Measurement Challenges for Cyber Cyber Digital Twins: Experiences from the Deployment of Facebook's WW Simulation System

K. Bojarczuk
Facebook Inc.
UK

I. Dvortsova
Facebook Inc.
UK

J. George
Facebook Inc.
UK

N. Gucevska
Facebook Inc.
UK

M. Harman
Facebook Inc.
UK

M. Lomeli
Facebook Inc.
UK

S. Lucas
Facebook Inc.
UK

E. Meijer
Facebook Inc.
UK

R. Rojas
Facebook Inc.
UK

S. Sapora
Facebook Inc.
UK

ABSTRACT

A cyber cyber digital twin is a deployed software model that executes in tandem with the system it simulates, contributing to, and drawing from, the system's behaviour. This paper outlines Facebook's cyber cyber digital twin, dubbed WW, a twin of Facebook's WWW platform, built using web-enabled simulation. The paper focuses on the current research challenges and opportunities in the area of measurement. Measurement challenges lie at the heart of modern simulation. They directly impact how we use simulation outcomes for automated online and semi-automated offline decision making. Measurements also encompas how we verify and validate those outcomes. Modern simulation systems are increasingly becoming more like cyber cyber digital twins, effectively moving from manual to automated decision making, hence, these measurement challenges acquire ever greater significance.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**; **Extra-functional properties**.

KEYWORDS

Simulation, Social Media, Digital Twin, Software Measurement

ACM Reference Format:
K. Bojarczuk, I. Dvortsova, J. George, N. Gucevska, M. Harman, M. Lomeli, S. Lucas, E. Meijer, R. Rojas, and S. Sapora. 2021. Measurement Challenges for Cyber Cyber Digital Twins: Experiences from the Deployment of Facebook's WW Simulation System. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '21), October 11–15, 2021, Bari, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3475716.3484196

ESEM '21, October 11–15, 2021, Bari, Italy

ACM ISBN 978-1-4503-8665-4/21/10.
https://doi.org/10.1145/3475716.3484196

1 INTRODUCTION

This paper is concerned with challenges in measurement of simulation systems. Measurement questions are essential for simulation since these have direct importance and impact on the decisions made from simulation outcomes.

Simulation is acquiring increasing importance due to the widespread application domains, and far-reaching impact of the critical decisions taken based on simulation results. These decisions affect diverse areas of human endeavour ranging from economics [50] and traffic safety [5], through to profound questions about the health of our species [1] and of our planet [30]. Without reliable, replicable, interpretable and soundly-based measurement, all of these important application areas become compromised.

At Facebook, we are building a digital twin, called WW [2], of all Facebook platforms. WW simulates Facebook user community behaviour, but uses the real Facebook platforms, as well as offline versions of them, thereby making simulations highly realistic [4]. WW is a cyber *cyber* digital twin, because it simulates virtual (i.e., cyber-based) systems; the Facebook WWW platforms in particular. [4]. The ability to execute simulations of user behaviour on the real software platforms is a unique property of cyber *cyber* digital twins, distinguishing them from their cyber *physical* cousins [4].

The 'cyber cyber' nature of the simulation also has implications for measurement, because we can deploy the same measurements that the real platform computes for real users, when computing values for simulated users (bots). However, as we shall see, despite this apparent advantage for measurement, there remain many important open challenges and research questions for measurement of cyber cyber digital twins in particular, and of modern software simulations in general.

This position paper outlines some of the open research challenges, highlighting the potential impact on software measurement theory and practice. We hope that the paper generates further interest amongst the scientific research community, in tackling some of these challenging, impactful, and intellectually stimulating problems.

2 WHAT IS MEASURED

Measurement is important, not only in deciding all the low-level signals to monitor but also plays a role in determining which high-level features or summaries to produce. The very essence of simulation is to produce measurements in the simulated world that are relevant to future real-world executions. Because our simulation is web-enabled [2], the measurements are taken from the execution of bot behaviours on the real platform. This allows us to base decisions on measurements taken directly from the execution of simulated behaviours on the real platform and not on simulated values from some *model* of the real world.

In order to support measurement, WW has an extensive set of monitoring facilities that can be selectively enabled to report a variety of signals including:

- Details of each action taken by each bot/user, whether it succeeded or failed and any exceptions it raised
- The observations provided to each bot/user
- The exact timestamp per action and observation (in both real-time and simulation-time)
- The changes or mutations made to the WW social graph as a result of the simulation

These measurements provide a rich set of signals from which to draw test-related inferences. In our definition of measurement, we include logging all the required low-level details of a run together with the higher-level signals we abstract from these. For example, we log exactly when each bot action was triggered and the reaction it produced. Subsequently, this raw output is processed and a meaningful summary needs to be extracted from it.

One of the advantages of the measurements we take from WW simulations is their realistic nature but there remain open problems and challenges in interpreting the measurement signal from simulations, more details of which are included in the next section.

3 FOUNDATIONS OF SIMULATION MEASUREMENT

A run from a simulation process encodes the entire emergent behaviour of the whole community of users on the WWW platform. It is of interest to determine whether two or more simulation runs come from the same underlying process. This can be done using a decision rule based on one or more shared attributes of each run. This boolean outcome is what constitutes our simulation measurement in this section. The task of determining whether two sets of runs X and Y are generated by the same underlying distribution, called simulation testing, gives us a form of software testing that lies above all other testing approaches in the test abstraction hierarchy, such as integration testing and unit testing. It has a greater fault revealing potential, since the simulation encodes the entire emergent behaviour of the whole community of users on the platform, rather than testing a specific aspect of the infrastructure.

As a consequence, simulation-based testing is able to find subtler bugs that might otherwise make their way into production. It achieves this greater fault revelation potential at the cost of greater computational effort than any other testing approach.

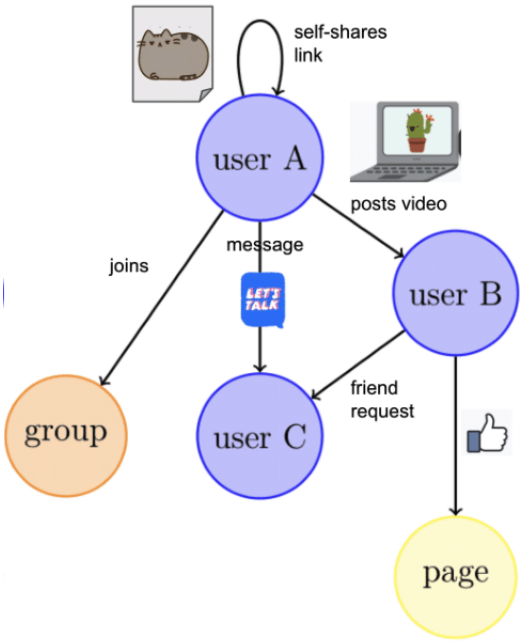

Figure 1: Some possible actions performed by users/bots that are part of the simulation output: bot A messaged bot C, posted video on bot's B profile, bot A reshared a URL link, bot B liked a page, bot A joined a group, bot B requested friendship to bot C.

The set of simulation runs used for simulation testing could include runs with underlying differences that need to be detected. These differences could be due to bugs or to infrastructural changes that indirectly affect the bot behaviour. An example of a given simulation output of a single run with $n+1$ steps from a community of three users A,B and C is:

$$S_n = \left\{ \left(A_0^b, \ldots, A_n^b \right)_{b \in \{A,B,C\}}, \left(\mathcal{G}_{A,B,C}^t \right)_{t=0}^n, \left(Q_A^t, Q_B^t, Q_C^t \right)_{t=0}^n \right\} \tag{1}$$

where A_t^b is the action performed by bot b at time t, $\mathcal{G}_{A,B,C}^t$ denotes the friendship graph for bots A,B and C at time t, Q_b^t denotes the content of the action performed by bot b at time t, $Q = \emptyset$ if there is no content. We can either select the number of steps in the simulation or select the maximum number of actions. The latter is useful when we move from virtual to real time. The initial friendship graph, $\mathcal{G}_{A,B,C}^0$, is specified in advance. Nevertheless, it can evolve during the simulation, for example when one bot becomes friends with another bot that joins the platform at $t > 0$. In the next section we construct tests based on attributes extracted from this simulation output.

3.1 Deterministic Simulation-based testing for WW at Facebook

In general, when we construct tests, we do so with respect to a set of attributes that we can observe from a simulation. These attributes are 'measured' as a byproduct of simulation, and form the connection between simulation and software measurement. We test WW using a software testing infrastructure called MIA (the Metamorphic Interaction Automaton) [3].

In this section, we review three types of deterministic simulation testing:

(1) **Social testing** [2] consists of verifying that the whole system meets social properties required of the system. It simulates the actions of an entire community of users, interacting with one another.
(2) **End-to-end testing** consists on verifying that the different parts of the system are working as intended, it simulates the actions of a single user of the system. An end-to-end test is usually regarded as having a successful outcome if there were no exceptions raised during the execution of the simulation. It is therefore, primarily concerned with whether the software system breaks when used by one of its users. However, this is a very rudimentary success criterion; equivalent to the default (aka implicit [13]) oracle. By contrast, social testing can reveal bugs where one user can harm another through the system under test; traditional end-to-end testing cannot do this because it concerns only a single user experience, and because it seeks bugs caused *by* the system not *through its use*.
(3) **Regression testing** [55] compares two or more executions of the system under test, typically comparing the current version with some previous ground truth. In previous work [3], we showed that regression testing is a special case of metamorphic testing, an observation we used to simplify the deployment of our MIA test system for social, end-to-end, regression and metamorphic testing.

Importantly, and unlike traditional simulation, the measurements are taken *directly* from the execution on the *real software system under test*. We call this "web-enabled" simulation [2], because the deployment of software systems on web-based platforms enables us to simulate user communities' behaviour on the real platform itself, rather than a mere simulation of that platform. This has the important implication that our measurements are taken directly from the engineering artefact simulated, thereby imbuing them with greater realism, and consequently actionability, compared to traditional simulations.

Based on the attributes we measure, we define properties that we expect to hold irrespective of the specific details of the simulations. For example, a simple property to verify is, if we create a new community of N bots, then at least N new user accounts have been created. We can therefore augment end-to-end testing with property testing, allowing the simulation developer to test that any changes they may make to the platform respect the properties of interest. In particular, end-to-end property testing in WW at Facebook is used to help us catch social bugs [2]. We use this form of simulation-based testing to identify integrity bugs; social bugs that affect the integrity of our user community.

We called this "property testing" rather than "property-based testing", because property-based testing typically involves simplifying fault-revealing test sequences [18]. Currently, we do not perform the simplification step, although there is no reason why, in future, we could not extend our test infrastructure to perform such simplifications, thereby making it fully property-based testing.

Property testing can be very powerful to measure expected outcomes in the simulation. Consider the scenario in which the social media platform needs to give certain guarantees about the deletion of offensive content. All three forms of simulation-based testing could be used to identify integrity bugs; social bugs that affect the integrity of our user community.

For example, we can check whether there exists a bug that breaks the part of the infrastructure responsible for deleting content. If we suppose further that the changes made in part of the code that is unrelated to content deletion have a subtle transitive effect on the deletion framework such that it can occasionally fail to delete content. Property testing would be the option of choice since the specific property – deletion of content – can be tested in all versions of the code to hold true.

A property or set of properties captures aspects of the social media system that should be preserved throughout any and all scenarios captured by the simulation. We can verify the following proposition: for all ordered tuples O, which contain a set of properties computed from the simulation output, and an ordered tuple O^*, which contains the expected property, then

$$\mathcal{ER}(O, O^*)$$

where $\mathcal{ER}$ denotes the conjunction of point-wise equality relationships.

One can test whether a simulation run contains a specific piece of content or whether a given action or set of actions has been completed. In the latter case, due to the stochasticity of simulations, testing that exactly n actions happened can lead to a false positive test failure so we instead test for the typicality per action. The statistical testing task is described in the next section, it consists on determining whether two sets of runs X and Y are generated by the same underlying distribution.

3.2 Statistical simulation testing

A simulation is an example of a randomized algorithm [10], since running it twice usually produces different output. For this reason, instead of testing whether a given property is present in all simulation runs, we can test whether it is *typically present* in a proportion of runs.

This is because there exists inherent stochasticity arising from both the bots' behaviours and infrastructural changes. Many systems, not just simulation systems, exhibit high degrees of non-determinism, arising from a diverse set of reasons, including:

- Remote Procedure Call (RPC)-related variability, arising from network delays and variable host performance and loads
- Parallel, multi-threaded and/or asynchronous execution
- Use of random number generators
- Dependence on real-world sensors

If we rely on measurements from such non-deterministic executions it could lead to flaky tests [28, 36].

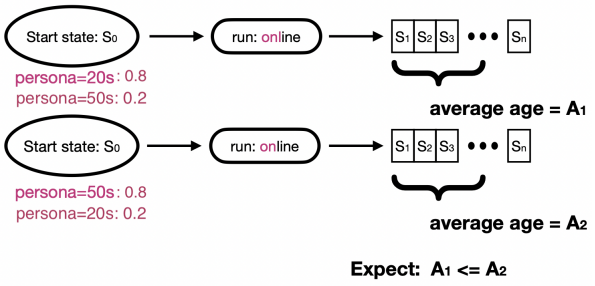

Figure 2: Two distinct runs of a simulation where we specify the proportion of users of a given age and sample the users age independently with a given probability. In the first run, the probability of a user being aged in their 50s is 0.2 and the probability of being aged in their 20s is 0.8. In the second run, users have a higher probability of being in their fifties. The average age of profiles visited during the second run is higher than the average age of visited profiles on the first run.

We also face this problem, in arguably its most pernicious form, with WW simulation outcomes, and the measurements and tests we extract from them. To continue the example of content deletion, a property test can check whether borderline content is removed from social networks. The property – deletion of content – could be checked to be *typically* fulfilled (rather than *always* fulfilled) since the classifiers might reasonably be expected to missclassify the content in a small number of scenarios. The typicality of the bots' behaviour can be assessed indirectly by analysing the frequency of the bots' actions per type.

This requires appropriate statistical tests to be properly analyzed in a sound manner [10]. Detailed surveys concerning the conduct of statistical testing can be found elsewhere [9, 27]. In the following two sections we review how these inferential statistical concepts apply to simulation measurement for cyber cyber digital twins.

3.2.1 Type I and Type II errors in Different Simulation Use Cases. In two sample tests, it is of interest to reject the null hypothesis of two probability distribution P and Q being equal. This is denoted by $H_0 : P = Q$ vs $H_1 : P \neq Q$. There are two possible types of errors when performing statistical testing: (I) reject the null hypothesis when it is true (i.e., claiming that there is a difference between two sets of outcome when actually there is none), and (II) H_0 is not rejected when it is false. The p-value of a statistical test denotes the probability of a Type I error. The significance level, α, of a test is the highest p-value that would lead to a rejection of H_0.

Naturally, a balance has to be struck between risk of committing each of these two types of error. Typically, because scientific advances ideally build on the most certain foundations, there has been a tendency to shift the balance in favour of not committing a Type I error. The ritualistic statistical behaviours this has introduced have been the subject of much discussion in the scientific community, for instance, the problem with common 'p-value fallacies' [23]. For simulation applications, this subtle balance needs to be determined differently depending on the use case sensitivity.

For example, when we are simulating whether a new feature might lead to a privacy violation, we would want an extremely high confidence before deploying the feature. For this scenario, we would much rather commit a Type II error and miss an opportunity to deploy the new feature.

However, when we are experimenting with a positive feature that may possibly reduce prevalence of harmful behaviour online, and which has no other risks, we would be much more concerned about committing a Type I error.

Clearly then, we cannot set an arbitrary threshold for the statistical confidence required of a simulation. Instead, we need to adapt the way in which we deploy inferential statistical analysis to each of the use cases to which we put simulation.

Non-deterministic behaviours, although varying between runs, typically do respect certain statistical properties of interest that can be expressed over multiple simulation runs. For example, consider Figure 2, which compares the results of two simulations according to the average age of profiles visited. Comparing the results of two or more executions is typically known as 'metamorphic' testing [17], although traditional end-to-end testing has been shown to be a special case [3].

For non-deterministic simulations, we are typically interested in statistical metamorphic testing, which caters for non-determinism and with which we can incorporate all the expressive power and conceptual framework of inferential statistical testing.

3.2.2 Parametric and non-parametric testing. There exist parametric two sample tests that rely on parametric assumptions about the data generation mechanism and non-parametric tests that do not have any distributional assumption. Parametric tests usually check whether certain moments of the two samples are the same, for example, the F-test checks for equality in variances, denoted by $H_0 : \sigma_1^2 = \sigma_2^2$ vs $H_1 : \sigma_1^2 \neq \sigma_2^2$.

Two-sample non-parametric tests directly compare the empirical observations of the probability distributions P and Q. Nonparametric tests are less prone to model miss-specification but require more samples to accurately estimate the empirical counterparts of the distributions.

In order to determine whether the two simulation outcomes can be considered to be 'equal', one also requires a divergence measure in the space of probability distributions $H_0 : D(P, Q) < \gamma$ vs $H_1 : D(P, Q) > \gamma$, where γ is a threshold that needs to be selected. Specifically, to compute the test threshold, the null distribution can be simulated via permutation or bootstrapping of the samples[6].

There are many well-known discrepancy measures between two probability distributions, such as the Kullback–Leibler divergence, the Hellinger and total variation distances, which belong to the class of f-divergences [19].

Alternatively, the class of integral probability metrics [41] has also been used to construct two sample tests, for instance, the maximum mean discrepancy [14, 46] is a kernel-based integral probability metric. Two sample tests based on metrics or divergences are available for different data types. For instance, for non-Euclidean data such as the friendship graph part of the simulation output, a two sample test has been proposed in [22]. A complete overview of both parametric and non-parametric two sampled tests is out of scope of the present paper.

In the next section, we outline our use of the Jensen-Shannon distance together with a classification approach as an illustration of how we can experiment with a statistically-based test and measurement approach. The motivation behind using a classifier approach is that a training set can be used to learn the regular variations present in the data. This helps us to account for natural simulation outcome variations and reduces the chance of wrongly rejecting the null hypothesis. The relationship between classifiers and two sample tests has been explored [16, 35]. Indeed, Kim *et al.* [31] formally demonstrated that classification accuracy is a proxy for two-sample testing.

3.3 Case Study: Using Jensen-Shannon Distance Measurement in Simulations

In this section we illustrate the use of the Jensen-Shannon distance for statistical metamorphic tests, based on the distributions of bot actions taken during simulation runs. The Jensen-Shannon distance is appropriate for more nuanced variations in behaviour. It provides a testing approach which does not require a particular deterministic property to be specified.

3.3.1 Setup. The setup involves:

- **A change request under test**: In Facebook parlance, pull requests are called 'diffs'. Diffs undergo code review in the normal continuous execution system deployed at Facebook, based on Phabricator [21].
- **A simulation story to run**: In this illustration, we use a story called `CommunityBuilder`, which creates a community of bots that then take actions within the community, such as creating posts, liking posts, sending 1-to-1 messages, uploading photos etc. At the time of writing there are over 60 individual actions in which bots can engage, a number that is regularly increasing as more simulations come on stream.
- **Selecting parameters**: Parameters include the number of users or bots, the community friendship graph, and the maximum total number of actions to execute.
- **Definition of the aspects of the simulation to be observed**: In this simple illustration, we count the total number of actions of each type that successfully executed during a simulation. This is returned as a dictionary of key-value pairs, for example `MobilePhotoUpload:23`, `ShareURL:533`, ...). We model these as drawn from an underlying multinomial distribution.
- **A set of simulation runs from which to collect observations**: These runs necessarily include those with known differences that need to be detected so that we can tune and test the test infrastructure itself. We create these differences by varying parameter choices. We also use mutation testing [29] in order to test the test system itself.

3.3.2 Statistical Distance Solution. In this section, we address how to detect anomalies in the distribution caused by bugs or by intended changes in the behaviour of the bots, compared to differences caused by "natural or regular" variations. More precisely, we want to estimate the probability that the action distributions represent a significant change in the underlying behaviour of either the platform or the community of bots.

A sample of pairs of runs, denoted by P_N, is collected from the action distribution where each run in the pair was made on the same diff with no known confounding platform effects. P_D is a sample of pairs where each run in the pair is drawn from different diffs that led to an expected difference in the action distribution between the pairs.

3.3.3 Jensen Shannon Distance (JSD). While any distance measure between two probability distributions can be used within our framework, we chose the JSD since it has some attractive features such as:

(1) JSD is simpler to implement that other distance measures which could have non-analytic forms
(2) JSD conforms to the requirements for a distance metric i.e. symmetry and triangle inequality
(3) JSD is bounded by 0.0 (the two distributions have nothing in common) and 1.0 (they are identical).

We use a frequentist approach to compute probabilities for each bot action type. That is, we count the number of each type of action successfully executed during a run, and normalise this by the total number of successful actions, thereby estimating the probability of each action. An alternative way to estimate action probabilities is to use a Bayesian method, where the the observation of each action is used to update a prior distribution (e.g. see [37], chapter 2). We could also adopt standard n-gram language modelling estimation techniques such as Laplace smoothing to include joint distributions as well as marginal distributions.

3.3.4 Threshold and Probability Estimation from Data. The JSD condenses the differences between two probability distributions into a single number. We need to set a threshold in order to determine the pass/fail outcome of the test. Alternatively, we can also calculate the probability that the distance between distributions signifies an underlying issue.

For each case, provided we have enough data in the test pair sets P_N and P_D, we proceed as follows:

(1) For each pair p_i we calculate the distance $d(p_i)$ and record whether it is a positive (P_N) or negative (P_D) example.
(2) We sort the entire list in increasing order of $d(p_i)$.
(3) Given an incoming run pair, we estimate the probability that it is a failed test by finding the proportion of negative pairs with a distance greater than or equal to $d(p_i)$.

If the list of pairs is ordered, this can be calculated directly in O(log(n)) time or compiled into a look-up table for constant-time computation. The Receiver Operator Characteristic Area Under Curve (AUC) metric[1] can be computed. The AUC gives us a single-figure summary of the power of the overall combined method setup (i.e. the simulation story, story parameters, action distributions extracted, distance measure used). The set up can be used to directly test alternatives.

When testing new product features or other counter-factual situations, we may not have sufficient existing negative run pairs from previously collected data. However, even in this data-scarce situation, there are two techniques we can employ to interpret the results from our analysis, to which we now turn.

[1] See https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc for concise description.

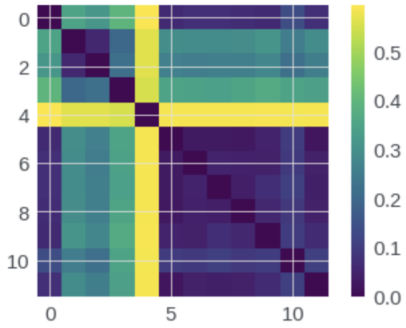

Figure 3: Visualising the JSD matrix; run number 4 broke and this clearly stands out compared to the normal background variations.

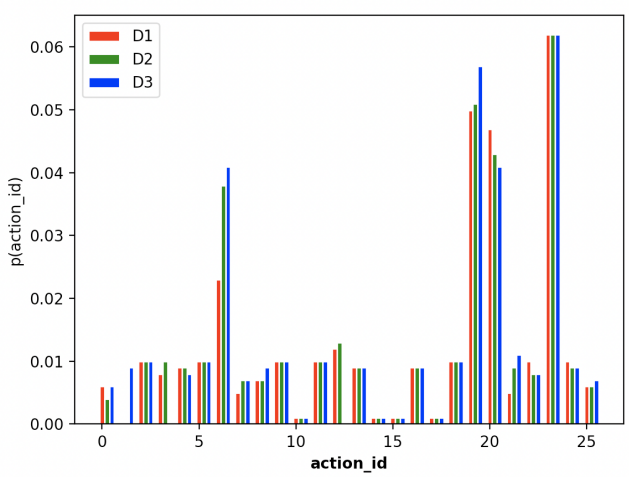

Figure 4: Histograms showing typical variations in action distributions (D1, D2) together with an anomalous run (D3). See Table 1 for distances between each of these distributions. For ease of visual inspection the most frequent action (React) is removed from these plots, but the JSD values are calculated on the true data.

3.3.5 Eyeballing Data Visualisations. For a small-scale trial across twelve individual runs (0 to 11 in the matrix in Figure 3) we computed the JSD between the action distributions of each run. Note that these were run at different times of day (hence different platform loads) and on different versions of the codebase, which are among the reasons why the action distributions naturally differ to this extent even on normally working versions of the platform.

Run number 4 clearly stands out as broken, and this was due to a problem with the Thrift RPC definitions becoming out of sync. This test can help us find and fix this kind of bug.

3.3.6 Synthetic Distributions and Differences. Another approach to gain more intuition regarding how values of JSD relate to differences in distributions is to study examples.

Figure 4 shows three different action distributions, with two being from normal runs and the third being from a run where a deliberate bug was introduced to test our method using mutation testing [29]. The mutant disables the normal follow-up actions available to a bot that has sent a 1-to-1 message.

On visual inspection it may be hard to pick out the clearly different distribution, but the differences are clearly exposed by the JSD. Table 1 shows these together with some everyday distribution pairs for comparison and their JSD values. $D_1 \ldots D_3$ are the distributions from Figure 4.

We also did an ablation test on D_1 and compared it with copies of itself but with each key:value pair removed. This synthesizes the case where a simulation runs normally, except for one type of action not functioning, and that failed function having no knock-on effects. D'_{max} is the same as D_1 except with the most frequent action set to zero, and similarly for D'_{min} (least frequent action set to zero).

The dice example is the JSD between the sum of dots when rolling two fair six-sided dice compared to a uniform distribution, and the coin example compares a fair coin to one with a biased $P(H) = 0.6$.

Table 1: JSD Examples

Pair	Distance
D_1, D_2	0.048
D_1, D_3	0.133
D_2, D_3	0.128
D_1, D'_{max}	0.43
D_1, D'_{min}	0.01
$D_{dice}, D_{uniform}$	0.22
$D_{coin}, D_{0.6}$	0.09

4 IMPLICATIONS FOR SOFTWARE TESTING

In Sections 3.1 and 3.2 we described deterministic and statistical testing of simulations, the latter using the Facebook MIA test infrastructure. MIA has proved to be useful technology for finding bugs in the simulation infrastructure, and helping us to build the WW simulation system itself. Statistical hypothesis testing provides a unified framework that caters for non-determinism over multiple runs, while retaining the traditional interpretation of a test that it either passes or fails; software testing's 'law of the excluded middle'. However, increasingly, we face the challenge of testing systems that are unavoidably stochastic in nature, as explained in Section 3.2.

Both MIA and statistical testing adopt the widely-accepted current approach to testing, used in all current testing tools and systems: a test signal is essentially pass or fail, often visualised as an unambiguous two-colour traffic light system (with red and green, for failing and passing tests respectively).

However, many underlying test outcomes are, in fact, neither red nor green but *orange*. The problems with current testing approaches lie in the following three assumptions, that the test process has to

(1) be entirely automated
(2) non interactive
(3) yield a simple boolean response. We call this assumption the 'boolean straight jacket'.

These assumptions are appropriate for unit testing, which resides at the lowest level of the test abstraction hierarchy. However, as we move up the hierarchy we encounter more expensive forms of testing and also tests with less certain outcomes. Over time, we aim to develop a more interactive approach to testing, better suited to simulation. We believe that such an approach to testing may benefit non-simulation based testing scenarios too. In this section, we outline the vision for this potentially new form of software testing. We envisage that our test technology will evolve into a bot that plays the role of a reviewer, interacting with other reviewers and code authors. In this way, test technology will yield much more nuanced and sophisticated signals than merely a simple boolean response.

To motivate the long term need to escape the boolean straight jacket, consider the current tip of the test abstraction hierarchy, where we encounter traditional end-to-end tests. End-to-end tests seek to mimic the behaviour of the whole system, as seen by a real user. At this level of testing, the test is computationally expensive, because it involves execution of the entire system. Social testing using simulation, lies at an even higher level of abstraction [2]. And it is even more computationally expensive.

As systems become increasingly non-deterministic, such high level tests also become increasingly flaky. One can remedy this by choosing an acceptable tolerance for Type I errors. However, it is not clear what this 'sensible' significance level should be *a priori*. The test flakiness problem has been highlighted previously, in both research [36] and practice [28, 40]. The research community has attempted to overcome the test flakiness problem by identifying, controlling and removing flaky tests [39]. However, an alternative is to *live with* a world in which "*all tests are assumed to be flaky*" [28]. This shift in perspective allows us to escape the boolean straight jacket. Such a non-boolean testing approach is clearly better suited to measurement challenges set out in the previous section. We also believe it may be forced on the wider software engineering community by increased non-determinism, whether or not the system under test is a simulation.

The boolean straight jacket is problematic from both a practical and a theoretical point of view:

- **Theoretical problems with the boolean straight jacket**: From a theoretical standpoint, the boolean straight jacket transforms an enormous amount of information into a single bit (pass/fail). It means that all testing systems have the worst possible domain to range ratio [8, 52], an observation which, perhaps, accounts for the difficulty of testing the test infrastructure itself; high domain to range ratio programs are the least testable [52]. From an information theoretic perspective [47], the boolean straitjacket also denotes a huge loss of information; why throw away all of that valuable test signal, especially when obtaining it was computationally expensive?
- **Practical problems with the boolean straight jacket**: From a practical perspective, there are mounting challenges in tackling test flakiness that come from unavoidably increasing non-determinism. Simulation resides at the vanguard of this trend, because it involves users' behaviour, which is typically highly stochastic. However, other software technologies, unrelated to simulation, are increasingly interconnected are thereby becoming increasingly non-deterministic. For instance, some of the first automated web based testing tools already encountered this problem; even the weather can affect test coverage achieved [7]. Taking account of this non-determinism in a testing process inherently involves making context sensitive and well-informed decisions about statistical thresholds, that balance false positives and false negatives. One cannot simply determine a threshold up front and adopt a 'one size fits all approach'.

Of course, it is undoubtedly heartening to receive a test signal which simply gives a green light or a red light. The old-fashioned testing approach is simple, intuitive, reassuring when green, and hopefully actionable when red. Sadly, as any practising software engineer will attest, the green signal can so easily give a false sense of security. Arguably worse, the red signal may prove to be a false positive, wasting developer effort on some of the most tedious and frustrating intellectual activities known to humankind.

Any test infrastructure that survives and thrives while respecting the boolean straight jacket can do so only by significantly reducing the chance of wasted developer effort. A test infrastructure that fails to achieve this will tend to be weeded out by natural selection; developers will simply abandon or ignore a system that wastes too much of their time.

As a result of this underlying evolutionary adoption process, the most pernicious effect of the boolean straight jacket lies not the wasted developer effort itself, but rather, it lies in the potentially valuable test signal that we lose by attempting to ensure developer effort is not wasted. In the presence of highly non-deterministic execution environments, testing systems resort to passing on a red signal to a developer, *only* when there is a high degree of certainty. What about those situations where the signal indicates there *may* be some problem, but an automated decision needs to be taken *not* to bother the developer?

4.1 Foggy Traffic Light Testing

What if we could give the developer a *foggy* traffic light signal instead of a crisp one?

Of course there is a lot to be sacrificed by migrating from sharp to foggy traffic lights, and this is why it has proved difficult to achieve. However, we need to rethink the fundamentals of testing and in particular, the software engineering workflows in which testing is deployed. Failure to do so will result in testing becoming ever more expensive, while simultaneously yielding ever poorer assurance of software correctness.

How can we go further than simply replacing a clear (actionable) signal with a foggy (vague and unactionable) signal?

The answer lies in code review: Instead of thinking of testing as a fully automated process that acts as a final gatekeeper on deployment, we should think of it as an extra member of the code review team, in the sense of modern code review [11]. Ever since the introduction of Fagin inspections [20], code review has proved to be one of *the most effective* practical techniques available for ensuring code quality and correctness [45].

Modern code review provides infrastructural support, whereby many software engineering tools become bots themselves [49]. However, despite increasing sophistication of test technology, the automated test tool is typically *not* thought of as an interactive bot.

Suppose we think of the automated test tool as a fully *interactive* member of the review team that comments on code changes, just as a human reviewer does. Such a 'test tool as a team member' approach shares its motivation with related approaches such as the automated statistician [48] and the robot scientist [32].

Let us even give this testing tool a name, 'Pat', to respect the fact that Pat truly is part of the team. Pat could be a Three-Letter Acronym for 'Practical Automated Tester', but let us anthropomorphise further, and imagine that Pat is a real person. What does Pat bring to the overall code review process? Quite a lot, it turns out. Pat is an exceptionally gifted member of the team, who never complains. Pat always responds, in a timely fashion, to questions and follow-up requests from other reviewers and the author of the proposed code change. Let us review Pat's unique attributes:

(1) **Anytime follow up**: Pat knows they have to respond quickly. If they are not the first one to comment on the diff, then the other human reviewers might accept, or reject based on supposition, bias or misinformation. This time pressure means that Pat typically cannot perform all the experiments they would like to undertake. Instead, Pat has to comment in a limited time window in order to ensure that they are a first responder. This time pressure forces Pat to choose a prioritised subset of test signal with which to initially comment. Fortunately, this subset prioritisation problem has been tackled in an extensive literature on test case selection and prioritisation, so there are many well-studied algorithms and techniques [34, 44] from which Pat can choose.
Furthermore, Pat is always willing, if asked by human reviewer, to "go away and perform extra experiments and checks, and come back to the review process with extra signal". Automated testing is an 'any time' algorithm; with additional computational resources we can run more test cases. Notice how making Pat and interactive reviewer, like a human reviewer, creates a natural and intuitive interface to help us efficiently find the necessarily context-aware balances required by test case selection and prioritisation problems [55].
In a foggy traffic light test system, Pat would offer a Service Level Agreement (SLA) in which their initial comment would arrive before that of any human tester. To achieve their SLA, Pat would adapt their test resources to compute this initial signal, based on observation of current human reviewer response times. When Pat's initial signal causes doubt as to the software correctness among the human reviewers, they can ask Pat to perform further testing. This is a process well-suited to a non-deterministic world; get the initial signal, make a human judgement, potentially ask for further signal, or decide to go ahead and deploy the code change anyway. Making Pat a natural part of the normal code review process creates natural and intuitive workflows that enable testing to be iterative and interactive.

(2) **Diligent Experimenter**: Pat can perform many tedious experiments to understand the precise implications of a code change. If one of the human reviewers requires Pat to do so, then Pat can perform experiments to provide follow-up signal, answering reviewer questions such as:
   (a) **Unrelated infra failures**: Could the test failures Pat reported be the result of recent (apparently) unrelated changes that landed into the code base? (Pat goes away to experimentally and locally revert the apparently unrelated changes and re-run the tests to check)
   (b) **Increased confidence and reduced flakiness**: could we wait a moment while Pat checks (apparently) flaky signal regarding property P for more evidence? (Pat goes away and runs further tests to gain greater statistical confidence wrt P)
   (c) **Counterfactual experimenter**: What if failing test T were to be run in an initial state where property P does not hold? Would it still fail (Pat goes away and checks). This ability to do counterfactual experimentation could form a natural jumping off point from testing into debugging. Notice how making Pat a team member establishes a very natural and seamless test-and-debug *blend*, rather than enforcing an arbitrary and abrupt step change (that would otherwise have come from the boolean straight jacket). Debugging technology has evolved little in the last 40 years; it is time for a change. Pat can help.
   (d) **Popperian Scientist**: Pat's follow-up need not focus only on deep diving on *failing* test signal. Pat can also play the role of 'Popperian scientist' [43]; attempting to falsify the claim that the software is correct, perhaps initiated by the doubts expressed by a human reviewer.
   For example, suppose the human reviewer feels queasy about a particular aspect of the system, perhaps due to a code smell [54], or maybe simply that awkward feeling that a corner case has been missed.
   Currently, the anxious human reviewer may have to do a lot of work to follow up. Depending on their temperament, they will either reject the code with too little explanation, or accept it while "holding their nose". How much misunderstanding and avoidable engineer conflict could have been averted if we had only had a *third* member of the team who *was* willing devote that extra effort to investigate apparently green signal? (Pat goes off and runs more tests on aspects with currently passing tests, in an to attempt to find a failing case and reports back).
(3) **Final Gate Keeper**: When Pat is sure that the code changes are incorrect, they can dig in their heels by demanding a fix. In this regard, adding Pat to the review team discards *none* of the existing properties of testing systems; foggy traffic light testing *subsumes* sharp traffic light testing. However, once again, there are interesting scientific challenges. How can Pat learn, from historical data, which parts of the code base are more likely to fail given the change introduced in the new code? This is a question already tackled by work on fault prediction [15, 25].

What human member of the review team is prepared to go to such lengths to help support the code review process? Why, for so long, have we ignored the very special characteristics a team member like Pat could bring to our team? Pat has so much to tell us, yet we are currently forcing this exceptionally gifted team member to forget all they know and to simply give us a pure and simple 'yes' or 'no' answer. As we all know, "the truth is rarely pure and never simple" [53].

Software testing workflows have to change to accommodate Pat. Ultimately, with further research and development, Pat may evolve beyond the confines of testing, to become a software engineer, able to suggest repairs [24, 38, 51], to recommend code transplants [12, 57], improvements and optimisations [42], and even to explore new features [26]. The first step on this exciting journey would be to make Pat a proper member of the review team, as a way to adapt software testing for a highly non-deterministic world.

5 OPEN CHALLENGES FOR THE SCIENTIFIC COMMUNITY

- **Non-boolean testing**: As outlined in Section 4, we need to rethink software testing for a world in which the developer gets recommendations during code review rather than merely a boolean pass or fail. Section 4 is part proposal and part polemic, setting out a possible vision of future software testing technology as a full member of the code review team, rather than it being merely a final gatekeeper. Much more work is required to realise this vision. Tackling it will undoubtedly surface many interesting scientific questions and challenges.
- **Statistical metamorphic testing**: As outlined in Section 3.2, we need techniques to incorporate statistical inference into the simulation system itself, the measurements we take from it, and the decisions and optimisations it recommends. Current deployment of A/B testing in many organisations [33] already involves sophisticated inferential statistical analysis, to support decision-making, based on the outcomes of the A/B test. We anticipate simulation will require similar statistical sophistication. However, the paradigm is more challenging than A/B testing, because simulation typically encompasses counterfactual scenarios.
- **Trade offs between simulation speed and precision**: For offline simulation modes [4], in which the simulation is not run directly on the real infrastructure, we need techniques to understand the trade-off between speed and fidelity of measurement.
- **Scalable fitness computation**: Measurements from simulation can used to guide optimisation, such as for mechanism design [2]. Mechanism design has huge potential to turn simulation into a technique for automated product improvement, using techniques such as genetic improvement[42]. However, the computational expense of simulation will raise scalability challenges. To tackle scalability, we need techniques that more closely integrate the optimisation algorithm with the simulation-based measurement process.
  We need a kind of 'lazy fitness computation', which would perform as much of the simulation as required, but no more, in order to determine an actionable fitness value. For earlier exploratory phases of the optimisation process we may need faster lightweight simulation, possibly using offline modes. For later optimisation stages in which exploitation of promising solution spaces requires higher fidelity, we will deploy more computationally expensive fitness computation, that can yield this higher fidelity.
  Methods such as early stopping are applicable here and are widely used in automated hyper-parameter optimisation for training neural networks. They aim to make the most of the available computation budget by terminating runs that are unlikely to provide good solutions [56].
  How ever we choose to tackle these scalability challenges, it is clear that the optimisation process cannot treat simulation merely as a black box that delivers fitness, but needs to have a white box approach to simulation in order to scale.

ACKNOWLEDGEMENTS

Author order is alphabetical. Mark Harman's scientific work is part supported by European Research Council (ERC), Advanced Fellowship grant number 741278; Evolutionary Program Improvement (EPIC) which is run out of University College London, where he is part time professor. He is a full time Research Scientist at Facebook. Simon Lucas is currently a full time Research Scientist at Facebook and also a part time professor at Queen Mary University of London.

REFERENCES

[1] David Adam. 2020. Special report: The simulations driving the world's response to COVID-19. *Nature* (April 2020).
[2] John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Ralf Laemmel, Erik Meijer, Silvia Sapora, and Justin Spahr-Summers. 2020. WES: Agent-based User Interaction Simulation on Real Infrastructure. In *GI @ ICSE 2020*, Shin Yoo, Justyna Petke, Westley Weimer, and Bobby R. Bruce (Eds.). ACM, 276–284. https://doi.org/doi:10.1145/3387940.3392089 Invited Keynote.
[3] John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Maria Lomeli, Erik Meijer, Silvia Sapora, and Justin Spahr-Summers. 2021. Testing Web Enabled Simulation at Scale Using Metamorphic Testing. In *International Conference on Software Engineering (ICSE) Software Engineering in Practice (SEIP) track*. Virtual.
[4] John Ahlgren, Kinga Bojarczuk, Sophia Drossopoulou, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Maria Lomeli, Simon Lucas, Erik Meijer, Steve Omohundro, Rubmary Rojas, Silvia Sapora, Jie M. Zhang, and Norm Zhou. 2021. Facebook's Cyber–Cyber and Cyber–Physical Digital Twins. In *25th International Conference on Evaluation and Assessment in Software Engineering (EASE 2021)*. Virtual.
[5] Saif Al-Sultan, Moath M. Al-Doori, Ali H. Al-Bayatti, and Hussien Zedan. 2014. A comprehensive survey on vehicular Ad Hoc network. *Journal of Network and Computer Applications* 37 (2014), 380 – 392.
[6] V. Alba Fernández, M.D. Jiménez Gamero, and J. Muñoz García. 2008. A test for the two-sample problem based on empirical characteristic functions. *Computational Statistics and Data Analysis* 52, 7 (2008), 3730–3748. https://doi.org/10.1016/j.csda.2007.12.013
[7] Nadia Alshahwan and Mark Harman. 2011. Automated Web Application Testing Using Search Based Software Engineering. In 26^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. Lawrence, Kansas, USA, 3 – 12.
[8] Kelly Androutsopoulos, David Clark, Haitao Dan, Mark Harman, and Robert Hierons. 2014. An Analysis of the Relationship between Conditional Entropy and Failed Error Propagation in Software Testing. In 36^{th} *International Conference on Software Engineering (ICSE 2014)*. Hyderabad, India, 573–583.
[9] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In 33^{rd} *International Conference on Software Engineering (ICSE'11)* (Waikiki, Honolulu, HI, USA). ACM, New York, NY, USA, 1–10.
[10] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing,*

Verification and Reliability 24, 3 (2014), 219–250. https://doi.org/10.1002/stvr.1486 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1486

[11] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 712–721.

[12] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*. 257–269.

[13] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (May 2015), 507–525.

[14] Karsten M. Borgwardt, Arthur Gretton, Malte J. Rasch, Hans-Peter Kriegel, Bernhard Schölkopf, and Alex J. Smola. 2006. Integrating structured biological data by Kernel Maximum Mean Discrepancy. *Bioinformatics* 22, 14 (07 2006), e49–e57. https://doi.org/10.1093/bioinformatics/btl242 arXiv:https://academic.oup.com/bioinformatics/article-pdf/22/14/e49/616383/btl242.pdf

[15] David Bowes, Tracy Hall, Mark Harman, Yue Jia, Federica Sarro, and Fan Wu. 2016. Mutation-Aware Fault Prediction. In *International Symposium on Software Testing and Analysis (ISSTA 2016)*. 330–341.

[16] Haiyan Cai, Bryan Goggin, and Qingtang Jiang. 2020. Two-sample test based on classification probability. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 13, 1 (2020), 5–13. https://doi.org/10.1002/sam.11438 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/sam.11438

[17] Tsong Yueh Chen, Jianqiang Feng, and T. H. Tse. 2002. Metamorphic Testing of Programs on Partial Differential Equations: A Case Study. In 26^{th} *Annual International Computer Software and Applications Conference (COMPSAC'02)*. IEEE Computer Society, 327–333.

[18] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279.

[19] I. Csiszar. 1967. Information-type measures of difference of probability distributions and indirect observation. *Studia Scientiarum Mathematicarum Hungarica* 2 (1967), 229–318. https://ci.nii.ac.jp/naid/10028997448/en/

[20] Michael E. Fagan. 1976. Design and code inspections to reduce errors in code development. *IBM Systems Journal* 15, 3 (1976), 182–211.

[21] Dror G. Feitelson, Eitan Frachtenberg, and Kent L. Beck. 2013. Development and Deployment at Facebook. *IEEE Internet Computing* 17, 4 (2013), 8–17.

[22] Han Feng, Xing Qiu, and Hongyu Miao. 2021. Hypothesis Testing for Two Sample Comparison of Network Data. arXiv:2106.13931 [stat.ME]

[23] Steven Goodman. 2008. A dirty dozen: twelve p-value misconceptions. In *Seminars in hematology*, Vol. 45. Elsevier, 135–140.

[24] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current Challenges in Automatic Software Repair. *Software Quality Journal* 21, 3 (2013), 421–443.

[25] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2012. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering* 38, 6 (2012), 1276–1304.

[26] Mark Harman, William B. Langdon, and Yue Jia. 2014. Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system. In 6^{th} *Symposium on Search Based Software Engineering (SSBSE 2014)*. Springer LNCS, Fortaleza, Brazil, 247–252.

[27] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. 2012. Search Based Software Engineering: Techniques, Taxonomy, Tutorial. In *Empirical software engineering and verification: LASER 2009-2010*, Bertrand Meyer and Martin Nordio (Eds.). Springer, 1–59. LNCS 7007.

[28] Mark Harman and Peter O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis (keynote paper). In 18^{th} *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2018)*. Madrid, Spain, 1–23.

[29] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (September–October 2011), 649 – 678.

[30] Gregory L Johnson, Clayton L Hanson, Stuart P Hardegree, and Edward B Ballard. 1996. Stochastic weather simulation: Overview and analysis of two commonly used models. *Journal of Applied Meteorology* 35, 10 (1996), 1878–1896.

[31] Ilmun Kim, Aaditya Ramdas, Aarti Singh, and Larry Wasserman. 2021. Classification accuracy as a proxy for two-sample testing. *The Annals of Statistics* 49, 1 (2021), 411 – 434. https://doi.org/10.1214/20-AOS1962

[32] Ross D. King, Kenneth E. Whelan, Ffion M. Jones, Philip G. K. Reiser, Christopher H. Bryant, Douglas B. Kell Stephen H. Muggleton, and Stephen G. Oliver. 2004. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature* (01 2004), 247–252.

[33] Ron Kohavi and Roger Longbotham. 2017. Online Controlled Experiments and A/B Testing. *Encyclopedia of machine learning and data mining* 7, 8 (2017), 922–929.

[34] Zheng Li, Mark Harman, and Rob Hierons. 2007. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering* 33, 4 (2007), 225–237.

[35] David Lopez-Paz and Maxime Oquab. 2017. Revisiting Classifier Two-Sample Tests. In *ICLR*.

[36] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In 22^{nd} *International Symposium on Foundations of Software Engineering (FSE 2014)*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne Storey (Eds.). ACM, Hong Kong, China, 643–653.

[37] David J. C. MacKay. 2002. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, USA.

[38] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. SapFix: Automated End-to-End Repair at Scale. In *International Conference on Software Engineering (ICSE) Software Engineering in Practice (SEIP) track*. Montreal, Canada.

[39] Atif M. Memon and Myra B. Cohen. 2013. Automated testing of GUI applications: models, tools, and controlling flakiness. In 35^{th} *International Conference on Software Engineering (ICSE 2013)*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, San Francisco, CA, USA, 1479–1480.

[40] Atif M. Memon, Zebao Gao, Bao N. Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. In 39^{th} *International Conference on Software Engineering, Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, Buenos Aires, Argentina, 233–242.

[41] Alfred Müller. 1997. Integral Probability Metrics and Their Generating Classes of Functions. *Advances in Applied Probability* 29, 2 (1997), 429–443. http://www.jstor.org/stable/1428011

[42] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (June 2018), 415–432. https://doi.org/doi:10.1109/TEVC.2017.2693219

[43] Karl R. Popper. 1959. The logic of scientific discovery. London: Hutchinson and Co. (Publishers) 480 p. (1959)..

[44] Gregg Rothermel, Roland Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing Test Cases For Regression Testing. *IEEE Transactions on Software Engineering* 27, 10 (Oct. 2001), 929–948.

[45] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. 181–190.

[46] Dino Sejdinovic, Arthur Gretton, Bharath Sriperumbudur, and Kenji Fukumizu. 2012. Hypothesis testing using pairwise distances and associated kernels (with Appendix). *Proceedings of the 29th International Conference on Machine Learning, ICML 2012* 2 (05 2012).

[47] Claude Elwood Shannon. 1948. A Mathematical Theory of Communication. *Bell System Technical Journal* 27 (July and October 1948), 379–423 and 623–656. http://cm.bell-labs.com/cm/ms/what/shannonday/paper.html,http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.ps.gz,http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf,http://djvu.research.att.com/djvu/sci/shannon/index.html

[48] Christian Steinruecken, Emma Smith, David Janz, James Lloyd, and Zoubin Ghahramani. 2019. *The Automatic Statistician*. Springer International Publishing, Cham, 161–173. https://doi.org/10.1007/978-3-030-05318-5_9

[49] Margaret-Anne D. Storey and Alexey Zagalsky. 2016. Disrupting developer productivity one bot at a time. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE 2016), Seattle, WA, USA, November 13-18, 2016*. ACM, 928–931.

[50] Sergio Terzi and Sergio Cavalieri. 2004. Simulation in the supply chain context: a survey. *Computers in Industry* 53, 1 (2004), 3–16.

[51] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. 2018. How to Design a Program Repair Bot? Insights from the Repairnator Project. In *40th International Conference on Software Engineering, Software Engineering in Practice track (ICSE 2018 SEIP track)*. 1–10.

[52] Jeffrey M. Voas and Keith W. Miller. 1995. Software Testability: The New Verification. *IEEE Software* 12, 3 (May 1995), 17–28.

[53] Oscar Wilde. 1895. *The Importance of Being Earnest*.

[54] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey. In *2013 20th working conference on reverse engineering (WCRE)*. IEEE, 242–251.

[55] Shin Yoo and Mark Harman. 2012. Regression Testing Minimisation, Selection and Prioritisation: A Survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[56] Tong Yu and Hong Zhu. 2020. Hyper-Parameter Optimization: A Review of Algorithms and Applications. arXiv:2003.05689 [cs.LG]

[57] Tianyi Zhang and Miryung Kim. 2017. Automated transplantation and differential testing for clones. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 665–676.