ISSN:0975-9646

C.P.V.N.J. Mohan Rao et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 2 (3) , 2011, 1202-1221

Modeling and Re-Employment of Differential Unit Test Cases from System Test Cases

Dr. C.P.V. N. J. Mohan Rao , Nandagiri R G K prasad, Satya P Kumar Somayajula

CSE Department,Avanthi College of Engg & Tech, Tamaram, Visakhapatnam, Andhra Pradesh, India.

Abstract- **In this project, 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. This project will investigate strategies for amplifying the power and applicability of testing resources. The strategies will transform existing tests into new tests that add complementary testing capabilities to the validation process. The developed strategies will be unique in their treatment of tests as data. This will require the development of test representations that can be efficiently manipulated, and test transformations to realize operations that generate new and valuable tests. We see Carving as the first of our transformations, but many others will follow.**

Keywords: **Modeling, DUT's, Unit Test cases, System Test Cases**

1. INTRODUCTION:

An important component of Empirical Software Engineering (ESE) research involves the measurement, observation, analysis and understanding of software engineering in practice. Results analyzed without understanding the contexts in which they were obtained can lead to wrong and potentially interpretation. Their exist several myths in software engineering, most of which have been excepted for years as being conventional wisdom without having been questioned. In this talk we deal briefly with a few popular mix in software engineering ranging from testing and static analysis to distributed development and high light the importance of context and generalization.

Our goal is to carve the behavior of a target unit or units from a whole system execution. We capture components that may influence the behavior of the targeted unit. Those components are then automatically assembled into a test harness that establishes the pre-state of the unit/s that was encountered during system test execution. From that state, the unit/s is replayed and the resulting state is queried to determine if there are differences with the recorded unit post-state.

SOFTWARE engineers develop unit test cases to validate individual program units (e.g., methods, classes, and packages) before they are integrated into the whole system. By focusing on an isolated unit, unit tests are not constrained or influenced by other parts of the system in exercising the target unit. This smaller scope for testing usually results in more efficient test execution and fault isolation relative to full system testing and debugging [1], [12]. Unit test cases are also key components of several development and validation methodologies, such as extreme programming (XP) [2], test-driven development (TDD) practices [3], continuous testing and efficient test prioritization and selection techniques. 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, which 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 coverage-adequacy criteria in testing 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) to make the units executable in isolation.

Software engineers also develop system tests, usually based on documents that are 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 its acceptability to 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 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 individual harnesses need to 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 [15]. In addition, even very thorough system testing may fail to exercise the full range of behavior implemented by a system's particular 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 trade-offs. In this paper, we present a general framework for the carving and replaying of what we call differential unit tests (DUTs) which aim to exploit those trade-offs. We termed them differential because their primary function is to detect 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 may influence the behavior of the targeted unit and that reflect the results of executing the unit; we term this carving because it involves extracting the relevant parts of the program state corresponding to the components exercised by a system test. Those components are automatically assembled into a test harness that establishes the prestate 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 poststate. Ideally, a set of DUT will.

1. retain the fault detection effectiveness of system tests on the target unit,

2. execute faster or use fewer resources than system tests, and

3. be applicable across multiple system versions. In addition, for program changes that are behavior preserving, effective DUTs will

4. report few differences that are not indicative of actual differences in system test results.

For changes that are intentionally behavior modifying, DUTs will, of course, detect differences. Rather than simply indicating that a difference is detected, our approach is able to provide a fine-grained view of the differences through the unit test outcomes. Using this information, developers will be able to quickly spot the effect of their intended modifications and to see where errors have been introduced.

In this paper, we investigate DUT carving and replay (CR) techniques with respect to the four numbered criteria. Through a set of controlled empirical studies within the context of regression testing, we compare the cost and effectiveness 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 in the presence of localized changes. When compared against emerging work on providing automated, extraction of powerful unit tests from system executions, [16], [18], the contributions of this paper are a framework for automatically carving and replaying DUTs that accounts for a wide variety of implementation strategies with different trade-offs, a novel state-based automated instantiation of the framework for CR at a method level that offers a range of costs, flexibility, and scalability, and an empirical assessment of the efficiency and effectiveness of CR of DUTs on multiple versions of three Java artifacts. We note that this paper is a revised version of an earlier paper presented at the Foundations of Software Engineering Symposium 2006 [11] that includes various framework extensions presented in the next section, the testing part described in section 1, key generation in section 2, a more complete and detailed implementation presented in Section 3, and additional assessments described in Section 4. Empirical study in Section 5, and related work in Section 6. Generating Regression Unit Tests using a Combination of Verification and Capture & Replay:

SECTION 1:

The combination of software verification and testing techniques is increasingly unit tests and regression test oracle. Hence, the two groups of techniques have complementary strengths, and therefore are ideal candidates for a tool-chain approach proposed in this paper. The first phase produces, for a given system, unit tests with high coverage. However, when using them to test a unit, its environment is tested as well – resulting in a high cost of testing. To solve this problem, the second phase captures the various executions of the program, which are monitored by the output of the first phase. The output of the second phase is a set of unit tests with high code coverage, which uses mock objects to test the units. Another advantage of this approach is the fact that the generated tests can also be used for regression testing. Testing techniques, in contrast, are powerful for detecting software faults and for gaining some degree of confidence that the program under test (PUT) behaves correctly in its runtime environment. VBT techniques use information gained from a verification attempt and can generate much targeted tests to reveal program faults or tests that exhibit high code coverage. Thus, both verification and testing techniques can profit when being combined. Yet, we can even go a step further in combining both approaches. We found that more traditional testing techniques have complementary strengths to VBT techniques. One such technique is capture and replay (CaR), whose strengths are the generation of isolated unit tests [13, 14] and regression test oracles [13, 17, 4].

Unit testing plays a major role in the software development process. A unit test explores a particular behavior of the unit that is tested. The unit that we deal with is a class. It explores a particular aspect of the behavior of the class under test, hereafter CUT. Testing a unit in isolation is an important principle of unit testing [10]. However, the behavior of the CUT usually depends on other classes, some of them not even existing yet. *Mock objects* [12] are used to solve this problem by replacing actual calls to methods of other classes by calls that simply return the required value, thus testing the unit in isolation. Furthermore, in order to gain confidence in the test result the test should have high code coverage. The maintenance phase is the most expensive part of the software life cycle, and is estimated to comprise at least 50% of the total software development expenses [16]. Unit testing enables programmers to re-factor code safely and make sure it works. Extreme Programming [19] adopts an approach that requires that *all* the software classes have unit tests; code without unit tests may not be released. Whenever code changes introduce a regression bug into a unit, it can quickly be identified and fixed. Hence, unit tests provide a safety net of regression tests and validation tests. This encourages developers to re-factor working code, i.e., change its internal structure without altering the external behavior [7]. Research related to regression testing often focuses on test selection and test prioritization techniques, e.g. [9, 11]. The focus of this paper is different.

We exploit the *synergies* of combining VBT and CaR tools for unit regression testing. We propose an approach for the automatic generation of unit and regression tests in the context of verification. Our goal is to improve test suites that are generated by VBT tools and CaR tools

separately. The proposed approach maintains the high test coverage provided by VBT tools while at the same time reduces the complexity of the tests through automatic generation of mock objects. Using mock objects facilitates the isolation of the unit under test. Some existing CaR tools enable to create mock objects. On the other hand, CaR tools do not provide means to achieve high code coverage, and can therefore benefit from being combined with coverage guaranteeing tools such as VBT tools. The advantage of using VBT tools is that the verification process can be used to ensure that only correct behavior is captured by the CaR tool. We identified that high code coverage and isolation are separate issues. They can be achieved independently using the two groups of techniques which have complementary strengths. Therefore we concluded that those groups of techniques are ideal candidates for the following tool-chain. The first phase produces, for a given system, unit tests with high code coverage. The second phase captures the various executions of the program, monitored by the output of the first phase. The output of the second phase is a set of unit tests with high coverage, which uses mock objects to test the units, in isolation.

The main contributions of the paper are described in Sections 4.We identify what the complementary strengths of VBT and CaR techniques are (Section 4). In Section 3 we present a novel tool-chain approach for unit regression testing in the context of verification and for unit regression testing in general. To the best of our knowledge, this tool-chain has not been considered with VBT tools so far. We have implemented the proposed approach using a concrete VBT and a concrete CaR tool resulting in the toolchain KeYGenU. By applying KeYGenU to a small banking application we provide a proof of concept of our approach, as described in Section 4. The advantages and possible limitations of the approach are then discussed in Sections 1 and 6. The other sections are related work (Section 6) and conclusions (Section 6). Complementary Strengths of the Regarded Techniques In the introduction we have described the complementary strengths of verification and testing in general. Both approaches should be combined in order to achieve reliable software and in order to optimize the verification and testing process. In this section we describe, by means of simple examples, advantages and disadvantages of CaR tools and coverage guaranteeing tools like VBT tools that are more specific to our tool-chain approach.

1.1 The Proposed Approach

We have analyzed the advantages and the problems of verification-based testing (VBT) tools and of capture and replay (CaR) tools separately. VBT tools support the verification process by helping to find software faults. They can generate test cases with high code coverage. These tools, however, usually generate neither mock objects nor regression test oracles that are based on previous program executions. CaR tools are strong at abstracting complicated program behavior and at automatically generating regression test oracles. The CaR tools, however, can do this only for specific program runs, that have to be provided somehow. In contrast, VBT tools can generate program inputs for distinct program runs.

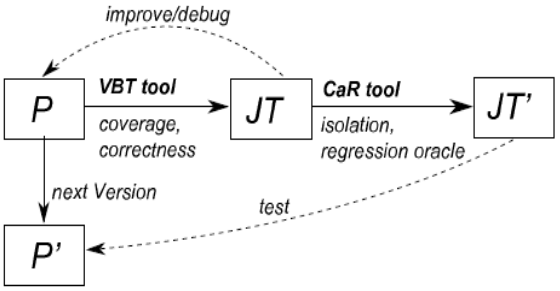


Fig. 1. The creation of a tool chain and its application to unit regression testing

From this analysis it becomes clear that these kinds of tools should be combined into a tool chain. Thus, the output of the VBT tool serves as input to the CaR tool, as shown in Figure 1. Our approach consists of two steps. In the first one the user tries to verify the program P using a verification tool that supports VBT. When a verification attempt fails, VBT is activated to generate a unit test suite JT for P. The so generated tests help in debugging P and the process is repeated until P is verifiable. When the verification succeeds the VBT tool is activated to generate a test suite JT that ensures coverage of the code of P. The generated test suite consists of one or more executable programs that are provided as input to the CaR tool. Thus when JT is executed the execution of the code under test is captured. The CaR tool in turn creates another unit test suite – JT'. If the CaR tool replays the observed execution of each test, consequently the high code coverage of JT is preserved by JT'. Furthermore, JT' benefits from the improvements that are gained by using the CaR tool. Depending on the capabilities of the CaR tool this can be the isolation of units and the extension of tests with regression-test oracles. Hence the tool chain employs the strengths of both kinds of tools involved. The test suite JT' can then be used to regression test P' that is the next development version of P.

Advantages and Limitations

We regard our approach from two perspectives. On the one hand, CaR tools can be used to further increase the quality of VBT. On the other hand, CaR tools can benefit from being combined with VBT tools. The VBT generated tests can be used to drive program's execution to ensure the coverage of the whole code. From this perspective our approach can be generalized by allowing general coverage ensuring tools for the first phase. However, for CaR tools, such as [4, 17, 13], it is important that during the capture phase only correct program behavior is observed – and this can be best ensured when a verification tool is used in the first phase. The approach combines also the limitations of the involved tools. CaR-based regression testing tools can discover changes in the behavior when a program is modified, but they cannot distinguish between intentional and not intentional changes. Another problem occurs with CaR tools that generate mock entities. It is often unclear under what preconditions the behavior of a mock entity is valid when the mock entity is executed in a state not previously observed by the CaR tool. Some advantages and limitations are specific to the particular tools and techniques. So are also the choice of the test target and mock objects. We advise the reader to refer to the referenced publications. Verification tools are typically

applicable to much smaller programs than testing tools. Our approach targets therefore at quality assurance of small systems that are safety or security critical. Building a tool-chain adds complexity to the verification process.

We expect, however, a payoff on the workload when the target system is modified and the quality of the software has to be maintained. Most VBT techniques are based on symbolic execution which is a challenging issue. Considering Listing 1.2 of Section 2, when symbolic execution reaches Line 8 the source code of write() may not be available or it may be too complicated for symbolic execution. Typically, in such situation method contracts that abstract the method call can be provided. Alternatively techniques such as [15] can be used that combine symbolic execution and runtime-execution. Regression testing techniques such as [11], for example, are often concerned with test selection and test prioritization. The goal is to reduce the execution time of the regression test suite and thus to save costs. Graves et al. [9] describe test selection techniques for given regression test suites. They reduce the scope of the PUT that is executed by selecting a subset of the test suite. Our approach provides an alternative partitioning of the PUT (Figure 2) that can reduce its tested scope and should be considered in combination with test selection techniques. Instead of reducing the number of tests, parts of the program are substituted by mock entities. When using selection techniques, a typical regression testing is usually described as follows (cf., for example, [9]). Let *P* be the original version of the program, *P0* the modified version that we would like to test, and *T* is the test suite for *P*, then:

1. Select *T'* $\subseteq$ *T*.
2. Test *P'* with *T'*, establishing the correctness of *P'* with respect to *T'*.
3. If necessary, create *T''*, a set of new functional or structural test cases for *P'*.
4. Test *P'* with *T''*, establishing the correctness of *P'* with respect to *T''*.
5. Create *T'''*, a new test suite and test execution profile for *P'*, from *T*, *T'*, and *T''*.

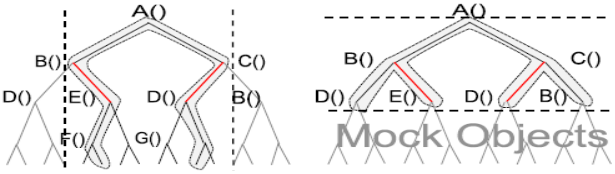


Fig. 2. The traditional test selection (left) versus our approach (right)

The authors of [9] point out the following problems associated with each of the steps:

1. It is not clear how to select a 'good' subset *T'* of *T* with which to test *P'*.
2. The problem of efficiently executing test suites and checking test results for correctness.
3. The coverage identification problem: the problem of identifying portions of *P'* or its specification that require additional testing.
4. The problem of efficiently executing test suites and checking test results for correctness.
5. The test suite maintenance problem: the problem of updating and storing test information.

We use a slightly different model, which seems to solve the above issues. This model can be summarized as follows. Let *P* be the original version of the program, *P'* the modified version that we would like to test, and *T* is the test suite which was generated for *P* after running the proposed tool-chain.

1. Introducing mock objects produces *P''* $\subseteq$ *P'*.
2. Test *P''* with *T*.
3. Rerun the tool-chain for the modified parts of *P'* to produce *T'*, covering new branches.

The problems are solved as follows:

1. There is no need to select a subset *T'* of *T*. Instead we have to consider how to create *P''*, i.e., which parts of the system *P'* should be replaced by mock objects.
2. The problem of efficiently executing test suites and checking test results for correctness is solved by using mock objects, thus not executing the whole system.
3. The coverage identification problem is solved since the whole program may be tested.
4. Same as step 2.
5. The problem of updating and storing test information is solved by rerunning the tool-chain on the modified system parts.

Safe regression test selection techniques guarantee that the selected subset contains all test cases in the original test suite that can reveal regression bugs [9]. By executing only the unit tests of classes that have been modified a safe and simple selection technique should be obtained.

Section 2:

We have implemented a concrete tool-chain according to Figure 1, called KeYGenU, and have applied it to several test cases. In this section we describe the two tools used by KeYGenU, namely KeY and GenUTest, and provide an example to demonstrate our ideas.

2.1 KeY

The KeY system [2] is a verification and test generation system for a subset of JAVA and a superset of JAVA CARD; the latter is a standardized subset of JAVA for programming of SmartCards. At its core, KeY is an automated and interactive theorem prover for firstorder dynamic logic, a logic that combines first-order logic formulas with programs allowing to express, e.g., correctness properties of the programs. KeY implements a VBT technique [6] with several extensions [5, 8]. The test generation capabilities are based on the creation of a *proof tree* (see Figure 3) for a formula expressing program correctness. The proof tree is created by interleaving first-order logic and symbolic execution rules where the latter execute the PUT with symbolic values in a manner that is similar to lazy evaluation. Case distinctions in the program are therefore reflected as branches of the proof tree; these may also be implicit distinctions like, e.g., the raising of exceptions. Proof tree branches corresponding to infeasible program paths, i.e., paths that can never be executed due to contradicting branch conditions in the program, are detected and not analyzed any further. Soundness of the system ensures that all paths through the PUT are analyzed, except for parts where the user chooses to use abstraction. Thus, creating tests for those proof branches often ensures full feasible path coverage of the regarded program part of the PUT. Based on the information contained in the proof tree, KeY creates test data using a built-in constraint solver. The

PUT is initialized with the respective test data of each branch at a time. In this way execution of each program path in the proof tree is ensured.

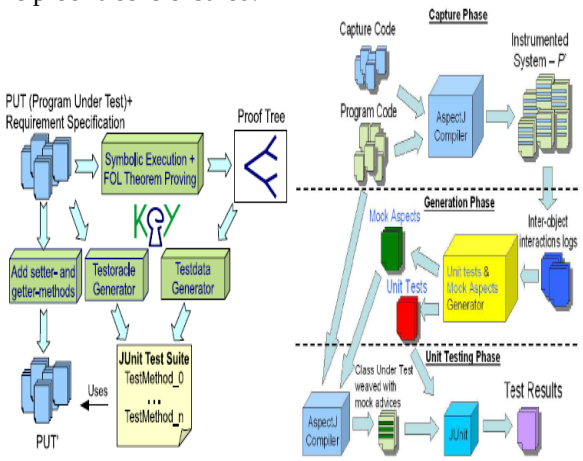


Fig. 3. Overview of verification-based testing implemented in KeY (left) and capture and replay implemented in GenUTest (right)

2.2 GenUTest

GenUTest is a prototype tool that generates unit tests [13]. The tool captures and logs inter-object interactions occurring during the execution of JAVA programs. The recorded interactions are then used to generate JUnit tests and mock-object like entities called mock aspects. These can be used independently by developers to test units in isolation. The comprehensiveness of the generated unit tests depends on the software execution. Software runs covering a high percentage generate in turn unit test with similar code coverage. Hence, GenUTest cannot guarantee a high coverage.

Figure 3 presents a high level view of GenUTest's architecture and highlights the steps in each of the three phases of GenUTest: the *capture phase*, the *generation phase*, and the *test phase*. In the capture phase the program is modified to include functionality to capture its execution. When the modified program executes, inter-object interactions are captured and logged. The interactions are captured by utilizing *AspectJ*, the most popular *Aspect-Oriented Programming* extension for the JAVA language .The generation phase utilizes the log to generate unit tests and *mock aspects*, mock-object like entities. In the test phase, the unit tests are used by the developer to test the code of the program.

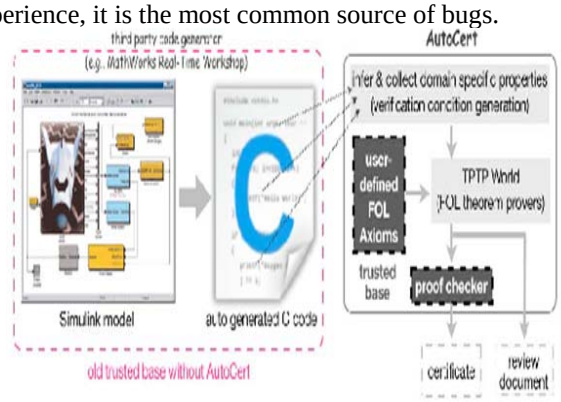
2.3 Testing First-Order Logic Axiom in Program Verification

Program verification system based on automated theorem proves rely on user provided axioms in order to verify domain specific properties of code. AUTOCERT is a source code verification tool for autogenerated code in safety critical domains, such as flight code generated from simulink models in the guidance, navigation and control (GN&C) domain using MathWorks' Real-Time Workshop code generator. AUTOCERT supports certification by formally verifying that the generated code complies with a range of mathematically specified requirements and is free of certain safety violation. AUTOCERT uses Automated Theorem Provers(ATP) based on First-Order Logic(FOL) to formally verify safety and functional correctness properties of auto generated code, as illustrated in next page figure.

AUTOCERT works by inferring logical annotation on the source code, and then using a verification condition generator (VCG) to check these annotations. This results in a set of first-order verification condition (VCs) that are then sent to a suite of ATP. These ATPs try to build proofs based on the user provided axioms, which can themselves be arbitrary First-Order Formulas (FOS).

If all the VCs are successfully proven, then it is graduated that the code complies with the properties with one important proviso: we need to trust the verification system, itself. The trusted base is the collection of components which must be correct for us to conclude that the code itself really is correct indeed, one of the main motivations for applying a verification tool like AUTOCERT to autocode is to remove the code generator a large, complex, black box-from the trusted base.

The annotation inference system is not part of the trusted base, since annotations merely serve as hints in the verification process-they are ultimately checked via their transaction into VCs by the VCG. The logic that is encoded in the VCG does need to be trusted since the proofs they generate can be sent to the proof checker. In fact, It is the domain theory, defined as a set of logical axioms, that is the most crucial part of the trusted base. Moreover, in our experience, it is the most common source of bugs.



AutoCERT narrows down the trusted base by verifying the geneated code.

Section 3:

A FRAMEWORK FOR TEST CARVING AND REPLAY

Java programs can have millions of allocated heap instances [29] 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 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 on which a method under test is guaranteed not to depend. The space of possible strategies is vast and a general framework for CR testing will aid in exploring possible cost-effectiveness tradeoffs in the space of CR testing techniques. For the purposes of explaining our framework, we consider a Java program to be a state transition system. At any point during the execution of a program, the program state S can be defined conceptually as all of the values in memory. 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 $\sigma = s_0, s_1, ..., $ 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 $\bar{\sigma}$. We denote the final state of an action sequence $s(\bar{\sigma})$. 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 correctness of the resulting state.

In the rest of this section, we define a general framework that allows different strategies to be applied in each of these steps.

3.1 Basic Carving and Replaying

Fig. 1 illustrates the general CR process. Given a system test case st_x carving a unit test case DUT_{xm} 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 m has executed.

The captured pair of states (s_{pre}, s_{post}) defines the DUT case for method m, denoted DUT_{xm}. States in this pair can be defined by directly capturing a pair of states in σ or by recording the cumulative effects of sequences of program actions $\bar{\sigma}_{pre}$ and $\bar{\sigma}_{post}$ i.e., recording s($\bar{\sigma}_{pre}$) and s($\bar{\sigma}_{post}$) .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 ($\bar{\sigma}_{pre}, \bar{\sigma}_{post}$). We note that action-state hybrid CR approaches that record, for example, pairs of actions sequences and states ($\bar{\sigma}_{pre}, s_{post}$) may also be useful.

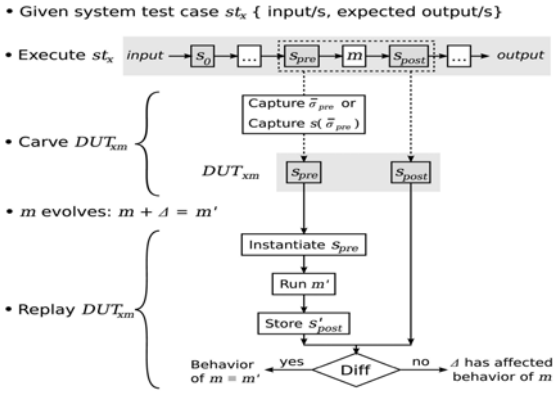


Fig. 1. Carving and replay process.

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 DUT_{xm} on m'. Replaying a DUT for a method m' requires the instantiation of s_{pre} by either loading the state s_{pre} into memory or by executing $\bar{\sigma}$ 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 s_{post}. At that point, the current execution state s'_{post} is compared to s_{post}. If the post states are the same, we can attest that the change did not affect the behavior of the target unit exercised by. DUT_{xm}.However, if the change altered the semantics of m, then further processing will be required to determine whether the alteration matches the developer's expectations (we discuss the support that provided by our implementation of CR in Section 3.1).

This basic CR approach suffers from several fundamental limitations that must be addressed in order to make CR cost-effective. First, the proposed basic carving procedure is at best inefficient and likely impractical. It is 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. It is 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 DUT_{xm} unexecutable in m'. Reducing the cost of CR testing is important, but we must produce DUTs that are robust to various types of changes so that they can be executed across a series of system versions in order to recover the overhead of carving, and provide further support to analyze the reasons behind DUTs detected differences. Finally, the use of complete poststates 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, algorithm improvements, and internal refactoring) leading to the generation of brittle tests. The following sections address these challenges.

3.2 Improving CR with Projections

We focus CR testing on a single method by defining projections on carved prestates that preserve information related to the unit under test and are likely to provide significant reduction in prestate size.

State-based projections.

A state projection function Π : S $\rightarrow$S preserves specific program state components and elides the rest. For example, a state projection may preserve the scalar fields in the object, a subset of the references to other objects, or a combination of both. Underlying many useful state projections is the notion of heap reachability. An object o' is reachable in one dereference from object o if the value of some field f references o'; let $reach(o) = \{o' | \exists_{f \in Fields(Class(o))} o.f = address(o')\}$, where Fields(c) denotes the set of (nonstatic) fields defined for class c and Class returns the class of an object. Objects reachable through any chain of dereferences up to length k from o are defined by using the iterated composition of this binary relation, $\bigcup_{1 \le i \le k} reach^i(o)$; as a notational

convenience, we will refer to this as $reach^k(o)$. The positive transitive closure of the relation, $reach^+(o)$, defines the set of all reachable objects from o in one or more dereferences.

To promote replay capabilities, state-based CR testing approaches at the method level should use projections that retain at most the set of heap objects reachable from a given calling context. That set includes heap objects reachable through the receiver object, the call's parameters, static fields within the method's class, and public static fields from other classes. More formally, given a call $r.m(p_1, \ldots, p_n)$, the reachable objects from the calling context include

1. $reach^+(r)$,
2. $\{o|\exists_{i\in 1..n}\ o \in reach^+(p_i)\}$,
3. $\{o|\exists_{f\in Fields_s(Class(r))}\ o \in reach^+(f)\}$ where $Fields_s$ is the set of static fields for a class and reach has been extended to fields, and
4. $\{o|\exists_{c\neq Class(m)}\ \exists_{f\in Fields_{ps}(c)}\ o \in reach^+(f)\}$

where $Field_{Sps}$ is the set of public static fields for a class and Class is the class declaring a given method.

This projection is lossless for reasoning about a method invocation since it retains all of the information in S_{pre} that could possibly be accessed by the call to m. More efficient projections might consider a subset of the heap elements captured by the calling context reachable projection. Some of these projections will use a notion of distance to determine what heap elements to preserve (e.g., retain all the heap elements that may be reached in up to k dereferences) while others may aim to maintain just the basic heap content and the heap structure (e.g., retain only the values of reference fields, thereby eliminating all scalar fields, which would maintain the heap shape of a program state). Some projections will determine the portion of the state to preserve ahead of time through some form of source code analysis (e.g., side-effects analysis or reachability analysis), while others will make that determination at runtime (e.g., retain the heap elements reachable or read during execution).

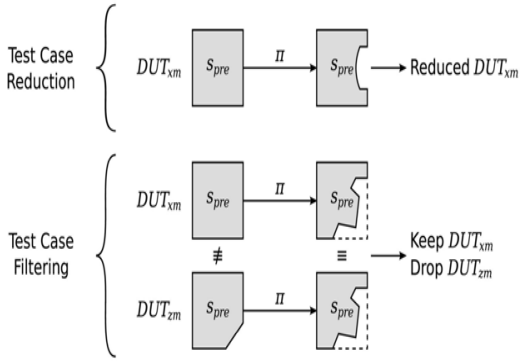


Fig. 2. Application of projections.

The range of projections makes it possible to trade robustness for reduction in carving cost and replaying time by defining projections that eliminate more state information. Section 3.2 presents five projections that exercise this trade-off.

Action-based projections and transformations. Projections can also operate on sequences of program actions, $\overline{\pi} : \overline{\sigma} \rightarrow \overline{\sigma}$ to distill the portion of a program run that affects the prestate of a unit method. Unfortunately, a purely action-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 native method 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 $\overline{\pi}$ that would replace any trace portion with a summarizing action sequence.

Applying projections. Fig. 2 illustrates two potential applications of projections on DUTs: test case reduction and test cases filtering. Reduction aims at thinning a single carved test case by retaining only the projected prestate (in Fig. 2, for example, the projection on S_{pre} carved from DUT_{xm} leads to a smaller S_{pre}). Reducing a DUT's prestate results in smaller space requirements and, more importantly, in quicker replay since loading time is a function of the prestate size. For example, a method like total Pages in Fig. 3 that returns the int field pages presents a clear opportunity to benefit from a reduction that retains just the scalar fields. Such reduction would avoid the need to load some potentially large objects such as the info hashtable, making replay faster. Depending on the type of projection, such gains may be achieved at the expense of additional analysis and carving time (e.g., using a more precise but expensive analysis to determine what to carve), or 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 when, for example, the data structures needed to successfully instantiate the object in memory become unavailable due to applied projections. In Fig. 3, the relevant program state for getAuthorName includes the field info of type java.util.Hashtable from the EnglishBook class, which stores the type of information for a particular book (such as publication date and author name). If the same scalar-only type reduction were applied, then DUTs for getAuthorName would not be replayable because the info field would be missing from the prestate. Under this circumstance, an alternative projection to enable reduction could aim for carving the fields of the parts of the hashtable just touched during the execution. The key is to identify the suitable level of reduction that would maximize efficiency, fault detection, and test executability at the same time.

Filtering aims at removing redundant DUTs from the suite. Consider a method that is invoked during program initialization and is independent of the program parameters. Such a method would be exercised by all system tests in the same way and likely result in multiple identical DUTs for that particular method. Filtering by comparing complete prestates could remove such duplicate tests, retaining just the DUTs that have a unique S_{pre}. Consider a simple accessor method with no parameters that returns the value of a scalar field. If this method is invoked by tests from different prestates, then multiple DUTs will be carved, and filtering based on complete prestates will retain all of the DUTs even

though they exercise the same behavior. For this method, filtering based on a projection that preserves just the subset of a prestate that is reachable from this in one dereference may remove multiple redundant DUTs (in Fig. 2, $\pi(s_{pre})$ for DUT_{xm} and for DUT_{zm} are identical so one of them can be removed). Clearly, in some cases, overaggressive filtering may result in a lower fault detection capability since we may discard a DUT that is different and, hence, potentially valuable. Note that, contrary to test case reduction, while filtering may sometimes only consider subsets of program states to judge equivalence, the stored program states are not modified; consequently, test executability is preserved since the DUTs that are retained 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.

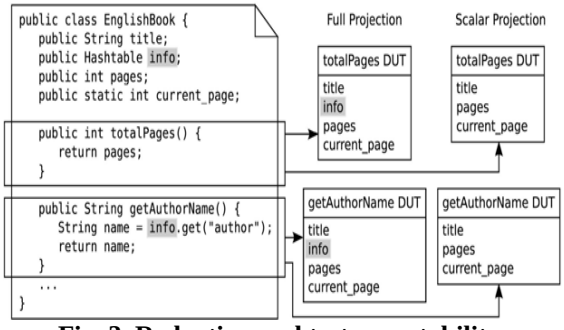


Fig. 3. Reduction and test executability.

3.3 Strategies to Manage Replay Anomalies

We have discussed how overly aggressive reductions can impair replay. Similarly, certain method changes such as modifications in a method's signature or key data structures may prevent a DUT from correct replay. For example, consider the scenario shown in Fig. 4 where we carved DUTxBook from v_0 of the Book class. Replaying the constructor for Book with the carved DUTxBook in v_1 encounters an error resulting from incompatible types in the words field between versions.

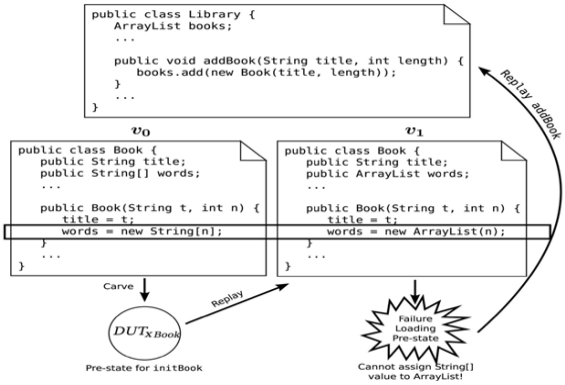


Fig. 4. An example of replay failure.

Effective CR testing must detect failures arising from carving limitations, differentiate them from regular application failures, and find ways to use this information to further guide the testing process and ensure the coverage of the target method. The detection and differentiation steps are implementation specific and are discussed in the following section, while this section focuses on what to do once a DUT fails to replay. When DUT_{xm} cannot be replayed, one could replay the system test case st_x on the new version of the software, while carving a new DUT_{xm} to replace the one invalidated by the program modification. The idea here is to use the DUT failure as a trigger for the system test case execution to ensure the proper coverage of the target method by the system test while creating DUTs for the future. An alternative approach that avoids system test case execution and immediate recarving takes advantage of the existing body of executable DUTs on other methods that exercise the target method. For instance, in Fig. 4, replaying the DUTs for the addBook method of the Library class would exercise the Book constructor (through the invocation of new Book(title, length)) without the explicit loading of DUTxBook. This approach is appealing because it eliminates the immediate need for recarving while still enabling the localized execution of a changed DUT. However, it does not account for the potential existence of multiple callers and the possibility that some callers may not be replayable themselves. When DUTxm fails, we can identify a set of DUTs whose execution reaches DUTxm's prestate; we call such a set a replay frontier of DUT_{xm}. There may be many replay frontiers for a given DUT. Selection of an appropriate frontier is guided by three criteria:

1) the ability of the frontier to successfully replay the behavior exercised in DUT_{xm},

2) the cost of executing the frontier, and

3) the localization of defect detection relative to DUT_{xm}.

At one extreme, DUT_{xmain}, i.e., the main program, comprises a replay frontier for any DUT. Intuitively, it maximizes replayability, since it is essentially an execution of system test case stx. On the other hand, this frontier will be more costly to execute than other frontiers and will provide a less focused characterization of detected defects. At the other extreme, one could identify the set of DUTs that directly invoke method m corresponding to the failed DUT. Executing these DUTs will provide localized replay of the behavior of DUT_{xm} and may be significantly less expensive than DUT_{xmain}. This frontier is more likely to exhibit replay anomalies due to the proximity to the change (e.g., when the caller and callee are methods in the same changed class). In Section 3.1, we explore a family of strategies that attempt to balance the three frontier selection criteria. 3.4 Adjusting Sensitivity through Differencing Functions

3.4 Adjusting Sensitivity through Differencing Functions

The basic CR testing approach described earlier compares a carved complete poststate to a poststate produced during replay to detect behavioral differences in a unit. The use of complete poststates is both inefficient and unnecessary for the same reasons as outlined above for prestates. While we could use comparison of poststate projections to address these issues, we believe that there is a more flexible solution that could also help control DUTs' sensitivity to changes.

Method unit tests are typically structured so that, after a sequence of method calls that establish a desired prestate, 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 differencing functions on poststates that preserve selected information about the results of executing the unit under test. These differencing functions can take the form of poststate projections or can capture properties of poststates, such as tree height or size, and consequently may greatly reduce the size of poststates while preserving information that is important for detecting just the meaningful behavioral differences. We define differencing functions that map states to a selected differencing domain, dif: $S \rightarrow D$. Differencing in CR testing is achieved by evaluating $dif(s_{post}) = dif(s_{post'})$. State projection functions are simply differencing functions where D = S. In addition to the reachability projections defined in the previous section, projections on unit method return values, called return differencing, and on fields of the unit instance referenced by this, called instance differencing, are useful since they correspond to techniques used widely in hand-built unit tests.

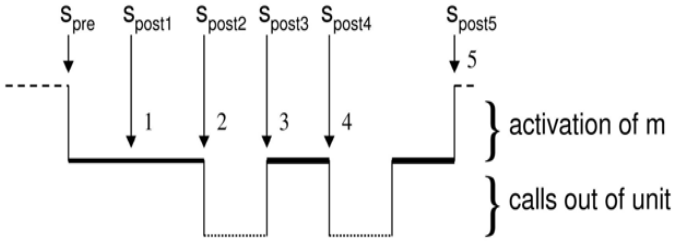


Fig. 5. Differencing sequences of poststates.

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 poststates 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 of the program state. Their incompleteness reduces cost and may add some level of insensitivity to changes in the implementation, but it could also 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 focus their attention on a potentially small portion of program state that may reflect the fault. DUTs can also be refined to increase their sensitivity in the temporal dimension by capturing sequences of poststate(s_{pre}, σ_{post})that capture intermediate points during the execution of the method under test. Such poststate sequences can be valuable to support fault isolation and debugging efforts since they provide additional observability on program states generated during the method execution.

Fig. 5 illustrates a scenario in which a DUT begins execution of m at s_{pre}. Conceptually, during replay, a sequence of poststates is differenced 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 poststates of the call out of the unit are compared.

Using a sequence of poststates 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 poststate comparison (as is illustrated in Fig. 5) or common points in the text of m and m' could be detected via textual differencing. Fault isolation information is enriched by using multiple poststates, 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 poststates may be expensive so its use can only be advocated to narrow the scope of code that must be considered for fault isolation once a behavioral difference is attributed to a fault.

4 INSTANTIATING THE FRAMEWORK

In this section, we describe the architecture and implementation details of a state-based instantiation of the framework for the Java programming language. Section 4 discusses alternative CR implementations.

4.1 System Architecture

Fig. 6 illustrates the architecture of the CR infrastructure. The carving activity starts with the Carver class which takes four inputs: the program name, the target method(s) m within the program, the system test case st_x inputs, and the reduction and filtering options.

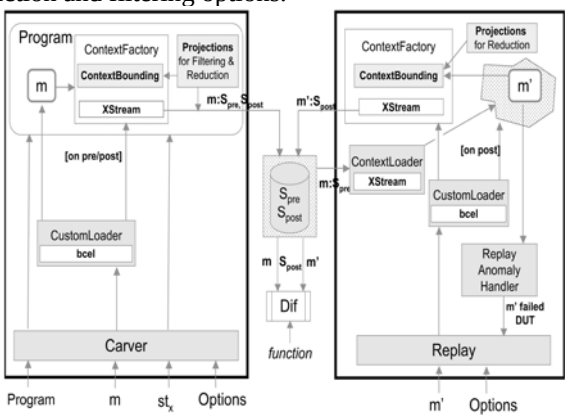


Fig. 6. CR tool architecture.

Carver utilizes a custom class loader CustomLoader (that employs the Byte Code Engineering Library (BCEL)) to incorporate instrumentation code into the program. We instrument the bytecode for all loaded program classes except the ones that are in the Java API, part of the CR tool,

and members of third-party libraries used by the tool. The instrumentation uses the singleton ContextFactory class to store pre and poststates of program methods at the entry and exit points of the methods (including exceptional exit points forced by throw instructions in m or methods called by m). Every execution of a method that is targeted for CR testing will lead to, at least, two invocations of the ContextFactory: one at the entry point of the method to store spre and one at the exit point of the method to store spost. As discussed earlier, carving the entire state of the program is impractical, therefore the ContextFactory utilizes ContextBounding to determine the parts of the program state to be stored during carving based on the chosen projections to perform reduction and filtering. Once the carving scope has been determined, ContextFactory utilizes an open source package, XStream [46], to perform the serialization to XML of the heap objects in the defined scope. Finally, ContextFactory stores the serialized program states. By default, ContextBounding applies the most conservative projection: an interface reachability projection (as described in Section 2.2), and filters DUTs based on that projection. Several other projections and lossy filters are available and introduced in the upcoming sections. While XStream is a powerful object serialization package, by default it does not serialize a class's static fields. However, to truly replay a method with the prestate that it encountered during a system test, we need to establish the values of static fields as well as instance fields since both influence the execution of the method. Fortunately, XStream allows a high level of customization. We implemented a custom extension for XStream that enables the serialization of and the application of projections to static fields by retrieving their contents including transitively reachable objects, serializing it using XStream, and placing the resulting XML in a special tag which we introduced to contain static fields. This XStream extension also takes care of deserializing the static fields and restoring them upon full object deserialization.

We have implemented two options for storing poststates:

1) complete poststate descriptions encoded in an XML format
2) unique fingerprints of poststates defined by hashing of XML encodings.

The complete representation is helpful in determining which part of the poststate was affected by the program changes, but carving execution time and storage requirements may be higher. Fingerprint storage allows for more efficient carving, storage, and difference detection, but does not allow for a detailed characterization of state differences. The other primary CR component, Replay, shares many of the core classes with Carver (CustomLoader, ContextBounding, ContextFactory) and works in a similar manner. To establish the desired prestate on which to invoke m0, Replay utilizes the ContextLoader class to obtain and load the carved s_{pre} of m, using XStream to deserialize the stored state. After that, m0 is invoked. Similar to Carver, Replay instruments the class of m0 and utilizes the ContextFactory, but only to store s_{post} after m0 is invoked. Once m0 has been replayed, we use Dif, the differencing mechanism, to compare the s_{post} of m0 generated during Replay with the carved s_{post} of m to determine whether the changes in m0 resulted in a behavioral difference. Currently, we have fully automated the differencing functions on return values, instance fields, state fingerprints, and complete XML state encodings which include static fields.

If Replay fails for m', the ReplayAnomalyHandler will begin the process of exploring the replayable frontier of m'. The current implementation to explore the frontier can use call graphs or the DUTs built-in caller information to guide the replay process in the presence of an anomaly.

These two mechanisms trade carving efficiency for replay efficiency. Keeping track of the DUT caller information requires an additional tracking method within ContextFactory that maintains a DUT call stack which increases carving overhead and storage per DUT. However, such information often leads to a more precise determination of what DUT needs to be replayed in the presence of a replay failure, which can cut down replay time.

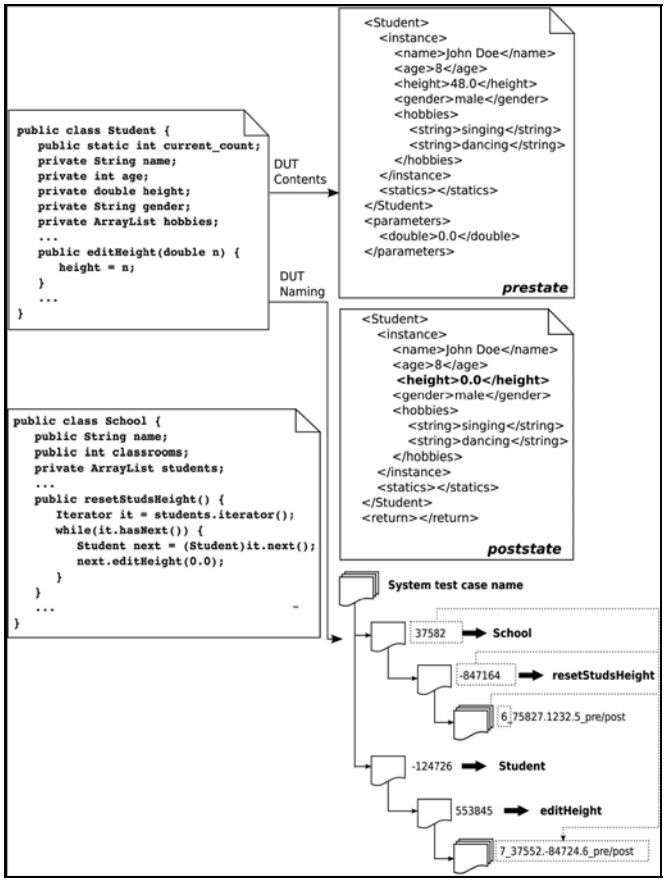


Fig. 7. DUT file contents and directory structure.

Each generated DUT is composed of two files: prestate, which includes the objects reachable through the method's parameters or the class fields, and poststate, which contains the reachable objects and return value for the method. DUTs are organized through a directory structure of four levels that includes a level for system tests, a level for classes, a level for methods, and one for DUTs. The DUTs are assigned an identification hashcode based on their corresponding method signature as well as the information identifying its caller DUT. Fig. 7 provides an example of DUT file contents for the after m0 is invoked. Once m0 has been replayed, we use Dif, the differencing mechanism, to compare the spost of m0 generated during Replay with the carved spost of m to determine whether the changes in m0 resulted in a behavioral difference. Currently, we have fully automated the differencing functions on return values, instance fields,

state fingerprints, and complete XML state encodings which include static fields.

If Replay fails for m0, the ReplayAnomalyHandler will begin the process of exploring the replayable frontier of m0. The current implementation to explore the frontier can use call graphs or the DUTs built-in caller information to guide the replay process in the presence of an anomaly. These two mechanisms trade carving efficiency for replay efficiency. Keeping track of the DUT caller information requires an additional tracking method within ContextFactory that maintains a DUT call stack which increases carving overhead and storage per DUT. However, such information often leads to a more precise determination of what DUT needs to be replayed in the presence of a replay failure, which can cut down replay time. Each generated DUT is composed of two files: prestate, which includes the objects reachable through the method's parameters or the class fields, and poststate, which contains the reachable objects and return value for the method. DUTs are organized through a directory structure of four levels that includes a level for system tests, a level for classes, a level for methods, and one for DUTs. The DUTs are assigned an identification hashcode based on their corresponding method signature as well as the information identifying its caller DUT. Fig. 7 provides an example of DUT file contents for the method edit Height of the class Student and illustrates the DUT naming scheme described above.

4.2 Implemented Projections

Here, we describe the types of projections implemented in the CR tool. These offer a degree of control over the carved test cases that can be generated. Interface k-bounded reachable projection. The interface k-bounded reachable projection for a method invocation $r.m(p_1,\ldots,p_n)$; defines the set of preserved objects to include only those reachable from the target method class via dereference chains of length up to k, i.e., $reach^k(r) \cup \{o|\exists_{i\in 1\ldots n} o \in reach^k(p_i)\} \cup \{o|\exists_{f\in Fields_s(Class(r))} o \in reach^k(f)\}$.

The intuition behind this projection is that DUTs that have identical heap structure up to depth k may exercise m in a similar manner and this could lead to significant filtering (e.g., a method working on link lists may only need to access the first k elements in a list to exhibit all of its interesting behavior). Using small values of k can greatly reduce the size of the recorded prestate and, in turn, this can lead to more DUTs being judged equivalent. For many methods, a small value of k 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. If a method changes to access data along a reference chain of length greater than the k set during carving, then the DUTs carved using the k-bounded projection would have retained insufficient data about the prestate to allow replay. Our implementation dynamically detects this situation and raises a custom exception to indicate a replay anomaly. During state storage, the heap is traversed and objects that are referenced at a depth of k +1, but no shallower, are marked. For each such marked objects, a sentinel attribute is introduced into the prestate XML encoding. When the prestate is deserialized, every object created from XML with a sentinel attribute is added to a Collection. Instrumentation is added after all GETFIELD instructions to check for the membership of the requested object in the Collection. If the object is a member, the instrumentation throws a SentinelAccess Exception. This prevents NullPointerExceptions from being thrown during sentinel object accesses which could be confused with normal application exceptions. It also prevents invalid replay results which would be caused by a program handling a null value and continuing execution when the value would not normally have been null. These SentinelAccessExceptions are one mechanism for identifying replay anomalies and triggering the ReplayAnomalyHandler.

May-reference reachable projection. The may-reference reachable projection uses a static analysis that calculates a characterization of the objects 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: $pf_1....f_n(F^+)?$ which captures an access path that is rooted at a parameter p and consists of n dereferences by the named fields f_i (e.g., p.next. next.val). 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. In general, F is calculated on a per-type basis whereF(c) is the subset of fields of class c that may be referenced by an execution of the method.

Let $reach_F(o) = \{o' \mathrm{I} \exists_{f\in F(class(o))} O.f = address(o')\}$ capture reachability restricted to the subset of fields encoded in F; reachf denotes reachability for the singleton set {f}. For a regular expression of the form p $f_1.....f_m$, where m $\leq$ n, we construct the set: $reach_{f1}(p) \cup ....\cup reach_{fm}(...(reach_{f1}(p)))$, since we want to capture all references along the path. If the regular expression ends with the term F^+, then we union an additional term of the form $reach_{F^+}(reach_{fm}(...(reach_{f1}(p))))$.

This projection can reduce the size of carved prestates while retaining arbitrarily large heap structures that are relevant to the method under test. We implement our projection using the context-sensitive interprocedural read-write analysis implemented in Indus [47]. This analysis handles all of the complexities of Java in its alias analyses including the safe approximation of readwrite operations performed in libraries. We configure this analysis to calculate l-bounded access path and then generate regular expressions that capture the set of all possible referenced access paths up to length l; we use a default of l=2.When traversing the program for serialization 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. Note that the lbounding controls the precision of the static analysis and does not limit the depth of the prestate carving, consequently no sentinel objects are introduced with this projection. This analysis is also capable of detecting when a method is side-effect free and in such cases the storage of poststates is skipped since method return values completely define the effect of such methods.

Touched-carving projection. The touched-carving projection utilizes dynamic information about all the fields that were read or written during the method execution (or the execution of methods called from that target method) to decide which parts of the program states to store. Our implementation of this projection starts with the instrumentation utilized by the interface k-bounded reachable projection, and it incorporates additional instrumentation to mark the parts of the heap referenced by the instrumented methods. During carving, the additional instrumentation helps to identify referenced fields and stores them. Fields that are not referenced are stored up to depth of k to ensure a level of robustness in the event of method changes that result in references to additional fields. There are two implementation aspects of this projection worth mentioning. First, given that we cannot know which fields will be read or written to prior to the execution of a method, we first store the method's complete s_{pre} in memory, then execute an instrumented version of the method that records all referenced fields for storage in the DUT.

This record is then used to write the XML structure or fingerprint to disk. Second, DUTxm's spre needs to store the fields referenced by m and also the fields referenced by all the methods m calls. To do this, we maintain object graphs during carving. Fig. 8 illustrates how this works for the class Person when a call to checkGrowth is made. The gray areas indicate fields that were referenced either directly or indirectly by the method. Fields in light gray were read, Light gray indicates read items and dark gray indicates written items. fields in dark gray were written, and fields in both were read and written. In the method checkGrowth, the field check is both read and written in the first line. The fields w, h, w:value, and h:value are read indirectly through calls to isTaller and isHeavier.

Clustering projection. The clustering projection attempts to identify a set of similar DUTs, like DUTxcallee;1; DUTx!callee;2; . . . , that result from the repeated invocation of callee from within the same DUT, DUTxcaller, of method caller. Fig. 9 illustrates an instance where this projection may be very effective. Every invocation of printbook results in a DUTxprintbook and one DUT for incIndex for each iteration of the while loop, i.e., length DUTsxincIndex. Consequently, there may be many DUTs generated for incIndex and the added value of those DUTs may be limited. Instead of carving such DUTs, through the clustering projection, we keep track of the number of invocations of incIndex from the context defined by DUTxprintBook. When that number exceeds a predetermined threshold, we replace the incIndex DUTs with a reference to DUT x printbook which enables their indirect replay.

This projection amounts to a heuristic for identifying a replay frontier and exploiting that frontier to filter DUTs lower in the call hierarchy. Normalizing transient data. Projections seek to retain relevant differences between states while eliminating data that is regarded as irrelevant. It is possible to eliminate differences, without eliminating data by normalizing values, for example, setting all java.util.Date fields to a fixed value, or fixing the seed in java.util.Random. In most Java programs, there are wealth of data types that have transient data. We have identified a number of those types and applied normalizing value transformations. For example, autoflushing Flushable implementations can be flushed at different times and differences in the contents of the backing Buffer objects, char[]s or byte[]s can occur under normal circumstances. To normalize buffer array contents, we check for Flushable types and Buffer types before serialization. If a Flushable type is found, the flush method is called, if a Buffer type is found, the clear method is called. Since the implementations for flush and clear do not truly clear the backing array (they just reinitialize a pointer), we use reflection to get all fields with type char[] or byte[] and overwrite them with zeros.

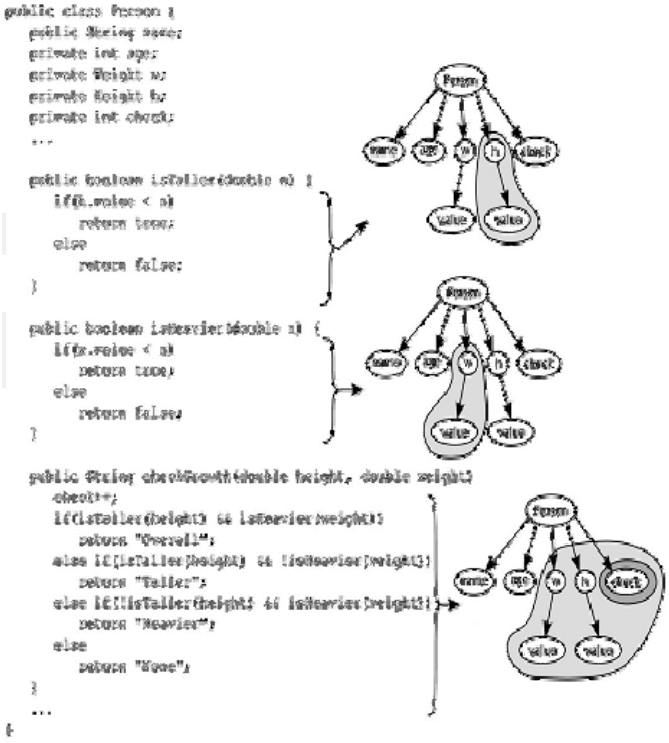


Fig. 8. Touched-carving projection.

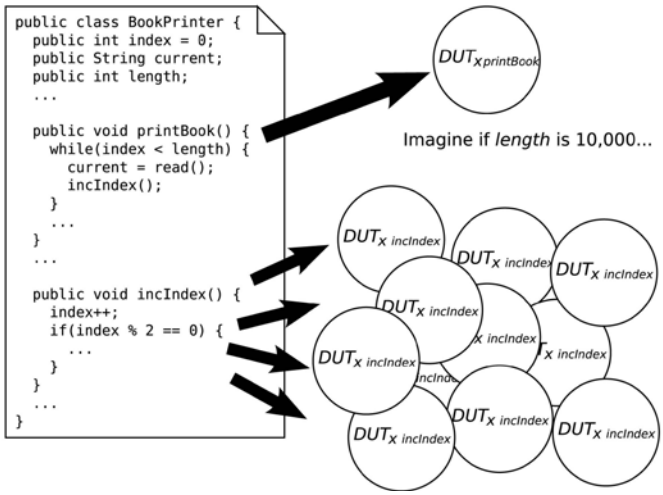


Fig. 9. A filtering strategy based on a caller's context.

Then, serialization continues as normal. This process guarantees that variable buffer contents are consistent across all poststates of multiple executions.

4.3 Toolset Limitations

The current CR toolset is robust in its support of the Java language and commonly used libraries and frameworks, but it has two limitations. Threading limitations. Our toolset was

originally developed for sequential programs and the instrumentation strategy we employ in the Carver is not thread safe. Rather than employ a basic locking strategy in instrumentation to assure thread-safety, we have deferred the treatment of thread-safety to pursue a more complex and potentially more efficient solution that avoids locking overhead in accessing Carver data structures. We note that, for replay, thread safety is not an issue. Serialization limitations. 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 (nonserializable) 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 [46] because it comes bundled with many converters for nonserializable 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 projections. In cases where XStream support was insufficient, we developed custom extensions such as the one mentioned before that enables the serialization of static fields. We anticipate that further extensions and customizations will accommodate other special object types.

Scope limitations.

Our toolset captures a large part of the program state relevant to a calling context, but it does not capture all of it. We do not capture public variables declared by other classes that are not reachable from the target method class. This implicit projection may cause false replay differences, but it is necessary to avoid bulky and inefficient DUTs. In addition, we do not capture fields declared static final since they cannot be restored during deSerialization. However, we note that such fields are often initialized to fixed values that are consistent across executions, limiting their influence in post state differences.

Section 5:
EMPIRICAL STUDY

The goal of the study is to assess execution efficiency, fault detection effectiveness, and robustness of 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 questions:

RQ1. Can DUTs reduce regression testing costs? We would like to compare the cost of carving and reusing DUTs versus the costs of utilizing regression test selection techniques that work on system test cases.

RQ2. What is the fault detection effectiveness of DUTs? This is important because saving testing costs while reducing fault detection is rarely an enticing tradeoff.

RQ3. How robust are the DUTs in the presence of software evolution? We would like to assess the reusability of DUTs on a real evolving system and examine how different types of change can affect the robustness and sensitivity of the carved tests.

5.1 Regression Test Suites

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 reuse T to the extent possible. In this study, we considered five types of test regression techniques, two that directly reuse with system tests (S) and three that reuse the DUTs carved from the system test suite (C). S-retest-All. When P is modified, creating P', we simply reuse all runnable test cases in T to test P0; this is known as the retest-all technique [33]. It is often used in industry [35] and as a control technique in regression testing experiments. S-selection. The retest all technique can be expensive: rerunning and rechecking the outcome of all test cases may require an unacceptable amount of time or human effort. Regression test selection techniques [21], [26], [34], [41] use information about P, P', and T to select a subset of T, T', with which to test P0. We utilize the modified entity technique [26], which selects test cases that exercise methods, in P, that 1) have been 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 often not 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 1 (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 a program change will mostly affect the parts of the heap reachable by the method under test or by the methods invoked by the method under test. C-selection-touched. Similar to C-selection-k except that it carves DUTs utilizing a touched-carving projection. This technique is based on the idea that modifications to the program are more likely to affect parts of the heap actually touched in the process of invoking the method under test. The touched-carving projection here is bounded to a depth of at least 1 so that the generated DUTs store at least all fields of primitive types.

5.2 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 some methods of P0. In other words, while system test case selection identifies the relevant test cases, CR adds another orthogonal dimension by identifying what methods are relevant. 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. By default, we applied the default lossless filter on all DUT test suites so that DUTs with unique prestates are kept for each program method. 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 type of change that render a DUT unexecutable or 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 are 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 aDUTxm on unit m0:

1) fault is detected, DUTxm causes m'to reveal a behavioral differences due to a fault;
2) false difference is detected, DUTxm causes m' to reveal a behavioral change from m to m0 that is not a fault (not captured by stx)
3) test is unexecutable, DUTxm is ill-formed with respect to m0.

TABLE 1
Siena's Component Attributes

Version	SLOC	Methods	Changed methods covered	Tests executing changed methods	Faults
v0	6022	109	-	-	-
v1	5848	100	2	494	3
v5	6098	111	2	494	1
v6	6066	111	2	8	1
v7	6034	107	10	503	2

Tests may be illformed for a variety of reasons (e.g., object protocol changes internal structure of object changes, invariants changes) and we refer to the degree to which a test set becomes ill-formed under a change as 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 sensitivity of the DUTs.

5.3 Artifact

The artifact we will use to perform this experiment study is Siena [25]. Siena is an event notification middleware implemented in Java. This artifact is available for download in the Subject Infrastructure Repository (SIR) [30], [40]. SIR provides Siena's source code, a system-level test suite with 503 unique 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 the 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. It also provides the number of physical source lines of code (SLOC) which was obtained using the wc utility.

5.4 Study Setup and Design

The activities in this study were performed on an Opteron 250 processor, with 4 Gbytes of RAM, running Linux-Fedora, and Java 1.5. The overall process consisted of the following steps as shown in Fig. 10. First, we prepare the base test suites, System tests, C _ k_, C-mayref, and C-touched. The preparation of the system-level test suite was trivial because it was already available in the repository.

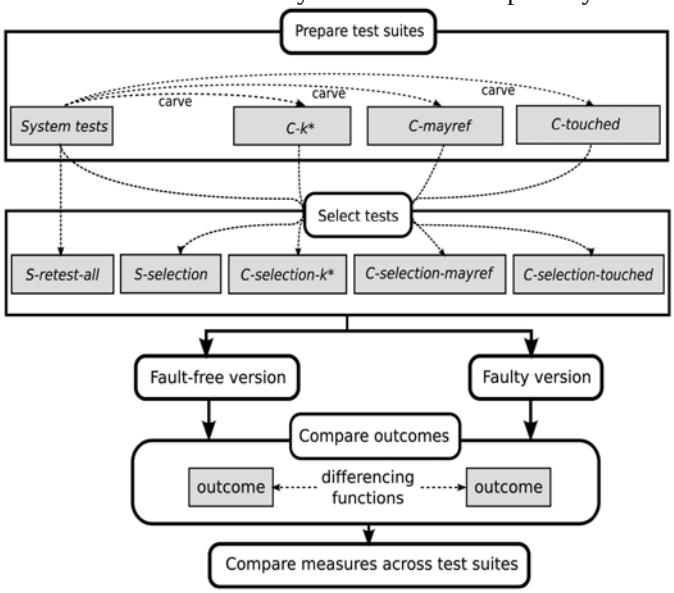


Fig. 10. Study process.

The preparation of the carved selection suites, required us to run the CR tool to carve all the DUTs for all the methods in v' executed by the system tests. Once the base test suites were generated, we performed test selection for each version, as described in Section 5, to obtain S-retest-all, S-selection, C-selection-k_, C-selection-mayref, and C-selection-touched. Second, we run each generated test suite on the fault-free versions of Siena to obtain an oracle for each version. For the system tests, the oracle consisted of 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}.

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 obscure the fault detection performance of the tests. Fourth, 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 faulty versions, we used predefined differencing functions

that are part of our implementation which ignore transient output data (e.g., dates, times, and random numbers). For the DUTs, we performed a similar differencing, but applied to the target method return values and S_{post} . When the outcome of a system test case differed between the fault-free and the faulty version, a fault is said to be found. For the differences on the carved tests, we performed a deeper analysis to determine whether the observed behavioral differences correspond to faults. Last, we compared all measures introduced in Section 4.2 across the test suites generated by S-retest-all, S-selection, C-selection-k_, C-selection-mayref, and C-selection-touched. We then repeated the same steps to collect data for the same techniques when utilizing an of-the-shelf compression package to reduce the size of the s_{pre}. The results emerging from this comparison are presented in the next section.

TABLE 2
Carving Times and Sizes to Generate Initial DUT Suites

Carving	Metric	Reduction				
		C-select-k			C-select	C-select
		1	5	∞	*mayref*	*touched*
Plain	Minutes	120	120	120	122	120
	MB	845	1.8K	1.8K	1.3K	1.0K
Compressed	Minutes	125	128	125	124	124
	MB	2.6	3.5	3.4	3.0	2.8
Number of DUTs		20698	23399	23514	22391	20961
Percentage of DUTs with sentinels		69%	3%	-	47%	61%

5 Results:

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

RQ1: Efficiency. We first focus on the efficiency of the carving process. Although our infrastructure completely 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 megabytes—MB) that it took to carve and store the complete initial suite carved from v' of approximately 20,000 DUTs utilizing the different CR techniques with and without the use of compression on the S_{pre} and S_{post} In the first row of Table 2, we observe that, for Siena, constraining the carving depth barely affects the carving time. However, we see that constraining the carving depth can greatly reduce the required space, as carving at k =1 requires 47 percent of the space required for carving with infinite depth. Observe that for depths greater than 1, the differences in storage space are small due to the rather "shallow" nature of the artifact (dereference chains with length greater than 2 are rare in Siena). C-select-mayref carving required additional time because of the extra static analysis performed up front, but consumed 55 percent of the space. Utilizing the touched-carving projection resulted in space requirements averaging those of k = 1 and k=∞. Compressing the stored DUTs with the open source utility bzip provided space savings of 99.7 percent when carving at unlimited depth, but added 4-8 minutes over the whole test suite carving process.

The last two rows of Table 2 reveal that 69 percent of the DUTs carved at k =1 contained sentinels while only 3 percent of the DUTs carved at a k =5 contained sentinels. The differences in sentinels mean that deeper differences in the heap are more often obscured by using k = 1, which explains why filtering is more effective on the smaller space captured by k =1. The touched suite size and carving costs resemble those of k = 1, while the mayref size and costs fit in between those of k = 1 and k = 5. 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 methods in Siena from each of 503 system tests that may execute each method. Carving was performed automatically without the tester's participation. As with regular unit tests, 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 (e.g., recarving the DUTs affected by the changes in v6 would only require 2 percent of the original carving time). Recarving will be necessary when it is determined that changes in the program may affect a DUT's relevant prestate. We now proceed to analyze replay efficiency. Replay efficiency is particularly important since, as with regular units tests, it is likely that a carved DUT will be repeatedly replayed as a method evolves while preserving its original intended behavior. Fig. 11 shows the time in minutes to execute the system regression test suites and to replay the C-selection-k1 suite (the most expensive of all carved suites). 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.

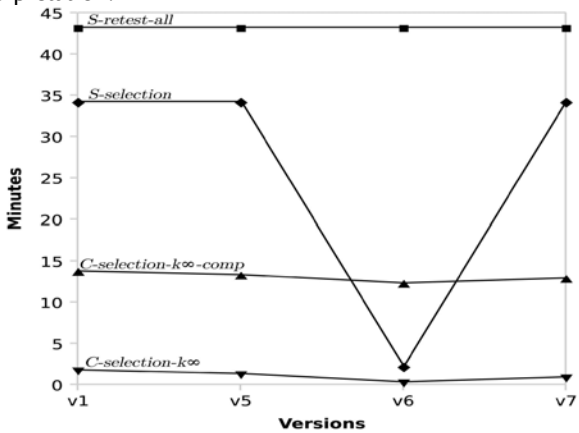


Fig. 11. Test suite execution times for system test suites and C-selection-k1 (with and without compression).

Replaying the C-selection-k1provides gains of at least an order of magnitude over the S-select suites, averaging less than a minute per version. On average, replaying carved suites take 2 percent of the time required by S-retest-all, and 3 percent of the time required by S-select. Utilizing Cselection- k1-comp incurs a large overhead to uncompress the DUTs content, rendering its application unlikely in spite of the storage savings. The test suite resulting from the S-retest-all technique consistently averages 43 minutes per version. The test suites resulting from S-select averages 28 minutes across versions, with savings over S-retest-all ranging from barely a minute in v7 to a maximum of 41 minutes in v6.1

We also measured the doffing time required by all techniques. For the system test suites the doffing times were consistently less than a minute, and for the

1. Factors that affect the efficiency of this technique are not within the scope of this paper but can be found in [48].
2. C-selection_ suites the time never exceeded 15 seconds, making both negligible compared with the replay time.

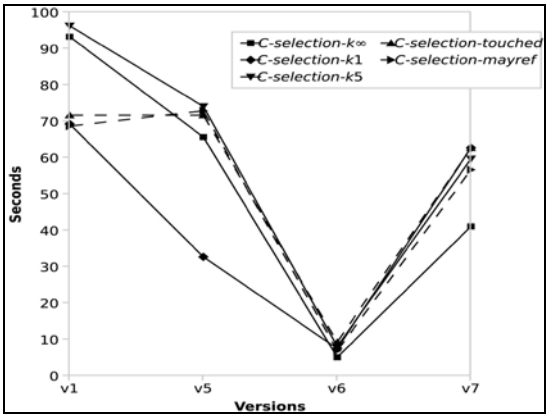


Fig. 12. Test suite execution times for the C-selection-k_ suites.

Fig. 12 summarizes the replay execution times for some of the other test suites we generated. We find that, on average, all the C-selection_ suites (excluding the one with compression) replay execution time was less than 1 minute. They all took less than 10 seconds to replay v6 and up to 96 seconds to replay the DUTs selected for v1. Constraining the carving depth with k = 1 consistently reduced replay time (over percent 50 reduction in v5). Similarly, constraining the carving space through either C-selection-mayref or C-selection-touched reduced the replay time in some versions (almost 20 percent reduction in v1).

RQ2: Fault detection effectiveness. Most of the test suites carved from S-selection, (with k_1), C-selectionmayref, and C-selection-touched detected as many faults as the S-retest-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 computing fault detection effectiveness over a whole DUT suite overlooks 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 artifact because many of the faults are detected by multiple system tests, so there were many carved DUTs that could have detected each fault. To address this situation, we perform an effectiveness analysis at the 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 DUTs passing, and 2) FF, the percentage of failing system tests that have at least one corresponding failing DUT. Table 3 presents the PP and FF values for the suites under all faults in each version. In general, we observe that most PP and FF values are over 90 percent indicating that DUTs carved from a system test case tend to preserve much of their effectiveness. But, we can also identify some interesting exceptions. For example, independent of the DUT suite, for v7: f1 (the first fault in version v7), only 24 percent of the passing system tests had all their associated DUTs passing. The rest of the system tests had at least one DUT that detected a behavioral difference that was not detected by the system test case oracle because it did not propagate to the output (the level at which the system test case oracle operated). This is one example where a DUT is more powerful than its corresponding system test. Another interesting instance is FF for C-selection-k1, v5, where we observed that replaying the carved test suite did not detect any of the behavioral differences exhibited by the selected system test cases. Upon further examination, we found that the changed method in v5 required access to references in the heap deeper than k = 1 which were not satisfied by the captured prestate of the C-selection-k1 suite, therefore resulting in SentinelAccessException.

Because of this, no poststates were stored for the method and the fault goes undetected. The other carved test suites on v5 did detect the fault since they either carved deeper prestates or, in the case of the touched-carving projection test suite, carved the parts of the prestates that were necessary for the methods under test. Still, for the other suites on v5, 3 out of the 300 failing system tests did not have any corresponding DUT on the changed methods failing (99 percent). We observed a similar situation in v7: f2 where 18 out of 203 DUTs (9 percent) 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 prestate for one of the changed methods because the system test case did not reach them;

TABLE 3
Fault Detection Effectiveness

	PP					FF				
	C-selection-k			C-selection	C-selection	C-selection-k			C-selection	C-selection
	1	5	∞	mayref	touched	1	5	∞	mayref	touched
v1:f1	100	100	100	100	100	100	100	100	100	100
v1:f2	100	100	100	100	100	100	100	100	100	100
v1:f3	100	100	100	100	100	100	100	100	100	100
v5	100	100	100	100	100	0	**99**	**99**	**99**	**99**
v6	100	100	100	100	100	100	100	100	100	100
v7:f1	**24**	**24**	**24**	**24**	**24**	100	100	100	100	100
v7:f2	100	100	100	100	100	**91**	**91**	**91**	**91**	**91**
Average	89	89	89	89	89	84	99	99	99	99

TABLE 4
Robustness and Sensitivity

	% of DUTs Detecting a Difference in	
	Changed Methods	Faulty Changed Methods
v1:f1	3	20
v1:f2	20	3
v1:f3	15	100
v5	7	7
v6	0	100
v7:f1	0	100
v7:f2	0	100

call graphs generated for the system test cases indicate that the faulty methods were not invoked during the execution of some of the system test cases on v' of Siena. Changes in the code structure (e.g., addition of a method call), 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 percent FF could have been generated by recarving the test cases in later versions (carve from vi instead of v' to replay in viþ1).

More generally, these observations point out again for the need for mechanisms to detect changes in the code that should trigger recarving.

RQ3: **Robustness and sensitivity.** We 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. We further evaluate the robustness
and sensitivity of DUTs by comparing their performance in the presence of methods that changed but are not faulty and in the presence of methods that changed and are indeed faulty. We performed such detailed comparison on the suites generated with C-selection-k1. Table 4 summarizes the findings and we now briefly discuss distinct instances of the scenarios we found.

In both faulty instances of v7, the version with the most methods changed (10), none of the DUTs revealing behavioral differences were found by methods other than the faulty ones.

This is clearly an ideal situation, which is also present in v6. V1: f3 represents perhaps a more common case where 15 percent of the DUTs going through nonfaulty changed methods detected differences, but 100 percent of the DUTs traversing faulty methods actually revealed a poststate difference. V 1 : f2 presents a scenario in which carving generates more behavioral differences for the nonfaulty changed methods than for the faulty changed methods, showing that even for correct changes the number of affected DUTs may be large (13 out of 65). In this case, the implementation change was such that the method switched the order of division and multiplication operations involving a variable which was eventually returned. Because of this, there was a difference in the return value, which was detected as a behavioral difference, and would probably be detected by other forms of unit tests as well.

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 could have detected the fault while reducing the number of false differences if the fault manifests itself in the return value. Mechanisms to select and appropriately combine these differencing functions will be important for the robustness and sensitivity of DUTs. In addition, we anticipate that as the CR components of the framework become parts of an IDE, the additional change information available in the developer's environment could help to reduce the number of false positives. For example, code modifications due to refactoring that do not affect the target unit's interface would be expected to retain the same behavior. However, changes that can be mapped to the bug repository would be expected to affect the unit's behavior.

5.6 Targeted Case Studies

The previous study addressed the stated research questions with respect to Siena, and we believe the findings generalize to similar artifacts. Still, we realize that our study suffers from threats to validity. Specifically, the selected artifact provided limited exposure to CR in the presence of deeper heap structures, extensive software changes, and high number of methods invocations. We have started to address those threats to the validity of our findings by investigating the performance of CR in the presence of such settings.

More specifically, we have studied the performance of CR on two other artifacts, NanoXML and Jtopas [30], that provide exposure to more complex heap structures, highfrequency executions sequences, and extensive changes between versions. These studies confirm our previous findings, but also show that the performance of the different carving strategies can vary significantly in programs with complex heap structures, that the Replay AnomalyHandler can enhance DUTs reuse and potential for fault detection with affordable replay costs, and that the clustering projection can be very effective to reduce the number of DUTs on high-frequency methods. Due to space constraints, the detailed settings and results are omitted here but available in a technical report [28].

Section 6: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 [49]. Although Weide's focus was 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 similar to Binkley's semantic guided regression testing in that it aims to reduce testing costs by running a subset of the program [21], [22]. Binkley's technique utilizes 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 our targets are not the semantic differences between versions, but rather methods in the program. The preliminary results from our original test carving prototype [38] 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 pinpoint 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 other research efforts related to the notion of test carving. First, Orso and Kennedy's and Clause et al.'s notion of selective capture and replay of program executions [36]. Orso and Kennedy's technique [36] aims to selectively capture and replay events and interactions between the selected program components and the rest of the application. The technique captures a simplified state representation composed of the object IDs, types, and scalar values directly utilized by the selected program components to enable replay. The approach is similar to carving with a touched projection with the difference that simplified heap structures are used to represent the program state. Second, the test factoring approach introduced by Saff and Ernst takes a similar approach to Orso's with the creation of mock objects that serve to create the scaffolding to support the execution of the test unit [43]. The same group introduced a tool set for a fully featured Java execution environments that can handle many of the subtle interactions present in this programming language (e.g., callbacks, arrays, and native methods) [42]. Saff et al.'s work [34] carves a method test case by recording

the sequence of calls that can influence the method, 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 __ such that $s(\overline{\sigma}) = s_{pre}$ for the method of interest and then calculating summarizing traces __calli that reflect the return value and side effects for each call out of the method and carving s_{pre}, the relevant prestate for each call. During replay, the same sequence of calls with the same parameters is expected—any deviation results in areport of a differenceduring replay. In our framework, we would identify the points at which the n calls out of the method occur as poststate locations to define a DUT of the form $((\overline{\sigma}_1.....\overline{\sigma}_j), s_{pre}, (s_{post1}, ...s_{posth}))$. The approaches introduced by Orso et al. and Saff et al. are action-based approaches that capture the interactions between 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 targeted unit. On the other hand, these approaches are likely to generate inefficient unit tests in the presence of long-running system tests and they may generate tests that are too sensitive to simple changes that do not effect meaning of the unit (e.g., changing the order of independent method calls). Saff et al. have identified this issue and propose to analyze the life span of the factored test cases across sequences of method modifications [42].

This is a critical factor in judging the cost-effectiveness of CR testing, and we have started to study it in Section 4.5. 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 statebased approach to CR testing. More recently, Xu et al. have proposed a hybrid approach that mixes action based with state based to enhance replay efficiency [50]. The approach only captures the set of runtime values required to reach a checkpoint and the values that could potentially be required to complete execution after the checkpoint. The set of runtime values required is obtained by computing the slice of the program required to generate those values (similar to action based). The set of values that could be required to complete execution is computed by walking the heap (similar to state based). In our framework, such a test could be defined by calculating traces $\overline{\sigma}_{control}$ control leading to checkpoint s_{pre} and a number of states s_{post1} corresponding to the method return points. This would result in a DUT of the form $((\overline{\sigma}_1.....\overline{\sigma}_j), s_{pre}, (s_{post1}, ...s_{posth}))$ where j is the number of relevant subtraces that lead to the checkpoint stack and h is the number of states that affect the postcheckpoint program execution.

All of these related efforts have shown their feasibility in terms of being able to replay tests and Saff et al.'s and Xu et al.'s approaches have provided initial evidence that they can save time and resources under several scenarios. None of these approaches, 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. JUnit, for example, is a popular framework for the Java programming language that lets programmers attach testing code to their classes to validate their behavior [31]. There are also multiple approaches that automate, to different degrees, the generation of unit tests. For example, commercial tools such as Jtest, developed by a company called Parasoft, develop unit test cases by analyzing method signatures and selecting test cases that increase some coverage criteria [51]. Some of these tools aim to assess software robustness (e.g., whether an exception is thrown [52]). Others utilize some type of specification such as pre and postconditions or operational abstractions, to guide the test case generation and actually check whether the test outcome meets the expectation results [23], [27], [37], [45]. Interestingly enough, a part of JTest called JTest Tracer can be used to monitor a deployed application in real time and capture inputs to generate realistic JUnit test cases [51], a process somewhat similar to carving. Although carving also aims to generate unit test cases, the approach we propose is different from previous unit test case generation 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 exposed by systems tests. In general, such interactions are hard to design and are rarely included in regular unit test cases. As stated, the poststate differencing functions that regulate the detection 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 [44] on the development of pseudo-oracles,

Jaramillo et al. [32] on using comparisons to check for optimization induced errors in compilers, or the comparison of program spectra [39] are instances of utilizing differencing-type oracles at the system or subsystem level. When focusing at the unit level of object-oriented programs, as we are doing, Binder suggests the term "concrete state" oracles, which aim to compare the value of all the unit's attributes against what is expected [20]. Briand et al. referred to this type of oracle as a "precise" oracle because it was the most accurate one employed in their studies [24]. Overall, the notion of testing being fundamentally differential has long been understood [44], since the pseudo-oracles against which systems 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 difference reflects an error.

CONCLUSION

We have presented a general framework for automatically carving and replaying DUTs. The framework accommodates two types of state representation, and incorporates sophisticated projection, anomaly handling, and differencing strategies that can be instantiated in various ways to suit distinct trade-offs. We have implemented a state-based instance of the framework that mitigates testing costs through a family of reachability-based projections, that enhances DUT robustness through replay anomaly handlers,

and that can adjust the sensitivity of DUTs through differencing functions. Our evaluation of this implementation on Siena, NanoXML, and JTopas provides evidence that DUTs can be generated automatically from system tests, can provide efficiency gains of orders of magnitude while retaining most of the effectiveness of system tests in a regression testing context, and can be robust to program changes and scale to large and complex heap structures. The experiences gained while instantiating and assessing the framework suggest several directions for future work.

First, we will perform further studies not only to confirm our findings on other artifacts under similar settings but also to compare DUTs with traditional 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. A longer-term direction is the exploration of other transformation techniques that utilize our current test representation. For example, we are investigating automated mechanisms that combine multiple DUTs to create an aggregated DUT for a larger program unit such as a class. This could be achieved by clustering multiple DUTs based on the identity of the receiver object, effectively transferring the effects of methods on the receiver object throughout the sequence, achieving a kind of interaction testing between calls. Ultimately, we envision a family of automated transformations of testing resources where carving is just one of those transformations.

Applications

Generating tests of different granularity.

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. 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.

Tool Support

Features included:

- Automatically carving DUTs from a running application
- DUTs reductions and filtering projections (e.g. identical pre-state representations)
- Ability to customize post-state output (simple hash for space savings, full state for easier analysis, etc.)
- Customizable replay specifications for individual methods or sequence of methods
- Anomaly replay handler (e.g., to replay caller of failed method or replayable frontier)
- DUT to JUnit Translation

Reference

[1]. B. Beckert and C. Gladisch. White-box testing by combining deduction-based specification extraction and black-box testing. In *TAP*. Springer, 007.

[2]. B. Beckert, R. H¨ahnle, and P. H. Schmitt, editors. *Verification of bject-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.

[3]. L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In *FME*, pages 422–439, 2003.

[4]. S. G. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE TSE*, 35(1):29–45, 2009.

[5]. C. Engel, C. Gladisch, V. Klebanov, and P. R¨ummer. Integrating verification and testing of object-oriented software. In *TAP*, pages 182–191, 2008.

[6]. C. Engel and R. H¨ahnle. Generating unit tests from formal proofs. In *TAP*, pages 169–188, 2007.

[7]. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[8]. C. Gladisch. Verification-based test case generation for full feasible branch coverage. In *SEFM*, pages 159–168, 2008.

[9]. T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *TOSEM*, 10(2):184–208, 2001.

[10]. P. Hamill. *Unit test frameworks*. O'Reilly, 2004.

[11]. M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, And A. Gujarathi. Regression test selection for Java software. *SIGPLAN Not.*, 36(11):312– 326, 2001.

[12]. T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: unit testing with mock objects. In *Extreme Programming Examined*, pages 287–301. Addison-Wesley, 2001.

[13]. B. Pasternak, S. Tyszberowicz, and A. Yehudai. Genutest: a unit test and mock aspect generation tool. *Journal on Software Tools for Technology Transfer*, 2009.

[14]. D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for java. In *ASE*, pages 114–123, 2005.

[15]. N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *TAP*, pages 134–153, 2008.

[16]. H. van Vliet. *Software Engineering: Principles and Practice (2nd ed.)*. John Wiley & Sons, Inc., 2000.

[17]. T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP*, pages 380–403, 2006.

[18]. T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant Object oriented unit tests. In *Proc. 19th ASE*, pages 196–205, September 2004.

[19]. Extreme Programming. http://www.extremeprogramming.org. Visited January 2010.

[20]. R. Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools, chapter 18, Object Technologies, pp. 943-951, first ed. Addison Wesley, Oct. 1999.

[21]. D. Binkley, "Semantics Guided Regression Test Cost Reduction," IEEE Trans. Software Eng., vol. 23, no. 8, pp. 498-516, Aug. 1997.

[22].D. Binkley, R. Capellini, L. Ross Raszewski, and C. Smith, "An Implementation of and Experiment with Semantic Differencing," Proc. IEEE Int'l Conf. Software Maintenance, pp.82-91, Nov. 2001.

[23].C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated Testing Based on Java Predicates," Proc. Int'l Symp. Softwar Testing and Analysis, pp. 123-133, July 2002.

[24].L.C. Briand, M. Di Penta, and Y. Labiche, "Assessing and Improving State-Based Class Testing: A Series of Experiments, IEEE Trans. Software Eng., vol. 30, no. 11, pp. 770-793, Nov. 2004.

[25].A. Carzaniga, D. Rosenblum, and A. Wolf, "Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service," Proc. 19th Ann. ACM Symp. Principles of Distributed Computing, pp. 219-227, July 2000.

[26].Y.-F. Chen, D.S. Rosenblum, and K.-P. Vo, "TestTube: A System for Selective Regression Testing," Proc. 16th Int'l Conf. Software Eng., pp. 211-220, May 1994.

[27].Y. Cheon and G.T. Leavens, "A Simple and Practical Approach to Unit Testing: The JML and JUnit," Proc. 16th European Conf. Object- Oriented Programming, pp. 231-255, June 2002.

[28].H.N. Chin, S. Elbaum, M.B. Dwyer, and M. Jorde, "DUTs: Targeted Case Studies," Technical Report TR-UNL-CSE-2007- 0005, Univ. of Nebraska, Aug. 2008.

[29].S. Dieckmann and U. Holzle, "A Study of the Allocation Behavior of the Specjvm98 Java Benchmark," Proc. 13th European Conf. Object-Oriented Programming, pp. 92-115, June 1999.

[30].H. Do, S.G. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact," Empirical Software Eng.: An Int'l J., vol. 10, no. 4, pp. 405-435, Oct. 2005.

[31.]E. Gamma and K. Beck, JUnit, http://sourceforge.net/projects/ junit, Dec. 2005.

[32].C. Jaramillo, R. Gupta, and M.L. Soffa, "Comparison Checking: An Approach to Avoid Debugging of Optimized Code," Proc. European Software Eng. Conf./Foundations of Software Eng., pp. 268- 284, Sept. 1999.

[33] H.K.N. Leung and L. White, "Insights into Regression Testing," Proc. IEEE Int'l Conf. Software Maintenance, pp. 60-69, Oct. 1989.
[34] H.K.N. Leung and L. White, "A Study of Integration Testing and Software Regression at the Integration Level," Proc. IEEE Int'l Conf. Software Maintenance, pp. 290-300, Nov. 1990.
[35] A.K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression Testing in an Industrial Environment," Comm. ACM, vol. 41, no. 5, pp. 81-86, May 1998.
[36] A. Orso and B. Kennedy, "Selective Capture and Replay of Program Executions," Proc. Third Int'l Workshop Dynamic Analysis, May 2005.
[37] C. Pacheco and M.D. Ernst, "Eclat: Automatic Generation and Classification of Test Inputs," Proc. 19th European Conf. Object- Oriented Programming, pp. 504-527, July 2005.
[38] S.K. Reddy, "Carving Module Test Cases from System Test Cases: An Application to Regression Testing," master's thesis, Dept. of Computer Science and Eng., Univ. of Nebraska, July 2004.
[39] T. Reps, T. Ball, M. Das, and J. Larus, "The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem," Proc. European Software Eng. Conf./Foundations of Software Eng.), pp. 432-449, Sept. 1997.
[40] G. Rothermel, S. Elbaum, and H. Do, Software Infrastructure Repository, http://cse.unl.edu/galileo/php/sir/index.php, Jan. 2006.
[41] G. Rothermel and M.J. Harrold, "Analyzing Regression Test Selection Techniques," IEEE Trans. Software Eng., vol. 22, no. 8, pp. 529-551, Aug. 1996.
[42] D. Saff, S. Artzi, J. Perkins, and M. Ernst, "Automated Test Factoring for Java," Proc. 20th Ann. Int'l Conf. Automated Softwar Eng., pp. 114-123, Nov. 2005.
[43] D. Saff and M. Ernst, "Automatic Mock Object Creation for Test Factoring," Proc.
SIGPLAN/SIGSOFT Workshop Program Analysis for Software Tools and Eng., pp. 49-51, June 2004.
[44] E.J. Weyuker, "On Testing Non-Testable Programs," The Computer J., vol. 25, no. 4, pp. 465-470, Nov. 1982.
[45] T. Xie and D. Notkin, "Tool-Assisted Unit-Test Generation and Selection Based on Operational Abstractions," Automated Software Eng. J., July 2006.
[46] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer and Matthew Jorde, "Carving and Replaying Differential Unit Test Cases from System Test Cases" IEEE Transactions On Software Engineering, vol. 35, no. 1, January/February 2009
[47] V.P. Ranganath and J. Hatcliff, "Pruning Interference and Ready Dependence for Slicing Concurrent Java Programs," Proc. 13th Int'l Conf. Compiler Construction, pp. 39-56, Apr. 2004.
[48] S. Elbaum, P. Kallakuri, A.G. Malishevsky, G. Rothermel, and S. Kanduri, "Understanding the Effects of Changes on the Cost- Effectiveness of Regression Testing Techniques," J. Software Testing, Verification, and Reliability, vol. 13, no. 2, pp. 65-83, June 2003.
[49] B. Weide, "Modular Regression Testing: Connections to Component-Based Software," Proc. Fourth ICSE Workshop Component- Based Software Engineering, pp. 82-91, May 2001.
[50] G. Xu, A. Rountev, Y. Tang, and F. Qin, "Efficient Checkpointing of Java Software Using Context-Sensitive Capture and Replay," Proc. ACM SIGSOFT Symp. Foundations of Software Eng., pp. 85-9 Oct. 2007.
[51] JTest, Jtest Product Overview, http://www.parasoft.com/jsp/products/home.jsp?product=Jtest, Oct. 2005.
[52] C. Csallner and Y. Smaragdakis, "Jcrasher: An Automatic Robustness Tester for Java," Software Practice and Experience, vol. 34, no. 11, pp. 1025-1050, Sept. 2004.
[53] Xstream—1.1.2, XStream, http://xstream.codehaus.org, Aug. 2005.

Authors Biography

Dr. C. P. V. N. J. Mohan Rao, is working as Professor & Principal of Avanthi College of Engineering & Technology, Tamaram, Makavarapalem, Narsipatnam (RD), Visakhapatnam, Andhra Pradesh, INDIA. He obtained PhD from Andhra University and having 15 years experience. He published more than 18 papers in reputed Journals.

Nandagiri R G K Prasad, Studying M.Tech in Software Engineering, in CSE Department, Avanthi College of Engineering & Technology, Tamaram, Makavarapalem, Narsipatnam (RD), Visakhapatnam, Andhra Pradesh, INDIA. I worked as an Assistant Professor in CSE Department at VITAM College of Engineering, Sontyam, Anadapuram Mandal, Visakhapatnam, Andhra Pradesh, India.

Somayajula Satya Pavan Kumar is working as Assistant Professor, in CSE Department, , Avanthi College of Engineering & Technology, Tamaram, Visakhapatnam, A.P., India. He has received his M.Sc(Physics) from Andhra University, Visakhapatnam and M.Tech (CST) from Gandhi Institute of Technology And Management University (GITAM), Visakhapatnam, Andhra Pradesh, INDIA. His research areas include Software Engineering and network security.