Analyzing the reliability of simulated distributed systems using Metamorphic Testing

Alberto Núñez
Universidad Complutense de Madrid
Alberto.Nunez@pdi.ucm.es

Pablo C. Cañizares
Universidad Autónoma de Madrid
Pablo.Cerro@uam.es

Pablo Gómez-Abajo
Universidad Autónoma de Madrid
Pablo.GomezA@uam.es

Esther Guerra
Universidad Autónoma de Madrid
Esther.Guerra@uam.es

Juan de Lara
Universidad Autónoma de Madrid
Juan.deLara@uam.es

ABSTRACT

Simulation is widely adopted by the research community to analyze and study complex systems. It is based on the idea of creating a model representing the target system under study, so that the experiments can be executed over the model instead of the target system. However, since the model is a simplification of a real-world system, the obtained results entail an accuracy loss, which makes determining the reliability of the experiments a complex task.

Testing can be applied to check the correctness of systems. Thus, an oracle is used to determine if a test is correct or not. In the field of simulation, an oracle can be applied to determine the reliability of the results, but in most cases, the oracle is not available or is computationally too expensive to be applied.

In this work, we propose to use metamorphic testing to detect faults in simulated distributed systems. In essence, we use metamorphic relations – representing the relevant properties of the system under study – as an oracle. Thus, the results provided are contrasted against these relations to determine their reliability. In order to show the applicability of this approach, we have modelled different distributed systems architectures using the SIMCAN simulator and a high performance application that is executed over the models.

CCS CONCEPTS

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

KEYWORDS

Metamorphic testing, Simulation, Distributed systems, HPC

ACM Reference Format:
Alberto Núñez, Pablo C. Cañizares, Pablo Gómez-Abajo, Esther Guerra, and Juan de Lara. 2022. Analyzing the reliability of simulated distributed systems using Metamorphic Testing. In *7th International Workshop on Metamorphic Testing (MET'22), May 9, 2022, Pittsburgh, P A, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3524846.3527342

MET'22, May 9, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9307-2/22/05…$15.00
https://doi.org/10.1145/3524846.3527342

1 INTRODUCTION

During the last decades, simulation has been widely adopted by the industry and the research community to analyze and study distributed systems [11]. According to the definition provided by Robert E. Shannon, simulation is "*the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behavior of the system and/or evaluating various strategies for the operation of the system*" [24]. In computer science, simulation can be seen as a technique to represent the behavior of real-world systems – like HPC data-centers or clouds – using a program that executes algorithms to imitate the internal processes of the target system being simulated [7].

One approach to validate a simulation is to reproduce the same experiment over the real-world target system and compare the obtained results against the ones provided in the simulated environment [20]. Unfortunately, in many cases, there is no real-world data available from the target system. Another approach consists in validating the simulator used to conduct the research. This implies determining whether the simulator program properly represents the behavior of the entities being simulated. In the case of distributed systems, these entities are disks, CPUs and communication networks, among others. Therefore, to demonstrate that a simulator is accurate, it is required to create – and validate – every potential model that can be reproduced by the simulator, which is unpractical and unfeasible due to time and computational restrictions. In this paper, we propose an approach for locating errors both in the model representing the target system, and in the simulator used to simulate its behaviour, by applying metamorphic testing (MeT) techniques.

Testing has been widely applied during the last decades to check the correctness of systems [2, 13]. There are two fundamental problems in testing: the *oracle problem* [6] and the *reliable test set problem* [9]. The former refers to the capability of having a mechanism – called oracle – to decide if the result provided by a test is correct, or not. The latter consists in generating a proper test suite for testing the target system. In our setting, we would like to create *oracles* to decide if the results provided by the simulators are reliable, or not. However, the *oracle problem* applies in our context, since for most simulation scenarios, the oracle is not available or it is computationally too expensive. In the past, those programs without an oracle have been considered as *non-testable programs* [29]. Some examples of this kind of applications are, among others, simulation of scientific and complex systems, machine learning algorithms, and search engines [23].

To attack these problems, we propose to combine MeT with simulation to detect errors in simulated distributed systems. The main objective of our approach is to provide valuable information for helping researchers to decide if a simulator is reliable enough to simulate a model representing real-world systems. In order to accomplish this objective, we use MeT techniques, that is, instead of checking the output provided by the simulation with the output provided by an oracle, we use metamorphic relations (MRs) which reflect the underlying behavior of the system under study by relating the inputs of different tests with their outputs.

In our approach, the researcher first executes the simulation of the model representing the target system. We refer to this execution as *source* test case. Next, a wide-spectrum of *follow-up* test cases is created. Generally, *follow-up* test cases are automatically generated by applying modifications over a *source* test case. For each *follow-up* test case, we use the results provided by the *source* test case against the one provided by the *follow-up* test case, to check if the MR is fulfilled. In such a case, we consider that the test passes. Otherwise, we say that the test fails, which means that for this test case, the model does not represent the expected behavior. Since MeT alleviates the two fundamental problems of testing [6], it can be considered adequate to verify the results provided by the simulations.

In order to show the applicability of this methodology, we present an empirical study directed to analyze the accuracy of the SIMCAN simulator. For this, we model a correct and a flawed architecture, and inject an artificial flaw in SIMCAN. The results show that the approach is able to detect both misbehaviors in the incorrect model, and in the faulty version of the simulator.

Paper organization. Section 2 analyses related work. Section 3 presents our approach to analyze the reliability of simulated distributed systems. Section 4 introduces the SIMCAN simulator, and Section 5 presents an empirical study to investigate its reliability. Finally, Section 6 concludes with some directions for future work.

2 RELATED WORK

During the last decade, the number of publications related to MeT has considerably increased [23]. In this section, we analyze the works applying MeT over simulation.

Lin and collaborators [12] present an approach, called exploratory metamorphic testing, to detect bugs. This proposal consists in defining the multiple iterations of tests for performing continuous simulations, and then keeping multiple MRs open for investigating the testing-result patterns. This approach has been applied to discover mismatches and constraints in automatically calibrating parameters for a water management model.

Olsen and Raunak proposed an overall framework and guidelines to apply MeT for simulation validation [21]. The authors of this work also present three different case studies to show the application of this approach: a networked agent-based models (ABM) of gossip propagation; a multiscale ABM of cancer; and a discrete-event simulation (DES) of airport arrival, check in, and security. The proposed process for simulation validation using MeT does not take into account the quality of the MRs, assuming these have been properly designed to detect flaws in the simulation.

A similar process has been followed to analyze and optimize cloud computing systems using simulation techniques. Núñez and Hierons presented a methodology to semi-automatically test and validate cloud configurations by combining simulation techniques and MeT [18]. To that end, the authors propose several MRs focusing on different aspects of the system: performance, functionality and energy consumption. The empirical study presented in that work analyses two different cloud configurations, clearly showing the existing flaws in the misconfigured cloud model. TEA-Cloud is a complete framework that integrates simulation with MeT for validating cloud system designs [16]. In that work, the behavior of the cloud is formally presented through a collection of MRs. The authors present an experimental study divided in two parts. First, different experiments are carried out to assess the validity of the MRs. Next, a complete empirical study is carried out to check the effectiveness and applicability of the framework. For this, 15 different mutants are used and around 16000 different follow-up test cases were automatically generated. Cañizares and collaborators [4] present an approach where cloud models are optimized to improve the energy consumption. In this case, Genetics Algorithms (GAs) are applied in the MeT process to generate variants of the original model to be improved.

In the field of intelligent and expert systems, there are some works in the current literature that combine MeT and simulation techniques. CloudExperts is an intelligent system that applies MeT to calculate the most appropriate cloud simulator for simulating a cloud configuration that fulfils the requirements of the user [15]. In that work, the proposed system uses the time required to execute the simulation, the features to be simulated, and the number of tests successfully executed by each simulator to make a final recommendation. An expert system for locating errors in memory systems is described in [3]. The system combines simulation and MeT enabling the automatic generation of appropriate test cases and deciding if their outputs are correct.

In the field of autonomous vehicles (AVs), MeT has been successfully applied to detect faults. Iqbal and collaborators [10] conducted a case study to investigate the fault-detection effectiveness of existing Advanced Driver-Assistance Systems (ADAS) testing standards. In particular, the authors focus the research in the Lane Keeping Assist System (LKAS). In order to check the robustness of that system, MeT was applied to detect bugs over 40 different scenarios. The results obtained in the study revealed a real-world bug in the LKAS system. Valle presents an approach to apply Quality-of-Service (QoS) aware MeT to test AVs modeled in MATLAB/Simulink [27]. In this work, MRs are defined by using a set of QoS measures applied to AVs. Additionally, mutation testing is applied to assess the approach in an AV modeled in Simulink.

In the field of social networks, Ahlgren and collaborators [1] describe how the system MIA (Metamorphic Interaction Automaton) is applied to test WW, Facebook's simulation of its platforms. WW is a multi-agent simulation environment where each agent is a bot – isolated from real users – that simulates classes of user behavior. WW simulates user interactions with bots trained by machine learning. The approach uses a suite of over 40 metamorphic test cases. The offline mode simulation shows that the test flakiness can be reduced from 50% (of all online tests) to 0% (offline).

Some of these works consist in validating and optimizing cloud system models using MeT. In contrast, our proposed approach differs in several aspects: i) Since this work focuses in detecting faults in both the model representing the target system, and in the simulator, the underlying architecture must be specified with a high level of detail, where parameters like the latency of the network, the read bandwidth of the storage disks and the configuration of the file systems, among others, must be specified to accurately represent the target system. On the contrary, cloud models use a more generalized configuration of the underlying architecture and, therefore, these parameters are not considered; ii) To study distributed systems, it is key to select appropriate distributed applications to be executed over the model, hence stressing the different components of the system, like disks, CPUs and memories, to analyze the overall system behavior. However, in cloud systems, the workload is usually represented by the users accessing the cloud for requesting resources, instead of the applications executed by the users; and iii) In this work we analyze the reliability of the simulator and the model for representing the behaviour of distributed systems. Thus, the goal is not to optimize the models under study.

3 ANALYZING THE RELIABILITY OF SIMULATED DISTRIBUTED SYSTEMS

A model of a distributed system is a simplification of a real-world system, reflecting its most relevant features, and discarding those that are not significant for its behavior of interest. Since the execution of the experiments is carried out over the model – not over the real system – the reliability of the results heavily depends of two main factors: the level of detail provided in the model to represent the target system, and the accuracy of the simulator to imitate the behavior of hardware components, like communication networks, disks and CPUs, among others.

The wide adoption of simulation by the research community is mainly due to the possibility of executing the experiments in a regular computer, since the target system is not required, while its main weakness is the reliability of the results. Simulators contain processes and algorithms that imitate the behavior of the devices of the target system. Increasing the level of accuracy of the simulator requires processing more features, which in turn requires more complex algorithms and computational power. Thus, the accuracy-performance tradeoff of the selected simulator must be carefully considered, to obtain reasonable simulation execution times.

In order to analyze the reliability of simulated distributed systems, it is crucial to execute a proper application that uses the simulated hardware – and software – parts of the target system. Thus, using different configurations of the application to be executed in the model, allows focusing the operations over different subsystems. For instance, a CPU-intensive application massively performs computing operations over the different CPUs of the system, while a data-intensive application massively performs data operations over the storage devices. This way, a model of a distributed system consists of two different parts: the underlying architecture of the system, and the application(s) to be executed over this architecture.

In this paper, we present an approach to analyze the reliability of simulators and models for representing the behavior of real-world distributed systems. Since we have not an oracle to determine if the results provided by the simulation are reliable, or not, we use MeT techniques to alleviate this issue. MeT uses metamorphic relations (MRs) to reflect the expected properties of the target system. The inputs of different tests are related to the observed outputs. Thus, an MR can be seen as a formula $i(MR) \Rightarrow o(MR)$, where $i(MR)$ refers to the relation between a source test case and a follow-up test case, and $o(MR)$ refers to the relation that must be fulfilled by the outputs obtained from the source test case and the follow-up test case.

The rest of this section is divided into three parts. Section 3.1 shows the main components of the distributed systems and how these are organized to create a model. Next, in Section 3.2, we describe a model of a distributed application that can be executed over architecture models. Finally, Section 3.3 shows the notation and MRs used to analyze the accuracy of distributed system simulators.

3.1 Modeling the architecture of distributed systems

Broadly speaking, a distributed system consists of different nodes interconnected through a communication network. In general, each node, which is also referred to as *blade*, consists of four basic subsystems: computing, storage, memory and networking.

In large distributed systems, like data-centers or HPC systems, the blades are organized in racks. The idea is to maximize the density of computers to reduce the required space to assemble the system. To that end, a rack accommodates several boards, each one containing different blades. Figure 1 shows a general architecture of a distributed system. The racks can be configured to contain internal switches for the interconnection of blades, which also provide a connection with the external communication network. Using this schema, a wide spectrum of scenarios can be represented, from a basic distributed system consisting of two different nodes interconnected through a switch, to a large HPC system consisting of thousands of blades organized in racks.

In order to study a distributed system using simulation, it is required to create a model of the target system. The model must reflect the underlying configuration of the system, which consists of different parameters like, just to name a few, the topology of the system, the hardware features of each blade, and the configuration of the storage system. However, the level of detail applied to create the model heavily depends on the simulator used to represent its behavior.

3.2 Modeling distributed applications

Usually, distributed systems are studied by executing a well-known application – or benchmark – and analyzing the provided results, which determine different aspects of the system, like performance, scalability or the existence of system bottlenecks, among others. Hence, it is key to use an adequate application to accurately analyze the required features of the target system. For instance, the computers that appear in the Top500 list [25], that is, the 500 most powerful computers in the world, are analyzed using the LINPACK Benchmark [8] that, basically, solves a dense system of linear equations. However, other applications can be used, for instance, to analyze the performance of the storage system [22].

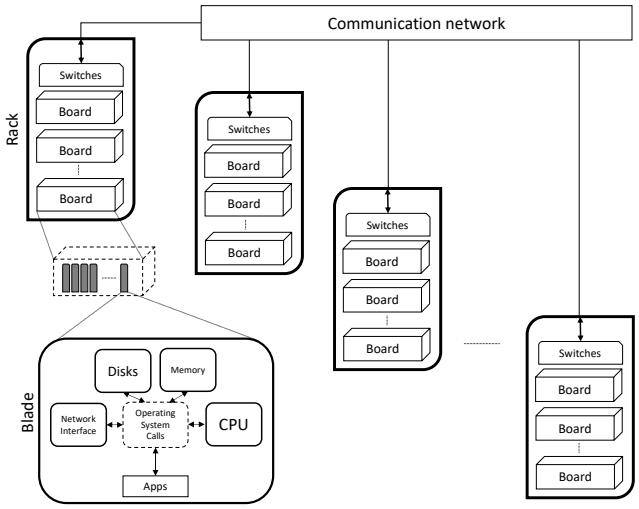

Figure 1: Architecture system model

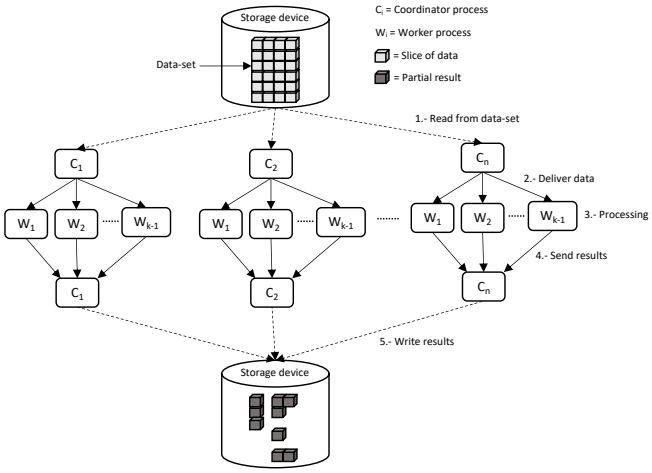

Figure 2: Application model

These applications are executed over a real production-ready system, which means that it is not possible to apply short-term architectural modifications anymore. Instead, in simulation, researchers are usually interested in contrasting the results of different models, where several configurations can be stressed to analyze their strengths and weaknesses. It is therefore required a flexible and versatile application that can be easily modified to represent a wide-range of configurations in a scalable way. To that end, we use an application model that has been used in the past to analyze the behavior of different distributed systems [14].

This application deploys two different types of process to compute a data-set: coordinators and workers. The former type focuses on delivering data among the worker processes and receiving the results. The latter type focuses on performing the computation over the input data to generate the results. Figure 2 shows a schema of this application, where the processes are grouped in n domains. Each domain contains k processes, one coordinator and k-1 workers. The processes of the same domain cannot communicate with those processes belonging to a different domain. The following steps are performed until the data-set is completely processed: 1) Each coordinator reads a slice of data from the data-set, which is stored in a storage device; 2) The slice of data is divided into different portions and are sent to the workers of the same domain; 3) Each worker computes the data and generates partial results; 4) The intermediate results are sent to the coordinator; 5) The coordinator receives the results from the worker processes and write them to disk.

3.3 Definition of MRs

In MeT, the properties that define the underlying behavior of the target system must be reflected in the MRs. Since a model of a distributed system contains a high number of inter-related parameters that affect the overall system performance, the MRs must be carefully designed. For instance, if the MRs are too general, that is, the features reflected in the relations represent basic characteristics of the underlying system under study, then the major part of the test cases will likely satisfy the MRs and, consequently, the existing errors will not be discovered. In MeT, defining high-quality MRs is key to reach valuable results.

Before defining the MRs, we need to create a model m that represents the target system and, next, use MeT techniques to determine if the results provided by the simulator S to simulate the behavior of m are reliable.

A model of a distributed system can be represented by a tuple $m = (s, a)$, with s the underlying architecture, and a the application to be executed on s. From the point of view of the simulator S, a test case is defined as the simulation of a model m (i.e., the simulation of the execution of a over s), denoted by $S(m)$. The *simulated time* estimated to execute a over s is denoted by $S_{time}(m)$.

We use the following notation to represent the features of the system architecture s:

- s_{comp} = Total number of computing nodes.
- s_{sto} = Total number of storage nodes.
- $s_{size} = s_{comp} + s_{sto}$.
- s_{nCores} = Number of CPU cores for each processor.
- $s_{CPUspeed}$ = Speed of each CPU core.
- s_{NetBan} = Network bandwidth.
- s_{NetLat} = Network latency.
- $s_{DiskSize}$ = Storage disk capacity.
- $s_{DiskRead}$ = Write bandwidth of each storage disk.
- $s_{DiskWrite}$ = Read bandwidth of each storage disk.

Similarly, we use the following notation to represent the features of the application a:

- $a_{totalProc}$ = Total number of processes deployed in a.
- a_k = Number of processes for each domain.
- $a_n = \frac{a_{totalProc}}{a_k}$ = Number of domains.
- a_{data} = Size of the data-set to be processed by a.
- $a_{deliver}$ = Amount of data (in MB) delivered to each worker process.
- a_{result} = Size of partial results (in MB) generated by each worker process.

$$MR_1 : s_{comp} < s'_{comp} \wedge a_{totalProc} < a'_{totalProc} \Rightarrow S_{time}(s,a) > S_{time}(s',a')$$
$$MR_2 : s_{nCores} < s'_{nCores} \wedge a_{totalProc} < a'_{totalProc} \Rightarrow S_{time}(s,a) > S_{time}(s',a')$$
$$MR_3 : s_{NetBan} < s'_{NetBan} \wedge s_{NetLat} \geq s'_{NetLat} \Rightarrow S_{time}(s,a) > S_{time}(s',a')$$
$$MR_4 : s_{sto} < s'_{sto} \wedge a_n < a'_n \Rightarrow S_{time}(s,a) > S_{time}(s',a')$$
$$MR_5 : s_{DiskRead} < s'_{DiskRead} \wedge s_{DiskWrite} < s'_{DiskWrite} \Rightarrow S_{time}(s,a) > S_{time}(s',a')$$

Figure 3: Metamorphic relations for validating distributed system simulations

- a_{cpu} = Computing performed (in MIs) for each worker process to compute a portion of data.

In order to check the reliability of the results provided by S, different MRs are used as an oracle. To that end, we define a source test case m, and create different follow-up test cases from m. Once the source test case and the follow-up test cases are executed using S, the results are contrasted against the MRs to analyze if the properties reflected in the MRs are satisfied.

Figure 3 presents several MRs for checking the reliability of simulated distributed systems, where the source test case is denoted by $m = (s, a)$ and the follow-up test case is denoted by $m' = (s', a')$. For clarity, we assume that the features that are not explicitly used in the MRs are equal in both m and m'.

4 OVERVIEW OF THE SIMCAN SIMULATOR

SIMCAN is a modular simulation platform for modeling and simulating both distributed systems and applications. It is publicly available as open source software[1]. SIMCAN has been written in C++ using OMNeT++ [28], a simulation framework to build network simulators, and INET [26], an open-source model library for OMNeT++ that provides protocols, agents and other models to simulate communication networks. SIMCAN has been successfully used in a wide variety of fields, like testing [5], teaching [19], and optimization of HPC systems [14].

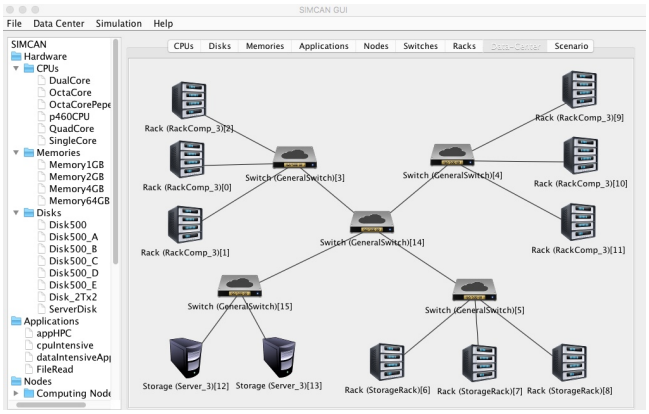

Figure 4: Snapshot of SIMCAN GUI

A repository of components organizes all the modules that can be used to build a distributed system. Figure 4 shows a snapshot of SIMCAN's GUI, where the left frame shows the repository of components like disks, CPUs and memories, and the right frame shows the architecture of the model. Thus, components from the left frame can be easily included in the model to represent the target system. Moreover, for each hardware device, different modules can be coded to represent its behavior using different techniques and algorithms, hence providing a high level of flexibility to create and configure a wide-spectrum of system configurations.

Additionally, SIMCAN provides different APIs that allow researchers to code distributed applications. These APIs have been carefully designed to ease the coding process. On the one hand, SIMCAN provides a POSIX-like API, which contains the main functions to interact with files, memory and network facilities. On the other hand, an API based on MPI (Message Passing Interface) can be used to develop high performance applications.

5 EMPIRICAL STUDY

Next, we present an empirical study to demonstrate the applicability of our approach. The main objective is to check if the behavior of different models, representing distributed systems, is the expected one. To that end, we use the MRs described in Section 3.3 and the SIMCAN simulator [17].

In this study, we consider the two models m_1 and m_2 shown in Figure 5. The architectural configuration of both models is the same, consisting of one rack having two boards – each one containing 8 blades – and one switch connected to the main network. The storage system consists of one storage server directly connected to the main network. However, while m_1 is correctly configured, model m_2 provides a wrong configuration of the file system. Specifically, m_2 only considers the first server in the servers list for accessing the data, keeping the rest ones unreachable. This is appreciated in Figure 5.b, where the processes executed in the racks cannot access the additional storage servers (included in the follow-up test cases). For simplicity, we consider both models homogenous, that is, every blade of the system has the same hardware characteristics. Table 1 shows the main configuration parameters of these models.

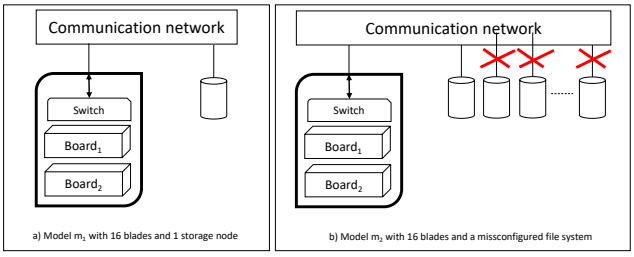

Figure 5: Models of distributed systems

[1] Available at http://www.simcansimulator.com

System architecture s	Application a
s_{comp} = 16	$a_{totalProc}$ = 16
s_{sto} = 1	a_k = 8
s_{nCores} = 1	a_n = 2
$s_{CPUspeed}$ = 23650 MIPS	a_{data} = 140 MB
s_{NetBan} = 1Gbps	$a_{deliver}$ = 2MB
s_{NetLat} = 0.46 μs	a_{result} = 1 MB
$s_{DiskRead}$ = 565 Mbps	a_{cpu} = 10000000 MIs
$s_{DiskWrite}$ = 318 Mpbs	-

Table 1: Main parameters of the models $m1$ and $m2$

```
1  int cpuIndex = 0;
2    ...
3    // Locate and assign CPU core
4    cpuIndex = searchIdleCPU();
5    sm_cpu->setNextModuleIndex(cpuIndex);
6
7    // Update state!
8    isCPU_Idle[cpuIndex]=false;
9
10   // Send request to CPU-core
11   sendRequestMessage (sm_cpu, toCPUGate[sm_cpu->getNextModuleIndex()]);
```

Listing 1: Excerpt of the correct CPU scheduler code

```
1  int cpuIndex = 0;
2    ...
3    // Locate and assign CPU core
4    sm_cpu->setNextModuleIndex(cpuIndex);
5    cpuIndex = searchIdleCPU(); ← // Bug!
6
7    // Update state!
8    isCPU_Idle[cpuIndex]=false;
9
10   // Send request to CPU-core
11   sendRequestMessage (sm_cpu, toCPUGate[sm_cpu->getNextModuleIndex()]);
```

Listing 2: Excerpt of the faulty CPU scheduler code

From now on, we refer to these two models as *source test cases*. In order to analyze them, we first generate different follow-up test cases using the MRs of Figure 3. Next, we execute the source test cases and the follow-up test cases using the SIMCAN simulator. Then, the results of each source test case are contrasted with the ones obtained from the follow-up test cases using the MRs. When these results satisfy the MRs, we assume that the behavior of the system is the expected one. Instead, when these results do not satisfy the relation, we consider that the model does not properly represent the behavior of the target system.

Additionally, we have created a faulty version of the SIMCAN simulator, which is denoted by S^f. In particular, we have injected a fault in the CPU scheduler, which manages the computing requests from the processes and assigns a CPU-core to execute the code. Listing 1 shows the correct code of the CPU scheduler, while Listing 2 shows the faulty version, which only considers one CPU-core to execute the code (line 5), keeping the rest of the cores idle.

Table 2 shows the results of the study, where the first three columns depict the MR used to generate the follow-up test cases, the modification applied over the source test case to generate the follow-up test case, and a description of the follow-up test case. The next four columns show the results obtained from the simulation. Thus, the columns labelled as $S_{time}(m'_1)$ and $S_{time}(m'_2)$ show the results of executing the follow-up test cases related to models m_1 and m_2, respectively, using the correct version of SIMCAN. The last two columns ($S^f_{time}(m'_1)$ and $S^f_{time}(m'_2)$) show the results of executing the follow-up test cases using the faulty version of SIMCAN.

Executing the source test cases using the correct version of SIMCAN – denoted by S – yields the same value for $S_{time}(m_1)$ and $S_{time}(m_2)$, 215.786 seconds. Likewise, the execution of the source test cases using the faulty version of SIMCAN (S^f_{time}) yields the same results, 215.786 seconds. Next, the results obtained from the source test cases and the ones obtained from the follow-up test cases are used to check if the MRs are satisfied. We use the symbol ✓to represent that a test case fulfills the MR and the system behaves as expected. The results are shown in bold text with the symbol ✘ when the MR is not satisfied. However, in those cases where an error produces a wrong behaviour of the system, which is not detected by the MRs, the results are shown in bold text with the symbol ⊗.

The first MR (MR_1) refers to the capacity of the system of increasing the number of blades. Thus, scaling up the system hardware should provide and improvement in the overall performance. In this case, the number of processes is also increased in such a way that each process has a dedicated CPU-core to be executed. The results show that every follow-up test case satisfies MR_1 using both versions of SIMCAN, S and S^f. MR_1 does not detect the fault of S^f because each blade has only one CPU-core.

However, it is worth noting that increasing the number of computing nodes does not guarantee – in all the cases – an increment in the overall performance. Particularly, for those follow-up test cases related to MR_1, this occurs when the target system contains 56 and 72 computing nodes. Essentially, the time required to fully process the data-set heavily depends on the number of iterations performed by the domains (see Figure 2). Thus, it is possible that, in the last iteration, the remaining data is delivered among some domains, keeping the rest ones idle, hence limiting the exploitation of parallelism. In this particular case, 24 computing nodes require 4 iterations to fully process the data-set, while 32 computing nodes require 3 iterations. Similarly, using 40, 48, 56, 64 and 72 computing nodes require 2 iterations and, consequently, the performance obtained is similar in these cases. Furthermore, since all the coordinator processes are concurrently requesting data from the same server, it is expected to generate a system bottleneck, which causes some performance variations like the ones presented in these tests.

The next MR, MR_2, refers to the capacity of the system to increase the number of CPU-cores per processor. Similar to the previous MR, we increase the number of cores in proportion with the number of processes. This MR is fulfilled by every follow-up test case generated from MR_2 when the simulator S is used to run the simulations, that is, $S_{time}(s,a) > S_{time}(s',a')$. However, when S^f is used to run the simulations, the two first test cases do not fulfill MR_2 since $S^f_{time}(s,a) \ngeq S^f_{time}(s',a')$ for the models m_1 and m_2. These results reveal the existing bug in the CPU scheduler that only makes use of one core of the CPU processor, hence forcing all the processes executed in the same blade use the same CPU-core. Although the next two follow-up test cases (using $a'_{totalProc}$ = 128

MR	Follow-up test case	Description	$S_{time}(m'_1)$	$S_{time}(m'_2)$	$S^f_{time}(m'_1)$	$S^f_{time}(m'_2)$
MR_1	s'_{comp} = 24, $a'_{totalProc}$ = 24	Increases computing nodes and processes to 24	173.042 (✓)	173.042 (✓)	173.042 (✓)	173.042 (✓)
	s'_{comp} = 32, $a'_{totalProc}$ = 32	Increases computing nodes and processes to 32	130.538 (✓)	130.538 (✓)	130.538 (✓)	130.538 (✓)
	s'_{comp} = 40, $a'_{totalProc}$ = 40	Increases computing nodes and processes to 40	88.339 (✓)	88.339 (✓)	88.339 (✓)	88.339 (✓)
	s'_{comp} = 48, $a'_{totalProc}$ = 48	Increases computing nodes and processes to 48	88.085 (✓)	88.085 (✓)	88.085 (✓)	88.085 (✓)
	s'_{comp} = 56, $a'_{totalProc}$ = 56	Increases computing nodes and processes to 56	88.138 (✓)	88.138 (✓)	88.138 (✓)	88.138 (✓)
	s'_{comp} = 64, $a'_{totalProc}$ = 64	Increases computing nodes and processes to 64	88.079 (✓)	88.079 (✓)	88.079 (✓)	88.079 (✓)
	s'_{comp} = 72, $a'_{totalProc}$ = 72	Increases computing nodes and processes to 72	88.130 (✓)	88.130 (✓)	88.130 (✓)	88.130 (✓)
	s'_{comp} = 80, $a'_{totalProc}$ = 80	Increases computing nodes and processes to 80	45.996 (✓)	45.996 (✓)	45.996 (✓)	45.996 (✓)
MR_2	s'_{nCores} = 2, $a'_{totalProc}$ = 32	Increases number of cores per chip to 2	130.452 (✓)	130.452 (✓)	**257.234 (✘)**	**257.234 (✘)**
	s'_{nCores} = 4, $a'_{totalProc}$ = 64	Increases number of cores per chip to 4	87.935 (✓)	87.935 (✓)	**257.04 (✘)**	**257.04 (✘)**
	s'_{nCores} = 8, $a'_{totalProc}$ = 128	Increases number of cores per chip to 8	45.316 (✓)	45.316 (✓)	**172.166** (⊗)	**172.166** (⊗)
	s'_{nCores} = 16, $a'_{totalProc}$ = 256	Increases number of cores per chip to 16	45.316 (✓)	45.316 (✓)	**172.166** (⊗)	**172.166** (⊗)
MR_3	s'_{NetBan} = 10Gbps	Increases network bandwidth to 10Gbps	214.845 (✓)	214.845 (✓)	214.845 (✓)	214.845 (✓)
	s'_{NetBan} = 40Gbps	Increases network bandwidth to 40Gbps	214.766 (✓)	214.766 (✓)	214.766 (✓)	214.766 (✓)
	s'_{NetBan} = 100Gbps	Increases network bandwidth to 100Gbps	214.751 (✓)	214.751 (✓)	214.751 (✓)	214.751 (✓)
MR_4	s'_{sto} = 2	Increases storage nodes to 2	214.618 (✓)	**215.821 (✘)**	214.618 (✓)	**215.821 (✘)**
	s'_{sto} = 4	Increases storage nodes to 4	214.417 (✓)	**216.104 (✘)**	214.417 (✓)	**216.104 (✘)**
	s'_{sto} = 8	Increases storage nodes to 8	214.482 (✓)	**216.146 (✘)**	214.482 (✓)	**216.146 (✘)**
	s'_{sto} = 16	Increases storage nodes to 16	214.485 (✓)	**216.177 (✘)**	214.485 (✓)	**216.177 (✘)**
MR_5	$s'_{DiskRead}$ = 600 Mbps $s'_{DiskWrite}$ = 350 Mbps	Increases disk read bandwidth to 600 Mbps and write bandwidth to 350 Mbps	215.534 (✓)	215.534 (✓)	215.534 (✓)	215.534 (✓)
	$s'_{DiskRead}$ = 650 Mbps $s'_{DiskWrite}$ = 400 Mbps	Increases disk read bandwidth to 650 Mbps and write bandwidth to 400 Mbps	215.225 (✓)	215.225 (✓)	215.225 (✓)	215.225 (✓)
	$s'_{DiskRead}$ = 700 Mbps $s'_{DiskWrite}$ = 450 Mbps	Increases disk read bandwidth to 700 Mbps and write bandwidth to 450 Mbps	214.962 (✓)	214.962 (✓)	214.962 (✓)	214.962 (✓)
	$s'_{DiskRead}$ = 750 Mbps $s'_{DiskWrite}$ = 500 Mbps	Increases disk read bandwidth to 750 Mbps and write bandwidth to 500 Mbps	214.758 (✓)	214.758 (✓)	214.758 (✓)	214.758 (✓)
	$s'_{DiskRead}$ = 800 Mbps $s'_{DiskWrite}$ = 550 Mbps	Increases disk read bandwidth to 800 Mbps and write bandwidth to 550 Mbps	214.57 (✓)	214.57 (✓)	214.57 (✓)	214.57 (✓)

Table 2: Results of the experimental process for the models *m1* and *m2*

and $a'_{totalProc}$ = 256) satisfy this MR, the obtained system performance is considerably less than the expected one. In these scenarios, using 128 or more processes, only one iteration is required to fully process the data-set and, therefore, the overall processing time is significantly reduced, which makes that MR_2 holds. Consequently, MR_2 is not capable of detecting the wrong behaviour of the system when the faulty version of the simulator (S^f) is used.

MR_3 checks the communication network. In this case, increasing the bandwidth of the network should result in an improvement of the overall system performance. Similar to the case of MR_1, all follow-up test cases related to MR_3 fulfill the relation for both models using S and S^f.

MR_4 deals with the storage system. Let us remark that m_2 has a misconfigured file system where only one storage server can be accessed. In this case, the results show a flaw in m_2, which is detected by both versions of the simulator (cf. fifth and seventh column in Table 2). In this case, increasing the capacity of the storage system does not provide a better performance because all accesses to the data are performed over the same server, hence limiting the parallelism to read and write data.

Finally, MR_5 studies the capacity of the disk drives, increasing the read and write bandwidth of the drives. Similarly to MR_1 and MR_3, every follow-up test case satisfies this MR using S and S^f.

The obtained results clearly show the flaws existing in the simulation process, both in the model and in the simulator component representing the behavior of hardware devices. When a correct simulator S is used to execute a properly designed model, every follow-up test case fulfills the MRs (see column $S_{time}(m'_1)$). On the contrary, using an incorrect model or a faulty simulator produces results that do not fulfill the MRs, showing a flaw in the system.

As a general conclusion, we think that MeT is adequate to verify the results obtained from simulating distributed systems and applications. It is important to remark that each MR focuses on a specific part of the system, such as computing, storage and networking. Thus, each MR must be applied to detect flaws in a specific part of the system. For instance, in this study, those MRs that do not reflect computing properties are not capable of detecting the bug in the CPU scheduler (see MR_3, MR_4 and MR_5 in Table 2). However, our proposed MRs are not capable of detecting the fault in the simulator for two specific configurations, which is a limitation of our approach that we will investigate in future work.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an approach to analyze the reliability of simulated distributed systems using metamorphic testing. For this purpose, we have designed different MRs focusing on different parts of the systems. The applicability of the approach has been

assessed by analyzing different models of distributed systems using the SIMCAN simulator. One of these models has been misconfigured – on purpose – to check if the MRs can detect the flaw. Additionally, we used a faulty version of the SIMCAN simulator that limits the applications to use only one CPU-core in multi-core scenarios. In this study, different follow-up test cases have been generated using the designed MRs. The MRs have been used both as an oracle to detect errors and as a template to generate the follow-up test cases. Notably, the results of this study show that MeT is able to detect both flaws in the model and in the simulator. Nevertheless, there are two specific configurations where the MR could not detect the wrong behaviour of the system using the faulty version of SIMCAN.

One of the main advantages of this approach is the capability of checking a specific part of the system. Thus, the testing process can be configured to execute the test cases generated from the selected MRs, which significantly increases the overall performance. On the other hand, the main limitation of this approach lies in the design of the MRs, as these must be carefully created to appropriately reflect the properties of the target system. Otherwise, the testing process may produce meaningless results. For instance, if the number of processes deployed by the application are not increased, it is possible that some resources remain idle, hence not providing an improvement in the overall performance. Moreover, the simulator used to represent the behavior of the models must be carefully analyzed. In this work, we assume the SIMCAN simulator correct (its code has been thoroughly revised by different experts), but generally, one cannot guarantee the absence of small bugs.

As future work, we are planning to investigate how to detect those cases where the MRs are satisfied by the results provided by the simulation and the system does not behave as expected. Additionally, we are planning to study the accuracy of this approach to detect wrong behaviors in large and complex models, and in the simulator. To this end, the experiments will be executed both in a simulator environment and in a real-world system. Finally, we are also interested in investigating the possibility of combining different MRs to detect faults in the system under study.

ACKNOWLEDGMENTS

This paper has been partially supported by the Madrid Government (Comunidad de Madrid-Spain) under the Multiannual Agreement with the Complutense University as part of the Program to Stimulate Research for Young Doctors in the context of the V PRICIT (Regional Programme of Research and Technological Innovation) under grant PR65/19-22452, the Spanish Ministry of Science (RTI2018-095255-B-I00), the Madrid region (P2018/TCS-4314), and project S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union and Comunidad de Madrid.

REFERENCES

[1] J. Ahlgren, M. Berezin, K. Bojarczuk, E. Dulskyte, I. Dvortsova, J. George, N. Gucevska, M. Harman, M. Lomeli, E. Meijer, S. Sapora, and J. Spahr-Summers. 2021. Testing web enabled simulation at scale using metamorphic testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'21)*. 140–149.

[2] P. Ammann and J. Offutt. 2017. *Introduction to Software Testing* (2nd ed.). Cambridge University Press.

[3] P. C. Cañizares, A. Núñez, and J. de Lara. 2019. An expert system for checking the correctness of memory systems using simulation and metamorphic testing. *Expert Systems with Applications* 132 (2019), 44–62.

[4] P. C. Cañizares, A. Núñez, J. de Lara, and L. Llana. 2020. MT-EA4Cloud: A methodology for testing and optimising energy-aware cloud systems. *Journal of Systems and Software* 163 (2020), 110522.

[5] P. C. Cañizares, A. Núñez, and M. G. Merayo. 2018. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *Journal of Systems and Software* 143 (2018), 187–207.

[6] T.Y. Chen, F.C. Kuo, H. Liu, P.L. Poon, D. Towey, T.H. Tse, and Z.Q. Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 4:1–4:27.

[7] C. Dobre, F. Pop, and V. Cristea. 2011. New trends in large scale distributed systems simulation. *Journal of Algorithms & Computational Technology* 5, 2 (2011), 221–257.

[8] J. J. Dongarra, P. Luszczek, and A. Petitet. 2003. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience* 15, 9 (2003), 803–820.

[9] M. Harman, Y. Jia, and Y. Zhang. 2015. Achievements, open problems and challenges for search based software testing. In *IEEE 8th International Conference on Software Testing, Verification and Validation (ICST'15)*. 1–12.

[10] M. Iqbal, J. C. Han, Z. Q. Zhou, and D. Towey. 2021. Enhancing Euro NCAP standards with metamorphic testing for verification of advanced driver-assistance systems. In *2021 IEEE/ACM 6th International Workshop on Metamorphic Testing (MET'21)*. 37–41.

[11] A. Kostin and L. Ilushechkina. 2010. *Modeling and simulation of distributed systems*. World Scientific Publishing. 1–440 pages.

[12] X. Lin, M. Simon, and N. Niu. 2020. Exploratory metamorphic testing for scientific software. *Computing in Science Engineering* 22, 2 (2020), 78–87.

[13] G. J. Myers, C. Sandler, and T. Badgett. 2011. *The Art of Software Testing* (3rd ed.). John Wiley & Sons.

[14] A. Nuñez, C. Andrés, and M. G. Merayo. 2012. Optimizing the trade-offs between cost and performance in scientific computing. In *International Conference on Computational Science, ICCS'12 (Procedia Computer Science)*, Vol. 9. Elsevier, 498–507.

[15] A. Núñez, P. C. Cañizares, and J. de Lara. 2022. CloudExpert: An intelligent system for selecting cloud system simulators. *Expert Systems with Applications* 187 (2022), 115955.

[16] A. Núñez, P. C. Cañizares, M. Núñez, and R. M. Hierons. 2021. TEA-Cloud: A formal framework for testing cloud computing systems. *IEEE Transactions on Reliability* 70, 1 (2021), 261–284.

[17] A. Nuñez, J. Fernández, R. Filgueira, F. García, and J. Carretero. 2012. SIMCAN: A flexible, scalable and expandable simulation platform for modelling and simulating distributed architectures and applications. *Simulation Modelling Practice and Theory* 20, 1 (2012), 12–32.

[18] A. Nuñez and R. M. Hierons. 2015. A methodology for validating cloud models using metamorphic testing. *Annales des Télécommunications* 70, 3-4 (2015), 127–135.

[19] A. Núñez, C. Mañoso, A. Pérez de Madrid, and S. Pickin. 2019. SIMCAN: A simulator to improve the learning of distributed and high-performance computing systems in engineering degrees. *Computer Applications in Engineering Education* 27, 5 (2019), 1126–1138.

[20] M. Olsen and M. Raunak. 2014. A survey of validation in health care simulation studies. In *Winter Simulation Conference*. 4089–4090.

[21] M. Olsen and M. Raunak. 2019. Increasing validity of simulation models through metamorphic testing. *IEEE Transactions on Reliability* 68, 1 (2019), 91–108.

[22] H.J. Park. 2013. Performance evaluation on journaling file systems using Iozone tool in the Linux: Focus on read, write. *The Journal of Korea Navigation Institute* 17 (2013), 39–46.

[23] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on Software Engineering* 42, 9 (2016), 805–824.

[24] R. E. Shannon. 1998. Introduction to the art and science of simulation. In *30th Conference on Winter Simulation (WSC'98)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 7–14.

[25] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. 2021. TOP500 Supercomputer site. http://www.top500.org

[26] G. Szaszko. 2017. INET framework for the OMNeT++ discrete event simulator. https://github.com/inet-framework/inet. [Online; Latest commit on Oct 2, 2017. Accessed on Oct 15, 2017].

[27] P. Valle. 2021. Metamorphic testing of autonomous vehicles: A case study on Simulink. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 105–107.

[28] A. Varga. 2001. The OMNeT++ discrete event simulation system,. In *European Simulation Multiconference*.

[29] E. J. Weyuker. 1982. On Testing Non-Testable Programs. *Comput. J.* 25, 4 (1982), 465–470.