Unit Tests of Scientific Software: A Study on SWMM

Zedong Peng, Xuanyi Lin, and Nan Niu(✉)

Department of Electrical Engineering and Computer Science (EECS),
University of Cincinnati, Cincinnati, OH 45221, USA
{pengzd,linx7}@mail.uc.edu, nan.niu@uc.edu

Abstract. Testing helps assure software quality by executing program and uncovering bugs. Scientific software developers often find it challenging to carry out systematic and automated testing due to reasons like inherent model uncertainties and complex floating point computations. We report in this paper a manual analysis of the unit tests written by the developers of the Storm Water Management Model (SWMM). The results show that the 1,458 SWMM tests have a 54.0% code coverage and a 82.4% user manual coverage. We also observe a "getter-setter-getter" testing pattern from the SWMM unit tests. Based on these results, we offer insights to improve test development and coverage.

Keywords: Scientific software · Unit testing · Test oracle · User manual · Test coverage · Storm Water Management Model (SWMM)

1 Introduction

Scientific software is commonly developed by scientists and engineers to better understand or make predictions about real world phenomena. Without such software, it would be difficult or impossible for many researchers to do their work. Scientific software includes both software for end-user researchers (e.g., climate scientists and hydrologists) and software that provides infrastructure support (e.g., message passing and scheduling). Because scientific software needs to produce trustworthy results and function properly in mission-critical situations, rigorous software engineering practices shall be adopted to assure software qualities.

Testing, which is important for assessing software qualities, has been employed extensively in business/IT software. However, developers of scientific software have found it more difficult to apply some of the traditional software testing techniques [14]. One chief challenge is the lack of the test oracle. An *oracle* in software testing refers to the mechanism for checking whether the program under test produces the expected output when executed using a set of test cases [2]. Many testing techniques—especially unit testing commonly carried out in business/IT software development projects—require a suitable oracle to set

V. V. Krzhizhanovskaya et al. (Eds.): ICCS 2020, LNCS 12143, pp. 413–427, 2020.
https://doi.org/10.1007/978-3-030-50436-6_30

up the expectation with which the actual implementation (e.g., sorting inventory items or calculating tax returns) can be compared.

Researchers have therefore proposed different approaches to overcoming the lack of oracle in scientific software testing. For example, a pseudo oracle—an independently developed program that fulfills the same specification as the program under test—has been used in numerical simulation and climate model testing [10,11]. A pseudo oracle makes the assumption that independently developed reference models will not result in the same failures; however, Brilliant *et al.* [5] reported instances of N-version programming that violated this assumption. Mayer [22] tested image processing applications with statistical oracles by checking the statistical characteristics of test results; yet a statistical oracle cannot decide whether a single test case has passed or failed.

Single test cases are commonly used in unit testing to verify individual program modules, each of which encapsulates some coherent computation (e.g., a procedure, a function, or a method). Although pseudo and statistical oracles are proposed in the research literature, their adoptions seem isolated among scientific software developers. Our ongoing collaborations with the U.S. Environmental Protection Agency's Storm Water Management Model (SWMM) team suggests limited applicability of N-version programming, and hence pseudo oracle especially at the unit testing levels, due to the constrained software development resources. More importantly, the SWMM team has been developing tests, unit tests and other kinds, throughout the project's more than four decades history [32]. To comply with the recent movements toward improving public access to data [31], these tests are released, sometimes together with the source code of SWMM, in GitHub and other repositories. However, little is known about the characteristics of the SWMM tests.

To shorten the knowledge gap, we report in this paper the tests that are publicly available for the SWMM software. We provide a detailed look at who wrote how many tests in what environments, and further analyze the coverage of the unit tests from two angles: how much they correspond to the user manual and to the codebase. The contributions of our work lie in the qualitative characterization and quantitative examination of the tests written and released by the scientific software developers themselves in the context of SWMM. Our results clearly show that oracle *does* exist in scientific software testing, and our coverage analysis reveals concrete ways to improve testing. In what follows, we provide background information and introduce SWMM in Sect. 2. Section 3 presents our search of SWMM tests, Sect. 4 analyzes the test coverage, and finally, Sect. 5 draws some concluding remarks and outlines future work.

2 Background

2.1 Oracle Problem in Testing Scientific Software

Testing is a mainstream approach toward software quality, and involves examining the behavior of a system in order to discover potential faults. Given an input for the system under test, the *oracle problem* refers to the challenge of distinguishing the corresponding desired, correct behavior from observed, potentially incorrect behavior [2]. The oracle of desired and correct behavior of scientific software, however, can be difficult to obtain or may not be readily available. Kanewala and Bieman [14] listed five reasons.

- Some scientific software is written to find answers that are previously unknown; a case in point is the program computing a large graph's shortest path of any arbitrary pair of nodes.
- It is difficult to determine the correct output for software written to test scientific theory that involves complex calculations, e.g., the large, complex simulations are developed to understand climate change [10].
- Due to the inherent uncertainties in models, some scientific programs do not give a single correct answer for a given set of inputs.
- Requirements are unclear or uncertain up-front due to the exploratory nature of the software [15,24].
- Choosing suitable tolerances for an oracle when testing numerical programs is difficult due to the involvement of complex floating point computations.

Barr *et al.* [2] showed that test oracles could be explicitly specified or implicitly derived. In scientific software testing, an emerging technique to alleviate the oracle problem is metamorphic testing [14,30]. For example, Ding and colleagues [8] tested an open-source light scattering simulation performing discrete dipole approximation. Rather than testing the software on each and every input of a diffraction image, Ding *et al.* systematically (or metamorphically) changed the input (e.g., changing the image orientation) and then compared whether the software would meet the expected relation (e.g., scatter and textual pattern should stay the same at any orientation).

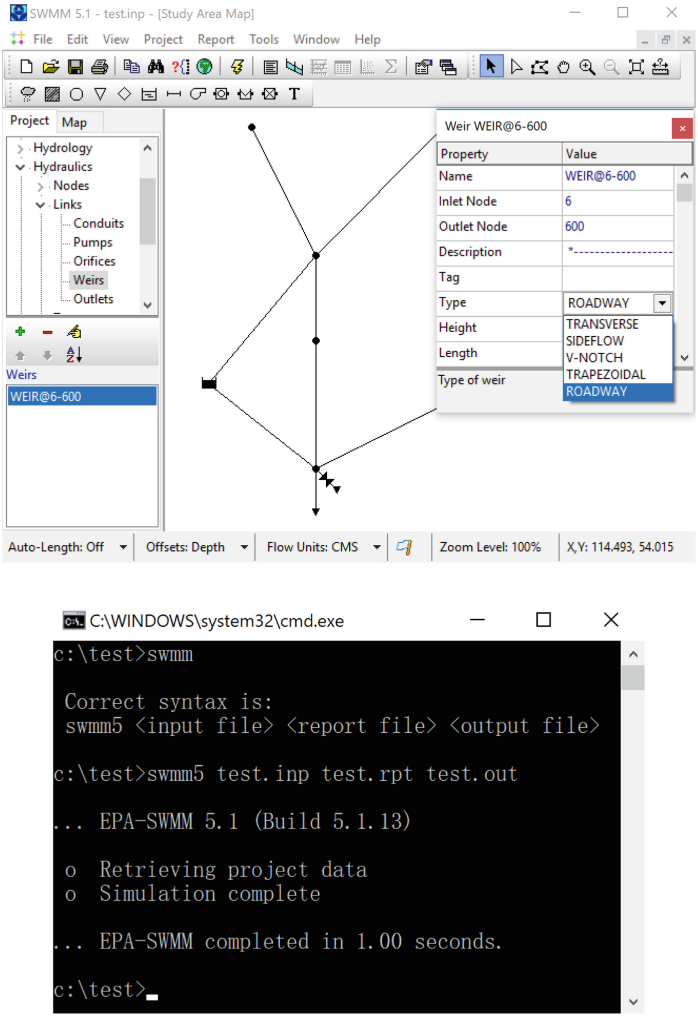
While we proposed hierarchical and exploratory ways of conducting metamorphic testing for scientific software [17,18], our work is similar to that of Ding *et al.*'s [8] by gearing toward the entire application instead of checking the software at the unit testing level. This is surprising given that one of the most prevalent stereotypes of metamorphic testing [30] is the trigonometry function, i.e., $\sin(x) = \sin(\pi - x)$, which is targeted at an individual computational unit. Unit tests are especially useful for guarding the developers against programming mistakes and for localizing the errors when they occur. Thus, we are interested in the unit tests written and released by the scientific software developers themselves, and for our current work, the focus is on SWMM.

2.2 Storm Water Management Model (SWMM)

The Storm Water Management Model (SWMM) [32], created by the U.S. Environmental Protection Agency (EPA), is a dynamic rainfall-runoff simulation model that computes runoff quantity and quality from primarily urban areas. The development of SWMM began in 1971 and since then the software has undergone several major upgrades.

Table 1. SWMM tests in six repositories.

Source	Author (#; Role)	# of Tests	Type	Method	Language	SWMM Version
https://github.com/michaeltryby/swmm-nrtests/tree/master/public/update_v5111	(N/A; N/A)	8	Numerical regression testing	numpy.allclose	Python, json	5.1.11
https://github.com/OpenWaterAnalytics/Stormwater-Management-Model/tree/feature-2dflood/tests/swmm-nrtestsuite/benchmark/swmm-5112	(N/A; N/A)	27				5.1.12
https://github.com/USEPA/Stormwater-Management-Model/tree/develop/tools/nrtest-swmm/nrtest_swmm	(1; EPA developer)	2				5.1.12
https://drive.google.com/drive/folders/16gImGSJV7iygX37P-XiRS4WcBK–YIzvT	(1; EPA developer)	58				5.1.13
https://github.com/michaeltryby/swmm-nrtests/tree/master/public	(1; EPA developer)	52				5.1.13
https://github.com/OpenWaterAnalytics/Stormwater-Management-Model/tree/develop/tests	(3+; EPA developer)	1,458	Unit testing	boost test	C++	5.1.13



Property	Value
Name	WEIR@6-600
Inlet Node	6
Outlet Node	600
Description	*------------------
Tag	
Type	ROADWAY
Height	
Length	

Auto-Length: Off | Offsets: Depth | Flow Units: CMS | Zoom Level: 100% | X,Y: 114.493, 54.015

```
c:\test>swmm

 Correct syntax is:
 swmm5 <input file> <report file> <output file>

c:\test>swmm5 test.inp test.rpt test.out

... EPA-SWMM 5.1 (Build 5.1.13)

 o  Retrieving project data
 o  Simulation complete

... EPA-SWMM completed in 1.00 seconds.

c:\test>_
```

Fig. 1. SWMM running as a Windows application (top) and the computational engine of SWMM running as a console application (bottom).

The most current implementation of the model is version 5.1.13 which was released in 2018. It has modernized both the model's structure and its user interface (UI). The top of Fig. 1 shows a screenshot of SWMM running as a Windows application. The two main parts of SWMM are the computational engine written in C/C++ with about 45,500 lines of code, and the UI written using Embarcadero's Delphi.XE2. Note that the computational engine can be compiled either as a DLL under Windows or as a stand-alone console application under both Windows and Linux. The bottom of Fig. 1 shows that running SWMM in the command line takes three parameters: the input, report, and output files.

The users of SWMM include hydrologists, engineers, and water resources management specialists who are interested in the planning, analysis, and design related to storm water runoff, combined and sanitary sewers, and other drainage systems in urban areas. Thousands of studies worldwide have been carried out by using SWMM, such as predicting the combined sewer overflow in the Metropolitan Sewer District of Greater Cincinnati [12], modeling the hydrologic performance of green roofs in Wrocław, Poland [6], and simulating a combined drainage network located in the center of Athens, Greece [16].

Despite the global adoptions of SWMM, the software development and maintenance remain largely local to the EPA. Our collaborations with the SWMM team involve the creation of a connector allowing for the automated parameter calibration [13], and through developing this software solution, we recognize the importance of testing in assuring quality and contribute hierarchical and exploratory methods of metamorphic testing [17,18]. In addition, we release our metamorphic tests in the connector's GitHub repository [19], promoting the open access to data and research results [31]. For similar purposes, we realize that the SWMM team has released their own tests in publicly accessible repositories. Understanding these tests is precisely the objective of our study.

3 Identification and Characterization of SWMM Tests

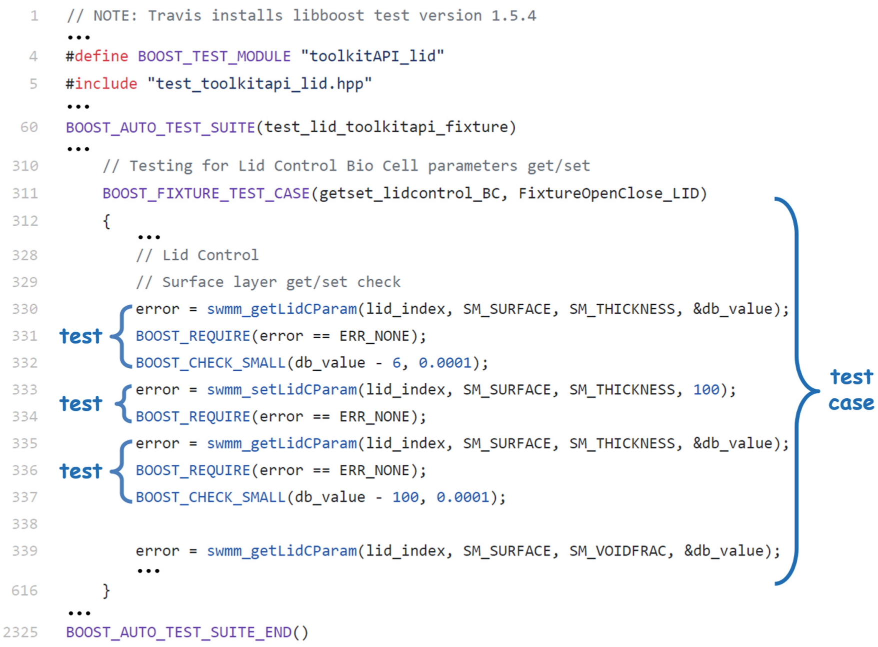
We performed a survey analysis of the SWMM tests released in publicly accessible repositories. Our search was informed by the SWMM team members and also involved using known test repositories to find additional ones (snowballing). Table 1 lists the six repositories that we identified, as well as the characteristics of the testing data. The "source" column shows that five repositories are based on GitHub, indicating the adoption of this kind of social coding environments among scientific software developers; however, one source is hosted on Google Drive by an EPA developer, showing that not all tests are embedded or merged in the (GitHub) code branches. Although we cannot claim the completeness of the sources, it is clear that searching only the code repositories like GitHub will result in only partial testing data.

Table 1 shows that three test sets are contributed by individual developers whereas one test set is jointly developed by more than three people. For the other two test sets, we are not certain about the number of authors and their roles, though other GitHub pages may provide inferrable information. The tests that we found can be classified in two categories: numerical regression testing and unit testing. Table 1 also shows that EPA developers adopt Python's numpy.allclose() function to write regression tests. The numpy.allclose() function is used to find if two arrays are element-wise equal within a tolerance, and for SWMM, this type of "allclose" checks whether the output from the newly released code is consistent with that from the previously working code. For regression testing, we count each SWMM input as a test, i.e., a single unit for different code versions to check "allclose". In total, there are 147 regression tests in five repositories.

In contrast, unit testing does not compare different versions of SWMM but focuses on the specific computations of the software. One source of Table 1 contains 1,458 tests written by a group of EPA developers by using the boost environment [7]. In particular, libboost test (version 1.5.4) is used in SWMM, and Boost.Test provides both the interfaces for writing and organizing tests and the controls of their executions. Figure 2 uses a snippet of test_toolkitapi_lid.cpp to explain the three different granularities of SWMM unit tests. At the fine-grained level are the assertions, e.g., line #334 of Fig. 2 asserts "error == ERR_NONE". The value of "error" is obtained from line #333. As shown in Fig. 2, we define a *test* in our study to be one instance that triggers SWMM execution and the associated assertions with that triggering. In Fig. 2, three tests are shown. A group of tests forms a *test case*, e.g., lines #311–616 encapsulate many tests into one BOOST_FIXTURE_TEST_CASE. Finally, each file corresponds to a *test suite* containing one or more test cases. Table 2 lists the seven test suites, and the number of test cases and tests per suite. Averagely speaking, each test suite has 8.7 test cases, and each test case has 23.9 tests.

4 Coverage of SWMM Unit Tests

Having characterized who developed how many SWMM tests in what environments, we turn our attention to the unit tests for quantitative analysis. Our rationales are threefold: (1) a large proportion ($\frac{1,458}{1,605} = 91\%$) of the tests that



```
1     // NOTE: Travis installs libboost test version 1.5.4
      ...
4     #define BOOST_TEST_MODULE "toolkitAPI_lid"
5     #include "test_toolkitapi_lid.hpp"
      ...
60    BOOST_AUTO_TEST_SUITE(test_lid_toolkitapi_fixture)
      ...
310       // Testing for Lid Control Bio Cell parameters get/set
311       BOOST_FIXTURE_TEST_CASE(getset_lidcontrol_BC, FixtureOpenClose_LID)
312       {
              ...
328           // Lid Control
329           // Surface layer get/set check
330           error = swmm_getLidCParam(lid_index, SM_SURFACE, SM_THICKNESS, &db_value);
331           BOOST_REQUIRE(error == ERR_NONE);
332           BOOST_CHECK_SMALL(db_value - 6, 0.0001);
333           error = swmm_setLidCParam(lid_index, SM_SURFACE, SM_THICKNESS, 100);
334           BOOST_REQUIRE(error == ERR_NONE);
335           error = swmm_getLidCParam(lid_index, SM_SURFACE, SM_THICKNESS, &db_value);
336           BOOST_REQUIRE(error == ERR_NONE);
337           BOOST_CHECK_SMALL(db_value - 100, 0.0001);
338
339           error = swmm_getLidCParam(lid_index, SM_SURFACE, SM_VOIDFRAC, &db_value);
              ...
616       }
      ...
2325  BOOST_AUTO_TEST_SUITE_END()
```

Fig. 2. Illustration of SWMM tests and test cases written in the boost environment.

we found are unit tests (2) the tests are intended for the most recent release of SWMM (version 5.1.13), and (3) unit testing requires oracle to be specified which will provide valuable insights into how scientific software developers themselves define test oracles.

Table 2. Test suites, test cases, and tests.

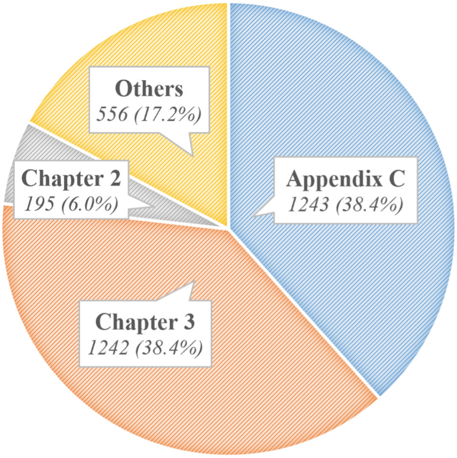
Test suite	Test case	Test
test_output.cpp	14	59
test_swmm.cpp	11	11
test_toolkitapi.cpp	12	128
test_toolkitapi_gage.cpp	1	11
test_toolkitapi_lid.cpp	17	679
test_toolkitapi_lid_results.cpp	5	555
test_toolkitapi_pollut.cpp	1	15
Σ	61	1,458

When unit tests are considered, *coverage* is an important criterion. This is because a program with high test coverage, measured as a percentage, has had more of its source code executed during testing, which suggests it has a lower chance of containing undetected software bugs compared to a program with low test coverage [4]. Practices that lead to higher testing coverage have therefore received much attention. For example, test-driven development (TDD) [23] advocates test-first over the traditional test-last approach, and the studies by Bhat and Nagappan [3] show that the block coverage reached to 79–88% at unit test level in projects employing TDD. While Bhat and Nagappan's studies were carried out at Microsoft, some scientific software demands even higher levels of test coverage. Notably, the European Cooperation for Space Standardization requires a 100% test coverage at software unit level, and Prause *et al.* [27] collected experience from a space software project's developers who stated that 100% coverage is unusual and brings in new risks. Nevertheless, the space software developers acknowledged that 100% coverage is sometimes necessary. Our work analyzes the coverage of SWMM unit tests not only from the source code perspective, but also from the viewpoint of the user manual. Compared to business/IT software, scientific software tends to release authoritative and updated user manual intended for the software system's proper operation. The rest of this section reports the 1,458 unit tests' coverage and discusses our study's limitations.

4.1 SWMM User Manual Coverage

We manually mapped the SWMM unit tests to its version 5.1 user manual [29], and for validation and replication purposes, we share all our analysis data in the institutional digital preservation site Scholar@UC [26]. The 353-page user manual contains 12 chapters and 5 appendices. Our analysis shows that 14, or 82.4%

($\frac{14}{17}$), are covered by at least one of the 1,458 unit tests. Figure 3 shows the distributions of the unit tests over the 14 user manual chapters/appendices. Because one unit test may correspond to many chapters/appendices, the test total of Fig. 3 is 3,236. The uncovered chapters are: "Printing and Copying" (Chapter 10), "Using Add-In Tools" (Chapter 12), and "Error and Warning Messages" (Appendix E). The error and warning messages are descriptive in nature, and printing, copying, and add-in tools require the devices and/or services external to SWMM. Due to these reasons, it is understandable that no unit tests correspond to these chapters/appendices.



(a)

Others	# of Tests
Chapter 5	84 (2.6%)
Chapter 8	84 (2.6%)
Chapter 7	76 (2.3%)
Chapter 11	71 (2.2%)
Chapter 6	69 (2.1%)
Chapter 4	52 (1.6%)
Appendix B	52 (1.6%)
Chapter 9	31 (1.0%)
Appendix D	23 (0.7%)
Chapter 1	8 (0.2%)
Appendix A	6 (0.2%)

(b)

Fig. 3. **(a)** Mapping unit tests to user manual chapters/appendices, and **(b)** Explaining the "Others" part of **(a)**.

Figure 3(a) shows that the unit tests predominantly cover "SWMM's Conceptual Model" (Chapter 3) and "Specialized Property Editors" (Appendix C). The same percentage, 38.4%, of these two parts is not accidental to us. In fact, they share the same subset of the unit tests except for one. We present a detailed look at these parts in Fig. 4. Chapter 3 describes not only the configuration of the SWMM objects (e.g., conduits, pumps, storage units, etc.) but also the LID (low impact development) controls that SWMM allows engineers and planners to represent combinations of green infrastructure practices and to determine their effectiveness in managing runoff. The units presented in §3.2 ("Visual Objects"), §3.3 ("Non-Visual Objects"), and §3.4 ("Computational Methods") thus represent some of the core computations of SWMM. Consequently, unit tests are written for the computations except for the "Introduction" (§3.1) overviewing the Atmosphere, Land Surface, Groundwater, and Transport compartments of SWMM. Surprisingly, more tests are written for the non-visual objects than

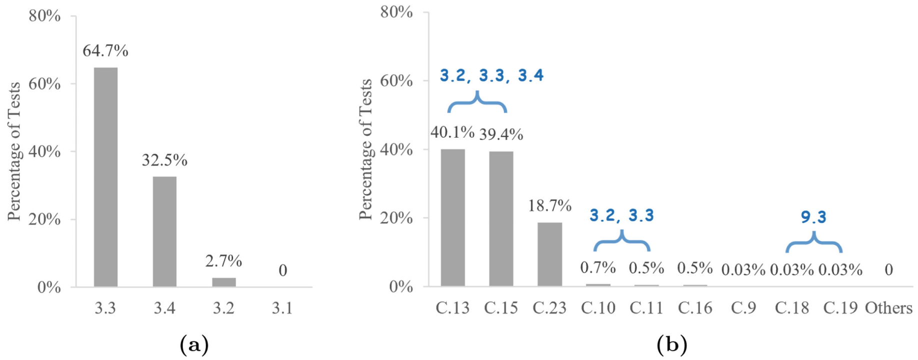


Fig. 4. (a) Breakdowns of unit tests into Chapter 3 ("SWMM's Conceptual Model") of the user manual, and **(b)** Breakdowns of unit tests into Appendix C ("Specialized Property Editors") of the user manual.

the visual objects, as shown in Fig. 4(a). The visual objects (rain gages, subcatchments, junction nodes, outfall nodes, etc.) are those that can be arranged together to represent a stormwater drainage system, whereas non-visual objects (climatology, transects, pollutants, control rules, etc.) are used to describe additional characteristics and processes within a study area. One reason might be the physical, visual objects (§3.2) are typically combined, making unit tests (e.g., single tests per visual object) difficult to construct.

The non-visual objects (§3.3), on the other hand, express attributes of, or the rules controlling, the physical objects, which makes unit tests easier to construct. For example, two of the multiple-condition orifice gate controls are RULE R2A: "IF NODE 23 DEPTH > 12 AND LINK 165 FLOW > 100 THEN ORIFICE R55 SETTING = 0.5" and RULE R2B: "IF NODE 23 DEPTH > 12 AND LINK 165 FLOW > 200 THEN ORIFICE R55 SETTING = 1.0". For units like RULE R2A and RULE R2B, tests could be written to check whether the orifice setting is correct under different node and link configurations. Under these circumstances, the test oracles are *known* and are given in the user manual (e.g., orifice setting specified in the control rules).

During our manual mappings of the unit tests, we realize the interconnection of the user manual chapters/appendices. One example mentioned earlier is the connection between Chapter 3 and Appendix C. It turns out that such interconnections are not one-to-one, i.e., Appendix C connects to not only Chapter 3 but also to other chapters. In Fig. 4(b), we annotate the interconnections grounded in the SWMM unit tests. For instance, §3.2, §3.3, and §3.4 are linked to §C.10 ("Initial Buildup Editor"), §C.11 ("Land Use Editor"), §C.13 ("LID Control Editor"), and §C.15 ("LID Usage Editor"), indicating the important role of LID plays in SWMM. Although only a very small number of unit tests connects §9.3 ("Time Series Results") with §C.18 ("Time Pattern Editor") and §C.19 ("Time Series Editor"), we posit more tests of this core time-series computation could be developed in a similar way as LID tests (e.g., by using the boost environment illustrated in Fig. 2). A more general speculation that we draw from our analysis is that if some core computation has weak links with the scientific software system's parameters and properties (e.g., Appendix C of the SWMM user manual), then developing unit tests for that computation may require other environments and frameworks like CppTest or CPPUnit; investigating these hypotheses is part of our future research collaborations with the EPA's SWMM team.

4.2 SWMM Codebase Coverage

There are a number of coverage measures commonly used for test-codebase analysis, e.g., Prause *et al.* [27] compared statement coverage to branch coverage in a space software project and showed that branch coverage tended to be lower if not monitored but could be improved in conjunction with statement coverage without much additional effort. For our analysis, we manually mapped the 1,458 unit tests to SWMM's computational engine (about 45,500 lines of code written in C/C++). Like the test-to-user-manual data, we also share our test-to-codebase analysis data in Scholar@UC [26]. At the code file level, the coverage of the 1,458 SWMM unit tests is 54.0%. In line with our user manual analysis results, the code corresponding to the greatest number of unit tests involves runoff and LID, including toposort.c, treatment.c, and runoff.c.

Different from our user manual analysis where we speculated that control rules, such as RULE R2A and RULE R2B, would be among the subjects of unit testing, the actual tests have a strong tendency toward getters and setters. This is illustrated in Fig. 2. Interestingly, we also observe a pattern of "getter-setter-getter" in the tests. In Fig. 2, the test of lines #330–332 first gets swmm_getLidCParam, ensures that there is no error in getting the parameter value (line #331), and compares the value with the oracle (line #332). A minor change is made in the next test where the new "&db_value" is set to be 100, followed by checking if this instance of parameter setter is successful (line #334). The last test in the "getter-setter-getter" sequence immediately gets and checks the parameter value (lines #335–337). Our analysis confirms many instances of this "getter-setter-getter" pattern among the 1,458 unit tests.

It is clear that oracle exists in SWMM unit tests, and as far as the "getter-setter-getter" testing pattern is concerned, two kinds of oracle apply: whether the code crashes (e.g., lines #331, #334, and #336 of Fig. 2) and if the parameter value is close to pre-defined or pre-set value (e.g., lines #332 and #337 of Fig. 2). One advantage of "getter-setter-getter" testing lies in the redundancy of setting a value followed immediately by getting and checking that value, e.g., swmm_setLidCParam with 100 and then instantly checking swmm_getLidCParam against 100. As redundancy improves reliability, this practice also helps with the follow-up getter's test automation. However, a disadvantage here is the selection of the parameter values. In Fig. 2, for example, the oracle of 6 (line #332) may be drawn from SWMM input and/or observational data, but the selection of 100 seems random to us. As a result, the test coverage is low from the parameter value selection perspective, which can limit the bug detection power of the tests.

A post from the SWMM user forum [1] provides a concrete situation of software failure related to specific parameter values. In this post, the user reported that: “The surface depth never even reaches 300 mm in the LID report file” after explicitly setting the parameters of the LID unit (specifically, “storage depth of surface layer” = 300 mm) to achieve the effect [1]. The reply from an EPA developer suggested a solution by changing: “either the infiltration conductivity or the permeability of the Surface, Soil or Pavement layers”. Although these layers are part of “LID Controls”, and even have their descriptions in §3.3.14 of the SWMM user manual [29], the testing coverage does not seem to reach “storage depth of surface layer” = 300 mm under different value combinations of the Surface, Soil or Pavement layers. We believe the test coverage can be improved when the design of unit tests builds more directly upon the SWMM user manual and when the parameter value selection alters more automatically than currently written in a relatively fixed fashion.

4.3 Threats to Validity

We discuss some of the important aspects of our study that one shall take into account when interpreting our findings. A threat to construct validity is how we define tests. Our work focuses on SWMM unit tests written in the boost environment, as illustrated in Fig. 2. While our units of analysis—tests, test cases, and test suites—are consistent with what boost defines and how the test developers apply boost, the core construct of “tests” may differ if boost evolves or the SWMM developers adopt other test development environments. Within boost itself, for instance, BOOST_AUTO_TEST_CASE may require different ways to define and count tests than BOOST_FIXTURE_TEST_CASE shown in Fig. 2.

An internal validity threat is our manual mapping of the 1,458 SWMM unit tests to the user manual and to the codebase. Due to the lack of traceability information [25,34,35] from the SWMM project, our manual effort is necessary in order to understand the coverage of the unit tests. Our current mapping strategy is driven mainly by keywords, i.e., we matched keywords from the tests with the user manual contents and with the functionalities implemented in the code (procedure signatures and header comments). Two researchers independently performed the manual mappings of a randomly chosen 200 tests and achieved a high level of inter-rater agreement (Cohen’s $\kappa = 0.77$). We attributed this to the comprehensive documentation of SWMM tests, user manual, and code. The disagreements of the researchers were resolved in a joint meeting, and three researchers performed the mappings for the remaining tests.

Several factors affect our study’s external validity. Our results may not generalize to other kinds of SWMM testing (numerical regression tests in particular), to the tests shared internally among the SWMM developers, and to other scientific software with different size, complexity, purposes, and testing practices. As for conclusion validity and reliability, we believe we would obtain the same results if we repeated the study. In fact, we publish all our analysis data in our institution’s digital preservation repository [26] to facilitate reproducibility, cross validation, and future expansions.

5 Conclusions

Testing is one of the cornerstones of modern software engineering [9]. Scientific software developers, however, face the oracle challenge when performing testing [14]. In this paper, we report our analysis of the unit tests written and released by the EPA's SWMM developers. For the 1,458 SWMM unit tests that we identified, the file-level code coverage is 54.0% and the user-manual coverage is 82.4%. Our results show that oracle *does* exist in at least two levels: whether the code crashes and if the returned value of a computational unit is close to the expectation. In addition to relying on historical data to define the test oracle [17,18], our study uncovers a new "getter-setter-getter" testing pattern, which helps alleviate the oracle problem by setting a parameter value and then immediately getting and checking it. This practice, though innovative, can be further improved by incorporating the user manual in developing tests and by automating parameter value selection to increase coverage.

Our future work will therefore explore these dimensions while investigating SWMM's numerical regression tests that we identified. While the 82.4% user manual coverage seems high, the 54.0% file-level code coverage begs the question as to whether our current analysis misses some code/functionality that is not user-facing. We will expand our analysis to address this question. We also plan to develop automated tools [20,21,28] for test coverage analysis, and will build initial tooling [33] on the basis of keyword matching drawn from our current operational insights. Our goal is to better support scientists in improving testing practices and software quality.

Acknowledgments. We thank the EPA SWMM team, especially Michelle Simon, Colleen Barr, and Michael Tryby, for the research collaborations. This work is partially supported by the U.S. National Science Foundation (Award CCF-1350487).

References

1. Adei, B., Dickinson, R., Rossman, L.A.: Some observations on LID output. https://www.openswmm.org/Topic/4214/some-observations-on-lid-output. Accessed April 2020
2. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: a survey. IEEE Trans. Software Eng. **41**(5), 507–525 (2015)
3. Bhat, T., Nagappan, N.: Evaluating the efficacy of test-driven development: industrial case studies. In: International Symposium on Empirical Software Engineering, pp. 356–363 (2006)
4. Brader, L., Hilliker, H., Wills, A.C.: Chapter 2 Unit Testing: Testing the Inside. Testing for Continuous Delivery with Visual Studio 2012, Microsoft Patterns & Practices (2013)
5. Brilliant, S.S., Knight, J.C., Leveson, N.G.: Analysis of faults in an N-version software experiment. IEEE Trans. Software Eng. **16**(2), 238–247 (1990)
6. Burszta-Adamiak, E., Mrowiec, M.: Modelling of green roofs' hydrologic performance using EPA's SWMM. Water Sci. Technol. **68**(1), 36–42 (2013)

7. Dawes, B., Abrahams, D.: Boost C++ libraries. https://www.boost.org. Accessed April 2020
8. Ding, J., Zhang, D., Hu, X.-H.: An application of metamorphic testing for testing scientific software. In International Workshop on Metamorphic Testing, pp. 37–43 (2016)
9. Dubois, P.F.: Testing scientific programs. Comput. Sci. Eng. **14**(4), 69–73 (2012)
10. Easterbrook, S., Johns, T.C.: Engineering the software for understanding climate change. Comput. Sci. Eng. **11**(6), 65–74 (2009)
11. Farrell, P.E., Piggott, M.D., Gorman, G.J., Ham, D.A., Wilson, C.R., Bond, T.M.: Automated continuous verification for numerical simulation. Geosci. Model Dev. **4**(2), 435–449 (2011)
12. Gudaparthi, H., Johnson, R., Challa, H., Niu, N.: Deep learning for smart sewer systems: assessing nonfunctional requirements. In: International Conference on Software Engineering (SE in Society Track) (2020)
13. Kamble, S., Jin, X., Niu, N., Simon, M.: A novel coupling pattern in computational science and engineering software. In International Workshop on Software Engineering for Science, pp. 9–12 (2017)
14. Kanewala, U., Bieman, J.M.: Testing scientific software: a systematic literature review. Inf. Software Technol. **56**(10), 1219–1232 (2014)
15. Khatwani, C., Jin, X., Niu, N., Koshoffer, A., Newman, L., Savolainen, J.: Advancing viewpoint merging in requirements engineering: a theoretical replication and explanatory study. Requirements Eng. **22**(3), 317–338 (2017). https://doi.org/10.1007/s00766-017-0271-0
16. Kourtis, I.M., Kopsiaftis, G., Bellos, V., Tsihrintzis, V.A.: Calibration and validation of SWMM model in two urban catchments in Athens, Greece. In: International Conference on Environmental Science and Technology (2017)
17. Lin, X., Simon, M., Niu, N.: Exploratory metamorphic testing for scientific software. Comput. Sci. Eng. **22**(2), 78–87 (2020)
18. Lin, X., Simon, M., Niu, N.: Hierarchical metamorphic relations for testing scientific software. In: International Workshop on Software Engineering for Science, pp. 1–8 (2018)
19. Lin, X., Simon, M., Niu, N.: Releasing scientific software in GitHub: a case study on SWMM2PEST. In: International Workshop on Software Engineering for Science, pp. 47–50 (2019)
20. Mahmoud, A., Niu, N.: Supporting requirements to code traceability through refactoring. Requirements Eng. **19**(3), 309–329 (2013). https://doi.org/10.1007/s00766-013-0197-0
21. Mahmoud, A., Niu, N.: TraCter: a tool for candidate traceability link clustering. In: International Requirements Engineering Conference, pp. 335–336 (2011)
22. Mayer, J.: On testing image processing applications with statistical methods. In: Software Engineering, pp. 69–78 (2005)
23. Niu, N., Brinkkemper, S., Franch, X., Partanen, J., Savolainen, J.: Requirements engineering and continuous deployment. IEEE Softw. **35**(2), 86–90 (2018)
24. Niu, N., Koshoffer, A., Newman, L., Khatwani, C., Samarasinghe, C., Savolainen, J.: Advancing repeated research in requirements engineering: a theoretical replication of viewpoint merging. In: International Requirements Engineering Conference, pp. 186–195 (2016)
25. Niu, N., Wang, W., Gupta, A.: Gray links in the use of requirements traceability. In: International Symposium on Foundations of Software Engineering, pp. 384–395 (2016)

26. Peng, Z., Lin, X., Niu, N.: Data of SWMM Unit Tests. http://dx.doi.org/10.7945/zpdh-7a44. Accessed April 2020
27. Prause, C.R., Werner, J., Hornig, K., Bosecker, S., Kuhrmann, M.: Is 100% test coverage a reasonable requirement? Lessons learned from a space software project. In: Felderer, M., Méndez Fernández, D., Turhan, B., Kalinowski, M., Sarro, F., Winkler, D. (eds.) PROFES 2017. LNCS, vol. 10611, pp. 351–367. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69926-4_25
28. Reddivari, S., Chen, Z., Niu, N.: ReCVisu: a tool for clustering-based visual exploration of requirements. In: International Requirements Engineering Conference, pp. 327–328 (2012)
29. Rossman, L.A.: Storm Water Management Model User's Manual Version 5.1. https://www.epa.gov/sites/production/files/2019-02/documents/epaswmm5_1_manual_master_8-2-15.pdf. Accessed April 2020
30. Segura, S., Fraser, G., Sánchez, A.B., Cortés, A.R.: A survey on metamorphic testing. IEEE Trans. Software Eng. **42**(9), 805–824 (2016)
31. Sheehan, J.: Federally funded research results are becoming more open and accessible. https://digital.gov/2016/10/28/federally-funded-research-results-are-becoming-more-open-and-accessible/. Accessed April 2020
32. United States Environmental Protection Agency. Storm Water Management Model (SWMM). https://www.epa.gov/water-research/storm-water-management-model-swmm. Accessed April 2020
33. Wang, W., Gupta, A., Niu, N., Xu, L.D., Cheng, J.-R.C., Niu, Z.: Automatically tracing dependability requirements via term-based relevance feedback. IEEE Trans. Ind. Inf. **14**(1), 342–349 (2018)
34. Wang, W., et al.: Complementarity in requirements tracing. IEEE Trans. Cybern. **50**(4), 1395–1404 (2020)
35. Wang, W., Niu, N., Liu, H., Wu, Y.: Tagging in assisted tracing. In: International Symposium on Software and Systems Traceability, pp. 8–14 (2015)