8-12-2008

DUTs: Targeted Case Studies

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

Sebastian Elbaum
University of Nebraska-Lincoln, elbaum@cse.unl.edu

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

Matthew Jorde
University of Nebraska - Lincoln, majorde@cse.unl.edu

University of Nebraska–Lincoln, Computer Science and Engineering
Technical Report TR-UNL-CSE-2008-0005
Issued August 12, 2008

DUTs: Targeted Case Studies

Hui Nee Chin, Sebastian Elbaum, Matthew B. Dwyer, Matthew Jorde
Dept. of Computer Science and Engineering, University of Nebraska
Lincoln, NE, USA
{elbaum,hchin,dwyer,majorde}@cse.unl.edu

I. Introduction

Our previous studies of DUTs addressed research questions of effectiveness, efficiency, and robustness with respect to one software artifact, Siena, and we believe the findings generalize to similar artifacts. Still, we realize that existing studies suffer 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 start to address those threats to the validity of our findings by investigating the performance of CR in the presence of such settings.

The findings in this report reveal that the performance of the different carving strategies can vary significantly in programs with complex heap structures, that the *ReplayAnomalyHandler* 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.

II. DUTs Robustness in the Presence of Program Changes

We are interested in answering the following questions:

- what is the impact of deeper and more complex heap structures on the performance of CR?
- how robust are DUTs under extensive software modifications?
- what is the cost of utilizing the *ReplayAnomalyHandler*
- what is the fault detection effectiveness of DUTs replayed using the *ReplayAnomalyHandler*?

We selected NanoXML, a command-line XML parser implemented in Java, as the artifact of study. NanoXML has two attributes that make it valuable for study. First, it has a large number of changes across versions which makes it appropriate to answer the robustness to change questions. Table I illustrates the degree of change through the number of modifications in class structures (e.g. changing the data structure of a class field from `java.util.Properties` to `java.util.Vector`). [1] Second, the heap structure of NanoXML is much deeper than Siena, which may expose further differences across the set of projections we have implemented. We obtained NanoXML from the SIR repository [1], and it includes the source code, system level test suites containing slightly over 120 test cases, and multiple versions corresponding to product releases. Since NanoXML is a component library, for the purposes of this analysis, we only consider the core components of NanoXML and not its test drivers. We utilize four versions of NanoXML that had faults exposed by its system tests.

[1]The actual number of changed methods may be slightly higher if we include changed methods in classes that did not undergo class structure changes.

Carving versions	Metric	Reduction *C-retest-all*			
		1	5	∞	touched
v0	Minutes	7	11	17	25
	Mb	56	311	671	83
DUTs		1030	6170	9491	1605
Percentage of DUTs with sentinels		83%	87%	-	86%

TABLE II
Carving Times and Sizes to Generate the Initial DUT Suites for NanoXML.

Similar to the study on Siena, the assessment was performed through the comparison of system tests and their corresponding carved unit test cases. We consider three types of regression testing techniques, *S-retest-all*, *C-retest-all-k**, and *C-retest-all-touched* (we decided not to employ regression test selection techniques because there are enough changed methods per version to cause the selection of the majority of the system tests.) For the robustness assessment, we want to know to what degree the DUTs from the initial carved test suites can be used to test the changed methods. We measure the fault detection effectiveness and the time to generate the carved DUTs and the time to replay the DUTs for the changed methods, as performed in the previous study.

A. Results.

Table II summarizes the time (in minutes) and the size requirements (in Mb) to carve the four initial *C-retest-all* test suites, as well as the number of DUTs generated for all methods executed in each suite. Constraining the carving depth greatly reduces *both* carving time and storage space for NanoXML, which is something we did not observed in Siena. This difference is caused by the longer reference chains of NanoXML as evidenced by the percentage of sentinels in $k = 5$ which indicate that there are plenty of references longer than length five. Another interesting difference with Siena is that applying the touched-carving projection resulted in carving execution times that are over 30% greater on average than the carving times for *C-retest-all-k*∞, caused by the higher level of bookkeeping required by such references.

To assess CR robustness in the presence of change, we analyzed in detail only the seven faulty methods of the artifact. We find that only one of the seven could be replayed without invoking the *ReplayAnomalyHandler*, whereas post-states for the other six faulty methods could be recorded only by traversing up the

Version	Physical SLOC	Classes	Change-covered classes	Number of methods		System test suites size	Faults
				In affected classes	In total		
v0	3708	10	-	-	106	120	-
v1	4334	12	7	71	120	123	1
v3	7185	15	3	73	213	128	1
v5	7646	19	4	34	232	128	5

TABLE I
NANOXML'S COMPONENTS ATTRIBUTES.

Fault instance	Reached `main`	Average # methods visited	Average traversal length	DUTs replayed	Replay Times (seconds)			
					k1	k5	k∞	*touched*
v1	Yes	4	3	86	6	15	26	20
v3	Yes	1	1	48	12	16	21	13
v5:f1	Yes	6	6	70	10	18	25	11
v5:f2	Yes	6	6	70	10	17	22	11
v5:f3	Yes	4	9	12	8	13	18	13
v5:f4	Yes	4	3	86	7	16	22	21
v5:f5	No	1	0	1	5	5	5	6

TABLE III
REPLAYING TIMES AND FRONTIER FOR NANOXML.

	PP				FF			
	C-retest-all				C-retest-all			
	k1	k5	k∞	touched	k1	k5	k∞	touched
v1	100	**50**	**54**	**54**	**0**	100	100	100
v3	100	100	100	100	100	100	100	100
v5:f1	**35**	**50**	**50**	**46**	100	100	100	100
v5:f2	**62**	**62**	**62**	**51**	100	100	100	100
v5:f3	100	100	**42**	**47**	**0**	**0**	100	100
v5:f4	100	**55**	**55**	**55**	**0**	100	100	100
v5:f5	100	**10**	**10**	**10**	**0**	100	100	100

TABLE IV
FAULT DETECTION EFFECTIVENESS FOR NANOXML.

call graph and replaying their caller(s) method(s). Columns 2-5 of Table III provide more details on the size of the frontier explored, while columns 6-9 show the required replay time. Note that exercising the faulty methods in the first six rows required to replay all methods up to *main*. Interestingly, utilizing the *ReplayAnomalyHandler* to identify the replayable frontier implied replaying an average of 62 DUTs but increased replay time by a factor of 5 at most. The general tendencies about the replay times across the different carving suites is similar to what was observed for Siena. Touched replay times were usually between those of $k = 1$ and $k = \infty$, however in some cases the replay time was closer to that of $k = 1$ and in other cases the replay time was closer to that of $k = \infty$. This variation is caused by the different dereference chain lengths used in the various methods tested.

To assess the fault detection effectiveness of these DUTs 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 IV presents the PP and FF values for the different carving projections under all version instances. All DUTs generated through *C-retest-all-k*∞ detected a difference in the faulty methods. Furthermore, many PP values are not 100, indicating that many DUTs have higher difference detecting power than their corresponding system test cases. For example, only 50% of the passing system tests in the *S-retest-all* test suite for $v5 : f1$ had all their corresponding DUTs passing in the *C-retest-all-k*∞ suite, while 46 DUTs failed even though they were carved from passing system tests. More importantly, we observed that limiting the carving depth has a profound impact on the FF proxy measure of fault detection effectiveness. Restricting the depth $k = 1$ led to lower fault detection effectiveness ($v1$, $v5 : f3$, $v5 : f4$, and $v5 : f5$). Even limiting the depth to $k = 5$ caused the fault in $v5 : f3$ to go undetected by the *C-retest-all-k5* suite. Despite the relatively low number of DUTs carved, applying the touched-carving projection yielded fault detection results similar to those of depth $k = \infty$.

III. DUTS SCALABILITY THROUGH CLUSTERING

Reducing the number of DUTs is crucial to make the CR approach scalable, and the projections we studied were helpful in that regard. However, during our investigation, we also noted that some systems performed a large number of method invocations

within the same calling context, which lead to the development of the *clustering* projection. This projection is unique in that it defines at run-time when to stop generating DUTs for a method once a threshold is met. The smaller number of DUTs may also result in less replay time. Through this study we aim to explore the degree to which *cluster-based* filtering can reduce the number of DUTs.

Since neither Siena nor NanoXML exhibited this attribute, we searched for an additional artifact. The artifact we chose for this analysis is JTopas, a Java library for tokenizing and parsing, which is available for download from the SIR repository [1]. On average, a JTopas system tests executes over 450,000 methods, and each method is invoked an average of over 6,000 times, making it appropriate for this study.

Since the focus of this study is on the effectiveness of the clustering projection, instead of providing a wide test suite characterization, we decided to focus in more depth on just 5 randomly selected JTopas tests. From these 5 tests we carved two DUT suites: *C-random*, which is the carved test suite generated from the 5 randomly selected system tests, and *C-random-c**, which corresponds to the carved test suites generated from the same system test cases utilizing the *clustering* filter with four different clustering thresholds, c, of 10, 100, 1000, and 2000. All carved test suites were subjected to the interface reachable projection as well. We assess the performance of the suites by measuring the carving time, and the number of DUTs carved, which serves as a proxy for replay time.

A. Results

Figure 1(a) illustrates the carving time for the test suite which shows that not all clustering threshold levels provided carving time savings; specifically, using the threshold of 10 resulted in a 52% increase in carving time. On further examination, however, we noticed that this was not the case for every test case. Carving two of the test cases ($t1$ and $t2$) did not result in any clustered methods, hence their carving times remained approximately the same regardless of the threshold values. One test case ($t3$) always benefited from clustering, while when carving the other two test cases ($t4$ and $t5$) the performance varied depending on the chosen clustering threshold.

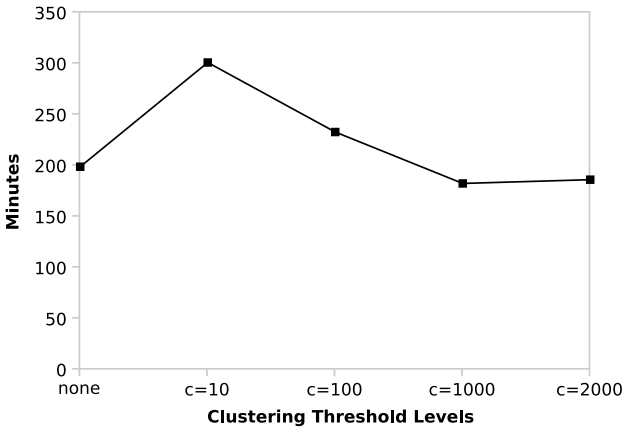
This performance variation is illustrated in Figure 1(b) and corresponds to the different test execution patterns exhibited under the different thresholds for $t1$–$t5$ individually. The variation in performance across the thresholds is caused by several factors. As the clustering threshold is lowered, more DUTs can be clustered. However, lowering the threshold also means that more method's DUTs must be rearranged (DUTs are replaced with markers pointing to the caller DUTs so that the target method can be selected for replay by the user without noticing the underlying clustering). As a result, for a given test, some thresholds will not benefit carving when there are not enough DUTs to offset the cost of removal and marking. Figure 2 presents the conversion (removal and marking) effort required for $t3$ and $t5$ at each threshold level to further illustrate this tradeoff. For instance, for $c = 10$, $t3$ performed 43 conversions and $t5$ performed 1104 conversions, while for $c = 100$ $t5$ performed 340 conversions.

Next we examine the effects of clustering on replay efficiency. Table V presents the number of DUTs carved at each threshold level. Clustering managed to reduce the number of DUTs by an average of 33% at the lowest clustering threshold (from 551202 to 368865). Note that, as observed before, not all test cases benefited the same way at each clustering threshold level. When looking at particular JTopas methods, we found that 24 of the 86 methods invoked by the test suite had DUTs that were clustered with $c = 10$, whereas 16 different methods had DUTs that were clustered with $c = 100$. Only one method had DUTs clustered at higher thresholds. Program methods invoked by each test case were affected differently; for example, DUTs from 9% of the methods invoked in $t3$ could be clustered while DUTs from 26% of the methods invoked in $t5$ could be clustered.

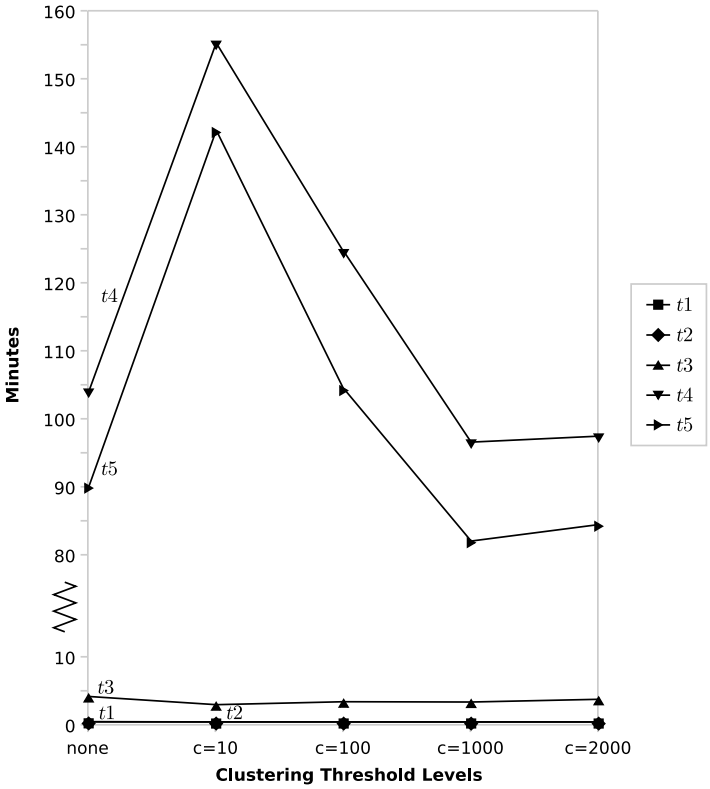
These data suggest that some analysis of individual test cases could be helpful in determining the suitable clustering threshold. One way to analyze the tests could be to measure the average number of times a method is invoked by each system test case. For example, the average number of times a method is invoked in $t3$ is 199 while each method is invoked an average of 5322 times in $t5$. This could serve as an indication that a threshold of 100 might be suitable for $t3$ while it might be too low for $t5$.

REFERENCES

[1] Gregg Rothermel, Sebastian Elbaum, and Hyunsook Do. Software infrastructure repository. http://cse.unl.edu/ galileo/php/sir/index.php, January 2006.



(a) Carving Time for Test Suite



(b) Carving Time for Tests

Fig. 1. Carving times with different clustering thresholds.

Test case	Number of DUTs to replay				
	Before clustering	After clustering			
		c=10	c=100	c=1000	c=2000
$t1$	59	59	59	59	59
$t2$	28	28	28	28	28
$t3$	11404	3523	4264	6481	6481
$t4$	309455	222226	250480	304533	304533
$t5$	230256	143029	171281	225333	225333
Total	551202	368865	426112	536434	536434

TABLE V
NUMBER OF DUTS TO REPLAY AT EACH CLUSTERING THRESHOLD.

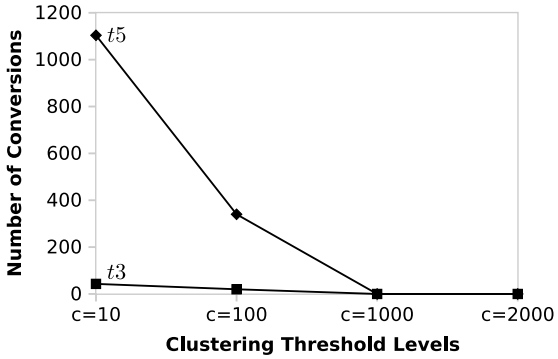


Fig. 2. Conversion effort for $t3$ and $t5$ at each threshold level.