Log-Based Slicing for System-Level Test Cases

Salma Messaoudi
University of Luxembourg
Luxembourg
salma.messaoudi@uni.lu

Donghwan Shin
University of Luxembourg
Luxembourg
donghwan.shin@uni.lu

Annibale Panichella
Delft University of Technology
The Netherlands
University of Luxembourg
Luxembourg
a.panichella@tudelft.nl

Domenico Bianculli
University of Luxembourg
Luxembourg
domenico.bianculli@uni.lu

Lionel C. Briand
University of Luxembourg
Luxembourg
University of Ottawa
Canada
lionel.briand@uni.lu

ABSTRACT

Regression testing is arguably one of the most important activities in software testing. However, its cost-effectiveness and usefulness can be largely impaired by complex system test cases that are poorly designed (e.g., test cases containing multiple test scenarios combined into a single test case) and that require a large amount of time and resources to run. One way to mitigate this issue is decomposing such system test cases into smaller, separate test cases—each of them with only one test scenario and with its corresponding assertions—so that the execution time of the decomposed test cases is lower than the original test cases, while the test effectiveness of the original test cases is preserved. This decomposition can be achieved with program slicing techniques, since test cases are software programs too. However, existing static and dynamic slicing techniques exhibit limitations when (1) the test cases use external resources, (2) code instrumentation is not a viable option, and (3) test execution is expensive.

In this paper, we propose a novel approach, called DS3 (Decomposing System teSt caSe), which automatically decomposes a complex system test case into separate test case slices. The idea is to use test case execution logs, obtained from past regression testing sessions, to identify "hidden" dependencies in the slices generated by static slicing. Since logs include run-time information about the system under test, we can use them to extract access and usage of global resources and refine the slices generated by static slicing.

We evaluated DS3 in terms of slicing effectiveness and compared it with a vanilla static slicing tool. We also compared the slices obtained by DS3 with the corresponding original system test cases, in terms of test efficiency and effectiveness. The evaluation results on one proprietary system and one open-source system show that DS3 is able to accurately identify the dependencies related to the usage of global resources, which vanilla static slicing misses. Moreover, the generated test case slices are, on average, 3.56 times faster than original system test cases and they exhibit no significant loss in terms of fault detection effectiveness.

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07.
https://doi.org/10.1145/3460319.3464824

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Automated static analysis.*

KEYWORDS

system level testing, log, program slicing

ACM Reference Format:
Salma Messaoudi, Donghwan Shin, Annibale Panichella, Domenico Bianculli, and Lionel C. Briand. 2021. Log-Based Slicing for System-Level Test Cases. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3460319.3464824

1 INTRODUCTION

Regression testing is a quality assurance technique applied when changes are made to an existing codebase. It provides confidence that the performed changes do not harm the behavior of the existing and unchanged parts of the code [35]. Although many techniques have been introduced for cost-effective regression testing, such as test case prioritization and test suite selection, their usefulness can be largely impaired if individual test cases (1) are poorly designed (e.g., the codebase contains test smells) and (2) require a large amount of time and resources to run [3, 28].

We observed both phenomena in the context of a collaborative industrial research project, with a large company in the aerospace domain. Due to the intrinsic complexity of the system under test (SUT) and to the accumulated technical debt over several years of software development, system test cases often contain multiple test scenarios combined into a single test case. These tests, often called *eager tests* [30], negatively impact both regression testing and test evolution. In our industrial context, *eager tests* are very expensive as they take several hours to run. This means that, for example, even if a state-of-the-art test case prioritization technique

is applied, no faults could be detected during the first few hours of test execution. Furthermore, eager tests are more difficult to read, understand, document, and evolve [30].

However, if such complex system test cases could be decomposed into smaller test cases without losing their test effectiveness, the execution time of the decomposed test cases would decrease and the cost-effectiveness of test case prioritization could improve. Moreover, having smaller test cases would facilitate fault localization [33] and program maintenance [3, 30] by providing more granular information about test results. Furthermore, as decomposed test cases have distinct test assertions, engineers would be able to easily pick specific test cases of interest and run them efficiently.

To achieve this, ideally, one would decompose a complex system test case containing multiple test scenarios into separate system test cases, each of them with only one test scenario and its corresponding assertions. Since system test cases are software programs too, they could be decomposed using static slicing techniques based on def-use analysis [32]. However, existing *static slicing* techniques cannot identify and deal with "hidden" dependencies between statements, originated from the usage of global resources such as external files or databases; such missing dependencies would lead to run-time errors in the resulting decomposed (sliced) test cases. *Dynamic slicing* [4] could be an alternative to static slicing, but it requires to instrument the source code and to collect coverage information. However, this alternative is not feasible for a system composed of third-party components and it would not address the problem of handling "hidden" dependencies as they are not captured by code coverage. Finally, *observational slicing* [4, 34] would require running each system test case multiple times. Such an approach is not applicable for typically long system test case execution times, as it is the case for the system developed by our industrial partner.

In this paper, we tackle the problem of slicing a complex system test case into simpler ones — without missing any "hidden" dependency between statements — by proposing a novel approach, called *DS3 (Decomposing System teSt caSe)*, which complements static slicing with a log-based analysis. The idea is to use test case execution logs, obtained from past regression testing sessions, to identify missing dependencies in the decomposed test cases generated by static slicing. Since logs include run-time information about the SUT, we can use them to extract the global resources accessed (e.g., files, databases) and the actions performed (e.g., read/write file, open/close database) upon executing each statement in the original (unsliced) system test case. In this way, we can reconstruct the additional dependencies between statements as defined by the usage of global resources.

DS3 first generates test slices (i.e., decomposed test cases) by applying backward static slicing using individual assertions included in the original system test case as slicing criteria. Then, DS3 complements the individual slices taking into account any missing dependencies identified by the analysis of the test execution logs.

We implemented DS3 in a prototype tool, on top of an off-the-shelf program slicing tool. We evaluated DS3 in terms of slicing effectiveness (i.e., ability to identify all required dependencies) and compared it with the vanilla static slicing tool. We also compared the test case slices obtained with DS3 with the corresponding unsliced system test case, in terms of efficiency and effectiveness, i.e., how quickly we can verify individual assertions and how many faults we can detect. In our evaluation, we used one proprietary system provided by our industrial partner and one open-source system. The results show that DS3 is able to accurately identify the dependencies related to the usage of global resources, which vanilla static slicing misses. Moreover, decomposed test cases are much faster than the corresponding original system test case (with an average speedup of $3.56x$) and there is no significant loss in terms of effectiveness (fault detection rate). Additionally, both the original system test cases and the decomposed test cases have the same function coverage with a small difference in branch coverage.

To summarize, the main contributions of this paper are:

(1) DS3, an approach for slicing complex system test cases using the global resources usage information available in test case execution logs;
(2) the evaluation of DS3 in terms of slicing effectiveness as well as test efficiency and effectiveness.

The rest of the paper is organized as follows. Section 2 provides some basic definitions. Section 3 illustrates the motivating example. Section 4 describes the main steps of DS3. Section 5 reports on the evaluation of DS3. Section 6 discusses the practical implications of using DS3. Section 7 summarizes the related work. Section 8 concludes the paper and provides directions for future work.

2 BACKGROUND

2.1 Logs

A *log* is a sequence of log entries; a *log entry* contains a timestamp (recording the time at which the logged event occurred) and a log message (recording the logged event). A log message can be further decomposed [10, 22] into a *message template*, characterizing the event type, and the parameter values of the event, which are determined at runtime. For example, given the log entry `14:26:00 read file X`, we can see that the event of the template **`read file`** `*` occurred at timestamp `14:26:00` with the value of `X`. Here, `*` denotes the position of the variable part in the template. More formally, let MT be the set of all the message templates that can occur in a program P, and V be the set of all mappings from template parameters to their concrete values, for all templates $mt \in MT$. A log L is a sequence of log entries $\langle e_1, \ldots, e_k \rangle$ where $e_i = (ts_i, mt_i, v_i)$, $ts_i \in \mathbb{N}$, $mt_i \in MT$, and $v_i \in V$ for $i = 1, 2, \ldots, k$.

2.2 Static Slicing

Static slicing is a technique, using def-use analysis [32], for isolating a "slice" of a program (i..e, a subset of the original program statements) that affects the computation of the value of one or more variables in a specific statement in the program. More formally, a slice S of a program P is constructed with respect to a slicing criterion (s, V) where s is a statement in P and V is a set of variables in s; a statement in P is removed to form S if it does not affect the computation of V at s.

3 MOTIVATING EXAMPLE

In this section, we present an example that motivates our work.

Figure 1 shows, on the top, an example system test case $T_{sys} = \langle s_1, s_2, \ldots, s_7 \rangle$ (where s_i is the i-th statement). Figure 2 shows T_{sys}'s corresponding execution log $L_{sys} = \langle e_1, \ldots, e_4 \rangle$; for simplicity, we

```
def test_example():
  (1) fdm = create_fdm_setup()
  (2) ref = read_csv('output.csv')
  (3) sim = deploy_proc('output.csv')
  (4) self.assertEqual(ref, sim)
  (5) new = run_ic()
  (6) diff = FindDiffs(ref, new, 1E-8)
  (7) self.assertEqual(len(diff), 0)
```

```
def test_example_ideal_slice1():
  (1) fdm = create_fdm_setup()
  (2) ref = read_csv('output.csv')
  (3) sim = deploy_proc('output.csv')
  (4) self.assertEqual(ref, sim)
```

```
def test_example_ideal_slice2():
  (1) fdm = create_fdm_setup()
  (2) ref = read_csv('output.csv')
  (5) new = run_ic()
  (6) diff = FindDiffs(ref, new, 1E-8)
  (7) self.assertEqual(len(diff), 0)
```

Figure 1: A system test case T_{sys} (top) and its ideal slices I_1 (middle) and I_2 (bottom)

ID	Statement	Template	Value
e_1	s_1	**read file** *	setup.xml
e_2	s_1	**write file** *	output.csv
e_3	s_2	**read file** *	output.csv
e_4	s_3	**read file** *	output.csv

Figure 2: The execution log L_{sys} for T_{sys}

show the structured log instead of its original, free-formed log and omit log entries not related to the usage of global resources. Notice that each log entry has a reference to the test case statement originating it. This example is a simplified version of a system test case in *JSBSim* [13], an open-source flight simulator.

The example test case has been designed to cover multiple test scenarios, with the presence of two assertions[1] (i.e., s_4 and s_7). Such test cases can become less than an ideal if an engineer is interested in testing only a specific scenario, and the execution of *both* test scenarios is expensive.

Ideally, T could be replaced with two test cases $I_1 = \langle s_1, s_2, s_3, s_4 \rangle$ and $I_2 = \langle s_1, s_2, s_5, s_6, s_7 \rangle$, as shown at the middle and at the bottom of Figure 1, each of which contains only an assertion referring to a specific test scenario. In this way, an engineer can select which test scenario to execute, reducing the overall testing cost. Notice that, though the new test cases are smaller than the original system test case, executing I_1 and I_2 is equivalent to executing T in terms of code coverage.

An engineer may try to use program slicing to generate I_1 and I_2 from T_{sys} since T_{sys} is a software program too and I_1 and I_2 can be seen as slices of T_{sys}. In particular, to have one assertion per slice, the engineer could apply backward static slicing using $\langle s_4, \{\texttt{ref}, \texttt{sim}\} \rangle$ and $\langle s_7, \{\texttt{diff}\} \rangle$ as slicing criteria. However, the variable `fdm` defined in s_1 is never used in the following statements in T_{sys}, and therefore none of the slices generated based on the data- and control-flow of the program code contains s_1. This is critical because, as recorded in L_{sys}, file `output.csv` needed in s_2 is internally generated by `create_fdm_setup()` in s_1. In practice, this means that the execution of the sliced test cases generated with vanilla static slicing will result in a crash, due to the missing resource (file `output.csv`).

Overall, because of the "hidden" dependency between s_1 and s_2, static slicing alone cannot properly generate I_1 and I_2 from T_{sys}.

This simple example shows the need for extending static slicing to identify hidden dependencies due to the usage of global resources. In the next section, we will present a method that achieves this goal leveraging the information contained in test case execution logs.

4 LOG-BASED TEST CASE DECOMPOSITION

Our new approach, called DS3, decomposes a complex system test case containing multiple test scenarios into multiple individual system test cases, each of them with only one test scenario and its related subset of assertions, while preserving the hidden dependencies due to the usage of global resources. The main idea is to complement static slicing with a log-based analysis. Since test execution logs include run-time information about the system under test, we can use them to extract the global resources accessed (e.g., files, databases) and the actions performed (e.g., read, write) upon executing each statement in the original (unsliced) system test case. In this way, we can reconstruct *hidden* dependencies between statements generated at run-time by global resources, which were not identified by static slicing.

DS3 takes as input a system test case, an execution log corresponding to the test case, and the log message templates related to global resources; it returns a set of slices, each of them exercising an individual test scenario and containing fewer assertions. In our running example, DS3 takes the system test case T_{sys} (Figure 1, top) and its corresponding log L_{sys} (Figure 2) and returns ideal slices I_1 and I_2 (Figure 1, middle and bottom). The engineer is only required to mark log message templates related to global resources, such as `output.csv`, in the log. For example, the log entry e_1 in L_{sys} indicates that the "read" operation is performed on file `setup.csv`. Hence, by looking at each message template, such as **`read file *`**, engineers can easily identify if it is related to the usage of global resources. Then, DS3 automatically identifies the hidden dependency between s_1 and s_2 using a log-based analysis, and generates I_1 and I_2 by refining the intermediate slices generated by static slicing.

Note that our approach is black-box: it does not require access to the source code. Therefore, it can be applied to software systems composed of 3rd-party components, whose source code is not accessible, as it is the case for the system developed by our industry partner. Nevertheless, we need two conditions to be satisfied to apply DS3: (1) there is a traceability information between statements in the test case and messages in the log and (2) the log contains some information on the usage of global resources. These conditions are required to identify 1) which messages were logged upon the execution of each statement of the test case and 2) the global resources used as part of the statement execution. Such conditions are satisfied by the system developed by our industry partner. In general, such conditions can easily be satisfied

[1] We remark that, though assertions are common practice, they are not the only kind of test oracles; for example, exceptions in the source code can be seen as test oracles. However, we focus on assertions in this work, since this is the current practice of our industrial partner.

by appropriately instrumenting test cases and adding a watchdog process (with logging capabilities) to monitor the usage of global resources at run time. Though DS3 additionally requires engineers to manually mark message templates related to global resources, the number of all templates is typically manageable (e.g., there are 14 message templates in our proprietary system), and it is easy for engineers with domain knowledge to identify the templates related to the usage of global resources.

Algorithm 1 provides the pseudo-code of DS3. It takes as input a system test case $T = \langle s_1, \ldots, s_n \rangle$, its corresponding execution log $L = \langle e_1, \ldots, e_k \rangle$, and the set of log messages templates $MT_G = \{mt_1, \ldots, mt_m\}$ marked as related to the usage of global resources in L; it returns a set of decomposed test cases (i.e., slices) $\mathcal{D} = \{D_1, \ldots, D_j\}$.

Algorithm 1 DS3: Decomposing System Test Case

Input: System Test Case $T = \langle s_i, \ldots, s_n \rangle$
Log $L = \langle e_1, \ldots, e_k \rangle$
Set of Templates $MT_G = \{mt_1, \ldots, mt_m\}$
Output: Set of Decomposed Test Cases $\mathcal{D} = \{D_1, \ldots, D_j\}$
1: Set of Test Cases $\mathcal{D} \leftarrow \emptyset$
2: **for** Assertion Statement $s \in T$ **do**
3: Set of Variables $V_s \leftarrow$ GET-VARS(s)
4: Test Case $D_s \leftarrow$ BACK-SLICE(s, V_s, T)
5: $\mathcal{D} \leftarrow \mathcal{D} \cup \{D_s\}$
6: **end for**
7: Set of Triples $G_{du} \leftarrow$ GLOBAL-DU(L, T, MT_G)
8: **for** Decomposed Test Case $D \in \mathcal{D}$ **do**
9: Test Case $D_{tmp} \leftarrow D$
10: **for** Statement $s \in D_{tmp}$ **do**
11: Set of Statements $W \leftarrow$ DEP-STMTS(s, G_{du}, T)
12: $D \leftarrow$ ADD(D, W)
13: **end for**
14: **end for**
15: **return** MINIMIZE($\mathcal{D}$)

Algorithm 1 consists of four major stages: (1) assertion-based backward slicing (lines 1–6), (2) def-use analysis for global resources using logs (line 7), (3) slice refinement (lines 8–14), and (4) slice minimization (line 15). The backward slicing stage generates static slices $\mathcal{D}$ from T. The global resources def-use analysis stage identifies the relationships between the statements in T, the log entries in L related to global resources, and the actions performed on the latter, using L and MT_G. The resulting set of triples G_{du} is then used to refine each of the static slices $D \in \mathcal{D}$ in the slice refinement stage. Last, the slice minimization stage removes any redundant slices in $\mathcal{D}$. The algorithm ends by returning the minimized $\mathcal{D}$. The four stages are described in detail in the following subsections.

4.1 Assertion-Based Backward Slicing

Algorithm 1 first performs backward static slicing on the assertions in T to generate a set of static slices $\mathcal{D}$ (lines 1–6). This guarantees that each $D \in \mathcal{D}$ has at most one test scenario by having one assertion. The algorithm starts by initializing $\mathcal{D}$ as an empty set (line 1). For each assertion statement $s \in T$ (lines 2–6), the algorithm gets the variables V_s in s (line 3), performs the backward static slicing on T using $\langle s, V_s \rangle$ as the slicing criterion (line 4), and adds the resulting slice D into $\mathcal{D}$ (line 5).

```
def test_example_static_slice1():
  (2) ref = read_csv('output.csv')
  (3) sim = deploy_proc('output.csv')
  (4) self.assertEqual(ref, sim)
```

```
def test_example_static_slice2():
  (2) ref = read_csv('output.csv')
  (5) new = run_ic()
  (6) diff = FindDiffs(ref, new, 1E-8)
  (7) self.assertEqual(len(diff), 0)
```

Figure 3: Static slice results: T_{static_1} (top) and T_{static_2} (bottom)

In our running example T_{sys}, for the assertion $s_4 \in T_{sys}$, the algorithms performs the backward static slicing using $\langle s_4, \{\texttt{ref}, \texttt{sim}\} \rangle$ as the slicing criterion, yielding the slice $D_{s_4} = T_{static_1} = \langle s_2, s_3, s_4 \rangle$, shown at the top of Figure 3. Note that s_1 is not included in T_{static_1} because the variable `fdm` is not used in any statements in T_{static_1} based on the static def-use analysis. Similarly, for the second assertion $s_7 \in T_{sys}$, the backward slicing with the slicing criterion $\langle s_7, \{\texttt{diff}\} \rangle$ yields another slice $D_{s_7} = T_{static_2} = \langle s_2, s_5, s_6, s_7 \rangle$, shown at the bottom of Figure 3. T_{static_2} does not include s_1, s_3, and s_4 as they do not affect the computation of the statements in T_{static_2} based on the static analysis.

4.2 Def-Use Analysis for Global Resources

The second stage of Algorithm 1 identifies a set of triples G_{du} in T using L and MT_G (line 7) where each triple $\langle s, g, a \rangle \in G_{du}$ indicates that an action a is performed on a global resource g when the statement $s \in T$ is executed. This is done by Algorithm 2.

Algorithm 2 GLOBAL-DU

Input: Log $L = \langle e_1, \ldots, e_k \rangle$
System Test Case $T = \langle s_1, \ldots, s_n \rangle$
Set of Templates $MT_G = \{mt_1, \ldots, mt_m\}$
Output: Set of Triples G_{du}
1: $G_{du} \leftarrow \emptyset$
2: **for** Statement $s \in T$ **do**
3: Set of Log Entries $E_s \leftarrow$ ENTRY-FOR-STATEMENT(s, L)
4: **for** Log Entry $e \in E_s$ **do**
5: **if** TEMPLATE(e) $\in MT_G$ **then**
6: String $g \leftarrow$ GET-GLOBAL-RESOURCE(e)
7: Action $a \leftarrow$ GET-ACTION-TYPE(e)
8: $G_{du} \leftarrow G_{du} \cup \{\langle s, g, a \rangle\}$
9: **end if**
10: **end for**
11: **end for**
12: **return** G_{du}

Algorithm 2 starts by initializing G_{du} as an empty set. For each statement $s \in T$ (lines 2–11), the algorithm identifies the set of log entries E_s originated from s using the traceability information between the statements in T and the log entries in L (line 3). Then, for each log entry $e \in E_s$ (lines 4–10) whose template is in MT_G (line 5), the algorithm identifies the global resource g from e's parameter value (line 6) and the action type a from e's template (line 7). If the template contains predefined keywords indicating a "use" of the resource, such as `read` and `access`, then a = `use`; similarly, keywords such as `write` and `update` indicate a "definition" of a resource and

we have a = `def`. The default set of keywords characterizing uses and definitions of global resources can be enhanced by engineers. The identified triple $\langle s, g, a\rangle$ is added into G_{du} (line 8); the algorithm ends by returning G_{du}.

In our running example, let us consider the case where GLOBAL-DU is called with parameters $T = T_{sys} = \langle s_1, \ldots, s_7\rangle$, $L = L_{sys} = \langle e_1, \ldots, e_4\rangle$, and MT_G = {**`read file *`**, **`write file *`**}. The algorithm first initializes G_{du} to $\emptyset$ and starts the iteration over statements in T_{sys}. For $s_1 \in T_{sys}$, the call to ENTRY-FOR-STATEMENT with s_1 and L_{sys} returns $E_s = \{e_1, e_2\}$, by checking the reference of each log entry to the test case statement originating in L_{sys}. Based on E_s, the algorithm starts the inner iteration over entries in E_s. For $e_1 \in E_s$, since e_1's template, i.e., **`read file *`**, is in MT_G, the algorithm identifies the global resource and action type of e_1 using GET-GLOBAL-RESOURCE and GET-ACTION-TYPE. Specifically, the call to GET-GLOBAL-RESOURCE with e_1 checks the parameter value of e_1 and returns g = `setup.xml`. Similarly, the call to GET-ACTION-TYPE with e_1 checks the template of e_1 and returns a = `use` because **`read file *`** contains the `read` keyword. The algorithm ends the inner iteration for e_1 by adding the triple $\langle s_1$, `setup.xml`, `use`$\rangle$ into G_{du} and moves on to the next iteration to process e_2. After processing all statements and corresponding log entries, the algorithm returns G_{du} = { $\langle s_1$, `setup.xml`, `use`$\rangle$, $\langle s_1$, `output.csv`, `def`$\rangle$, $\langle s_2$, `output.csv`, `use`$\rangle$, $\langle s_3$, `output.csv`, `use`$\rangle$ }.

4.3 Log-Based Slice Refinement

The two previous stages of Algorithm 1 compute $\mathcal{D}$ and G_{du}. In this stage, we aim to refine the slices $\mathcal{D}$ using the information in G_{du} (lines 8–14). Specifically, for each slice $D \in \mathcal{D}$ and for each statement $s \in D$, the algorithm finds all the statements of T needed for s using algorithm DEP-STMTS (line 11, described in detail below) and adds the found statements into D (line 12). As a result, we ensure that all $D \in \mathcal{D}$ have no missing statements, and can be executed without resulting in a crash due to the improper usage of a global resource (e.g., writing to a file before opening it).

Algorithm 3 presents the pseudo-code of DEP-STMTS, the core of the slice refinement stage. It *recursively* finds dependent statements using both the information in G_{du} and backward static slicing[2].

For a given s, G_{du}, and T, the algorithm first gets a set of variables V_s in s and calculates a set of statements S_{dir} directly needed for s using DEFS and BACK-SLICE. Specifically, the call to DEFS with s and G_{du} returns all statements s' such that $\langle s', g,$ `def`$\rangle \in G_{du}$ and $\langle s, g,$ `use`$\rangle \in G_{du}$. The call to BACK-SLICE with s, V_s, and T returns all statements in the static slice of T constructed by the slicing criterion of $\langle s, V_s\rangle$. If $S_{dir} = \emptyset$, the algorithm returns $\emptyset$ (line 4); otherwise, the algorithm collects another set of statements S_{rec} needed for each statement $s_{dir} \in S_{dir}$ (lines 6–9) by recursively calling DEP-STMTS with s_{dir} (line 8) and then returns $S_{dir} \cup S_{rec}$ (line 10).

In our running example, let us consider the case where DEP-STMTS is called with parameters $s = s_2, G_{du} = \{\langle s_1$, `setup.xml`, `use`$\rangle$, $\langle s_1$, `output.csv`, `def`$\rangle$, $\langle s_2$, `output.csv`, `use`$\rangle$, $\langle s_3$, `output.csv`, `use`$\rangle\}$,

[2]Note that Algorithm 3 could be called for the same statement multiple times. To reduce the execution time, Algorithm 3 internally keeps a cache of the dependency information for each statement.

Algorithm 3 DEP-STMTS

Input: Statement s
Set of (Global Resource def-use) Triples G_{du}
System Test Case $T = \{s_1, \ldots, s_n\}$
Output: Set of Statements S
1: Set of Variables $V_s \leftarrow$ GET-VARS(s)
2: Set of Statements $S_{dir} \leftarrow$ DEFS(s, G_{du}) $\cup$ BACK-SLICE(s, V_s, T)
3: **if** $S_{dir} = \emptyset$ **then**
4: **return** $\emptyset$
5: **else**
6: Set of Statements $S_{rec} \leftarrow \emptyset$
7: **for** Statement $s_{dir} \in S_{dir}$ **do**
8: $S_{rec} \leftarrow S_{rec} \cup$ DEP-STMTS(s_{dir}, G_{du}, T)
9: **end for**
10: **return** $S_{dir} \cup S_{rec}$
11: **end if**

and $T = T_{sys}$. Since s_2 has no variables, $V_s = \emptyset$ and the call to BACK-SLICE returns $\emptyset$. On the other hand, the call to DEFS returns $\{s_1\}$ since $\langle s_1$, `output.csv`, `def`$\rangle \in G_{du}$ and $\langle s_2$, `output.csv`, `use`$\rangle \in G_{du}$. Thus, $S_{dir} = \{s_1\}$, and the algorithm recursively calls DEP-STMTS for s_1. Since s_1 does not depend on any other statement, the recursive call returns $\emptyset$, leading to $S_{rec} = \emptyset$. The algorithm ends by returning $S_{dir} \cup S_{rec} = \{s_1\}$.

Recall that the assertion-based backward slicing stage calculated $D_{s_4} = T_{static_1} = \langle s_2, s_3, s_4\rangle$ and $D_{s_7} = T_{static_2} = \langle s_2, s_5, s_6, s_7\rangle$ for T_{sys}. Since the call to DEP-STMTS for s_2 returns $\{s_1\}$ as described above, both D_{s_4} and D_{s_7} can be refined, thanks to the inclusion of s_1. Note that, while s_3 also depends on s_1 according to G_{du}, function ADD in Algorithm 1 (line 12) does not redundantly add s_1 to D_{s_4} and D_{s_7}. As a result, D_{s_4} and D_{s_7} become the same as the ideal slices I_1 and I_2, respectively.

4.4 Slice Minimization

After the slice refinement stage, $\mathcal{D}$ may contain slices that (excluding the assertions) are subsets of others. A slice D_i is a *subset* of another slice D_j, denoted by $D_i \sqsubseteq D_j$, if all statements (except assertions) of D_i are in D_j. As we are dealing with test case slices, we make the following assumption, based on our observations in real-word codebases: $D_i \sqsubseteq D_j$ (or $D_j \sqsubseteq D_i$) holds when D_i and D_j belong to the same test scenario and thus share the same test fixture (e.g., the same setup and teardown code). For example, a test scenario for the initialization routine of an object can be implemented with multiple assertions that verify the initialization of the various properties of the object; in such a case, the assertions will rely on the same setup code. Based on this assumption, DS3 includes a minimization stage to remove any redundant slices in terms of test scenarios. Specifically, the MINIMIZE function in Algorithm 1 analyzes each obtained slice (ignoring assertions) in $\mathcal{D}$, to ensure that the property $\forall D_i, D_j \in \mathcal{D},\ \ D_i \not\sqsubseteq D_j \wedge D_j \not\sqsubseteq D_i$ holds on $\mathcal{D}$.

If MINIMIZE finds two slices $D_i, D_j \in \mathcal{D}$ such that $D_i \sqsubseteq D_j$, it merges D_i and D_j by moving the assertion from D_i into D_j (preserving the order among statements defined in the original test case) and removing D_i from $\mathcal{D}$.

In our running example, when ignoring the assertion statements, $D_{s_4} = I_1$ contains s_3 and s_4 that are not included in $D_{s_7} = I_2$,

and therefore $D_{s_4} \not\subseteq D_{s_7}$; similarly, $D_{s_7} \not\subseteq D_{s_4}$. So MINIMIZE returns $\{D_{s_4}, D_{s_7}\} = \{I_1, I_2\}$, and Algorithm 1 ends by returning $\mathcal{D} = \{I_1, I_2\}$.

Note that the minimization stage results in some slices having multiple assertions since MINIMIZE merges slices without taking them into account. Nevertheless, based on the aforementioned assumption, the assertions included in each slice belong to the same test scenario. Furthermore, the assertions of the original test case will be distributed over the generated slices, thus reducing the overall number of assertions per slice.

5 EVALUATION

We implemented DS3 as a `Python` program, using the `Python-Program-Analysis` toolkit [26] to perform static slicing.

In this section, we report on the assessment of DS3 in slicing system test cases, and the *effectiveness* and *efficiency* of the obtained slices. More specifically, our research is steered by the following research questions:

RQ1: *How effective is DS3 in slicing system test cases compared to standard static slicing?*

RQ2: *How efficient are the slices produced by DS3 compared to the original test cases?*

RQ3: *What is the code coverage and fault detection capability of the slices produced by DS3 compared to the original test cases?*

RQ1 investigates how effective DS3 is in slicing system test cases into sliced test cases that successfully compile and have no run-time errors (i.e., no "hidden" dependency is missing). These two aspects are essential when decomposing complex system test cases since slices yielding compilation or run-time errors are useless.

RQ2 assesses the running time (efficiency) of the generated test slices compared to the corresponding original (non-sliced) test cases. Efficiency is important in the context of regression testing because developers would execute only a subset of the test cases (and their assertions) to find regression faults within the available time budget. The efficiency of regression testing depends on the running time of the test cases that are selected.

RQ3 analyzes the code coverage and fault detection capability of the generated test slices compared to the non-sliced ones. Since DS3 decomposes system test cases into slices, a potential drawback is that the latter may be less effective than the former in terms of structural coverage and fault detection capability. Thus, it is essential to investigate how structural coverage and fault detection capability may be affected by applying DS3.

5.1 Benchmarks

A candidate benchmark for our evaluation should meet the following requirements: (1) it contains system or integration level test cases, (2) the test cases should generate logs when executed, and (3) the test cases should access/use global resources (e.g., external files, databases, remote resource).

These requirements are fulfilled by a proprietary benchmark, hereafter referred to as *Prop*, provided by one of our industrial partners active in the satellite industry. This benchmark includes 30 complex system test cases written in Python, each of which takes on average 53 minutes to execute, as it triggers multiple cyber-physical components[3].

To increase the diversity of our experimental subjects and support open science, we aimed to include in our benchmark one open-source system as well. Among the top 10 trending repositories on GitHub, we filtered out those that do not satisfy the above requirements, ending up with one open-source system, namely *JSBSim*, an open-source flight simulator already introduced in our running example in section 3. It is mainly written in C++ with about 300 source files (in total over 20 KLOC). It also includes 81 system-level test cases written in Python.

Our approach uses logs to detect dependencies originated from the usages of global resources. The proprietary test cases generate log messages with detailed information regarding the usage of global resources (file, database, and network connection) and the timestamp of each executed test statement. Therefore, *Prop* did not require further modifications to generate appropriate logs.

In contrast, the logs generated by the *JSBSim* test cases do not include the usages of global resources by default. Thus, we implemented a *watchdog* script that monitors the usage of global resources during test executions. Since the system test cases in *JSBSim* access and modify external files only, our script captured the names of the changed files as well as the timestamp of each file access. Besides, we also instrumented the test cases to store the timestamps of each executed test statement. This allowed us to determine precisely which statement read or wrote which external files. Notice that implementing a watchdog and instrumenting system test cases can easily be automated, without requiring access to the code base.

5.2 RQ1: Slicing Effectiveness

5.2.1 Methodology. To answer RQ1, we considered static slicing (using the same slicing criterion used in the assertion-based backward slicing stage of DS3, see 4.1) as the baseline for comparison; we used the implementation provided by an open-source toolkit, `python-program-analysis`, since all the test cases in our benchmarks are written in Python. We consider neither dynamic slicing nor observational slicing as alternative baselines. The former is not feasible as our proprietary system *Prop* includes several third-party components; hence, instrumenting these components and building the test execution traces is not possible. The latter is too expensive for system test cases since it requires executing each test case multiple times, each time by deleting one single test statement and observing whether the test case fails [4]. Considering the average execution time of 53 minutes of the system test cases in the *Prop* benchmark, observational slicing was not applicable from a practical standpoint.

To assess the slicing effectiveness, we ran both DS3 and the baseline static slicing tool on the two benchmarks, obtaining two sets of sliced test cases (one generated by each tool). To measure the slicing effectiveness of the two approaches, we used the following metric:

$$Eff(\mathcal{D}) = \frac{|\{\mathcal{D}_i \in \mathcal{D} \mid \mathcal{D}_i \text{ has no errors}\}|}{|\mathcal{D}|}$$

[3] Due to non-disclosure agreements, we cannot divulge more details about this system.

Table 1: Comparison between DS3 and static slicing in terms of slicing effectiveness $Eff(\mathcal{D})$

System	# Test Cases	Approach	#Slices			$Eff(\mathcal{D})$
			Total	#Pass	#Fail	
JSBSim	76	DS3	84	84	0	1
		Static Slicer	169	56	113	0.33
Prop	30	DS3	137	137	0	1
		Static Slicer	166	40	126	0.24

where $\mathcal{D}$ is the set of slices produced by a given approach (DS3 or the baseline). In the formula, the numerator indicates the number of test slices that do not lead to compilation or run-time errors; the denominator represents the total number of generated slices. The value of *Eff* ranges between 0 and 1; larger values are preferable as they indicate fewer failing test slices. In our context, generated slices may fail due to missing dependencies; therefore, larger $Eff(\mathcal{D})$ values mean that the technique under analysis is more effective in generating correct slices that do not miss any dependency.

Notice that the same static slicer used in the first stage of DS3 is also used as baseline static slicer. This means that the comparison between DS3 and the baseline actually shows the effect of log-based refinement on vanilla static slicing.

Note that each test case was executed in a sandbox to avoid any side effects on other test cases. Furthermore, to avoid non-deterministic behaviors, we ran each of the original test cases multiple times and removed flaky test cases (i.e., passing in some runs and failing in others).

5.2.2 Results. Table 1 reports the number of slices produced by both our approach and the baseline. Column "System" indicates the name of the benchmark; column "# Test Cases" indicates the number of the original test cases to be sliced; columns "Total", "Pass", and "Fail" indicate, respectively, the total number of obtained slices, the number of test case slices that successfully passed when executed, and the number of test case slices that failed when executed; column "$Eff(\mathcal{D})$" indicates the slicing effectiveness of an approach. The total number of test cases to be sliced for *JSBSim* is 76 because three of them were flaky and two additional test cases could not be properly parsed by the static slicer.

The results show that the static slicer has lower slicing effectiveness than DS3 for both subject systems. For *JSBSim*, the static slicer achieved an effectiveness score of 0.33. Indeed, 67% of the test slices it generated resulted in compilation or run-time errors; such errors occurred because the static slicer missed hidden dependencies. For *Prop*, only 24% of the slices generated by the static slicer ran successfully, without run-time errors, as they had no missing dependencies. Instead, all the slices generated by DS3 ran successfully, without errors, resulting in an effectiveness score of 1. Overall, this means that DS3, leveraging the global resources usages recorded in the logs, is able to identify many hidden dependencies that a vanilla static slicer would have missed.

5.3 RQ2: Efficiency of the Sliced Test Cases

5.3.1 Methodology. To answer RQ2, we compared the execution time of the generated test slices with the execution time of the original (non-sliced) test cases. As the test case execution can alter the environment (e.g., by creating or modifying a file), we reset the environment before running each test case, to avoid any incorrect results. We ran each test 10 times to account for the uncertainty in test execution time. We also assessed the overhead of DS3 by measuring, over the 10 executions, the average time taken by DS3 internal stages (see Algorithm 1 in section 4) and the total execution time for slicing a given test case.

The *JSBSim* test cases have a very short execution time (a few seconds), so they are not adequate to realistically assess the efficiency of the sliced test cases. For this reason, to answer RQ2, we only considered the results obtained for the *Prop* test cases.

All test cases (original and sliced) were executed on an Apple MacBook Pro computer with a 2.50 GHz Intel Core i7 processor and 16 GB of memory.

5.3.2 Results. Table 2 shows the time (in seconds) for running DS3 to generate the test slices, as well as the time for executing the generated slices and the original test cases. More specifically, columns "Others", "Static", and "Refine" indicate the average time for executing the different stages of DS3: finding global resources defs and uses and the minimization step (column "Others"), static slicing (column "Static"), slice refinement (column "Refine"); column "Total" indicates the average total execution time of DS3; column "Slices" indicates the number of slices produced by DS3; column "Org" indicates the execution time of the original (non-sliced) test cases; columns "Sl. Avg" and "Sl. Tot" indicate the average and the cumulative execution time of the sliced test cases, respectively, where the latter represents the sum of the execution time of all slices obtained for each individual system test case; column "Speedup" indicates the speedup ratio between the execution time of the non-sliced test cases and the cumulative execution time of the sliced test cases.

The results shows that, for 24 out of 30 test cases, DS3 produced test case slices whose cumulative execution time is shorter than the one of the original test case, with an average speedup of 3.56x.

The largest speedup (23.14x) can be observed for test case PTC30. In this case, the cumulative execution time of the five slices is 1004 s (≈ 17 minutes) on average; instead, executing the original test case requires 23 231 s (≈ 387 minutes, i.e., more than six hours). This large difference is due to the execution, within the original test case, of an expensive procedure that actually does not have any dependencies with other statements in the test case; indeed, DS3 successfully determined and excluded this procedure in the test case slices it generated.

In the remaining six test cases (characterized by a speedup ratio lower than one), the total execution time of the test slices was higher than the one of the original test case. We observed the lowest value of the speedup ratio (i.e., the highest slowdown, 0.53) for PTC5: the execution time of the original test case was 465 s while the cumulative execution time of the two slices produced by DS3 was 874 s. To further understand the root cause of this large increase in execution time, we manually analyzed PTC5 and its corresponding slices. We discovered that, for this case, DS3 created two independent test slices, each with the same copy of the test

Table 2: Execution time (in seconds) of DS3 and of the original and sliced system test cases.

STC	DS3 (s)				Slices	Test Cases (s)			Speedup
	Others	Static	Refine	Total		Org	Sl. Avg	Sl. Tot	
PTC1	2.76	5.06	204.42	212.23	2	1139	520.00	1040	1.10
PTC2	2.19	3.25	108.79	114.21	2	498	342.50	685	0.73
PTC3	0.14	3.08	32.99	36.15	3	245	132.60	398	0.62
PTC4	3.51	3.07	54.11	60.70	1	330	210.00	210	1.57
PTC5	0.10	3.57	84.65	88.35	2	465	437.00	874	0.53
PTC6	2.77	3.07	36.31	42.25	2	2720	139.00	278	9.78
PTC7	13.84	18.53	258.58	290.94	3	3080	372.00	1116	2.76
PTC8	18.78	25.36	260.06	304.18	4	3765	404.75	1619	2.33
PTC9	0.75	5.48	15.49	21.72	1	197	103.00	103	1.91
PTC10	10.24	68.15	89.47	167.62	7	2280	296.43	2075	1.10
PTC11	9.50	4.95	20.13	34.58	3	1041	121.00	363	2.87
PTC12	4.26	9.95	69.89	84.10	3	913	182.67	548	1.67
PTC13	3.65	50.81	28.29	82.74	5	4150	328.60	1643	2.53
PTC14	0.01	27.81	91.25	119.07	7	15480	144.00	1008	15.36
PTC15	4.16	5.66	9.50	19.32	2	541	89.00	178	3.04
PTC16	0.51	49.58	142.25	192.32	6	922	128.50	771	1.20
PTC17	5.08	11.43	596.04	612.53	8	4975	212.50	1700	2.93
PTC18	0.04	11.46	11.14	22.61	5	4091	116.00	580	7.05
PTC19	0.02	5.08	36.11	41.19	3	183	68.33	205	0.89
PTC20	0.08	15.10	223.85	239.02	3	342	70.00	210	1.63
PTC21	0.06	20.79	136.81	157.65	2	4172	953.50	1907	2.19
PTC22	0.03	9.84	27.52	37.37	5	1201	209.00	1045	1.15
PTC23	0.11	15.95	250.19	266.24	3	6383	281.67	845	7.55
PTC24	0.03	9.24	31.20	40.46	7	2173	201.30	1409	1.54
PTC25	0.74	7.79	89.80	98.33	10	1060	187.00	1870	0.57
PTC26	0.05	28.32	312.50	340.85	3	3768	382.33	1147	3.29
PTC27	0.03	217.56	194.08	411.64	13	3051	137.15	1783	1.71
PTC28	0.55	53.06	24.06	77.22	7	2802	120.00	840	3.34
PTC29	1.69	8.83	128.98	138.88	10	680	99.10	991	0.69
PTC30	0.23	5.07	14.94	20.29	5	23231	200.80	1004	23.14
Average	2.86	23.56	119.45	145.82	5	3196	239.66	948	3.56

set-up code; executing this set-up code takes a large portion of the execution time of the text case slice.

Statistical Analysis. We further analyzed the results reported in Table 2 using statistical and effect size tests. In particular, we used the Wilcoxon rank sum test [6] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [31]. Both tests are non-parametric; therefore, they do not make any assumption on the data distributions. We used the Wilcoxon test to assess whether the difference in running time between the original test cases and the corresponding slices are statistically significant. For the sake of the analysis, we considered the cumulative execution time for all slices obtained for each individual system test case. We considered a level of significance $\alpha = 0.05$.

According to the Wilcoxon tests, the slices generated by DS3 have a statistically significant lower execution time than the corresponding non-sliced test cases (p-value=0.01). The Vargha-Delaney's statistic reports a medium effect size $\hat{A}_{12} = 0.69$.

DS3 Overhead. As shown in the left side of table 2, DS3 takes, on average, 145.82 s to slice a complex system test case. The most time-consuming step is the refinement step, which recursively derives the hidden dependencies and performs backward slicing to guarantee that all related statements are included. This step takes, on average, 82% of the overall DS3 execution time. The second most expensive step is the generation of the initial set of slices using the static slicer; this step takes 16% of the overall DS3 execution time, on average. The remaining steps take, on average, only 2% of the total DS3 execution time.

It is worth noting that DS3 will be used just once to obtain the test case slices, so its overhead will be limited in any case. Further, using DS3 is particularly advantageous for those test cases that are executed many times a day, a common situation in continuous integration and deployment (CI/CD) environments, for example [27].

5.4 RQ3: Coverage and Fault Detection Capability

5.4.1 Methodology. To answer RQ3, we first compared the cumulative coverage of the original test cases and the test slices obtained through DS3. To measure code coverage, we used `Bullseye Coverage` [29], an advanced C++ code coverage tool used to improve software quality in critical system domains such as industrial control, medical, automotive, communications, aerospace, and defense. Next, we used mutation testing to assess the difference in fault detection capabilities between the original test cases and slices. Mutation testing is widely used in the literature to systematically assess the fault detection effectiveness of test cases [5, 8, 14], especially when not enough real-faults have been recorded, which is our case. Mutation testing introduces syntactic changes (mutations) into the production code using well-established mutation rules (mutation operators). The variants of the program produced by the mutation operators are often referred to as *mutants*. Effective test suites should pass on the original program but fail when executed against the *mutants*. In this scenario, the mutant is said to be *killed*; otherwise, the mutant is said to be *alive*.

For mutation analysis, we used `Mutate++` [18], an open-source mutation testing tool. `Mutate++` provides 15 mutations operators, including *arithmetic*, *conditional*, *Boolean*, *numeric*, and *line-deletion* operators. For our analysis, we selected six mutation operators based on the following observations. First, for some mutation operators, the majority of the generated mutants were killed at the build stage due to compilation errors; an example of such a mutation operator is the *line-deletion* operator, which removes a source code statement. Second, Lin et al. [17] reported that some mutation operators are sufficient in *selective mutation*. Selective mutation aims to reduce the number of mutants to consider without compromising the measurement of test effectiveness [11]. Based on the above observations, in our evaluation we considered the following mutation operators: (1) Logical Operator, (2) Conditional Operator, (3) Increment Decimal Operator, (4) Arithmetic Operator, (5) Boolean Literal Operator, (6) Decimal Number Operator.

To assess whether the slicing process of DS3 did not impact the fault detection capability of the test slices (group 1) compared to the original test cases (group 2), we compared the number of mutants killed by each group. We use K_O to denote the set of mutants killed by the original test cases and K_S to denote the set of mutants killed by the test slices generated by DS3.

We performed mutation testing and coverage analysis only on *JS-BSim*. We could not consider *Prop* for the following reasons: (1) the proprietary system includes a large number of software components, written with different programming languages. Analyzing such a heterogeneous codebase would require a powerful and sophisticated mutation testing tool. (2) We did not have access to the system source code. (3) Running mutation testing on *Prop* would have required to run the test cases hundreds of times, once for

Table 3: Comparison between original and sliced test cases in terms of code coverage

Subject	FunctionCov			BranchCov		
	Total	Org	Slices	Total	Org	Slices
JSBSim	2980	1831	1831	13541	5736	5698

Table 4: Comparison of the mutation testing results (K_O: the set of mutants killed by the original test cases, K_S: the set of mutants killed by the test slices produced by DS3)

Subject	ALL Mutants	K_O	K_S	$K_O \cap K_S$	$K_O \setminus K_S$
JSBSim	2678	289	288	288	1

each generated mutant. One execution for all *Prop* test cases requires ≈ 26 hours; hence, mutation testing for *Prop* would have been prohibitively expensive.

5.4.2 Results. Table 3 reports code coverage for the original test cases and the generated test slices. Columns "FunctionCov" and "BranchCov" indicate, respectively, function coverage and branch coverage scores; sub-columns "Total" indicate the total number of functions/branches in the system source code; sub-columns "Org" and "Slices" indicate, respectively, the number of functions/branches covered by the original test cases and by the slices.

In terms of function coverage, both the original test cases and the generated slices achieved 61% (1831/2980). We observe a very small difference in branch coverage: the original test cases cover 42.36% (5736/13541) of the code branches whereas the generated slices covered 42.08% (5698/13541) of them.

We manually analyzed the test slices to understand the root cause of such (minor) differences in branch coverage. We observed that a few original test cases contain spurious function calls whose execution results are neither asserted nor used as input for other method calls. As such, these function calls do not contribute to the test scenario under test. Since DS3 uses an assertion-based slicing criterion, it can successfully identify these spurious statements as they are not included in any of the generated slices.

Table 4 shows the mutation testing results. For the original system test cases, 289 mutants were killed. The generated test slices were able to kill 288 mutants which, as expected, were all killed by the original test cases. To better understand why this single mutant was not killed by the generated slices, we manually analyzed the original (non-sliced) test case that killed that mutant as well as the slices generated by DS3. We found out that the mutant was injected within a function that was invoked by the original test case but removed in the generated slices. The function call was removed by DS3 because it does not contribute to the test cases' assertions; it is not used to create inputs for other method calls, and it does not access global resources (i.e., it does not have hidden dependencies). Though the original test case kills the mutant, it is not due to an assertion failure but rather a run-time error when invoking the function call after the mutant is injected. This negligible difference shows that *DS3 does not negatively impact the effectiveness in terms of code coverage and fault detection capability.*

5.5 Threats to Validity

Threats to *construct validity*. We evaluated DS3 using different metrics, namely (1) number of generated test slices with no compilation or run-time error, (2) running time, (3) code coverage, and (4) killed mutants. These metrics are widely used in testing [19, 35]. To give a reasonable estimate of the test execution cost, we ran each test (both slices and original tests) 10 times and reported the average (arithmetic mean) results. To have a more reliable measure of the DS3 overhead, we also ran our approach 10 times.

Threats to *external validity*. We assessed DS3 in the context of a collaborative industrial research project with a large company in the aerospace domain. Hence, we reported the achieved results for one industrial, proprietary system. To improve the generalizability of our results and promote open science, we also included an open-source project, namely *JSBSim*, which implements a multi-platform, object-oriented Flight Dynamics Model written in C++.

6 PRACTICAL IMPLICATIONS

Test Smells and Maintainability. We argue that DS3 contributes to addressing two test smells: *assertion roulette* and *eager tests*. *Assertion roulette* is a test with multiple assertion statements, which make root cause analysis more difficult in case of test failure [30]. *Eager tests* check multiple different functionalities at once, negatively affecting test code readability and understandability [30]. Executing DS3 on a given system test case will yield multiple slices, each with fewer assertion statements and one individual test scenario. Notice that the slice minimization stage further contributes to reduce the overall number of slices. Thanks to DS3, the assertions of the original test case will be distributed over the generated slices, thus reducing the number of assertions per slice (assertion roulette). Furthermore, splitting the test cases into independent slices helps address eager tests.

Regression testing. Although DS3 has a non-negligible overhead (see Section 5.3), it is applied only once to generate the test slices. Therefore, its impact on the testing cost is limited. The advantage of using DS3 is that the generated slices are less expensive: this plays an important role in regression testing [35]. Reducing test execution time is one of the objectives of test case selection and prioritization. Using test slices, lightweight code analysis, and domain knowledge, test engineers can select and run the slices covering the changed or impacted portion of the production code, instead of the original test cases. In addition, test slices can be grouped by setup configuration, input values, or target functionalities. Slices can be further selected for each group to run only the most representative test slices and further reduce the overall test execution cost. Indeed, after reviewing the results on *Prop*, some of our industry partner's engineers stated that the sliced test cases were easier to understand and modify and of course faster to execute; they also found it interesting to extract useful information from the large number of logs they already have.

According to the results for RQ3, the test slices generated by DS3 have the same coverage and fault detection capability as the original test cases. Since the test slices are also statistically less expensive

to run (see RQ2 results), which positively affects regression testing, there is a clear gain in using DS3.

Two common objectives used in test case selection and prioritization are coverage (to maximize) and test execution (to minimize) [35]. With DS3, developers can select test slices that reach the same coverage and mutation scores as the original test cases but with a significantly lower execution time.

7 RELATED WORK

The approach presented in this paper is mainly related to work done in the areas of *test case decomposition*, *program slicing*, and *test refactoring*.

Test Case Decomposition. The closest approaches to DS3 are those based on *test carving* techniques [7, 15], which automatically carve unit test cases from system test cases. Such techniques consist of capturing, for a specific target unit method, the system states before (pre-state) and after (post-state) the invocation of the unit method during the execution of the system test case. From the pre-state, the unit method is replayed and the resulting state is queried to determine if there are differences with the recorded post-state. One of the main differences between these techniques and ours is that DS3 preserves the system level characteristics of the obtained (sliced) test cases (which can potentially entail a very complex usage of the various units), whereas carved tests target unit methods. Another difference is that carving techniques require to instrument the program code to capture pre- and post-states and to replay the unit method, whereas DS3 relies on program slicing and on information, regarding global resource usage, recorded in execution logs.

Xuan and Monperrus [33] propose *test case purification* for improving fault localization, by separating existing test cases into small fractions (called purified test cases). Similar to DS3, they use program slicing on assertions in an original test case to generate single-assertion test cases. Since they mostly target inexpensive, unit test cases (rather than expensive system test cases), dynamic slicing represents a viable solution for their approach. In contrast, DS3 deals with system test cases, relies on static slicing extended with log-based refinement to capture all dependencies in test case statements, and does not require the execution of test cases.

Bach et al. [2] propose to extract unit test cases from system test cases through the reverse execution (called time travel debugging) of a program flow for reconstructing object creation and modification statements from the source code. The approach also uses differential analysis to identify the test statements from which the unit test case will be extracted. Different from DS3, which requires only access to the test cases and to the execution logs, this approach requires access to the source code of the SUT.

Finally, Jorde et al. [12] investigated whether coarser granularity tests could be automatically generated by aggregating unit tests using Differential Unit Tests (DUT), initially developed for test carving by Elbaum et al. [7]. Such a strategy is, conceptually speaking, dual to test case decomposition.

Program Slicing. DS3 represents an enhancement of vanilla *static* slicing [32]; as discussed in section 3, the latter is most likely to miss hidden dependencies among statements, originated from the usage of global resources (e.g., external files) within the test case program. Compared to vanilla *static* slicing, DS3 requires one single execution of the test cases (from previous regression testing activities) to collect and parse the execution log files.

Dynamic slicing [16] is another form of enhancement of static slicing, which considers only specific executions of the program for a given objective (e.g., to perform debugging and root cause analysis). Dynamic slicing requires running test cases and accessing the source code for instrumentation and coverage analysis. Slices can be generated by analyzing the execution paths and the dependencies across the executed statements. However, this alternative is not feasible for a system composed of third-party components, and it does not handle "hidden" dependencies. Indeed, code coverage does not include information about which external resource has been accessed during the test execution.

Binkley et al. [4] presents a particular type of dynamic slicing technique, called *observation-based* slicing, which aims to slice programs independently from the programming language used. The key idea is not to purely rely on dependency analysis, but rather to use iterative observations on the validity of obtained slices (i.e., to execute the obtained slice to make sure it runs without compilation or run-time errors). Although this technique can be used for multi-language systems that include (3rd-party) binary components, it requires a large number of executions of the test case under analysis to iteratively slice candidates and check their validity. This requirement makes observation-based slicing impractical for complex system test cases (especially those in the cyber-physical system domain [1, 9]), whose execution is time-consuming. For this type of systems, DS3 is preferable as it does not require additional test case executions for assessing the validity of the generated slices.

Test Refactoring. A side benefit of the application of DS3 is the removal of some test code smells [21]. In this sense, DS3 is related to approaches for (test) code smells refactoring [20, 30], which introduced catalogues of test smells together with (manual) refactoring operations to address them. Based on existing catalogues, there have been proposals [23, 25] to automatically detect (rather than refactor) test smells. These approaches rely on detection rules that raise warnings when some metrics (e.g., size, number of method calls, number of assertions) in the test code exceed given thresholds. However, a recent study [24] showed that detection rules are far less accurate than previously reported, especially when tests use external resources.

Although the primary goal of DS3 is neither to detect nor to refactor test smells, by slicing test cases into separate sliced test cases with fewer assertions, DS3 addresses the *eager tests* and *assertion roulette* test smells. Furthermore, eager tests are identified through static slicing and log analysis, instead of using detection rules, as proposed in the literature.

8 CONCLUSION

In this paper, we addressed the problem of dealing with complex system test cases containing multiple test scenarios, which negatively impact both regression testing and test evolution. We proposed DS3, a novel approach to decompose a complex system test case with multiple test scenarios into separate sliced test cases, each of them running one test scenario. DS3 leverages static slicing and

the execution logs collected during past regression testing sessions. The main idea is to use logs containing run-time information about the SUT to identify dependencies between test statements due to the access and usage of global resources; these dependencies are used to refine sliced test cases generated by static slicing, which tend to miss such dependencies. The evaluation results, conducted on one proprietary system and one open-source system, show that log-based slice refinement is indeed effective at avoiding, in the generated sliced test cases, compilation or run-time errors due to missing dependencies. Furthermore, the generated test case slices are, on average, 3.56 times faster than the original system test case, with no significant loss in fault detection capability.

As a part of future work, we plan to assess the impact of using DS3 on the cost-effectiveness of regression testing activities, such as test case prioritization. We also plan to extend DS3 to support different programming languages, and to evaluate it on additional benchmarks. Since the idea of log-based slicing is not only applicable to test cases but also to program source code in general, we will further investigate possible applications beyond regression testing.

ACKNOWLEDGMENTS

This work has received funding from the Luxembourg National Research Fund (FNR) under grant agreement No C-PPP17/IS/11602677 and from NSERC of Canada under the Discovery and CRC programs.

REFERENCES

[1] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2018. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE)*. IEEE, New York, NY, USA, 143–154. https://doi.org/10.1145/3238147.3238192

[2] Thomas Bach, Ralf Pannemans, Johannes Haeussler, and Artur Andrzejak. 2019. Dynamic unit test extraction via time travel debugging for test cost reduction. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. ACM, New York, NY, USA, 238–239. https://doi.org/10.1109/ICSE-Companion.2019.00093

[3] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? An empirical study. *Empirical Software Engineering* 20, 4 (2015), 1052–1094.

[4] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery, 109–120. https://doi.org/10.1145/2635868.2635893

[5] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. IEEE, New York, NY, USA, 597–608. https://doi.org/10.1109/ICSE.2017.61

[6] W. J. Conover. 1998. *Practical Nonparametric Statistics* (3rd edition ed.). Wiley, New York, NY, USA.

[7] Sebastian Elbaum, Hui Nee Chin, Matthew B Dwyer, and Jonathan Dokulil. 2006. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. Association for Computing Machinery, New York, NY, USA, 253–264. https://doi.org/10.1145/1181775.1181806

[8] R. Gopinath, C. Jensen, and A. Groce. 2014. Mutations: How Close are they to Real Faults?. In *Proceeding of the 25th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Los Alamitos, CA, USA, 189–200. https://doi.org/10.1109/ISSRE.2014.40

[9] Ines Hajri, Arda Goknil, Fabrizio Pastore, and Lionel C Briand. 2020. Automating system test case classification and prioritization for use case-driven testing in product lines. *Empirical Software Engineering* 25, 5 (2020), 3711–3769. https://doi.org/10.1007/s10664-020-09853-4

[10] P. He, J. Zhu, Z. Zheng, and M. R. Lyu. 2017. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *Proceedings of the International Conference on Web Services (ICWS)*. IEEE Press, Piscataway, NJ, USA, 33–40. https://doi.org/10.1109/ICWS.2017.13

[11] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678. https://doi.org/10.1109/TSE.2010.62

[12] M. Jorde, S. Elbaum, and M. B. Dwyer. 2008. Increasing Test Granularity by Aggregating Unit Tests. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*. ACM, New York, NY, USA, 9–18. https://doi.org/10.1109/ASE.2008.11

[13] JSBSim Team [n.d.]. *JSBSim*. Retrieved January 28, 2021 from https://github.com/JSBSim-Team/jsbsim

[14] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929

[15] Alexander Kampmann and Andreas Zeller. 2019. Carving parameterized unit tests. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. ACM, New York, NY, USA, 248–249. https://doi.org/10.1109/ICSE-Companion.2019.00098

[16] Bogdan Korel and Janusz Laski. 1988. Dynamic program slicing. *Information processing letters* 29, 3 (1988), 155–163. https://doi.org/10.1145/93548.93576

[17] Huan Lin, Yawen Wang, and Yunzhan Gong. 2018. Subsuming Mutation Operators. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. ACM, New York, NY, USA, 236–237.

[18] Niels Lohmann. [n.d.]. *Mutate++*. Retrieved January 28, 2021 from https://github.com/nlohmann/mutate_cpp

[19] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 643–653. https://doi.org/10.1145/2635868.2635920

[20] Rogério Marinke, Eduardo Martins Guerra, Fábio Fagundes Silveira, Rafael Monico Azevedo, Wagner Nascimento, Rodrigo Simões de Almeida, Bruno Rodrigues Demboscki, and Tiago Silva da Silva. 2019. Towards an Extensible Architecture for Refactoring Test Code. In *Proceedings of the International Conference on Computational Science and Its Applications*. Springer, Cham, Switzerland, 456–471. https://doi.org/10.1007/978-3-030-24305-0_34

[21] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on software engineering* 30, 2 (2004), 126–139. https://doi.org/10.1109/TSE.2004.1265817

[22] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, and R. Sasnauskas. 2018. A Search-Based Approach for Accurate Identification of Log Message Formats. In *Proceedings of the 26th International Conference on Program Comprehension (ICPC)*. IEEE Press, Piscataway, NJ, USA, 167–177. https://doi.org/10.1145/3196321.3196340

[23] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. 2018. Automatic test smell detection using information retrieval techniques. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. Association for Computing Machinery, New York, NY, USA, 311–322. https://doi.org/10.1109/ICSME.2018.00040

[24] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. 2020. Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities. In *Proceeding of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Los Alamitos, CA, USA, 523–533. https://doi.org/10.1109/ICSME46990.2020.00056

[25] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2019. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., USA, 193–202.

[26] ppa [n.d.]. *Microsoft python-program-analysis library*. Retrieved January 28, 2021 from https://github.com/microsoft/python-program-analysis

[27] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access* 5 (2017), 3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629

[28] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the relation of test smells to software code quality. In *Proceedings of the International Conference on Software Maintenance and Evolution. (ICSME)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1109/ICSME.2018.00010

[29] Bullseye Testing Technology. [n.d.]. *BullseyeCoverage*. Retrieved January 28, 2021 from https://www.bullseye.com

[30] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP)*. ACM, New York, NY, USA, 92–95.

[31] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. https://doi.org/10.3102/10769986025002101

[32] M. Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

[33] Jifeng Xuan and Martin Monperrus. 2014. Test Case Purification for Improving Fault Localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 52–63. https://doi.org/10.1145/2635868.2635906

[34] Shin Yoo, David W. Binkley, and Roger D. Eastman. 2017. Observational slicing based on visual semantics. *J. Syst. Softw.* 129 (2017), 60–78. https://doi.org/10.1016/j.jss.2016.04.009

[35] S. Yoo and M. Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stvr.430