Answers to Reviewer Questions

A. Answer to R2Q1: When should/should not developers use Semter?

Motivation and Approach. As suggested by Reviewer 2, we try to figure out whether Semter is recommended for developers when they fix a bug or introduce new requirements between the two versions. Specifically, we manually collect description information from the version updates, including the readme files of the new version, change logs, and GitHub commit, for our studied web applications and summarize the change points between the two successive versions in Table I of the original submission. We group each change point into (a) bug fixing and (b) new requirements and employ the ratio of the number of bug fixing points to the total number of change points as the metric to indicate how version updates are related to bug fixing[1]. In this RQ, we aim to check the correlation between the effectiveness of our method and this ratio. Through our investigation, we observe that a few version updates contain descriptions of the change points in *AddressBook* and *Claroline*. As a result, we only consider the other two web applications, *Collabtive* and *Password-Manager*. We count the related the effectiveness of our method (i.e., the breakage repair ratio) and the ratio of bug fixing points to total change points for the two web applications, and calculate their Pearson correlation values.

Results. Our results are shown in Fig. 1, where the x-axis indicates the ratio of bug fixing points after each version update, i.e., the number of bug fixing points / the total number of bug fixing points and new requirements points, and the y-axis represents the breakage repair ratio for each new version. From Fig. 1, it can be observed that the breakage repair ratio has no apparent relationship with the bugs fixed or new requirements introduced. Besides, Pearson correlation values on *Collabtive* and *Password-Manager* are 0.02 and 0.32, respectively. This also indicates no strong correlation between the repair effectiveness and the bugs fixed or new requirements introduced after the new version update.

From the previous experiment in our paper, we observe that semantic information of most UI elements is relatively reliable, i.e., unchanged or changed slightly, as the web application under test evolves. That is the critical factor that makes Semter achieve the best repair effectiveness compared with other repair techniques. To illustrate, we take Addressbook 3.0 and 4.0 as an example. Fig. 2 shows the GUIs of the home pages on the two versions. As can be seen, the GUI (e.g., the layout and background color) has significantly changed; on the contrary, semantic information (e.g., text) of UI elements is changed slightly. As reported in our paper, 149 test breakages occurred between Addressbook 3.0 and 4.0. The repair ratio of Semter is as high as 96.0%, which is much better than baselines. This example would illustrate that Semter works better on test repairs when the semantic information of UI elements is not significantly changed, and the other aspects (e.g., the layout and background color) could be greatly changed between the two releases.

[1]See details in the excel file fix+requirements.xlsx in the same directory.

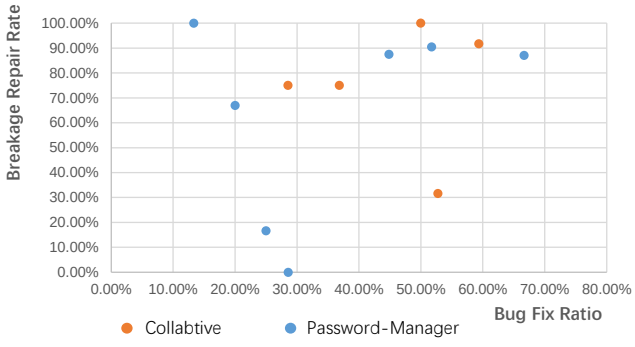


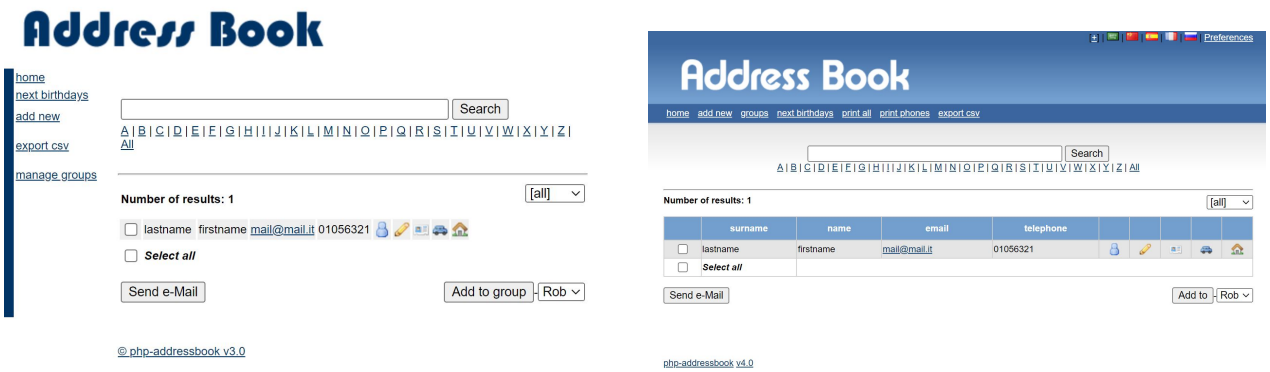
Fig. 1. Relationship between the breakage repair ratio and the bug fix ratio after each version update on Collabtive and Password-Manager.

> Answer to R2Q1: We observe no strong correlation between the transformation motivation (e.g., bug fixing and new requirements) and the repair effectiveness of Semter. Besides, we suggest that developers should use Semter on test repairs when the semantic information of UI elements is not significantly changed between the two releases.

B. Answer to R2Q2: Does Semter work better when analyzing course-grained or fine-grained transformations?

Motivation and Approach. We aim to analyze whether Semter works better for course-grained or fine-grained transformations. We employ the code change rate (for adjacent versions of the web applications) to divide the transformations into course-grained and fine-grained. To calculate the code change rate, we use the git diff to count each updated version's number of code change lines. We use the median value[2] to classify the code change rate into two groups as low change rate (equal to or less than the median) and high change rate (greater than the median). Naturally, version updates with a low code change rate are considered fine-grained transformations, while version updates with a high code change rate are

[2]The median in the data set is the value such that half of the data set is less than the median and half is greater than it.



(a) AddressBook-3.0 (b) AddressBook-4.0

Fig. 2. The comparison of UI elements between AddressBook-3.0 and AddressBook-4.0

considered course-grained transformations[3]. Then we calculate the average breakage repair ratios corresponding to each web application's fine-grained version update and course-grained version update.

Results. Fig. 3 presents the result of the average breakage repair ratios for fine-grained and course-grained transformations. From Fig. 3, it can be seen that for *AddressBook*, *Claroline*, and *Collabtive*, repair effect is better for fine-grained transformations, while for *Password-Manager*, Semter has very similar effects on fine-grained and course-grained transformations. On the whole, Semter performs well for both course-grained and fine-grained transformations and does slightly better when analyzing fine-grained transformations than course-grained ones.

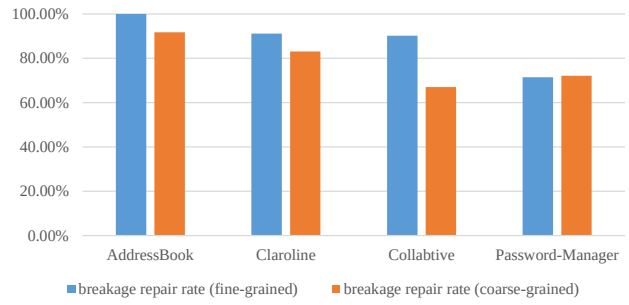


Fig. 3. The average breakage repair ratios for fine-grained and course-grained transformations.

Answer to R2Q2: Semter performs well for both course-grained and fine-grained transformations, and it does slightly better when analyzing fine-grained transformations than course-grained.

[3]See details in the excel file versions_diff.xls in the same directory.

C. R2Q3: Correctness in the evaluation section?

Motivation and Approach. As suggested by Reviewer 2, to evaluate the correctness of test repairs, we conduct an empirical study on Claroline. We generate 330 mutants of Claroline. Mutation operators are presented in Table I. We reuse some GUI mutation operators [1] and design some new operators that simulate GUI changes leading to real test breakages. Mutation operators can be roughly divided into three groups, namely removing, adding, and modifying the UI element. We adopt the same parameter setting as our previous experiment, and each mutant is used as the updated release. One test breakage occurs on each mutant. The correctness of repairs is checked manually.

Results. Table I shows the results of our methods on mutations and the distribution of breakages and correctly repairs across different mutations. SEMTER is capable of repairing all 330 test breakages that occur in the experiment. The manual check shows that, in total, 92.4% (i.e., 305/330) of repairs are correct, demonstrating that Semter can repair most test breakages correctly.

Answer to R3Q2: Semter can repair most test breakages correctly.

D. R2Q5: Can Semter achieve better results if it uses other strategies to calculate similarities?

Motivation and Approach. To measure the similarity strategy used by Semter, we design another similarity formula as follows:

$$sim(e,e') = \alpha \times sim_i(e,e') + \frac{(1-\alpha)}{\sqrt{|e.cs| \times |e'.cs|}} \times \sum_{c_k \in e.cs} sim_c(c_k, c'_j)$$

where $sim_i(e,e')$ and $sim_c(e,e')$ represent the individual semantic similarity and the context semantic similarity between e and e', respectively, and α is a weighting parameter. The

TABLE I
MUTATIONS STUIDED IN OUR EXPERIMENTS

	Mutations	#Breakages	#Repaired
1	Remove the element	5	2
2	Remove a neighbouring element	3	3
3	Set the element invisible	3	3
4	Add an identical element	7	7
5	Add a semantically similar element	7	7
6	Add an semantically dissimilar element with similar attributes	8	8
7	Modify a neighbouring element	5	5
8	Reduce the size of windows to hide the element	15	15
9	Move the element to a proper location	10	10
10	Move the element to overlap with a neighbouring element	11	11
11	Modify the relative positions between the element and a neighbouring element	36	33
12	Swap the attributes of the element with a neighbouring element	8	8
13	Modify the type of the element	15	4
14	Modify the position of the form to which the element belongs	3	3
15	Scale the whole page or form to which the element belongs	8	8
16	Replace the element with a semantically similar one	33	28
17	Modify the size of the element	28	28
18	Modify the style of the element	30	29
19	Modify the background color	26	25
20	Modify the font style	69	68
	Total	**330**	**305**

computation methods of $sim_i(e, e')$ and $sim_c(e, e')$ are the same as Semter. We replace the formula used by Semter with the new formula and conduct an experiment on AddressBook and Claroline. We select 4 different values for α (i.e., 0.6, 0.7, 0.8, and 0.9) and 4 different threshold values for t (i.e., 0.4, 0.5, 0.6, and 0.7). Due to the limited response time, we have only completed the experiments on the first two web applications and reported the result here.

Results. When both α and t in the new formula are selected as 0.6, we achieve the best effectiveness of repair. The repair ratio for AddressBook is 83.9% whereas 82.4% for Claroline. The total repair ratio is 83.0%. Figure 4 compares the repair results for Semter and the new similarity formula. The total repair ratio of Semter is 88.8%, which clearly surpasses the new similarity formula.

> Answer to R3Q2: Semter achieves a better repair effectiveness than the new similarity formula.

E. Answer to R3Q2: Most of the breakages in the experiment subjects are NSSP. Is this also true for other real-world applications?

Motivation and Approach. This RQ aims to evaluate whether most of the test breakages occurring in other applications are also NSSP. We select two real-world web applications, i.e., Mantis Bug Tracker (MantisBT) and Meeting Room Booking System (MRBS), to conduct an empirical study. The two applications and test cases are available at https://sepl.dibris.unige.it/2014-Visual-DOM.php. For each application, we select 6 different releases. For each release, we resuse 35 test cases[4] for MantisBT and 24 for MRBS respectively. We manually count the number of test breakages between two successive releases and categorize them into NSSP, MSDP, NSNP, and NSR.

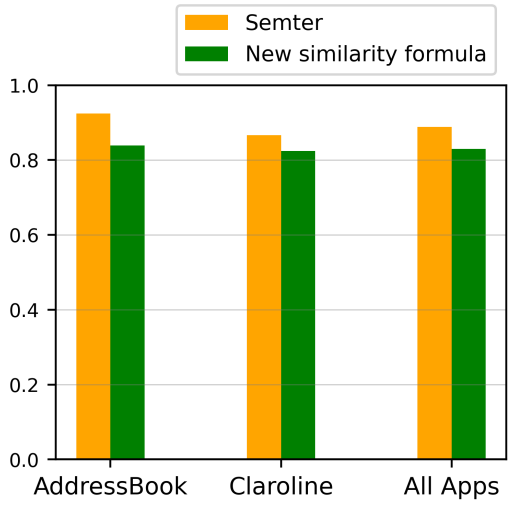


Fig. 4. Comparison of repair ratios between Semter and the new similarity formula (with the optimal parameters).

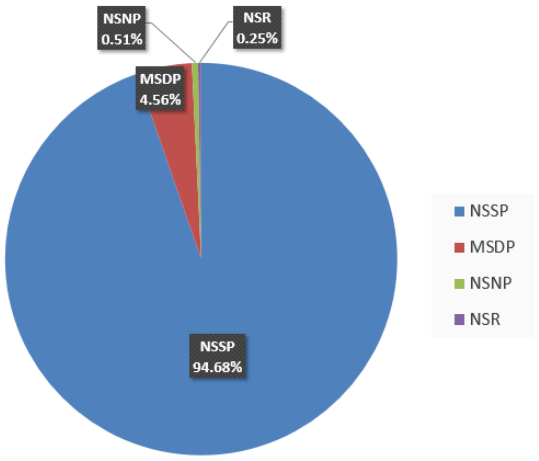


Fig. 5. Percentage of each type of breakages on MRBS.

Results. In MantisBT, 69 breakages occur, and all of them are NSSP. Concerning MRBS, 395 breakages are found, and 94.68% (374/395) of breakages are NSSP (see details in

[4]We reuse all aviable test cases on MantisBT and MRBS.

Figure 5). Moreover, there are 18 MSDP, 2 NSNP, and 1 NSR. Overall, 95.47%, i.e., (69+374)/(69+395), of test breakages in these two applications are still NSSP.

Answer to R3Q2: Most of test breakages occurring in other real-world applications are NSSP.

REFERENCES

[1] R. A. P. Oliveira, E. Alégroth, Z. Gao, and A. M. Memon, "Definition and evaluation of mutation operators for gui-level mutation analysis," in *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–10. [Online]. Available: https://doi.org/10.1109/ICSTW.2015.7107457