Hierarchical Metamorphic Relations for Testing Scientific Software

Xuanyi Lin
Department of Electrical Engineering and Computer Science
University of Cincinnati
Cincinnati, Ohio
linx7@mail.uc.edu

Michelle Simon
EPA Office of Research and Development
Water Systems Division
Cincinnati, Ohio
simon.michelle@epa.gov

Nan Niu
Department of Electrical Engineering and Computer Science
University of Cincinnati
Cincinnati, Ohio
nan.niu@uc.edu

ABSTRACT

Scientist developers have not yet routinely adopted systematic testing techniques to assure software quality. A key challenge is the oracle problem, a situation in which appropriate mechanisms are unavailable for checking if the code produces the expected output when executed using a set of test cases (TCs). Metamorphic testing alleviates the oracle problem by specifying the relationship that a source TC and its follow-up TC shall meet. Such relationships are called metamorphic relations (MRs) which are necessary properties of the intended program's functionality. Existing approaches handle the MRs in a flat manner. This paper introduces a novel way to facilitate a hierarchy of MRs to be developed incrementally. We illustrate our approach by testing U.S. EPA's Storm Water Management Model (SWMM). The results offer concrete insights into developing effective MRs to systematically test scientific software.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Scientific software, metamorphic testing, hierarchical metamorphic relations, SWMM, PEST++

ACM Reference Format:
Xuanyi Lin, Michelle Simon, and Nan Niu. 2018. Hierarchical Metamorphic Relations for Testing Scientific Software. In *SE4Science'18: IEEE/ACM International Workshop on Software Engineering for Science, June 2, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3194747.3194750

1 INTRODUCTION

Scientific software is commonly developed by scientists to better understand or make predictions about real world processes. Such software is widely used for assisting critical decision making in many fields. For example, in nuclear weapons simulations, code is used to determine the impact of modifications, since these weapons cannot be field tested [32].

Testing, which is important for assessing software qualities, has not been performed systematically by scientific software developers, and the key problem to address is the oracle problem [21]. An *oracle* refers to the mechanism for checking whether the program under test produces the expected output when executed using a set of test cases [16]. Due to reasons like the inherent uncertainties in simulation models and the exploratory nature of scientific software written to find answers previously unknown, oracles may not be available. The lack of suitable oracles presents the oracle problem, which makes it difficult to detect faults in scientific code [22].

One automated way to alleviate the oracle problem is *metamorphic testing* [6] which shifts software testing from one input at a time to multiple ones whose outputs shall follow certain relationships. The prototypical example is a program that computes the sine function [35]. When testing an individual input, the oracle may be difficult to obtain, e.g., the exact value of sin(12) could depend on how the specific scientific software handles round-off and truncation. However, if two test cases, e.g., 12 and (π-12), are executed one after the other, then their outputs shall be equivalent regardless the implementation specifics. The relationship, $\sin(x)=\sin(\pi-x)$, is an example of a *metamorphic relation* (MR), and it is through checking whether the MRs hold or not that metamorphic testing alleviates the oracle problem.

Researchers have applied metamorphic testing in scientific software. For example, earlier work by Chen *et al.* [7] identified one MR for numerical programs and illustrated the MR's effectiveness via testing a program that solves an elliptic partial differential equation with Dirichlet boundary conditions. More recently, Ding and colleagues [11] developed 5 MRs based on the input biological-cell images to validate the open-source light scattering simulation software that performs discrete dipole approximation. Current approaches define the MRs individually and check them independently, e.g., the 5 MRs in [11] were tested separately.

Drawing from our ongoing research on scientific software integration [20], we propose to relate the MRs in a hierarchical manner. Specifically, we use one MR's testing results to create more MRs in order to better locate the defects in scientific software. The MR hierarchy is then extended to increase metamorphic testing's effectiveness. The main contributions of this paper are twofold: (1) we present a novel approach that facilitates a hierarchy of MRs to be developed incrementally, and (2) we demonstrate our approach via a case study where the integration of multiple scientific software systems is tested.

SE4Science'18, June 2, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5748-7/18/06…$15.00
https://doi.org/10.1145/3194747.3194750

The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 describes the engineering and integration of the scientific software systems in which our work is grounded. Section 4 details our MR hierarchy and analyzes the testing results. Section 5 discusses the implications of our work, and finally, Section 6 concludes the paper.

2 BACKGROUND AND RELATED WORK

2.1 Testing Scientific Software

Several definitions and classifications of scientific software exist in the literature. Kreyman *et al.* [26] defined scientific software as the software with a large computational component providing data for decision support. Kanewala and Bieman [21] adopted a broad view and referred to scientific software as the software used for scientific purposes. The types of scientific software [4, 23] include research software written with the goal of publishing papers, end user application software [17–19] written to achieve scientific objectives (e.g., hydraulics modeling or digital preservation [24, 29, 30]), and production software written as tool support for the above (e.g., MATLAB xUnit [14]).

Despite the differences, there appear to be some common aspects. First, the size of scientific software ranges from 1,000 to 100,000 lines of code [34]. Second, the person in charge of a scientific software development project is generally a scientist and the scientists often develop the software themselves [27]. Third, the process typically involves scientists' development of discretized models, followed by transforming the models into algorithms that are then coded using a programming language [21].

These characteristics present challenges in testing scientific software. Size-wise, although the codebase may seem moderate, a complexity for testing lies in the many interdependent input parameters of a scientific program. In [39], the large input space is modeled as a directed graph whose nodes are the input parameters and the edges are the specific values accepted by the parameters annotated with the probability of a parameter taking that value. The resulting directed graph not only satisfies Markov chains, but also allows the nodes to be split or merged to manage input domain complexity. Nevertheless, identifying input domain boundaries as well as parameter dependencies *a priori* is difficult, making testing techniques like equivalence partitioning less suitable for scientific software [5, 34].

Because the goal of scientists is to do science, writing software is a means to an end but not an end in itself. Testing to assure the quality of software, therefore, has not been perceived as ultra-critical by scientist developers. For example, computational scientists would rather sacrifice the program's runtime done by code tuning in the face of achieving higher-fidelity approximations of the problems being tackled [2]. Testing remains a qualitative judgment for many scientist developers who have no distinction between debugging and testing, and adopt only ad-hoc and unsystematic testing methods (if at all) [21, 33]. Advancing toward more systematic testing thus requires automated test case generation, execution, and analysis.

The two-step process—translating the problem to an algorithm and translating the algorithm to code [2]—is so intertwined and inseparable that it is insufficient to perform testing only as a verification activity (e.g., checking the correctness of a single computation unit like a method with respect to a well-defined specification). Rather, testing must also be carried out as part of the validation that ensures the scientific model is correctly modeling the physical phenomena of interest. Consequently, oracles specifying the expected test outputs are not always available for scientific software. Causes of such oracle problems include simulation software commonly produces an approximation to a set of equations that cannot be solved exactly, some software is written to find answers that are previously unknown, some programs do not give a unique correct answer for a given set of inputs, the computations constantly involve complex floating point calculations, and so on [21].

In summary, the unique aspects of size, developer, and process present special challenges for scientific software testing. Techniques addressing these challenges shall account for input parameter interdependencies, be amenable to automation, and facilitate the validation of software without assuming the availability of suitable oracles. Next is a review of such a testing technique.

2.2 Metamorphic Testing

Metamorphic testing was introduced by Chen *et al.* [6] as a technique to alleviate the oracle problem. The basic idea is illustrated in Figure 1. For the software under test (SUT), the executions of a source test case (TC) and a follow-up TC are compared against some metamorphic relation (MR). In Figure 1, even if no oracle is available to verify each individual output, $\sin(x_1)$ or $\sin(x_2)$, the sine function can still be tested by comparing the pair of outputs against the given MR: $\sin(x_1)=\sin(x_2)$.

MRs can be of various forms, such as equalities, inequalities, periodicity properties, convergence constraints, subsumption relationships, etc. They resemble the concept of program invariants [15] but important distinctions must be made. While an invariant has to hold for every possible program execution, an MR is a relation between some executions (e.g., a pair of source TC and follow-up TC executions in Figure 1). As a result, MRs are necessary properties of the intended program's functionality: If an MR violation is detected, then metamorphic testing reveals the presence of defects in the SUT.

One of the earliest case studies of applying metamorphic testing to scientific software was performed by Chen *et al.* [7]. Their SUT was a numerical program implementing the alternating direction implicit method to solve the partial differential equation (Laplace

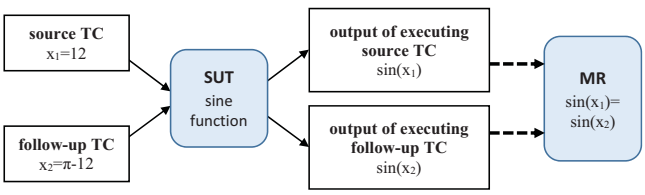

Figure 1: Illustration of metamorphic testing: "TC" stands for "test case", "SUT" stands for "software under test", "MR" stands for "metamorphic relation", solid arrows represent execution, and dotted arrows represent comparison.

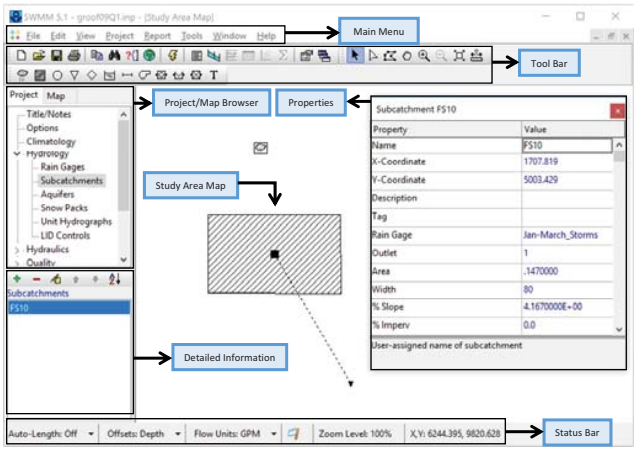

Figure 2: Screenshot of SWMM running as a Windows application, annotated with functional areas in the UI. In this example, the subcatchment "FS10" is used as input for SWMM to simulate the runoff from the given area of land modeled by the study area map. The properties of "FS10" are input parameters impacting SWMM's calculation of how the subcatchment is routed into a downstream outfall.

equation) with Dirichlet boundary conditions. Because one cannot find exact solutions to such numerical problems, an MR was developed to alleviate the oracle problem. The case study showed the effectiveness of the MR by detecting the subtle errors in the SUT that special TCs (e.g., symmetric boundary conditions in a square plate) failed to reveal. Building on the work by Chen *et al.* [7], metamorphic testing has been applied to a variety of scientific software systems, including casting simulation [36], ad-hoc network protocol simulators [8], open queuing network modeling [9], and Monte Carlo modeling for the simulation of photon propagation [10].

The construction of MRs is one of the first and most important steps in the process of metamorphic testing [35]. The identification of MRs requires knowledge of the problem domain, and is therefore a manual process [40]. Due to the relatively moderate size of scientific software ranging from 1,000 to 100,000 lines of code [34], many MRs have been developed for verifying the functional correctness of the SUT, e.g., the MR in [7] checks the faults in the implementation of the numerical program.

In addition to verifying implementation correctness, metamorphic testing has also been used as a validation activity to assess the degree to which the SUT meets the users' real needs. Zhou *et al.* [40], for instance, developed 5 MRs to assess the quality of such search engines as Google, Bing, and Baidu. Because search engine's design and algorithms are commercial secrets, the MRs in [40] serve validation purposes. In scientific software testing, the recent study by Ding *et al.* [11] used 5 MRs to validate an open-source light scattering simulator. The source TCs were defined by the input biological-cell images, and the MRs were built to test the effects of the software on the follow-up TCs which altered the image size, shape, orientation, refractive index, etc.

In summary, metamorphic testing has been applied to verify and validate scientific software that produces complex output like complicated numerical simulations. Most approaches like [11, 40] develop the MRs in a one-shot manner, organize them in the same hierarchy, and analyze their executions separately. The study presented next motivates our novel way of hierarchically creating the MRs.

3 SWMM AND ITS INTEGRATION WITH PEST++

The main subject system of our study is the United States Environmental Protection Agency's Storm Water Management Model (SWMM) [38]. SWMM 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.

The most current implementation of the model is version 5.1 which was released in 2017. It has modernized both the model's structure and its user interface (UI). Figure 2 shows a screenshot of SWMM 5.1 running as a Windows application. The two main parts of SWMM are the computational engine written in 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 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 throughout the world have been carried out by using SWMM, such as generating the spatial distribution of precipitation for the Ballona Creek Watershed (a large urban catchment in Southern California) [1], predicting the pollution in rainy weather in a combined sewer system catchment in Santander, Spain [37], and simulating a combined drainage network located in

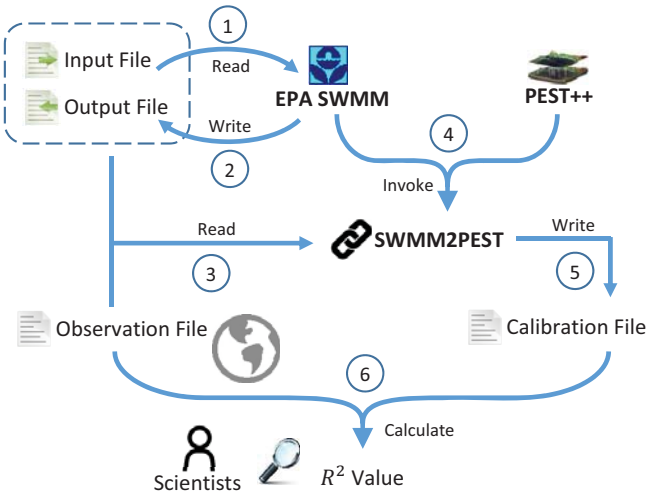

Figure 3: Process illustration of SWMM integrated with PEST++: numbers indicate the execution sequence.

Table 1: Input parameters of the FS 10 Green Roof dataset.

ID	Name	Description	ID	Name	Description
1	area	area of subcatchment	18	percent_slope	subcatchment slope (percent)
2	width	width of subcatchment	19	percent_impervious	percent imperviousness of subcatchment
3	n_imperv	Manning's n for overland flow over the impervious sub-area	20	n_perv	Manning's n for overland flow over the pervious sub-area
4	percent_zero_ impervious	percent of impervious area with no depression storage	21	surface_layer_ storage_depth	the height of any surface depression storage
5	soil_layer_thickness	thickness of the soil layer	22	surface_layer _roughness	Manning's n for overland flow over surface soil cover
6	suction	soil capillary suction	23	surface_layer_slope	slope of a roof/pavement surface or vegetative swale
7	soil_layer_field_capacity	soil field capacity	24	hydraulic_ conductivity	soil saturated hydraulic conductivity (in/hr or mm/hr)
8	soil_layer_wilting_point	soil wilting point	25	soil_layer _conductivity	soil's saturated hydraulic conductivity
9	initial_moisture_deficit	initial soil moisture deficit	26	soil_layer_slope	slope of the curve of log versus soil moisture content
10	number_replicate _units	number of replicate LID units deployed	27	soil_layer_porosity	soil porosity (volume of pore space relative to total volume)
11	top_width_overland_ flow_surface	width of the outflow face of each identical LID unit	28	surface_layer _vege tative_ cover _fraction	fraction of the surface storage volume that is filled with vegetation
12	percent_initially_ saturated	the degree to which the unit's soil is initially filled with water	29	surface_layer_ swale_side_slope	slope of the side walls of a vegetative swale's cross section
13	drainmat_thickness	thickness of the drainage mat	30	soil_layer _suction_head	soil capillary suction (in or mm)
14	area_each_unit	area of each replicate unit	31	storage_depth _perv	depression storage for pervious sub-area
15	storage_depth_imperv	depression storage for impervious sub-area	32	drainmat_roughness	Manning's n used to compute the horizontal flow rate of drained water through the mat
16	drainmat_void_fraction	ratio of void volume to total volume in the mat	33	percent_impervious_ area_treated	percent of the impervious portion of the subcatchment's non-LID area whose runoff is treated by the LID practice
17	send_outflow_ pervious_area	if the flow should be routed back onto the pervious area			

the center of Athens (Kypseli), Greece for preparing extreme events like pluvial flooding [25].

Because scientific software like SWMM takes many input parameters [39], manually calibrating their values is tedious and error-prone. This is especially the case when the catchment is large and complex [1]. For this reason, we are currently investigating the integration of SWMM with PEST++, a software package for automated parameter estimation [13]. PEST++ is written in C++ and has around 180,000 lines of code. We use Python to develop a connector called SWMM2PEST [20] which enables the integration of SWMM and PEST++. The size of SWMM2PEST is about 2,500 lines of code.

Figure 3 depicts the interactions of the three software systems: SWMM, SWMM2PEST, and PEST++. In the figure, the observation file records the real-world values of the SWMM output variables, and hence defines the ideal simulation results. This observation file, together with SWMM's input and output files, are taken by SWMM2PEST which then invokes PEST++ for calibrating the set of input parameters in order to match the observation values as closely as possible. Such a matching can be measured by the R^2 coefficient statistically indicating the goodness of fit of a model: the greater the R^2 value, the better the regression line approximates the real data points.

In the process of Figure 3, it is important to note that, from the scientist developer's perspective, no oracle is available to determine what the exact value of R^2 should be for each individual run. The lack of suitable oracles motivates us to leverage metamorphic testing to validate the integration of SWMM, SWMM2PEST, and PEST++.

4 HIERARCHICAL METAMORPHIC RELATIONS

We propose to develop a *hierarchy* of MRs as a way to improve the current practices of devising all the MRs at once and executing them separately. Our hierarchical MRs inform and give rise to each other.

4.1 FS 10 Green Roof Dataset

In this paper, we use FS 10 Green Roof [3] as our research dataset. FS 10 represents the fire station 10 located in Seattle, Washington. The Green Roof is defined as a building's roof that is partially or completely covered with vegetation together with the growing medium. The dataset was used in [3] to build an input file with a total of 24,355 lines of storm data and 133 lines of configuration of FS 10 Green Roof related parameters, based on which SWMM will produce an output file after simulation.

Thirty-three input parameters of the FS 10 Green Roof dataset are used in our study. Table 1 lists these parameters and provides brief descriptions. The interdependencies of the input parameters are common in scientific software [39]; here in our case, for instance, the soil layer thickness (p_5)[1] and soil layer porosity (p_{27}) are related and their values, consistent or not, will influence the result of calibration when the integrated SWMM and PEST++ are executed.

4.2 Before MR: Single and Pairwise Parameters

We began investigating the single parameter's applicability in the process shown in Figure 3. Specifically, each of the 33 input parameters was tested to identify the ones that would lead to runtime

[1]For the rest of the paper, we use "p_n" to refer to the input parameter n of Table 1.

Table 2: MR_{SP} testing on 431 parameter pairs.

ID	R_i^2 ? $R_{i\wedge j}^2$	R_j^2 ? $R_{i\wedge j}^2$	#	Percentage
SP_1	<	<	42	9.75%
SP_2	<	=	61	14.15%
SP_3	<	>	23	5.34%
SP_4	=	<	17	3.94%
SP_5	=	=	205	47.56%
SP_6	=	>	31	7.19%
SP_7	>	<	11	2.55%
SP_8	>	=	32	7.43%
SP_9	>	>	9	2.09%

failure, i.e., no calibration file could be generated when the given parameter was passed through the SWMM, SWMM2PEST, and PEST++ cycle. Such checking is *not* metamorphic testing because we did expect the integrated software to successfully complete the calibration, i.e., the oracle was known. The results show that 3 parameters: p_1, p_{10}, and p_{14} led to no calibrations, indicating their inapplicability to further metamorphic testing. We therefore excluded them from the subsequent analyses.

Similar to the single-parameter tests, we performed a round of pairwise-parameter tests for the 30 applicable parameters because our interest was in input parameter interdependencies. We ran a total of $\binom{30}{2}$=435 pairs of tests. As a result, 4 pairs caused the integrated software not to perform any calibration: $p_8 \wedge p_{12}$, $p_8 \wedge p_{27}$, $p_{15} \wedge p_{31}$, and $p_{19} \wedge p_{33}$. Possible failure reasons might be the calibration solver's non-convergence and instability, or the varying Marquardt lambda changed in PEST++ during an optimization iteration in order to test the effectiveness of different parameter values in lowering the objective function [12]. From the validation perspective (i.e., the standpoint of the scientist developer in Figure 3), the 3 parameters (p_1, p_{10}, and p_{14}) and the 4 pairs identified above are unproductive for metamorphic testing which will be presented next. The 431 pairs of input parameters served as a baseline for our development of hierarchical MRs.

4.3 First-Level MR_{SP}: Singleton versus Pair

The first MR that we created was to compare the pair of input parameters run together versus them running separately as singletons. We call this relation "MR_{SP}" whose source TC consists of two distinct parameters undergoing calibration by themselves: p_i and p_j. Thus, two R^2 values, R_i^2 and R_j^2, correspond to the source TC singletons. The follow-up TC of MR_{SP} is $p_i \wedge p_j$, i.e., passing both parameters together for calibration. The execution of the follow-up TC results in one R^2 value: $R_{i\wedge j}^2$.

We compared R_i^2 versus $R_{i\wedge j}^2$ as well as R_j^2 versus $R_{i\wedge j}^2$. The comparisons were organized in 9 cases shown in Table 2. For the 431 pairs of input parameters, the MR_{SP} distributions over the 9 cases are also provided in Table 2. The first result row, for example, shows that 42 (9.75%) pairs of parameters behaved in such a way that their joint calibration was better than the calibration done on either parameter alone, i.e., $R_i^2 < R_{i\wedge j}^2$ and $R_j^2 < R_{i\wedge j}^2$. This is a desired situation illustrating the benefit of allowing more parameters to be automatically calibrated.

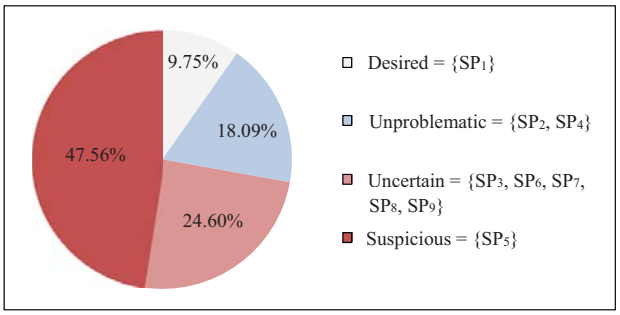

Figure 4: Classifying the results of Table 2 to provoke the development of next-level MRs to address the suspicious category.

Two cases, SP_2 and SP_4, represent the next best scenario where the joint calibration outperformed one singleton's calibration but was not worse than the other's. Although not as desired as SP_1, SP_2 and SP_4 are unproblematic in our opinion and therefore require no further hierarchical development of MRs. SP_3, SP_6, SP_7, SP_8, and SP_9 are uncertain cases due to the complex interdependencies between the input parameters which cause the calibration of a whole/pair to perform worse than calibrating its part. While the uncertain cases require further examination, we focus on investigating SP_5 which we classify as suspicious. This is not only because each pair is indifferent in the context of automated calibration ($R_i^2 = R_j^2 = R_{i\wedge j}^2$), but also because almost half of the parameter pairs (47.56%) exhibit the sense of indifference. Figure 4 shows how we regroup the results of Table 2 into the desired, unproblematic, uncertain, and suspicious categories. The largest proportion of the suspicious category in Figure 4 provokes our subsequent MR development.

4.4 Second-Level MR_{PT}: Pair to Triplet

The next level of our hierarchical MR development addresses the indifferent pairs resulted from the first level of metamorphic testing. Our intent is to check how such a matter of indifference may

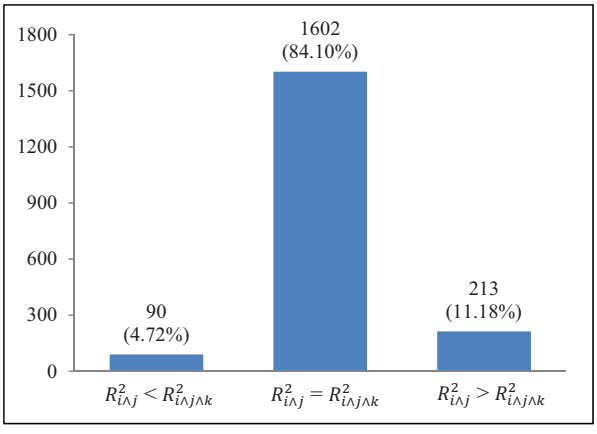

Figure 5: MR_{PT} testing on 1,905 triplets.

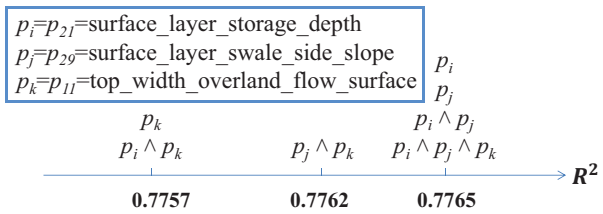

Figure 6: Example of $p_i \wedge p_j \wedge p_k$ such that $R^2_{i\wedge j} = R^2_{i\wedge j\wedge k}$.

change when a third input parameter is taken into consideration. We call this second-level relation "MR_{PT}" whose source TC is a suspicious pair: $p_i \wedge p_j$ such that $(p_i \wedge p_j) \in SP_5$. The follow-up TC is a triplet: $p_i \wedge p_j \wedge p_k$ such that $p_k \neq p_i$, $p_k \neq p_j$, and p_k appears in SP_1. Our choice of p_k is based on the assumption that if p_k and another input parameter form a desired pair belonging to SP_1, then p_k can be considered as a calibration-contributing parameter. We therefore conjecture p_k would also positively impact $p_i \wedge p_j$.

The comparisons made about MR_{PT} were between two R^2 values: $R^2_{i\wedge j}$ resulted from the source TC and $R^2_{i\wedge j\wedge k}$ resulted from the follow-up TC. A sample of 1,905 triplets were compared with their $p_i \wedge p_j$ pairs. Figure 5 shows the results. Surprisingly, p_k positively impacted $p_i \wedge p_j$ in less than 5% of the cases. Its negative effects were present in around 10% of the triplets. The vast majority (84.10%) retained the indifference. Considering MR_{PT} extends MR_{SP}, for these 1,602 triplets, $R^2_i = R^2_j = R^2_{i\wedge j} = R^2_{i\wedge j\wedge k}$.

Figure 6 provides a concrete example and orders the seven R^2 values: 3 from the singletons, 3 from the pairs, and 1 from the triplet. The base pair, $p_{21} \wedge p_{29}$, achieved the greatest R^2 in Figure 6. Having p_{11} join the automated calibration process kept the R^2 value unchanged. However, p_{11} was not orthogonal to $p_{21} \wedge p_{29}$. When p_{11} was calibrated together either with p_{21} or with p_{29}, R^2 decreased compared with $R^2_{21\wedge 29\wedge 11}$. This implies that, though p_{21} and p_{29} might be neutral to each other, their pairing helped mitigate negative effect from a third parameter like p_{11}. The example here illustrates the importance of MR_{PT} in that oftentimes one has to go beyond a pair to test and reason about the pairwise input-parameter relationships.

4.5 Third-Level MR_{FL}: Fault Localization

Based on the results of MR_{PT}, we further defined a third-level MR, named MR_{FL}, for localizing the fault. In our study, a fault can be located in three different software systems: SWMM, SWMM2PEST, and PEST++. Our purpose here is to use metamorphic testing for more precise diagnosis of the fault so that the correction could be applied to one specific software in order for the integrated system to work properly. To that end, we designed two procedures of MR_{FL}: "Check SWMM" and "Check SWMM2PEST". Figure 7 and Figure 8 show these procedures respectively. Following the method of elimination, if "Check SWMM" and "Check SWMM2PEST" both pass, then the fault is likely to be located in PEST++.

"Check SWMM" of Figure 7 builds on the previous hierarchy's MR_{PT} where $R^2_{i\wedge j} = R^2_{i\wedge j\wedge k}$. The particular parameter, p_k, would undergo its own metamorphic testing only on SWMM. The source

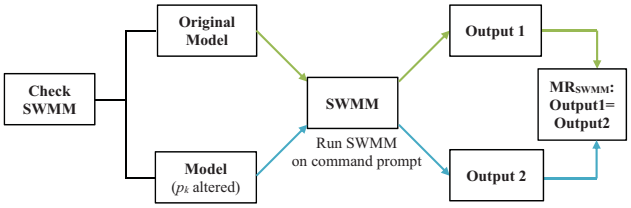

Figure 7: Extending $R^2_{i\wedge j} = R^2_{i\wedge j\wedge k}$ of MR_{PT} to locate fault in SWMM via MR_{SWMM}.

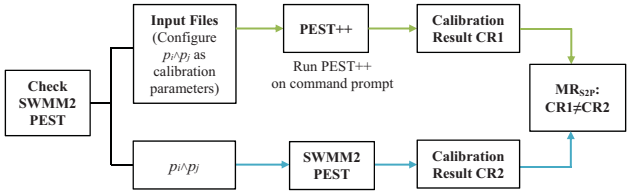

Figure 8: Extending $R^2_{i\wedge j} > R^2_{i\wedge j\wedge k}$ of MR_{PT} to locate fault in SWMM2PEST via MR_{S2P}.

TC and the follow-up TC had only one difference, i.e., the value of p_k was changed while all the other parameters' values were kept intact. Running SWMM on command prompt for the source and follow-up TCs would result in two outputs, and if they were equivalent as shown by MR_{SWMM} in Figure 7, then the fault could be in SWMM, e.g., p_k was involved in some dead code.

"Check SWMM2PEST" of Figure 8 extends MR_{PT}'s $R^2_{i\wedge j} > R^2_{i\wedge j\wedge k}$ results. The metamorphic testing here had the source TC inputting $p_i \wedge p_j$ as the parameters to be calibrated directly to PEST++ on command prompt. The follow-up TC was still aimed at automatically calibrating $p_i \wedge p_j$, but the execution was initiated from SWMM2PEST as shown in Figure 8. If the source and follow-up TCs had different calibration results, as specified in MR_{S2P}, then SWMM2PEST was likely faulty.

5 DISCUSSION

In this section, we discuss a couple of limitations of our work and also share the lessons learned which we believe could advance the state-of-the-art in applying metamorphic testing for scientific software.

One of the limitations was our pursuit of only one of the categories in Figure 4 (namely, the suspicious category). Although our operation was informed by the largest proportion of MR_{SP} testing results, other categories, especially the uncertain cases of $\{SP_3, SP_6, SP_7, SP_8, SP_9\}$, could be hierarchically extended. When doing so for the uncertain category, one could potentially iterate the execution of the process shown in Figure 3. In this way, the random values initially assigned to input parameters could be gradually replaced with the calibrated values, ideally converging to the optimal values. Such iterative executions, which seem to be common among simulation software, present new challenges for metamorphic testing in

terms of defining source and follow-up TCs that should be sensible to iteration/termination conditions.

Another limitation refers to the determination of p_k in MR_{PT} (cf. Section 4.4). Ideally, one should have chosen such p_k to form the triplet with $p_i \wedge p_j$ that $p_i \wedge p_k \in SP_1$ and $p_j \wedge p_k \in SP_1$, i.e., p_k's positive effects on the base pair have been directly observed in testing MR_{SP}. However, such p_k rarely existed. We therefore relaxed the criteria to select some parameters from SP_1 as long as it was different from p_i and p_j. This was based on our observation that the parameters in SP_1 and SP_5 had little overlap. This may not hold if other datasets were used and/or deeper semantic relationships of parameters were determined. Nevertheless, our choice of p_k in MR_{PT} precluded exhaustive generation of testing data. Rather, a sample of 1,905 triplets was tested, which caused about 50 hours executed on a MacBook Pro. This echoes the long execution time being a challenge for scientific software testing [21].

The most important lesson to share is our way to hierarchically develop MRs in testing scientific software. To the best of our knowledge, the hierarchical MRs are novel as existing approaches such as [11, 40] define the MRs all at once and reason about their validity without inherently connecting one to another. Our work is in stark contrast to the existing approaches. Rather than viewing the exploratory nature of scientific software as a challenge to software testing [21], we take advantage of it to enable a hierarchy of MRs to be built incrementally. When constructing MR_{PT}, for example, the selection of p_k was fully informed by the MR_{SP} tested earlier. Consequently, p_k was chosen not randomly but intentionally from SP_1. Similarly, the procedures of "Check SWMM" and "Check SWMM2PEST" shown in Figures 7 and 8 may seem so general that they can be applied to any p_k and $p_i \wedge p_j$. On the contrary, we made more informed decisions to scope how to "Check SWMM" and to "Check SWMM2PEST" (the third-level MRs) based on the results from the second-level MR_{PT}.

Another insight is our synergy of metamorphic testing and testing with known oracles (cf. Section 4.2). Many metamorphic testing approaches focus exclusively on MRs. We realize that not every testing for scientific software faces the oracle problem. In our work, testing the 33 individual FS 10 Green Roof input parameters and their pairwise combinations provides more sensible foundations for our hierarchical development of MRs. While our testing of the third-level MR_{FL} is ongoing, we want to emphasize the practical value of fault localization so as to facilitate proper removal of the fault. This practicality is especially important in our study since the three software systems are built for different purposes, written in different programming languages, and maintained by different scientific computing communities.

6 CONCLUSIONS

In this paper, we have presented a novel way to hierarchically develop MRs in order to systematically test scientific software. We demonstrated the approach in testing the U.S. EPA's SWMM software integrated with a parameter estimation system. We showed how testing with known oracles could be synthesized with metamorphic testing, and more importantly, how the MR hierarchy should be targeted at fault localization. The three levels of MRs not only offer specific mechanisms to overcome the oracle problems in SWMM testing, but also provide concrete examples for the software engineering and scientific computing communities to discuss the hierarchical MRs' pros and cons.

Our future work includes executing the MR hierarchy with other SWMM datasets than FS 10 Green Roof with systematic reuse [31] as a goal. We are also interested in developing new MRs to tackle the uncertain category of Figure 4, which may require the iteration/termination conditions to be accounted for. Moreover, though the MRs in our current work focus on the comparison of different R^2 values, the absolute values of R^2 should not be neglected. In the example shown in Figure 6, the three R^2 values are very close to each other. Based on the feedback from the domain expert of SWMM, the overall 77% level of R^2 represents a good fit for the model. The work on heuristic test oracles [28] is therefore relevant for us to investigate whether the outputs (e.g., R^2 values) are close enough to be semantically or practically considered equals.

7 ACKNOWLEDGMENTS

This work is funded in part by the U.S. National Science Foundation (Award CCF 1350487) and the U.S. Environmental Protection Agency via the Oak Ridge Institute Student Employment Agreement.

Disclaimer: The U.S. Environmental Protection Agency, through its Office of Research and Development, funded and managed, or partially funded and collaborated in, the research described herein. It has been subjected to the Agency's peer and administrative review and has been approved for external publication. Any opinions expressed in this paper are those of the author(s) and do not necessarily reflect the views of the Agency, therefore, no official endorsement should be inferred. Any mention of trade names or commercial products does not constitute endorsement or recommendation for use.

REFERENCES

[1] J. Barco, K. M. Wong, and M. K. Stenstrom. Automatic calibration of the U.S. EPA SWMM model for a large urban catchment. *Journal of Hydraulic Engineering*, 134(4):466–474, April 2008.

[2] V. R. Basili, J. C. Carver, D. Cruzes, L. Hochstein, J. K. Hollingsworth, F. Shull, and M. V. Zelkowitz. Understanding the high-performance-computing community: a software engineer's perspective. *IEEE Software*, 25(4):29–36, July/August 2008.

[3] Cardno TEC, Inc. Green Roof performance study. https://www.seattle.gov/Documents/Departments/OSE/Green-Roof-Performance-Study-2012.pdf. Last accessed: March 2018.

[4] J. C. Carver, R. A. Bartlett, D. Heaton, and L. Hochstein. What scientists and engineers think they know about software engineering: a survey. Technical report, Sandia National Laboratories, Albuquerque, NM, USA, 2011.

[5] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post. Software development environments for scientific and engineering software: a series of case studies. In *International Conference on Software Engineering (ICSE)*, pages 550–559, Minneapolis, MN, USA, May 2007.

[6] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, The Hong Kong University of Science and Technology, Hong Kong, China, 1998.

[7] T. Y. Chen, J. Feng, and T. H. Tse. Metamorphic testing of programs on partial differential equations: a case study. In *International Computer Software and Applications Conference (COMPSAC)*, pages 327–333, Oxford, England, August 2002.

[8] T. Y. Chen, F.-C. Kuo, H. Liu, and S. Wang. Conformance testing of network simulators based on metamorphic testing technique. In *International Conference on Formal Techniques for Distributed Systems (FMOODS/FORTE)*, pages 243–248, Lisboa, Portugal, June 2009.

[9] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and W. K. Tam. Testing an open source suite for open queuing network modelling using metamorphic testing technique. In

International Conference on Engineering of Complex Computer Systems (ICECCS), pages 23–29, Potsdam, Germany, June 2009.

[10] J. Ding, T. Wu, D. Xu, J. Q. Lu, and X.-H. Hu. Metamorphic testing of a Monte Carlo modeling program. In *International Workshop on Automation of Software Test (AST)*, pages 1–7, Honolulu, HI, USA, May 2011.

[11] J. Ding, D. Zhang, and X.-H. Hu. An application of metamorphic testing for testing scientific software. In *International Workshop on Metamorphic Testing (MET)*, pages 37–43, Austin, TX, USA, May 2016.

[12] J. Doherty. Getting the most out of PEST. http://www.pesthomepage.org/Downloads.php. Last accessed: March 2018.

[13] J. Doherty, C. Muffels, J. Rumbaugh, and M. Tonkin. http://www.pesthomepage.org. Last accessed: March 2018.

[14] S. L. Eddins. Automated software testing for MATLAB. *Computing in Science and Engineering*, 11(6):48–55, November/December 2009.

[15] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, December 2007.

[16] R. M. Hierons. Oracles for distributed testing. *IEEE Transactions on Software Engineering*, 38(3):629–641, May/June 2012.

[17] X. Jin, C. Khatwani, N. Niu, M. Wagner, and J. Savolainen. Pragmatic software reuse in bioinformatics: how can social network information help? In *International Conference on Software Reuse (ICSR)*, pages 247–264, Limassol, Cyprus, June 2016.

[18] X. Jin, Niu, and M. Wagner. On the impact of social network information diversity on end-user programming productivity: a foraging-theoretic study. In *International Workshop on Social Software Engineering (SSE)*, pages 15–21, Seattle, WA, USA, November 2016.

[19] X. Jin, Niu, and M. Wagner. Facilitating end-user developers by estimating time cost of foraging a webpage. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 31–35, Raleigh, NC, USA, October 2017.

[20] S. Kamble, X. Jin, N. Niu, and M. Simon. A novel coupling pattern in computational science and engineering software. In *International Workshop on Software Engineering for Science (SE4Science)*, pages 9–12, Buenos Aires, Argentina, May 2017.

[21] U. Kanewala and J. M. Bieman. Testing scientific software: a systematic literature review. *Information & Software Technology*, 56(10):1219–1232, October 2014.

[22] D. Kelly, D. Hook, and R. Sanders. Five recommended practices for computational scientists who write software. *Computing in Science and Engineering*, 11(5):48–53, September/October 2009.

[23] D. Kelly, S. Smith, and N. Meng. Software engineering for scientists. *Computing in Science and Engineering*, 13(5):7–11, September/October 2011.

[24] C. Khatwani, X. Jin, N. Niu, A. Koshoffer, L. Newman, and J. Savolainen. Advancing viewpoint merging in requirements engineering: a theoretical replication and explanatory study. *Requirements Engineering*, 22(3):317–338, September 2017.

[25] I. M. Kourtis, G. Kopsiaftis, V. Bellos, and V. A. Tsihrintzis. Calibration and validation of SWMM model in two urban catchments in Athens, Greece. In *International Conference on Environmental Science and Technology (CEST)*, Rhodes, Greece, August-September 2017.

[26] K. Kreyman, D. L. Parnas, and S. Qiao. Inspection procedures for critical programs that model physical phenomena. Technical report, McMaster University, Hamilton, Canada, 1999.

[27] C. Morris and J. Segal. Some challenges facing scientific software developers: the case of molecular biology. In *International Conference on e-Science (eScience)*, pages 216–222, Oxford, UK, December 2009.

[28] C. Murphy and G. E. Kaiser. Empirical evaluation of approaches to testing applications without test oracles. Technical report, Columbia University, New York, NY, USA, 2010.

[29] N. Niu, S. Brinkkemper, X. Franch, J. Partanen, and J. Savolainen. Requirements engineering and continuous deployment. *IEEE Software*, 35(2):86–90, March/April 2018.

[30] N. Niu, A. Koshoffer, L. Newman, C. Khatwani, C. Samarasinghe, and J. Savolainen. Advancing repeated research in requirements engineering: a theoretical replication of viewpoint merging. In *International Requirements Engineering Conference (RE)*, pages 186–195, Beijing, China, September 2016.

[31] N. Niu, J. Savolainen, Z. Niu, M. Jin, and J.-R. C. Cheng. A systems approach to product line requirements reuse. *IEEE Systems Journal*, 8(3):827–836, September 2014.

[32] D. E. Post and R. P. Kendall. Software project management and quality engineering practices for complex, coupled multiphysics, massively parallel computational simulations: lessons learned from ASCI. *International Journal of High Performance Computing Applications*, 18(4):399–416, November 2004.

[33] R. Sanders and D. Kelly. The challenge of testing scientific software. In *Conference for the Association for Software Testing (CAST)*, pages 30–36, Toronto, Canada, July 2008.

[34] R. Sanders and D. Kelly. Dealing with risk in scientific software development. *IEEE Software*, 25(4):21–28, July/August 2008.

[35] S. Segura, G. Fraser, A. B. Sánchez, and A. R. Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, September 2016.

[36] K. Y. Sim, W. K. S. Pao, and C. Lin. Metamorphic testing using geometric interrogation technique and its application. *ECTI Transactions on Computer and Information Technology*, 1(2):91–95, November 2005.

[37] J. Temprano, Ó. Arango, J. Cagiao, J. Suárez, and I. Tejero. Stormwater quality calibration by SWMM: a case study in Northern Spain. *Water SA*, 32(1):55–63, 2005.

[38] US EPA Research. https://www.epa.gov/water-research/storm-water-management-model-swmm. Last accessed: March 2018.

[39] S. A. Vilkomir, W. T. Swain, J. H. Poore, and K. T. Clarno. Modeling input space for testing scientific computational software: a case study. In *International Conference on Computational Science (ICCS)*, pages 291–300, Kraków, Poland, June 2008.

[40] Z. Zhou, S. Xiang, and T. Y. Chen. Metamorphic testing for software quality assessment: a study of search engines. *IEEE Transactions on Software Engineering*, 42(3):260–280, March 2016.