selection-k\_*, and C-selection-mayref for their automatic execution. The preparation of the system level test suites was trivial because they were already available in the repository. The preparation of the carved selection suites (C-selection-k\_* and C-selection-mayref), required for us to run the CR tool to carve all the DUTs for all the methods in v0 executed by the system tests.

Second, we run each of the generated test suites on the fault-free versions of Siena to obtain an oracle for each version. In the case of the system test suite, the oracle consisted in the set of outputs generated by the program. For the carved tests, the oracle consisted of the method return value and the relevant s\_post (we later explore several alternative projections to define the relevant state).

Third, we run each test suite on each faulty instance of each version (some versions contained multiple faults) and recorded their execution time. We dealt with each fault instance individually to control for potential masking effects among faults that might negatively affect the fault detection performance of the tests.

Fourth, to assess fault detection effectiveness, for each test suite, we compared the outcome of each test case between the fault-free version (oracle) and the faulty instances of each version. To compare the system test outcomes between correct and fault versions, we used pre-defined differencing functions that are part of our implementation which ignore “non-deterministic” output data (e.g., dates, times, random numbers). For the unit tests, we performed a similar differencing, but applied to the target method return values and s\_post. When the outcome of a test case differed between the fault-free and the faulty version, a fault is found.

Last, we compared the measures across the test suites generated by S-retest-all, S-selection, C-selection-k\_*, and C-selection-mayref. We then repeated the same steps to collect data for the same techniques when utilizing test case filtering and compression. The results emerging from this comparison are presented in the next section. All these activities were performed on an Opteron 250 processor, with 4GB of RAM, running Linux-Fedora, and Java 1.5.

4.5 Results

In this section we provide the results addressing each research question regarding carving and replaying efficiency, fault detection effectiveness, and robustness and sensitivity of the DUTs suites.

<table>
<thead>
<tr>
<th>Version</th>
<th>Methods</th>
<th>Changed-covered methods</th>
<th>Tests executing changed methods</th>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>109</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>v1</td>
<td>100</td>
<td>2</td>
<td>494</td>
<td>3</td>
</tr>
<tr>
<td>v5</td>
<td>111</td>
<td>2</td>
<td>494</td>
<td>1</td>
</tr>
<tr>
<td>v6</td>
<td>111</td>
<td>2</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>v7</td>
<td>107</td>
<td>10</td>
<td>550</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1: Siena’s components attributes.

<table>
<thead>
<tr>
<th>Carving</th>
<th>Metric</th>
<th>Reduction</th>
<th>C-select-k*</th>
<th>C-select-mayref</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain</td>
<td>Minutes</td>
<td>113 1.1K</td>
<td>155 1.9K</td>
<td>2K</td>
</tr>
<tr>
<td>Compressed</td>
<td>Minutes</td>
<td>129 1.9K</td>
<td>188 1.9K</td>
<td>496</td>
</tr>
</tbody>
</table>

Table 2: Carving times and sizes to generate initial DUT suite.

RQ1: Efficiency. We first focus on the efficiency of the carving process. Although our infrastructure automates carving, this process does consume time and storage so it is important to assess its efficiency as it might impact its adoption and scalability. Table 2 summarizes the time (in minutes) and the size (in MB) that took to carve and store the complete initial suite of DUTs utilizing the different techniques without and with the use of compression on the s\_pre and s\_post. Each column in the table contains a test case reduction technique.

For Siena, constraining the carving depth affects the carving time. This is more noticeable when carving at k = 1 which approxi-
ately a quarter of the time required to carve with either $k = 5$ or the whole $s_{pre}$. We observe the same patterns in terms of storage requirements. Note again that for depths greater than one the differences in storage space are minimal due to the rather “shallow” nature of the subject (dereference chains with length greater than 2 are rare in Siena). The may-reference projection requires almost three times of additional analysis time, but as we shall see, it is able to provide some gains in replay time. In the second row of Table 2 we see that simply compressing the state data increased the carving time in proportion to the carved state size (and it will add uncompression time as well for the DUTs selected for replaying), but it consistently provided between two and three-orders of magnitude reduction in the space required by the DUTs, offering a very interesting tradeoff.

It is important to note that the carving numbers reported in Table 2 correspond to the initial carving of the complete DUT suite – DUTs carved for each of the over 100 methods in Siena from each of the over 560 system tests that may execute each method – and can be performed automatically without the tester’s participation. During the evolution of the system, DUTs will be replayed repeatedly amortizing the initial carving costs, and only a subset of the DUTs will need to be recarved. Recarving will be necessary when determined that changes in the program may affect a DUT’s relevant pre-state. We believe that existing impact analysis techniques [24] could be used, for example, to determine what DUTs must be recarved when the a unit is changed, and we plan to integrate those into our infrastructure in the future.

We now proceed to analyze the replay efficiency. Replay efficiency is particularly important since it is likely that a carved DUT will be repeatedly replayed as the target unit evolves. Figure 5 summarizes the replay execution times for some of the techniques we consider. Each observation corresponds to the replay time of each generated test suite under each version, while the lines joining observations are just meant to assist in the interpretation. Note that the plots for $C$-select-$k_5$ and $C$-select-$k_{oc}$ overlapped almost completely so we display only one of them.

The test suite resulting from the $S$-restest-all technique consistently averages 135 minutes per version. The test suites resulting from $S$-select for each version averages 92 minutes per version, with savings over $S$-restest-all ranging from a minimum of 4 minutes in $v_6$ maximum of 132 minutes in $v_7$. (Factors that affect the efficiency of this technique are not within the scope of this paper but can be found at [16]). On average, $S$-select takes 67% of the time required by $S$-restest-all.

The test suites selected by the $C$-selection-$k^*$ techniques show very similar tendencies. On average, all the $C$-selection-$k^*$ techniques replay execution time was less than 9 minutes, and they took less than a minute to replay $v_6$ and up to 24 minutes for $C$-selection-$k_{oc}$ to replay $v_5$. On average, these suites takes 6% of the time required by $S$-restest-all, and 9% of the time required by $S$-select. The test suites selected by $C$-selection-mayref takes 6% of the time required by $S$-restest-all, 10% of the time required by $S$-selection, and 92% of the time required by $C$-selection-$k_{oc}$. Last, we observe that handling compressed files (only shown for $k = 5$) increased the replaying time by up to a factor of four.

We also measured the differencing time required by all techniques. For the system test suites the differencing times were consistently less than a minute, for the $C$-selection-$k^*$ suites it averaged 12 minutes, and for the $C$-selection-mayref averaged 6 minutes. When filtering was employed, differencing time for the $C$-selection-$k^*$ techniques was reduced by an average of 54%. Overall, although the differencing activity is important to the performance of the carved suites, implementing simple incremental differencing functions could dramatically improve their differencing performance. For example, we currently compare all the program post-state, but we could instead first compare the return values to see if it reveals any differences, and if it does not, then compare the rest of the post-state. This simple technique would suffice to reduce $v_5$ differencing time by 96%.

**RQ2: Fault detection effectiveness.** The test suites directly generated by $S$-selection, $C$-selection-$k^*$, and $C$-selection-mayref detected as many faults as the $S$-restest-all technique. This indicates that a DUT test suite can be as effective as a system test suite at detecting faults, even when using aggressive projections. It is worth noting, however, that when computing fault detection effectiveness over a whole DUT suite we do not account for the fact that, for some system tests, their corresponding carved DUTs may have lost or gained fault detection effectiveness. We conjecture that this is a likely situation with our subject because many of the faults are detected by multiple system tests. To address this situation we perform an effectiveness analysis at the individual test case level.

For each carving technique we compute: 1) PP, the percentage of passing selected system tests (selected utilizing $S$-Selection) that have all corresponding carved unit test cases passing, and 2) FF: the percentage of failing system tests that have at least one corresponding failing carved unit test case. Table 3 presents the PP and FF values for all the techniques under all version instances. In general we observe that the average most for PP and FF are over 90% indicating that DUTs carved from a system test case tend to conserve much of its effectiveness. We now discuss some interesting instances of the PP and FF values.

--- FROM HERE ---

When using the test suite resulting from $C$-selection-$k_1$ we find that for $v7 : f_1$, only 24% of the passing system tests had all their associated DUTs passing. The rest of the tests had a DUT that detected a behavioral difference that was not detected by the system test case oracle because it did not propagate to the output to be detected.

Using this restrictive reduction, the FF values are on average 97%.
In the cases where FF is not 100% such as in v6, we observed that replaying the test suite carved utilizing C-selection-k1 did not detect all the behavioral differences exhibited by the selected system test cases (1 out of the 8 system tests exposed a behavioral difference that was not exposed by any of its corresponding DUTs). This reduction in FF was due to the depth-1 projection which did not capture enough pre-state to detect a behavioral difference. The other carved suites, however, did detect this fault.

In v5, and independently of the carved test suite used, 3 out of 300 failing system tests did not have any corresponding DUT on the changed methods failing (99%). We observed the same situation in v7 : f2 where 18 out of 203 DUTs (9%) did not expose behavioral differences even though the corresponding system tests failed. When we analyzed the reasons for this reduction in FF we discovered that in both cases the tool did not carve in v0 the pre-state for one of the changed methods. The tool did not carve any pre-state for those methods because the system test case did not reach them. Changes in the code structure (e.g., addition of a method call, handling of an exception), however, made the system test cases reach those changed methods (and expose a fault) in later versions. In both circumstances, improved DUTs that would have resulted in 100% FF could have been generated by re-carving the test cases in later versions (carve from v1 to re-carve in v1.f1). More generally, these observations point out again for the need to establish mechanisms to detect changes in the code that should trigger re-carving.

**RQ3: Robustness and sensitivity.** We previously examined how DUTs obtained through C-selection-k1 are quite fragile in terms of their executability, and how certain code changes may make a method reach a new part of the heap that was not originally carved. A complementary way to evaluate the robustness and sensitivity of DUTs is to compare their performance in the presence of methods that changed, and in the presence of methods that changed and are indeed faulty. We performed such detailed comparison on the filtered suites for C-selection-∞ and C-selection-mayref, and now briefly discuss three distinct instances of the scenarios we found.

In both faulty instances of v7, the version with the most methods changed (10), none of the behavioral differences were found by methods other than the faulty ones. This is clearly a valid situation. V1 : f1 represents perhaps a more common case were none of the DUTs going through non-faulty changed methods failed, but only 78% of the DUTs traversing faulty methods actually failed. Yet a different perspective is offered by v5. Only two methods changed in this version, and one them is invoked exclusively by the other. The fault is located in the callee. The caller method is exercised by 6354 DUTs out of which 736 detect behavioral differences (12%). The faulty callees method is exercised by 26173 DUTs out of which 928 detect behavioral differences (less than 4%). This last scenario, in which carving still generates more behavioral differences for the faulty method than for change one, is interesting because it shows that even for correct changes the number of affected DUTs may be large.

It is worth noting that the differencing functions offer an opportunity to control this problem. For example, a more relaxed differencing mechanism focused on just return values would have detected all the faults in v5 and v6, while reducing the number of false differences significantly since both faults manifest themselves in the return value. Such a differencing function, however, leads to a reduced fault detection in the case of v1. Mechanisms to select and appropriately combine these differencing functions will be important for the robustness and sensitivity of DUTs.

### Table 3: Fault Detection Effectiveness.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>C-selection-k</th>
<th>5</th>
<th>C-selection-mayref</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PP</td>
<td>FF</td>
<td>PP</td>
<td>FF</td>
</tr>
<tr>
<td>v1.f1</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>v1.f2</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>v1.f3</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>v5</td>
<td>100</td>
<td>0</td>
<td>100</td>
<td>99</td>
</tr>
<tr>
<td>v6</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>v7.f1</td>
<td>24</td>
<td>100</td>
<td>24</td>
<td>100</td>
</tr>
<tr>
<td>v7.f2</td>
<td>100</td>
<td>91</td>
<td>100</td>
<td>91</td>
</tr>
<tr>
<td>Average</td>
<td>89</td>
<td>64</td>
<td>89</td>
<td>99</td>
</tr>
</tbody>
</table>

5. RELATED WORK

Our work was inspired by Weide’s notion of modular testing as a means to evaluate the modular reasoning property of a piece of software [36]. Although Weide’s focus was not on testing but on the evaluation of the fragility of modular reasoning, he raised some important questions regarding the potential applicability of what he called a “modular regression technique” that led to our work.

Within the context of regression testing, our approach is also similar to Binkley’s semantic guided regression testing in that it aims to reduce testing costs by running a subset of the program [6, 5]. Binkley’s technique proposes the utilization of static slicing to identify potential semantic differences between two versions of a program. He also presents an algorithm to identify the system tests that must be run on the slices resulting from the differences between the program versions. The fundamental distinction between this and our approach is that we do not run system level tests, but rather smaller and more focused unit tests. Another important distinction is that the testing target are not the semantic differences between versions, but rather methods in the program.

The preliminary results from our original test carving prototype [28] evidenced the potential of carved tests to improve the efficiency and the focus of a large system test suite, identified challenges to scale-up the approach, and defined some scenarios under which the carved test cases would and would not perform well. We have built on that work by presenting a generic framework for differential carving, extending the type of analysis we performed to make the approach more scalable, and by developing a full set of tools that can enable us to explore different techniques on various programs.

We are aware of two other research efforts related to the notion of test carving. First, Orso et al. prototyped the notion of selective record and replay mechanisms of program executions by capturing the interactions between the observed subsystem and its context, and then replaying just the result of those interactions [25]. Second, the test factoring approach introduced by Saff et al. takes a similar approach to Orso’s with the creation of what they called mock objects that serve to create the scaffolding to support the execution of the test unit [34]. The same group introduced a tool set for fully-featured Java execution environments that can handle many of the subtle interactions present in this programming language (e.g., callbacks, arrays, native methods) [33]. In terms of our framework, both of these approaches would be considered action-based CR approaches. We have presented, what is to the best of our knowledge, the first state-based approach to CR testing.
Saff et al. describe their approach in detail allowing us to provide
more in depth comparison with our approach. While carving a
method test case, their infra-structure records the sequence of calls
that can influence the method and then they record the sequence
of calls made by the method and the return values and unit state
side-effects of those calls. In our framework, this would amount
to calculating \( \bar{s} \) such that \( s(\bar{s}) = s_{pre} \) for the method of interest
and then calculating summarizing traces \( T(\bar{s}_{calls}) \) that reflect the
return value and side effects for each call out of the method
and carving \( s_{pre} \), the relevant pre-state for each call. During replay
the same sequence of calls with the same parameters is expected -
any deviation results in a report of a difference during replay. In
our framework, we would identify the points at which the, \( n \), calls
out of the method occur as post-state locations to define a DUT of
the form \( \{ \bar{s}, (s_{pre1}, \ldots, s_{pre_n}) \} \).

Both of these action-based approaches, capture the interactions be-
tween the target unit and its context and then build the scaffolding
to replay just those interactions. Hence, they do not incur in costs
associated with capturing and storing the system state for each tar-
geted unit. On the other hand, this approach may generate tests that
are sensitive to changes that do not effect meaning, e.g., changing
the order of independent method calls. Saff et al. have identified
this issue and propose to analyze the lifespan of the factored test
cases across sequences of method modifications [33]. This is a
critical factor in judging the cost-effectiveness of CR testing and
we have studied this issue in Section 4.5.

These two related efforts have shown their feasibility in terms of
being able to replay tests and the latter approach has provided initial
evidence that it can save time and resources under several scenarios.
Neither approach, however, has been evaluated in terms of its fault
detection effectiveness which ultimately determines the value of the
carved tests, or in the context of regression testing.

Our work also relates to efforts aimed at developing unit test cases.
Several frameworks grouped under the umbrella of Xunit have been
developed to support software engineers in the development of unit
tests. Junite, for example, is a popular framework for the Java pro-
gramming language that lets programmers attach testing code to
their classes to validate their behavior [17].

There are also multiple approaches that automate, to different de-
gressions, the generation of unit tests. For example, commercial tools
such as Jtest and Unit tests can be analyzed by monitoring code
structures and selecting test cases that increase some coverage criteria
[20]. Some of these tools aim to assess software robustness (e.g.,
whether an exception is thrown [12]). Others utilize some type of
specification such as pre and post conditions or operational abstrac-
tions, to guide the test case generation and actually check whether
the test outcome meets the expectation results [7, 11, 26, 38]. In-
terestingly enough, Parasoft new version of JTest enhances the unit
test case generated with “Sniffer”, a tool that monitors running ap-
plications to pick interesting values to exercise the target unit [20],
which can be perceived as a primitive type of carving projection.

Although carving also aims to generate unit test cases, the
approach we propose is different from previous unit test case gen-
eration mechanisms since it consists of the projection of a system
test case onto the targeted software unit. As such, we expect for
carved unit tests to retain some of the interesting interactions ex-
posed by systems tests that are harder to design into regular unit
test cases that do not consider the system context.

As stated, the post-state differencing functions that regulate the de-
tection of differences between encodings of unit behavior belongs
to a larger body of testing work on differential-based oracles. For
example, the work of Weyuker [37] on the development of pseudo-
oracles, Jaramillo et al. [19] on using comparisons to check for
optimization induced errors in compilers, or the comparison of pro-
gram spectra [29] are instances of utilizing differencing-type or-
acles at the system or subsystem level. When focusing at the unit
level of object oriented programs, as we are doing, Binder sug-
gests the term “concrete state” oracles, which aim to compare the
value of all the unit’s attributes against what is expected [4]. Briand
et al. refer to this type of oracle as a “precise” oracle because it
was the most accurate one employed in their studies [8]. Overall,
the notion of testing being fundamentally differential has long
been understood [37], since the pseudo-oracles against which sys-
tems are judged correct are themselves subject to error. Thus, the
question we aimed to answer is not whether our CR method judges
a system correct or incorrect, but rather whether it is capable of

cost-effectively detecting differences between encodings of system
behavior that developers can easily mine to judge whether the dif-
ference reflects an error.

6. CONCLUSION

We have presented a general framework for automatically carving
and replaying DUTs. The framework incorporates sophisticated
projection and differencing strategies that can be instantiated in
various ways to accommodate distinct trade-offs. We have imple-
mented a state-based instance of the framework that mitigates
testing costs through two types of reachability-based projections,
and that can adjust the DUTs sensitivity through two differencing
functions. Our evaluation of this implementation has revealed that
DUTs can be automatically generated from system tests, reduce
average test suite execution time to a tenth of our best system se-
lection technique\(^1\), and still retain most of the fault detection power
of system tests.

The experiences gained while instantiating and assessing the frame-
work suggest several directions for future work. First, we will
perform further studies not only to confirm our findings on other
subjects under similar settings, but also to compare DUTs with tra-
ditional unit tests developed by software engineers. We conjecture
that software engineers develop rather shallow unit tests and that
we can effectively complement those with DUTs that expose the
target units to more complex execution settings.

Second, we will extend our implementation with additional features
to reduce the cost of CR testing while preserving test effectiveness.
We will store the results of applying differencing functions to post-
states rather than storing post-states themselves. We will provide
mechanisms for testers to define differencing functions besides the
ones provided by the framework. We expect that experience apply-
ing these techniques to a broad collection of examples will expose
additional opportunities for cost-reduction. For example, when col-
lecting the data for Siena we realized that applying some “lossy”
projections to filter DUTs may yield more interesting tradeoffs be-
tween scalability and fault detection effectiveness.

Third, we believe that it is possible to combine multiple DUTs
to create a compound DUT for a larger program unit, for exam-
ple a class. This can be achieved by correlating multiple DUTs

\(^1\) See the Filtered-C-selection-k5 and Filtered-C-selection-mayref results in Figure 5

\[^{1}\text{See the Filtered-C-selection-k5 and Filtered-C-selection-mayref results in Figure 5}\]
based on the identity of the receiver object. For a sequence of method calls, \( c_i, \ldots, c_j \), on an object in a system test, the set of DUTs for those calls is replaced by a single DUT that captures \( (s_{pre_i}, (s_{post_i}, \ldots, s_{post_j})) \). In this test, the sequence of calls are replayed for each method as \( (s_{post_k}, s_{post_{k+1}}) \) where \( k > i \); for \( k = i \) the replay is for \( (s_{pre_i}, s_{post_i}) \). This effectively transfers the effects of methods on the receiver object throughout the sequence achieving a kind of interaction testing between calls. We plan to implement this approach and assess it relative to other class-oriented testing techniques.

Last, we will develop a supporting infrastructure to increase the use of DUTs in practice. We will leverage some of the static analysis techniques already at our disposal to determine, for example, when changes in a method may suggest a re-carving operation targeted at that specific method. We would also like to extend the analysis performed after a DUTs detects a behavioral difference on a unit that is later deemed correct. In this situation, we would like to know what other DUTs might be obsolete and require re-carving.

Acknowledgments

This work was supported in part by the National Science Foundation through CAREER award 0347518, and awards 0429149, 0444167, 045203, and 0411043, and by the Army Research Office through DURIP award W911NF-04-1-0104. We would like to thank B. Weide for inspiring this effort and S. Reddy for the feasibility exploration she provided through her thesis. We would also like to thank V. Ranganath for supporting our use of Indus and O. Tkachuk for implementing preliminary versions of the static analysis.

7. REFERENCES


