Test transplantation through dynamic test slicing

Test Transplantation through Dynamic Test Slicing

Mehrdad Abdi
Dept. Computer Science
University of Antwerp
Antwerp, Belgium

Serge Demeyer
Dept. Computer Science
University of Antwerp
Antwerp, Belgium

***Abstract*—Previous research has demonstrated that the test coverage of libraries can be expanded by using existing test inputs from their dependent projects. In this paper, we propose an algorithm for test transplantation based on *test slicing*. The algorithm extracts test inputs, isolates them by creating mocks, and then transplants the test code onto the test suite of the libraries. To achieve test slicing, we dynamically execute the tests in the dependent project and create its graph of histories. Then, we traverse back from the interesting object state and collect the corresponding edges. Finally, we reverse the collected edges and create a sequence of method calls to reconstruct the same object state. We have implemented a proof-of-concept in Pharo-Smalltalk, in this paper we discuss the lessons learned so far.**

***Index Terms*—test amplification, program slicing, Pharo, Software ecosystems**

I. Introduction

Modern software repositories contain a test suite covering some of its code. In a software ecosystem, projects usually import modules from other libraries and invoke their public interfaces to fulfill their tasks. A recent study by Schittekat et al. [1] illustrated that the tests in the user projects (source) indirectly cover some new parts in libraries (destination). This shows the opportunity of exploiting these test suites to amplify libraries' test coverage.

One easy solution to generate new unit tests is taking snapshots of the interesting object states during test execution and restoring them in test methods. However, these tests may not comply with the unit testing pattern in object-oriented programming languages and be less readable. In object-oriented programming languages, a *unit test* typically is initializing an instance of the class-under-test, updating it to the desired state, and finally, asserting the expected states. In addition to readability, the snapshots may depend on some classes from the source project that do not exist in the target project.

This paper introduces a method to synthesize valid sequences of method calls to reconstruct the state of the object-under-test and other necessary objects based on tests in the source project. The method-call sequence will be installed as a unit test in the destination project. We call this process *dependency-base test transplantation*. In this paper, we adopt the terminology from the original paper positioning the idea of code transplantation [2]. Hence, we refer to the dependent project (source) as *donor project*, the library (destination) as *host project*, and the object state to be transplanted as *organ*. We also refer to the test method containing the organ in the donor project as *donor test*.

We propose an algorithm to slice the donor test dynamically. First, we execute it and collect execution traces, including method calls. We form a *graph of histories* using these traces. Then, we spot the interesting object state (organ) in the graph and extract a subgraph. In extracting this subgraph, we traverse the graph backward, starting from the organ, and collect the list of edges. Finally, we reverse the collected edges and synthesize a sliced test method. In this process, we isolate the slices by mocking those classes not belonging to the host project. Once we obtain a precise slice containing the organ, we can transplant the test into the host project.

To conclude, we reduce the test transplantation problem into a test slicing problem and consequently a test slicing problem into a graph traversal problem. We also write about our learned lessons from implementing (work in progress) this approach in a proof-of-concept tool called SMALL-MINCE in Section IV.

II. Background

a) Test amplification: Software repositories contain a considerable amount of test code written by developers to prevent regression in software evolution. Exploiting this source of knowledge to improve software testing is called *test amplification* [3]. SMALL-AMP [4] is a test amplifier in Pharo that synthesizes new unit tests that increase the mutation coverage. SMALL-AMP is based on four main components: (1) a *profiler* which captures variables' type information, (2) an *input amplifier* which transforms existing test methods and generates new test inputs, (3) an *assertion amplifier* which regenerates the assertion statements, and (4) a *selector* that runs mutation testing and identifies the tests introducing new coverage.

b) Program slicing: Program slicing is finding a smaller set of statements from a program based on a slicing criteria [5]. A slicing criterion is defined as a target statement and a variable. So, the goal is to find a program slice in which the value of the target variable in the specified statement is identical to the same variable value in the same statement in the original program. The slicing algorithm uses the statements dependency graph and computes a slice by a backward graph traversing.

Static slicing of the program considers all possible inputs in a program and may produce slices with unnecessary statements. Dynamic program slicing [6] uses a specific input and

```
state := primitive | history | self | special
primitive := int | str | null | ...
history := uid, event*
event := <message,
          state:= version,
          state′:= version,
          args:= state*,
          args′:= state*,
          returns:= state>
version := A set of var_name ↦ state pairs
special := uid, predefined representations (lists, streams,...)
```

Fig. 1: A model for object representation

performs the slicing based on the executed statements. As a result, it produces more precise slices, which help debug the program based on specified inputs.

c) Pharo: Pharo[1] is a pure object-oriented, dynamically typed language. In Pharo, all actions are done by sending *messages* to objects which is the equivalent to method invocation in other languages. The instance variables are private and can only be updated by the methods. The Pharo environment offers several facilities for dynamically inspecting the internal state of execution, making it well suited for dynamic program analysis.

III. Dynamic Test Slicing

The traditional program slicing technique models the program as a set of statements and their relations. In this paper, we model a program as a set of object states and their relations and create a *graph of histories*. The result of the slicing is a sequence of method calls that produces the same state of objects in the defined location. In this section, we introduce the object representation and the graph representation methods inspired by related work on the subject [7], [8], [9].

A. Model

a) Language Model: We use an object-oriented language model, similar to what Pharo provides: Everything in the language are objects, all objects have a default constructor (`new`), and the instance variables in it are private and can be updated only by its methods. All objects are passed by their reference as arguments.

b) Object representation: Values in this language are either *primitive* values (integers, strings, ...) or objects. Each object is initiated by its constructor (`new`), and its state is updated by sending different messages. We refer to sending messages as *events* that create a new *version* of the receiver object. The set of all object's versions is its *history*. We adopt two representations for object values: (1) as a *history* which is a unique identifier uid and a list of *events* (2) as *versions* which are concrete state of objects: it includes a mapping of instance variables to their values; if the value is not a primitive, we represent it as an object history (Figure 1).

An event shows that the *message* with the *args* has been sent to the receiver object (identified by uid in history) when it had the internal state of *state*. This call has led it to the internal state *state'*, produced the return value of *returns*, and the states of arguments are changed to *args'* after the method call.

B. The Graph of histories

Each event on an object creates a new version. We create an acyclic graph using the versions of all objects as nodes, and their relations as edges ($G =< V, E_{events}, E_{args}, E_{rets}, E_{argrets} >$). For simplicity, we skip showing the primitive values in the graph. The nodes are connected with four types of edges in this graph:

1) **Event edges:** The messages sent to the object which have updated its version. We use solid arrows to represent these edges and annotate them with the message name in Figure 2.
2) **Argument edges:** Shows that an object version is used as an argument in an event. Figure 2 uses hollow arrows to represent these edges.
3) **Return edges:** Shows that an object version is returned from a method call. We use dashed arrows to represent these edges in Figure 2. We also use the node after the method call as the source of these edges. For example, the event `md1` on the object d version 1 leads it to its version 2 and meanwhile returns the object o version 2. We draw a dashed edge from d version 2 to o version 2.
4) **Argument return edges:** Shows the state of arguments after a method call. We use dotted arrows to represent these edges. We also omit these edges when the state of the argument is not updated within an event. Similar to return edges, we use the node after the method call as the source of these edges.

Listing 1: Example code for dynamic test slicing

```
1  DriverTest >> test1
2    d := Driver new.
3    o := d md1: #ClassO.
4    o mo2.
5    o mo3.
6    d md2.
7    d md3: o
8
9  Driver >> md1: aSymbol
10   "..."
11   aSymbol = #ClassO ifTrue: [
12     retVal := ClassO new.
13     a := ClassA new.
14     x := ClassX new.
15     retVal mo1A: a X: x
16     ^ retVal ]
17   "..."
18
19 ClassO >> mo1A: aObj X: xObj
20   xObj mx1.
21   y := ClassY new.
22   xObj mx2Y: y.
23
24 ClassO >> mo2
25   r := ClassR new.
26   r mr1.
27   ^ r
28
29 ClassX >> mx2Y: yObj
30   z := ClassZ.
31   z mz1.
32   yObj my1Z: z.
33   ^ 1
```

Figure 2 shows the graph related to the code in Listing 1. As an example, we focus on the history of the object o: We see it at line 3, but looking deeper, this object is initialized at line 12 as the variable name $retVal$. Both variables o (line 3) and $retVal$ (line 12) refer to the same object in execution time. The message `#mo1A:X:` is sent to it using the arguments a version 1 and x version 1 (lines 15). This message brings this object to its second version. From the outcoming dotted edge

[1] https://pharo.org/

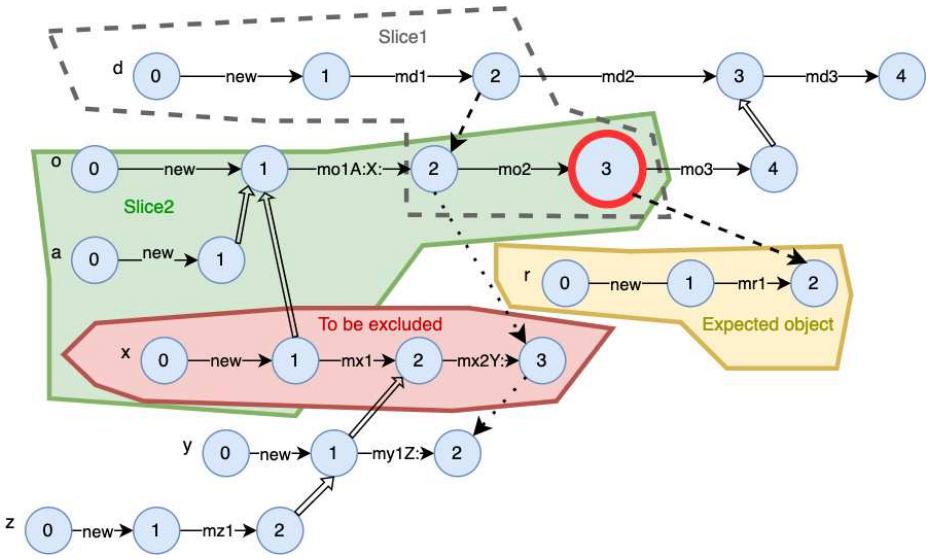

Fig. 2: An example of versions graph

(argument return), we understand that the state of the object x is updated from version 1 to 3 inside `#mo1A:X:` (lines 19 to 22). The incoming dashed edge shows that version 2 of the object o is returned when the message `#md1:` is sent to the object d (line 3). The event edge of `#mo2` is the transition to version 3 for object o (line 4). The outgoing dashed edge shows that an object r version 2 is returned from this event (line 27). The object o is updated to version 4 by accepting `#mo3` (line 5). We also understand that the version 4 of o is used in calling `#md3` on the object d.

C. Slicing tests

A *unit test* in an object-oriented language typically consists of initializing an instance of class-under-test, updating it to the desired state, and finally, asserting the expected states. In other words, a unit test is the history of the object-under-test. For example, the method `#test1` in Listing 1 is the history of the object d (the assertion statements are removed for simplicity).

If we recognize that a version of an object is interesting from the testing point of view, we can synthesize a unit test to regenerate the same state based on the graph of histories. We start from the target node, traverse the graph backward, and collect all events in order to reconstruct the same order.

In this method, we use a mapping V to store the list of objects to be traversed and their latest visited version. We also use the list of S including visited edges. Because of the space considerations, we only use an example to describe the algorithm. Our example graph considers version 3 of the object o as the slicing criteria. We start traversing from this node ($V = \{o \mapsto 3\}, S = []$):

There is only one node to be traversed in V. We pick it and see that it has only one incoming event edge and does not have any other incoming or outgoing edges to other visited objects, so it is safe to visit (There is an outgoing dashed edge to r version 2, but r is not in the list of nodes to be traversed: $r \notin V$). We update the version of o in V and also add the visited edge to S: $V = \{o \mapsto 2\}, S =$ [call o.mo2].

Again, we have one node in V. We see that it has two incoming edges: an event edge and a return edge. A version of the object can be obtained either by sending a message to its previous version or being returned from another event as a return value or argument. So we can take either of them. We explore both scenarios, which will create two different slices.

- We take the return edge. It says that the current state is the return value from calling `#md1` on the object d version 1. We discard o from V and add d: $V = \{d \mapsto 1\}, S =$ [call o.mo2, return of d.md1]. Traversing the remaining of this path is straightforward ($V = \{d \mapsto 0\}, S =$ [call o.mo2, return of d.md1, call d.new]). When all objects in V are reached their version 0, the algorithm finishes.
- We consider the event edge (calling `#mo1A:X:`). This edge will require two other objects a and x with version 1 as arguments. We check V to check if these objects are in the list of nodes to be traversed. If we found them in V, we check that their version is 1. If they exist in V and have a version higher than 1, we postpone traversing the current edge (`#mo1A:X:`) and continue traversing argument objects to bring their version to 1. Finally, if they do not exist in V, we add them to the list. In our case, we take the event edge and add a version 1 and x version 1 to V: $V = \{o \mapsto 1, a \mapsto 1, x \mapsto 1\}, S =$ [call o.mo2, call o.mo1A:X:].
  In the next step, we select one of the nodes from V. Let us take a and traverse its edge: $V = \{o \mapsto 1, a \mapsto 0, x \mapsto 1\}, S =$ [call o.mo2, call o.mo1A:X:, call a.new]. Traversing nodes x and o is also straightforward: $V = \{o \mapsto 0, a \mapsto 0, x \mapsto 0\}, S =$ [call o.mo2, call o.mo1A:X:, call a.new, call o.new, call x.new]).

We synthesize a test by reversing the traversed edges and synthesizing each event. Listing 2 shows the two sliced tests from this example.

Listing 2: Example of sliced tests

```
testSlice1
  d := Driver new.
  o := d md1: #ClassO.
  o mo2.


testSlice2
  x := ClassX new.
  o := ClassO new.
  a := ClassA new.
  o mo1A: a X: x.
  o mo2.
```

a) Method call sequence minimizing: In real traces, the number of events on the objects is considerable, and synthesizing all events may result in a lengthy unreadable test. We can analyze the $state$ changes in objects to reduce some extra events.

As an example, consider state preserving methods (getter) are called inside a loop. In the generated test slice, we will see plenty of unnecessary calls to these getter methods. We can minimize the slice length by detecting and removing these invocations. They can be detected by evaluating the difference between the state after the event and the state before each event: $state' - state == \varnothing$.

b) Assert generation: Generating the assertion statements to assert the primitive types is straightforward because the actual value of the primitive can be found in the event traces. However, there are opportunities for generating advanced assertions, such as asserting the *expected objects*. In the example graph in Figure 2, the object r is returned from sending `mo2` to o. Listing 3 shows a sliced test that reconstructs the expected object $r\_expected$ and asserts it is equal to the returned value.

Listing 3: Asserting expected objects

```
testSlice2_withAsserts
  o := ClassO new.
  x := ClassX new.
  a := ClassA new.
  o mo1A: a X: x.
  r_expected := ClassR new.
  r_expected mr1.
  r_actual := o mo2.
  self assert: r_actual equalsTo: r_expected
```

D. Test isolation

In test isolation, our goal is to exclude some classes from the sliced tests. We replace the excluded classes with mock objects.

In our example in Listing 1, we deduce that the class $ClassX$ needs to be excluded. This exclusion will invalidate the sliced test `testSlice2` because it depends on $ClassX$. We can see from the graph that the object x, which is an instance of this class, is passed as an argument to the method `mo1A:X:`. We also see that two other messages `mx1` and `mx2` are sent to x when it was being processed in the method `mo1A:X:` (the version of x is updated from 1 to 3 when it is returned). We create the mock object $xMocked$ that simulates the required behaviors. Listing 4 illustrates a test method with the mocked $ClassX$ based on `Mocketry` mocking library[2].

Listing 4: Isolated sliced test

```
testSlice2_mocked
  o := ClassO new.
  xMocked := Mock new.
  xMocked stub mx1 willReturn: nil.
  (xMocked stub messageWith: (Instance of: ClassY)) willReturn: 1.
  a := ClassA new.
  o mo1A: a X: xMocked.
  o mo2
```

IV. Proof-of-concept

We implemented our algorithm in a proof-of-concept tool called SMALL-MINCE[3] in the Pharo language (work still in progress). The tool consists of three main components: (1) tracer, (2) slicer, and (3) synthesizer.

The tracer component manipulates the classes in the project to enable them to log the details of message invocations. We employed *method proxies* to capture the receiver and arguments state before and after an invocation. We use an integer instance variable as the object's version (increases by each event), and also a stack to reject the internal method invocations. After the manipulation, the donor test is executed, and traces are collected.

The slicer component creates the graph and extracts some subgraphs based on the identified target organ (as the program input). The graph does not need to be entirely loaded in the memory at this stage, and we can mine the logs to traverse it. After traversing the graph and obtaining the list of events, we minimize it by skipping the state-preserving events.

The synthesizer module converts the traversed paths to test methods, installs them in the system, and verifies that they are runnable. At the current proof-of-concept, we do not generate assertion statements, and we will use the assertion-amplification component from another project (SMALL-AMP) in the host project after transplantation.

The main lessons we have learned so far from this proof-of-concept are:

- Tracer needs to manipulate the classes in advance. However, it is difficult to find a list of classes to be manipulated. We use all defined classes in the project as default.
- Manipulating the system classes like `Array`, `Stream` and `Dictoriary` is challenging. It is why we skip manipulating these classes and use some predefined representation for them. However, the number of system classes is considerable, and language-specific knowledge bases are required beforehand.
- When all methods in a project are proxied, the program's execution gets dramatically slow. This shows that it is important to keep the instructions in the method proxies as minimized as possible.

[2]https://github.com/dionisiydk/Mocketry
[3]https://github.com/mabdi/small-mince

V. Related Work

The works by Artzi et al. [10], and Zhang et al. [8] use a similar graph representation to extract a model from the class-under-test and guide a random-based test generator. However, we use this graph to reconstruct a call sequence as a test slice. Staff et al. [11] use a capture and replay technique to replace the environment part of the program with mocks and create unit tests from slower system tests.

GENTHAT is a unit test extraction tool for R language [14]. It analyses execution traces from running in example code and reverse-dependency projects and synthesizes new unit tests. However, they only work with primitive data types.

Messaoudi et al. introduce DS3, an approach to slice system level test by a static analysis enhanced with log analysis [15]. In their work, it is considered that the system test is huge, and the dynamic slicing is not possible, so they use the software-under-test as a black-box and do not analyze it. They only consider the code in the test method and the log produced in the execution. Therefore, they only generate smaller tests with the same statements from the original test method.

Generating differential unit tests by carving [12], [13] represents related work that captures the state of the program before and after of execution of the unit. When the unit evolves, the recorded (carved) pre-state is loaded to memory and the unit is executed, and the state after is compared to the carved post-state. Tiwari et al. [16] also introduce PANKTI which observes the program execution in production and generates a set of differential tests that expands the test coverage. It manipulates the methods under test and serializes the state of receiving object, arguments, and the return value. Then it generates a test method by deserializing the states from trace files and reconstructing the execution state. We see our proposed approach as a complement to this work, whereas we can use objects' history to create a sequence of method calls to reconstruct the same state instead of deserializing the states from files.

VI. Conclusion

This paper addressed the problem of test transplantation from a dependent project to the imported libraries. We choose a test in the dependent project that amplifies the coverage in the library, then we slice it, isolate it if necessary, and move it to the library's test suite. We reduced the test slicing problem to a subgraph finding and backward traversing problem. We illustrated the proposed traversing algorithm using an example and also mentioned our learned lessons from the implementation of a proof-of-concept.

In the future, we will evaluate the algorithm for the test transplantation problem on real projects. In addition, we will explore different use-cases for test slicing: (1) Slicing amplified tests can improve their readability by removing unnecessary statements. (2) Amplifying sliced tests can increase the input amplification surface and, consequently, the test amplification performance. (3) By slicing tests in code clones, similarity-based test transplantation can be possible where we can cover untested clones.

Acknowledgments

This work is supported by (a) the Fonds de la Recherche Scientifique-FNRS and the Fonds Wetenschappelijk Onderzoek - Vlaanderen (FWO) under EOS Project 30446992 SECO-ASSIST (b) Flanders Make vzw, the strategic research centre for the manufacturing industry.

References

[1] I. Schittekat, M. Abdi, and S. Demeyer, "Can we increase the test-coverage in libraries using dependent projects' test-suites?" in *The International Conference on Evaluation and Assessment in Software Engineering 2022*, ser. EASE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 294–298.

[2] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 257–269.

[3] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A snowballing literature study on test amplification," *Journal of Systems and Software*, vol. 157, p. 110398, 2019.

[4] M. Abdi, H. Rocha, S. Demeyer, and A. Bergel, "Small-amp: Test amplification in a dynamically typed language," 2022.

[5] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, p. 446–452, jul 1982.

[6] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[7] T. Xie, "Augmenting automatically generated unit-test suites with regression oracle checking," *Lecture Notes in Computer Science*, pp. 380–403, 2006.

[8] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Combined static and dynamic automated test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 353–363.

[9] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Acm Sigplan Notices*, vol. 49. ACM, 2014, pp. 419–428.

[10] S. Artzi, M. D. Ernst, A. K. Zun, C. P. Jeff, and H. P. Csail, "Finding the needles in the haystack: Generating legal test inputs for object-oriented programs," in *In 1st Workshop on Model-Based Testing and Object-Oriented Systems (M-TOOS)*. Citeseer, 2006.

[11] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for java," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 114–123.

[12] A. Kampmann and A. Zeller, "Carving parameterized unit tests," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '19. IEEE Press, 2019, p. 248–249.

[13] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, "Carving and replaying differential unit test cases from system test cases," *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 29–45, 2009.

[14] F. Křikava and J. Vitek, "Tests from traces: Automated unit test extraction for r," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 232–241.

[15] S. Messaoudi, D. Shin, A. Panichella, D. Bianculli, and L. C. Briand, "Log-based slicing for system-level test cases," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 517–528.

[16] D. Tiwari, L. Zhang, M. Monperrus, and B. Baudry, "Production monitoring to improve test suites," *IEEE Transactions on Reliability*, pp. 1–17, 2021.