1 Introduction

- **Differential unit test (DUT)** are a hybrid of unit and system tests that exploits their strengths.
- DUT 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 unit and then reassembling those components so that the unit can be exercised as it was by the system test.

1.1 Unit Tests

- Unit tests aim to validate individual program units (e.g., method, classes and package) before they are integrated into the whole system. Unit tests are not constrained or influenced by other parts of the system. Thus unit tests are more efficient in test execution and fault isolation to system testing and debugging.
- Challenges or limitations: 1. Specifications of unit behavior are usually informal and incomplete.
- 2. Specifications may evolve independently of implementations requiring additional maintenance of unit tests.

1.2 System Tests

- System tests may expose faults that unit tests do not, for example, faults that emerge only when multiple units are integrated.
- In system tests, no individual harnesses need to be constructed.
- Several disadvantages:
  - 1. System tests are expensive to execute.
  - 2. System testing may fail to exercise the full range of behavior implemented by a system’s particular unit.
  - 3. Fault isolation and repair during system testing can be more expensive than during unit testing.
1.3(part1) Differential Unit Test (DUTs)

- System and unit tests have complementary strengths and that they offer a rich set of trade-offs.
- So, this paper proposes a general framework for the carving and replacing of what they call differential unit tests (DUTs) which aims to exploit those trade-offs.
- DUTs are created from system tests by capturing components of the exercised system that may influence the behavior of the targeted unit and reflect the results of executing the unit; they term this carving. 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 replaying and the resulting state is queried to determine if there are differences with record unit prestate.

1.3(part2) DUTs

Ideally, a set of DUT will
1. Retain the fault detection effectiveness of system test on the target unit,
2. Execute faster or use fewer resources than system tests, and
3. Be applicable across multiple system versions.
4. In addition, for program changes that are behavior preserving, effective DUTs will report few differences that are not indicative of actual differences in system test results.

2.1 A Framework for Test Carving and Replay

- 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.

2.2 Basic Carving and Replaying

A sequence of program states is written as \( \sigma = s_0, s_1, \ldots, s_i \in S \) and \( s_0 \) is the initial state.

Given a system test case \( s_t \), carving a unit test \( \text{DUT}_{xm} \) for target unit \( m \) during the execution of \( s_t \), consists of capturing \( s_{pre} \), the program state immediately before the first instruction of an activation of method \( m \), and \( s_{post} \), the capture pair of states \((s_{pre}, s_{post})\) define the DUT case for method \( m \), denoted \( \text{DUT}_{xm} \).

In practice, a method \( m \) can change over the program lifetime. We replay \( \text{DUT}_{xm} \) on \( m' \). By either loading state \( s_{pre} \) into memory or by executing \( \text{DUT}_{xm} \), execution of \( m' \) is initiated and it reaches the point corresponding to \( s_{post} \). Then current state \( s'_{post} \) compared to \( s_{post} \). If they are the same, the change did not affect the behavior of the target unit exercised by \( \text{DUT}_{xm} \). Otherwise, if the change alter the \( m \), we need to determine whether the alteration is expected.

![Fig. 1. Carving and replay process.](image-url)
2.3 CR approach limitations

- CR approach has some limitations.
- First, the proposed basic carving procedure is inefficient and impractical.
- Second, two distinct complete program state may be identical from the point of view of a given method, thus carving complete states would yield redundant unit tests.

2.4.1 Improving CR with Projections-

- State-based projections is the notion of heap reachability.
- An object o’s reachable in one deference from object o if the value of some field f references o’;
- let reach(o)={o'| ∃ f ∈ Fields(c) Class(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.
- State-based projections retain at most the set of heap objects reachable from a given calling context.

2.4.2 Improving CR with Projections-

- Action-based projections and transformation
- Projections can also operate on sequences of program actions but a purely action-based approach to state capture may fail in some program. For example, a program that calls native method does not access to native method instructions.
- To accommodate this, they allow for transformation of actions during carving, i.e., replace one sequence of instructions with another.

2.4.3 Applying projections on DUTs

1. Two potential applications of projection on DUTs: test case reduction and test cases filtering
2. Reduction aims at thinning carved test case by retaining only the projected prestate.
3. Filtering aims at removing redundant DUTs from the suite.

For example, a method that is invoked during program initialization and is independent of the program parameters would be exercised by all system tests result in redundant DUTs. Filtering by comparing complete prestate could remove such duplicate tests.
2.5 Strategies to Manage Replay Anomalies

- Overly aggressive reductions can impair replay. Similarly, certain method changes such as modification’s signature or key data structures may prevent a DUT from correct replay.

1. When DUT$_{xm}$ fails, 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.

2. An alternative approach that avoids system test case execution and immediate carving takes advantages of the existing body of executable DUTs on other methods that exercise the target method.

2.6 Adjusting Sensitivity Through Differencing Functions

- Differencing in CR testing is achieved by evaluating $\text{dif}(s_{\text{post}}) = \text{dif}(s_{\text{post}}')$.

- A central issue in differential testing is the degree to which differencing functions are able to detect changes, called sensitivity, that corresponding to faults while masking implementation changes.

1. They address this by allowing for multiple differencing functions to be applied in CR testing, which has the potential to increase fault sensitivity.

2. Another method to increase sensitivity in the temporal dimension by capturing sequences of poststates $(s_{\text{post}}, \sigma_{\text{post}})$ that capture intermediate points during the execution of the method under test.

3. Evaluation

- After some empirical study, they assess execution efficiency, fault detection effectiveness, and robustness of DUTs.

- 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 structure.

Conclusion

- 1. They propose a general framework for automatically carving and replaying DUTs.

- 2. 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.

- Questions?