Exploring an Inclusion Relation on Test Cases to Identify Unit and Integration Tests

Ryu Okamoto, Shinsuke Matsumoto, and Shinji Kusumoto

Graduate School of Information Science and Technology, Osaka University, Japan
{r-okamot, shinsuke, kusumoto}@ist.osaka-u.ac.jp

Abstract. In software testing, among the various types of tests, two commonly conducted ones are unit and integration tests. Unit tests verify individual functionalities, and integration tests verify the combination of multiple functionalities. If we can identify unit/integration tests and measure them as ordinal values, such as the degree of integration-*ness*, we can utilize them to improve testing efficiency. However, the definitions of unit/integration are ambiguous, making it difficult to distinguish between them. To the best of our knowledge, there is currently no method for detecting this distinction. In this study, aiming to support the testing process, we will consider a measurement method for unit/integration tests. The key idea is to utilize an inclusion relation, which naturally exists among test cases. As an application of the inclusion relation, we propose a method for ordering failed tests to streamline debugging. We conducted a mutation analysis to evaluate how much our proposal reduces debugging effort compared to a naive method. The results showed that our proposal was effective in 29.7% of cases and confirmed an average reduction of 20.7% in debugging effort.

Keywords: unit test, integration test, inclusion relation, line coverage

1 Introduction

Software testing is one of the fundamental and essential activities in software development [8]. This paper focuses on program-based automated software testing, which uses some test cases describing validation processes, inputs, and expected outputs. Henceforth, we will simply refer to test cases as tests.

In testing, individual functionalities are usually verified by unit tests, and combinations of multiple functionalities are verified by integration tests [3][6]. Unit/integration tests have different roles and aims. Thus, we have to use them properly, depending on the software functionality that will be tested.

If we can identify unit/integration tests and further determine the degree of integration-*ness*, this information is helpful for debugging. One potential application is suggesting the order to resolve when multiple failed tests exist. In such a scenario, priority should be given to resolving unit-leaning tests. This is because unit-leaning tests focus more narrowly on individual functionalities,

making it easier to identify defective areas. In contrast, integration-leaning tests span multiple functionalities, making pinpointing defects more difficult.

However, in practice, it is difficult to distinguish whether a test is a unit or integration test [10]. This is due to the ambiguous definitions of unit/integration tests. We consider the ambiguity arises from the unclear definition of "individual functionality". There are various interpretations of what constitutes an individual functionality within the software. For example, a single use case can be considered an individual functionality that directly provides value to end users [4]. On the other hand, a use case is often implemented by multiple internal functionalities, such as methods. Although a method can be seen as an atomic unit of functionality, it is often composed of multiple other methods.

This study examines methods for identifying of unit/integration tests to support developers in debugging. The key idea is introducing an inclusion relation, which naturally exists in tests. From the typical unit/integration definitions, unit tests can be interpreted as being conducted as part of integration tests. Therefore, when the lines of code executed by one test include those of another, we consider there to be an inclusion relation between them. Furthermore, we measure the degree of inclusion to determine the integration-*ness* of tests.

As an application of the inclusion relation, we propose a method for ordering failed tests to resolve efficiently. We conducted mutation analysis with real projects to evaluate how much debugging effort our proposal can reduce compared to a naive method. The results showed that our proposal was effective in 29.7% of cases. Furthermore, when our proposal was effective, we confirmed that the average reduction rate of debugging effort was 20.7%.

2 Related Works

The definitions of unit/integration tests vary and are often qualitative. Traditionally, the portion of product code called "unit" is defined. If a test verifies only one "unit", it is considered a unit test; otherwise, an integration test [3][6]. The definition of "unit" can be interpreted variously, such as a method, a class, or a package. Titus et al. attempt to classify tests, focusing on two attributes of tests: size and scope [11]. The size represents the resources required during testing, such as memory and time. The scope is the portion of the product code intended to be verified by testing. They call tests unit/integration, focusing on the scope. However, they do not specify any numerical indicator but only classify them on a small/medium/large qualitative size.

Trautsch and Grabowski investigated whether unit tests are conducted in Python OSS according to the classical definitions [10]. They reported that many of the unit tests intended by the developers did not match the traditional definitions and that the number of matches varied across definitions. This suggests that the definitions of unit/integration tests are conceptual, and it is not easy for actual developers to distinguish between them.

Kanstén defines a numerical indicator called test level, which expresses the granularity of a test [5]. Tests are arranged on a number line according to the

number of called functions and grouped at regular intervals from 0. Then, the test level is determined based on which group it belongs to. This method has the advantage of being quantitative and capable of automatic measurement. However, since each test is projected onto a one-dimensional number line first, information about which tests are included by which other tests are abstracted.

There are studies similar to ours that consider and utilize an inclusion relation. The difference lies in how they define it. We define an inclusion relation based on the lines executed by tests. On the other hand, Gälli et al. focuses on the functions called by tests [2]. Through experiments, they demonstrated that failures were more likely to propagate between tests with the inclusion relation and concluded that the included tests verify more specific functionalities. Marré and Bertolino proposed test reduction principles using an inclusion relation in program elements to achieve efficient coverage improvement [7]. They define sets of program entities using a flow graph. A set of program entities includes another if the execution of all entities in the set results in the execution of all entities in the other set.

3 Identifying Unit and Integration Tests

To identify unit/integration tests, we define an inclusion relation among tests based on the executed lines and quantify the degree of inclusion. We consider a test that does not include any other as a unit test.

Let T_A and T_B are tests. We define $T_A \subset T_B$ as $\mathrm{Stmt}(T_A) \subsetneq \mathrm{Stmt}(T_B)$, where $\mathrm{Stmt}(T)$ is a set of lines executed by test T. As an example, the product code of a user master and its test code are shown in Figure 1. Before registering a user, the given `id` and `name` formats are checked. There are three tests: two to verify the format checks and the other to verify the registration process. The executed lines of each test are as follows:

$$\begin{aligned}\mathrm{Stmt}(\texttt{test\_validate\_id}) &= \{\ell 3\}\,,\\ \mathrm{Stmt}(\texttt{test\_validate\_name}) &= \{\ell 5\}\,,\\ \mathrm{Stmt}(\texttt{test\_register\_user}) &= \{\ell 3, \ell 5, \ell 8, \ell 9, \ldots\}\,,\end{aligned}$$

where ℓx represents line x in Figure 1. From the above, there are the following inclusions among these tests:

$$\begin{aligned}\texttt{test\_validate\_id} &\subset \texttt{test\_register\_user},\\ \texttt{test\_validate\_name} &\subset \texttt{test\_register\_user}.\end{aligned}$$

We call a directed graph $G = \langle V, E\rangle$ an inclusion graph, where V represents a set of all the tests and E the inclusion relation on V. That is to say if $T_A, T_B \in V$ and $T_A \subset T_B$, then $\langle T_B, T_A\rangle \in E$. In accordance with the definition of inclusion relation, inclusion graphs are DAG (Directed Acyclic Graph). We can assign the height for each node in DAG. Thus, we define the inclusion level of a test as its height in the inclusion graph. In the example shown in Figure 1, the level of each test can be calculated as 0, 0, and 1 respectively from the top one.

```
## Product
def validate_id(id: str) -> bool:
    return bool(re.match(id, r'[a-z][0-9]{3}'))
def validate_name(name: str) -> bool:
    return bool(re.match(name, r'[a-z][a-z0-9_]*'))
class UserMaster:
    def register_user(self, id: str, name: str):
        if not validate_id(id) or not validate_name(name):
            self.user_map[id] = name
    ...

## Test
def test_validate_id():
    assert_that(validate_id('x099')).is_true()
def test_validate_name():
    assert_that(validate_name('alice')).is_true()
def test_register_user():
    master = UserMaster()
    master.register_user('a001', 'bob')
    assert_that(master.lookup('a001')).is_equal_to('bob')
```

Fig. 1. An example of product and test code

Following the established definitions of unit/integration, unit tests can be interpreted as being executed as part of integration tests. Therefore, a test is considered a unit test when its execution lines are not a complete subset of another test's; otherwise, the test will be an integration test. In other words, a test with level 0 is considered equivalent to a unit test. Furthermore, the higher the inclusion level of a test is, the more integration-leaning the test is. Using the inclusion relation on tests, we can detect unit tests relatively rather than absolutely, thus avoiding ambiguity in the definition of "unit".

4 An Application of the Inclusion Relation

As an application of the inclusion relation, we propose a method for ordering failed tests. When several tests fail, the method suggests in what order programmers should resolve the failures for less debugging effort. Its output is an order of the failed to resolve.

The procedure of our proposal is as follows. First, we run failed tests as input and collect sets of executed lines. Next, the inclusion graph is constructed, and the inclusion level for each test is calculated. Finally, the failed tests are sorted with the level as the first key and the number of covered lines as the second key.

If several tests fail, programmers should prioritize fixing more independent ones. This is because highly coupled tests broadly verify multiple functionalities, making it difficult to localize faults. Hence, our proposal sorts tests in ascending

order, with the inclusion level as the first key. When multiple tests with the same level exist, the number of executed lines is used as the key to sort in ascending order. It is based on the simple idea that tests with the smallest number of execution lines should be resolved first.

The executed lines of a failed test are often incomplete due to handling exceptions or being interrupted. However, in this context, we do not need to care about it. On the contrary, included tests in such situations focus more on certain statements, i.e., functionalities, and can help localize faults.

5 Evaluation

To evaluate the effectiveness of our proposal as a debugging support, we conducted an experiment on OSS projects. The goal is to confirm that a suggested debugging order of failed tests can reduce identifying faults effort.

As targets for the experiment, we selected four projects: commons-lang[1], commons-io[2], jsoup[3], and gson[4]. Each project achieves approximately 90% line coverage, and sufficient tests are prepared. We selected them for two reasons: their stars on GitHub are more than 1,000 and they use JUnit[5] as their test framework. We treat a method annotated with `@Test`, `@ParameterizedTest`, or `@RepeatedTest` as a test. We used JaCoCo[6] to collect executed lines.

Regarding the premise of the evaluation experiment, we conducted a preliminary experiment to verify the existence of the inclusion relation in real projects. Table 1 shows the number of tests at each level. According to the table, the inclusion relation existed. Furthermore, multi-level inclusions also existed since there were tests with level 2 or higher. Jsoup had the highest percentage, 90.7%, of tests with level 0. This means unit testing in jsoup was the most thorough. On the other hand, commons-lang had the lowest percentage, 53.7%, of level 0 and a relatively more significant number of integration tests.

[1] Accessed at 2024/04/12. https://github.com/apache/commons-lang.
[2] Accessed at 2024/04/14. https://github.com/apache/commons-io.
[3] Accessed at 2024/04/19. https://github.com/jhy/jsoup.
[4] Accessed at 2024/04/19. https://github.com/google/gson.
[5] Accessed at 2024/05/17. https://junit.org/junit5/.
[6] Accessed at 2024/05/17. https://www.jacoco.org/jacoco/.

Table 1. The number of tests at each level

	Inclusion level					
	0	1	2	3	4	5
commons-lang	2,122	1,129	441	168	79	10
commons-io	1,397	540	157	36	10	2
jsoup	1,079	93	16	2	0	0
gson	1,015	210	86	23	5	0

Table 2. The results of the mutant analysis. N is the number of generated mutants. $|F|_{Q_i}$ is the $25i$ percentile of $|F|$. $|F|$ is the number of failures when mutating. $P_{<1}$ and $P_{>1}$ are the percentage of cases where the proposal was effective and not. $\overline{E}$ is the average reduction rate of debugging effort in cases where the proposal was effective.

	N	$\|F\|_{Q_1}$	$\|F\|_{Q_2}$	$\|F\|_{Q_3}$	$P_{<1}$	$P_{>1}$	$\overline{E}$
org.apache.commons.lang.text	184	6	11	24	17.4	0.0	40.6
org.apache.commons.lang.function	350	2	3	4	7.7	0.0	15.5
org.apache.commons.io.input	237	5	8	29	20.7	9.7	26.6
org.apache.commons.io.file	236	15	31	57	46.2	9.3	15.7
org.jsoup.nodes	298	13	19	38	26.8	0.0	18.6
org.jsoup.parser	371	21	39	310	32.3	0.3	15.1
com.google.gson.internal.bind	294	21	50	71	52.4	0.3	23.2
com.google.gson.stream	46	59	156	266	58.7	0.0	28.3
All packages	2,016	5	19	49	29.7	2.3	20.7

Method. First, we generate mutants with two or three mutations, using mutation operators introduced by Sasaki et al. [9]. At this time, we make sure that two or more tests fail. Next, we determine an order to resolve by applying our proposal sorting and ascending sorting by the number of executed lines. We will refer to the results of the former as suggested orders and the latter as naive orders. Finally, we calculate an evaluation metric, which is defined later. We evaluate the proposal by repeating the above steps multiple times.

We define the debugging effort value for a given resolution order of failures $F = \langle F_1, F_2, \ldots, F_n \rangle$. Suppose it is not until programmers check execution parts from F_1 to F_k that all mutation lines are detected. We calculate the debugging effort value $\text{effort}(F) = \sum_{i=1}^{k} 3.20|\text{Stmt}(F_i)|^{1.05}$. Let R be the ratio of the debugging effort value of a suggested order to that of a naive. Moreover, if $R < 1$, i.e., if the suggested order is superior to the naive, then calculate the debugging effort reduction rate $E = 100(1 - R)$. The formula $3.20|\text{Stmt}(F_i)|^{1.05}$ is based on the definition of the nominal development effort in software estimation [1]. Its unit is man-months. The reason for weighting the executed size is that the relation between the effort required to identify faults and the number of lines of product code to check is not linear. The effort required is expected to increase due to the increase not only in the code volume but also in its structural complexity.

Result. Table 2 shows the results of the experiment. For each target package, the results are shown when two or three mutations are introduced. Two target packages were randomly selected from each project, with 85% or more line coverage and 5 or more classes. N column is the number of generated mutants. $|F|_{Q_i}$ column shows the $25i$ percentile of $|F|$. $|F|$ is the number of failures when mutating. $P_{<1}$ column indicates the percentage of generated mutants whose evaluation value R is less than 1, and similarly for $P_{>1}$. $\overline{E}$ column represents the average debugging effort reduction rate.

Based on the total of all packages, for 29.7% of all generated mutants, the suggested order had an average debugging effort value of 20.7% smaller than

the naive. In particular, good evaluation values were obtained in the majority of cases, 52.4% for gson.internal.bind and 58.7% for gson.stream. Additionally, cases where the evaluation value is poor, i.e., $R > 1$, account for 2.3% across all packages, with less than 0.5% in many packages, indicating their rarity. Therefore, we can confirm that our proposal effectively reduces the debugging effort compared to the naive method.

However, in 2.3% of the cases, the debugging effort value was more significant for our proposal than for the naive method. This was especially noticeable in the case of commons-io, and there were some cases, in about 9%, where the evaluation values were inferior. One possible cause is that the inclusion level of tests is calculated too high due to excessive inclusion detection. Resolving a test executing a mutation line is put off if the test includes other failed tests. The leading cause of excessive inclusion is error handling using the ternary operator. At this time, the difference in the execution paths of normal and abnormal tests disappears, making it easier for one to include the other. When the evaluation value was poor, we observed the lines executed by failed tests. We found many error handlings using the ternary operator.

6 Discussion

In this section, we discuss the advantages and limitations of the inclusion relation based on four respects: universality, versatility, sensitivity, and prerequisite.

Universality. The defined inclusion relation can be detected independently in a specific language, testing framework, or scene. We interpret a test as a set of executed lines and use the natural concept of set inclusion to define the test inclusion. The executed lines can be easily collected with coverage measurement tools. Coverage is a widely used metric in practice, and many measurement tools have been implemented.

Versatility. The idea of the inclusion relation is so natural that it has scope for use in various testing supports. For example, it can be used to extend coverage measurement. In conventional measurement, only whether each product code line has been executed by any test is measured. It is not measured whether the tests lean towards unit or integration tests. Therefore, by using the inclusion relation, integration-leaning tests are identified, and the portions executed only by those tests are reported. The existence of such parts may suggest inadequacy in unit testing. Integration testing verifies the combination of several functionalities and is not responsible for verifying the behavior of each functionality. Functionalities only verified by integration tests ought to be validated by unit tests for those functionalities as possible.

Sensitivity. Intuitively, the magnitude of the inclusion level represents the integration-*ness*, but several counterexamples exist to this notion. For instance, consider tests for insertion sort where sorted data and randomly ordered data are provided as inputs. According to the conventional understanding of unit/integration testing, both tests verify a single functionality, sorting, and thus would be equivalent to unit tests. However, in many implementations of insertion sort,

the execution portion for randomly ordered data encompasses that for sorted. As a result, there is a possibility of finer unit/integration tests distinctions than expected or even the occurrence of the opposite.

Prerequisite. The inclusion relation is a relative concept. Thus, it presupposes that tests are sufficiently prepared. If tests are insufficient, some traditionally identified as integration tests may be assigned inclusion level 0.

7 Conclusion

In this study, we introduced the coverage-based inclusion relation on test cases to identify unit/integration tests. We also proposed the specific application and confirmed that it effectively reduces debugging effort. However, we have not yet validated whether our defined inclusion relation matches the generally recognized unit/integration relations. Thus, in future work, we have to verify the validity of the identification results through human inspection.

Acknowledgments. This research was partially supported by JSPS KAKENHI Japan (Grant Number: JP24H00692, JP21H04877, and JP21K18302).

References

1. Boehm, B.: Software Engineering Economics. Trans. Software Engineering **SE-10**(1), 4–21 (1984)
2. Gälli, M., Lanza, M., Nierstrasz, O., Wuyts, R.: Ordering Broken Unit Tests for Focused Debugging. In: Proc. International Conference on Software Maintenance (ICSM). pp. 114–123. IEEE (2004)
3. IEEE: ISO/IEC/IEEE International Standard - Systems and Software Engineering – Vocabulary. ISO/IEC/IEEE 24765:2010(E) pp. 1–418 (2010)
4. Jacobson, I., Spence, I., Kerr, B.: Use-case 2.0. Trans. Communications of the ACM **59**(5), 61–69 (2016)
5. Kanstrén, T.: Towards a Deeper Understanding of Test Coverage. Trans. Software Maintenance and Evolution: Research and Practice **20**(1), 59–76 (2008)
6. Khorikov, V.: Unit Testing Principles, Practices, and Patterns: Effective Testing Styles, Patterns, and Reliable Automation for Unit Testing, Mocking, and Integration Testing with Examples in C#. Manning Publications (2021)
7. Marré, M., Bertolino, A.: Using Spanning Sets for Coverage Testing. Trans. Software Engineering **29**(11), 974–984 (2003)
8. Myers, G., Badgett, T., Sandler, C.: The Art of Software Testing. John Wiley & Sons, Ltd (2012)
9. Sasaki, Y., Higo, Y., Matsumoto, S., Kusumoto, S.: SBFL-Suitability: a Software Characteristic for Fault Localization. In: Proc. International Conference on Software Maintenance and Evolution (ICSME). pp. 702–706 (2020)
10. Trautsch, F., Grabowski, J.: Are There Any Unit Tests? An Empirical Study on Unit Testing in Open Source Python Projects. In: Proc. International Conference on Software Testing, Verification and Validation (ICST). pp. 207–218. IEEE (2017)
11. Winter, T., Manshreck, T., Wright, H.: Software Engineering at Google: Lessons Learned from Programming over Time. Oreilly & Associates Inc (2020)