Leveraging metamorphic testing to automatically detect inconsistencies in code generator families

Mohamed Boussaa, Olivier Barais, Gerson Sunyé, Benoit Baudry

SPECIAL ISSUE PAPER

Leveraging metamorphic testing to automatically detect inconsistencies in code generator families

Mohamed Boussaa[1,*,†], Olivier Barais[2], Gerson Sunyé[3] and Benoit Baudry[4]

[1]*School of Computer Science, McGill University, Montréal, Canada*
[2]*IRISA-INRIA, University of Rennes 1, Rennes, France*
[3]*LS2N, University of Nantes, Nantes, France*
[4]*CASTOR, KTH Royal Institute of Technology, Stockholm, Sweden*

SUMMARY

Generative software development has paved the way for the creation of multiple code generators that serve as a basis for automatically generating code to different software and hardware platforms. In this context, the software quality becomes highly correlated to the quality of code generators used during software development. Eventual failures may result in a loss of confidence for the developers, who will unlikely continue to use these generators. It is then crucial to verify the correct behaviour of code generators in order to preserve software quality and reliability.

In this paper, we leverage the metamorphic testing approach to automatically detect inconsistencies in code generators via so-called "metamorphic relations". We define the metamorphic relation (i.e., test oracle) as a comparison between the variations of performance and resource usage of test suites running on different versions of generated code. We rely on statistical methods to find the threshold value from which an unexpected variation is detected. We evaluate our approach by testing a family of code generators with respect to resource usage and performance metrics for five different target software platforms. The experimental results show that our approach is able to detect, among 95 executed test suites, 11 performance and 15 memory usage inconsistencies. © 2019 John Wiley & Sons, Ltd.

Received 30 November 2018; Revised 23 August 2019; Accepted 8 October 2019

KEY WORDS: code generators; metamorphic testing; non-functional properties; software quality; test automation; test oracle

1. INTRODUCTION

The intensive use of generative programming techniques has become a common practice in software development to deal with the heterogeneity of platforms and technological stacks that exist in several domains such as mobile or Internet of Things [1]. Generative programming [2] offers a software abstraction layer that software developers can use to specify the desired system behaviour (e.g., using domain-specific languages, models, etc.) and to automatically generate software artifacts to different platforms. As a consequence, multiple code generators are used to transform the specifications/models represented in a graphical/textual languages to general-purpose programming languages such as C, Java, C++, etc.

Code generators have to respect different requirements that preserve not only the functional properties but also the reliability and quality of delivered software. As long as the quality of generators is maintained and improved, the quality of generated software artifacts also improves. Any issue

*Correspondence to: Mohamed Boussaa, School of Computer Science, McGill University, Montréal, Canada.
†E-mail: mohamed.boussaa@mail.mcgill.ca

with the generated code leads to a loss of confidence in generators, and users will unlikely continue to use them during software development. Defective (or "nonmature") code generators can generate defective software artifacts that range from uncompilable or semantically dysfunctional code that causes serious damage to the generated software to non-functional issues that lead to poor quality code that can affect system reliability and performance (e.g., high resource usage, low execution speed, etc.). As a consequence, checking the correctness and the quality of generated code has to be done with almost the same expensive effort as it is needed for the manually written code in order to ensure the correct behaviour of delivered software.

Nevertheless, code generators are known to be difficult to understand because they involve a set of complex and heterogeneous technologies that make the activities of design, implementation and testing very hard and complex [3, 4]. In addition, the non-functional testing of code generators remains a challenging and time-consuming task because developers have to analyse and verify the non-functional behaviour of generated code for different target platforms using platform-specific tools (e.g., debuggers, profilers, monitoring tools, etc.) [5, 6]. The problem becomes more critical when automating the non-functional tests. Particularly, test automation raises the oracle problem because there is no clear definition of how the oracle might be defined when evaluating the non-functional properties (e.g., execution speed, utilization of resources, etc.). Proving that the generated code respects one of these non-functional requirements is difficult because there is no reference implementation.

To alleviate the oracle problem, numerous approaches have been proposed [7, 8] to automatically verify the functional outcome of generated code by applying techniques such as back-to-back or differential testing [9, 10]. However, there is a lack of solutions that pay attention to automatically evaluate the properties related to the performance and resource usage of generated code.

In this paper, we leverage the metamorphic testing approach to alleviate the oracle problem in the context of non-functional testing of code generators. We define the metamorphic relation (i.e., test oracle) as a comparison between the variations of performance and resource usage of test suites running on different versions of generated code. We rely on statistical methods to find the threshold value from which an unexpected variation is detected. Compared with the original work by Boussaa et al. [11], published in the International Conference on Generative Programming: Concepts & Experiences (GPCE 2016), where they manually identify code generator issues, this paper presents a fully automatic approach based on metamorphic testing to detect unexpected behaviours (or inconsistencies). We evaluate our approach by analysing the performance of Haxe, a popular high-level programming language that involves a family of code generators. We evaluate the properties related to the resource usage and performance for five different target software platforms. The experimental results show that our approach is able to detect, among 95 executed test suites, 11 performance and 15 memory usage inconsistencies, violating the metamorphic relation. This work is based on the results of the first author's PhD thesis [12].

The contributions of this paper are the following:

- We describe our adaptation of the metamorphic approach to the problem of automatic non-functional testing of code generators. We also describe two statistical techniques that are applied to define the metamorphic relation. This contribution addresses mainly the oracle problem when automating the non-functional tests.
- We propose an elastic and reproducible testing environment, based on a virtualized infrastructure, to ensure the deployment, execution and monitoring of generated code. This contribution addresses the problem of resource usage monitoring in heterogeneous environments.
- We also report the results of an empirical study by evaluating the non-functional properties of Haxe code generators. The obtained results provide evidence to support the claim that our proposed approach is able to automatically detect real issues in code generator families.

The paper is organized as follows. Section 2 presents the context and motivations behind this work. In particular, we discuss three motivation examples and the challenges we are facing. Section 3 presents an overview of our approach. In particular, we present our metamorphic testing approach and the infrastructure used to ensure the automatic non-functional testing of code generators.

The evaluation and results of our experiments are discussed in Section 4. Finally, related work, concluding remarks and future work are provided in Sections 5 and 6.

2. CONTEXT AND MOTIVATIONS

2.1. Code generator families

Today, many customizable code generators are used to easily and efficiently generate code for different software platforms, programming languages, operating systems, etc. This work is based on the intuition that a code generator is often a member of a family of code generators [13].

Definition 1 (Code generator family)
We define a code generator family as a set of code generators that takes as input the same language/model and generate code for different target software platforms.

For example, this concept is widely used in industry when applying the "write once, run everywhere" paradigm. Users can benefit from a family of code generators (e.g., cross-platform code generators [14]) to generate from the manually written (high-level) code, different implementations of the same program in different languages. This technique is very useful to address diverse software platforms and programming languages. As motivating examples for this research study, we can cite three approaches that intensively develop and use code generator families:

2.1.1 Haxe. Haxe[‡] [15] is an open source toolkit for cross-platform development, which compiles to a number of different programming platforms, including JavaScript, Flash, PHP, C++, C# and Java. Haxe involves many features: the Haxe language, multiplatform compilers and different native libraries. The Haxe language is a high-level programming language that is strictly typed. This language supports both functional and object-oriented programming paradigms. It has a common type hierarchy, making certain API available on every target platform. Moreover, Haxe comes with a set of code generators that translate the manually written code (in Haxe language) to different target languages and platforms. This project is popular (more than 1940 stars on GitHub).

2.1.2. ThingML. ThingML[§] is a modelling language for embedded and distributed systems [16]. The idea of ThingML is to develop a practical model-driven software engineering tool-chain that targets resource-constrained embedded systems such as low-power sensors and microcontroller-based devices. ThingML is developed as a domain-specific modelling language that includes concepts to describe both software components and communication protocols. The formalism used is a combination of architecture models, state machines and an imperative action language. The ThingML tool-set provides a code generator family that translates ThingML-based programs to C, Java and JavaScript. It includes as well a set of variants for the C and JavaScript code generators to support specific embedded systems with resource constraints.

2.1.3. TypeScript. TypeScript[¶]is a typed superset of JavaScript that compiles to plain JavaScript [17]. In fact, it does not compile to only one version of JavaScript. It can transform TypeScript to EcmaScript 3, 5 or 6. It can generate JavaScript that uses different system modules ("none", "commonjs", "amd", "system", "umd", "es6" or "es2015")[‖]. This project is popular (more than 23 750 stars on GitHub).

Functional testing of a code generator family is straightforward. Because the produced programs are generated form the same input program, the oracle can be defined as a comparison between their functional outputs, which should be the same. This is commonly known as differential or back-to-back testing [9, 10]. In fact, based on the three sample projects presented above, we remark

[‡]http://haxe.org/
[§]http://thingml.org/
[¶]https://www.typescriptlang.org/
[‖]Each of this variation point can target different code generators (e.g., function *emitES6Module* vs. *emitUMDModule* in emitter.ts).

that all GitHub code repositories of the corresponding projects use pass/fail unit tests to check the correctness of the generated code for the target software platform.

In terms of non-functional tests, we observe that ThingML and TypeScript do not provide any specific tests to check the consistency of code generators in terms of non-functional properties. Haxe provides two test cases[**] to benchmark the resulting generated code. One serves to benchmark an example in which object allocations are deliberately (over) used to measure how memory access/GC mixes with numeric processing in different target languages. The second test evaluates the network speed across different target platforms.

2.2. Challenges when testing the non-functional properties

The automatic non-functional testing of code generators raises different challenges. In the following, we discuss some of them.

2.2.1. The oracle problem. To automate the testing process, test oracles are required to assess whether a test passes or fails. However, test oracles are not always available and may be hard to define or too difficult to apply [18]. When it comes to testing the non-functional properties such as the resource usage or execution speed, this problem becomes more critical. In fact, there is no clear definition about how the oracle should be defined except the few research efforts [19, 20], where the generated code performance is compared with the handwritten code as a reference. The main difficulty when testing the generated code is that we cannot just observe the execution of produced code, but we have to observe and compare the execution of generated programs with equivalent (or reference) implementations (e.g., in other languages). In the absence of any specification or reference implementation, the automatic detection of non-functional issues becomes impossible.

2.2.2. Software platforms diversity. Automating the testing process of code generators requires many system configurations (i.e., execution environments, libraries, compilers, etc.) to efficiently generate and test the code. For example, the generated software artifacts, for example, written in different languages such as Java, C#, C++, etc., have to be compiled, deployed and executed across different target platforms, for example, Android, ARM/Linux, JVM, x86/Linux, etc. Setting up the testing environment and running tests across multiple software platforms and machines become then very tedious and time-consuming. In the meantime, to evaluate the memory or CPU usage for example, developers have to collect and visualize the resource usage data of running tests across different platforms. This requires several platform-specific profilers, trackers, instrumenting and monitoring tools in order to find some inconsistencies or bugs during code execution [5, 6]. Finding inconsistencies within code generators involve analysing and inspecting the code for each execution platform. For example, one way to handle that is to analyse the memory footprint of software execution and find memory leaks [21]. Developers can then inspect manually the generated code and find some fragments of the codebase that have triggered this issue. In short, evaluating the quality of generated code (e.g., in terms of resource usage and performance) is a manual and hard task. It requires many technologies and system configurations to handle the diversity of software and hardware platforms.

3. APPROACH OVERVIEW

The key objective of our approach is to address the challenges discussed in Section 2.2. Our contribution in this work is twofold:

- We first describe our testing infrastructure based on a virtualized environment to provide effective support to conduct the non-functional testing of code generators. This contribution addresses the problem of software diversity and resource monitoring.
- Second, we present a methodology, based on metamorphic testing, to automatically detect inconsistencies in code generator families. This approach addresses the oracle problem when testing the non-functional properties.

[**] https://github.com/HaxeFoundation/haxe/tree/development/tests/benchs

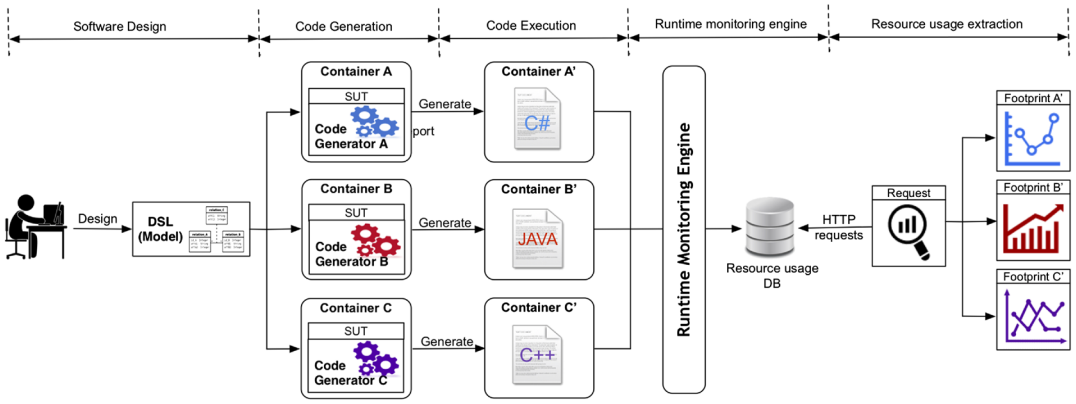


Figure 1. An overview of the virtualized infrastructure used to ensure the automatic code generation and resource usage monitoring of produced code.

3.1. A virtualized infrastructure for code generators testing and monitoring

As discussed earlier, evaluating the resource usage of automatically generated code is complex because of the software platforms diversity that exist in the market. One way to overcome this problem is to use the virtualization technology. We aim to benefit from the recent advances in lightweight system virtualization, in particular container-based virtualization [22], in order to offer effective support for automatically deploying, executing and monitoring the generated code in heterogeneous environment. This technology enables to mimic the execution environment settings and reproduce the tests in isolated and highly configurable system containers. When it comes to evaluating the resource consumptions of automatically generated code, this technology becomes very valuable because it allows a fine-grained resource management and isolation. Moreover, it facilitates resource usage extraction and limitation of programs running inside containers.

3.1.1. System containers as a lightweight execution environment. Instead of running tests on multiple machines and environment settings, we rely on system containers as a dynamic and configurable execution environment for running and evaluating the generated programs in terms of resource usage. System containers are operating system-level virtualization method that allows running multiple isolated Linux systems on a control host using a single Linux kernel. Container-based virtualization reduces the overhead associated with having each guest running a new installed operating system such the case for virtual machines [23]. For instance, Docker[††] is a popular container-based technology that automates the deployment of any application as a lightweight, portable and self-sufficient container, running virtually on a host machine [24, 25]. Using Docker, it is possible to define preconfigured applications and servers to host as virtual images. It also defines the way the service should be deployed in the host machine using configuration files called Dockerfiles. We use then this technology to: (i) configure code generators inside different containers, where we install all the libraries, compilers and dependencies needed to ensure the code generation and compilation for the target platform (see code generation in Figure 1) and (ii) run tests in different container instances dedicated to specific software platforms (see code execution in Figure 1). We resume the main advantages of this approach as follows:

- The use of containers induces little to near zero performance overhead compared with the full stack virtualization solution [23]. Indeed, instrumentation and monitoring tools for memory profiling like Valgrind [21] can induce too much overhead.
- Thanks to the use of Dockerfiles, it is possible to easily configure the execution environment in order to build and customize applications using numerous settings (e.g., generator version, dependencies, host IP and OS, optimization options, etc.). Thus, we can use the same

[††]https://www.docker.com

configured Docker image to execute different instances of the same application. For hardware architecture, containers share the same platform architecture as the host machine (e.g., x86, x64, ARM, etc.).
- Although containers run in isolation, they can share data with the host machine and other running containers. Thus, non-functional data relative to resource consumption can be gathered and managed by other containers (i.e., for storage purpose, visualization)

3.1.2. Runtime monitoring engine. In order to monitor the applications (i.e., tests) running within containers, we provide a monitoring engine that collects at runtime the resource usage metrics (see runtime monitoring engine in Figure 1). Docker containers rely on Control groups (Cgroups) file systems provided by the Linux kernel to group processes running within containers and expose a lot of metrics about the accumulated CPU cycles, memory, block I/O usage, etc. Our monitoring engine automates then the extraction of these runtime resource usage metrics stored in the Cgroups files. For example, we access to live resource consumption of each container available at the Cgroups file system via stats found in "/sys/fs/cgroup/cpu/docker/(longid)/" (for CPU consumption) and "/sys/fs/cgroup/memory/docker/(longid)/" (for stats related to memory consumption). So we extract automatically the runtime resource usage statistics relative to the running container (i.e., the generated code that is running within a container). Then, because we are collecting time series data that correspond to the resource utilization profiles of programs execution, we save these resource usage metrics within a time series database. Hence, we can run queries and define non-functional metrics from historical data (see resource usage extraction in Figure 1). For example, the following query reports the maximum memory usage of container *generated_code_v1* since its creation:

```
select max (memory_usage) from stats where container_name=''generated_code_v1''
```

For now, we presented the technical solution adopted in this work to ensure the automatic code generation and resource usage extraction. We describe in the following our contribution for automatic inconsistencies detection in code generator families.

3.2. A metamorphic testing approach for automatic inconsistencies detection in code generator families

One of the most important aspects we are interesting in while testing code generators is the *test oracle*. The testing community has proposed several approaches [18, 26] to alleviate the oracle problem (e.g., specified, implicit and derived oracles). Among the attractive solutions that can be applied to test code generators, we distinguish the metamorphic testing approach to derive oracles from properties of the system under test. In the following, we describe the basic concept of metamorphic testing and our adaptation of this method to the problem of non-functional testing of code generators.

3.2.1. Basic concept of metamorphic testing. In this section, we shall introduce the basic concept of metamorphic testing (MT), proposed by Chen et al. [27]. The idea of MT is to derive test oracles from the relation between test cases' outputs instead of reasoning about the relation between test inputs and outputs.

Metamorphic testing recommends that, given one or more test cases (called "source test cases", "original test cases" or "successful test cases") and their expected outcomes (obtained through multiple executions of the target program under test), one or more follow-up test cases can be constructed to verify the necessary properties (called Metamorphic Relations [MRs]) of the system or function to be implemented. In this case, the generation of the follow-up test cases and verification of the test results requires the respect of the MR.

The classical example of MT is that of a program that computes the *sin* function. A useful metamorphic relation for *sin* functions is $sin(x) = sin(\pi - x)$. Thus, even though the expected value for the source test case $sin(50)$, for example, is not known, a follow-up test case can be constructed to verify the MR defined earlier. In this case, the follow-up test case is $sin(\pi - 50)$ that must produce an output value that is equal to the one produced by the original test case $sin(50)$. If this property is violated, then a failure is immediately raised. MT generates follow-up test cases as long as

the metamorphic relations are respected. This is an example of a metamorphic relation: an input transformation that can be used to generate new test cases from existing test data and an output relation MR that compares the outputs produced by a pair of test cases. MR can be any properties involving the inputs and outputs of two or more executions of the target program such as equalities, inequalities, convergence constraints and many others.

Because MT checks the relation among several executions rather than the correctness of individual outputs, it can be used to fully automate the testing process without any manual intervention. We describe in the next section our adaptation of MT to the problem of non-functional testing of code generator families.

3.2.2. Adaptation of the MT approach to detect code generator inconsistencies. In general, MT can be applied to any problem in which a necessary property involving multiple executions of the target function can be formulated. Some examples of successful applications are presented by Zhou et al. [28]. We note that MT is recently applied to compilers testing [29–31].

To apply MT, there are four basic steps to follow:

1. Find the properties of the system under test: The system should be investigated manually in order to find intended MRs defining the relation between inputs and outputs. This is based on the source test cases.
2. Generate/select test inputs that satisfy the MR: This means that new follow-up test cases must be generated or selected in order to verify their outputs using the MR.
3. Execute the system with the inputs and get the outputs: Original and follow-up test cases are executed in order to gather their outputs.
4. Check whether these outputs satisfy the MR, and if not, report failures.

We develop now these four points in details to show how we can adapt the MT approach to the code generator testing problem.

3.2.3. Metamorphic relation. Step 1 consists of identifying the necessary properties of the program under test and represents them as metamorphic relations. Our MT adaptation is based on the MR definition presented by Chan et al. [31, 32]. In fact, a code generator family can be seen as a function: $C : I \rightarrow P$, where I is the domain of valid high-level source programs, and P is the domain of target programs that are automatically generated by the different code generators of the same family. The property of a code generator family implies that the generated programs P share the same behaviour as it is specified in I.

The availability of multiple generators with comparable functionality (i.e., code generator family) allows us to adapt the MT in order to detect non-functional inconsistencies. In fact, if we can find out our proper relations R_1 and R_2 (see Equation 1) between the inputs and outputs, we can get the metamorphic relation and conduct MT to evaluate the behaviour of our code generators. Let $f(P(t_i))$ be a function that calculates the non-functional output (such as execution time or memory usage) of the input test suite (t_i), running on a generated program (P). Because we have different program versions generated in the same family, we denote by $(P_1(t_i), P_2(t_i),\ldots, P_n(t_i))$ the set of generated programs. The corresponding outputs would be $(f(P_1), f(P_2),\ldots, f(P_n))$. Thus, our MR looks like this:

$$\begin{aligned} &R_1(P_1(t_i), P_2(t_i), \ldots, P_n(t_i)) \Rightarrow \\ &R_2(f(P_1(t_i)), f(P_2(t_i)), \ldots, f(P_n(t_i))) \end{aligned} \tag{1}$$

To define the R_1 relation, we use the following equation $P_1(t_i) \equiv P_2(t_i)$ to denote *the functional equivalence relation* between two generated programs P_1 and P_2 from the same family. This means that the generated programs P_1 and P_2 have the same behavioural design, and for any test suite t_i, they have the same functional output. If this relation is not satisfied, then there is at least one faulty code generator that produced incorrect code. In this this work, we focus on the non-functional properties, so we ensure that this relation is satisfied by excluding all tests that do not exhibit the same behaviour.

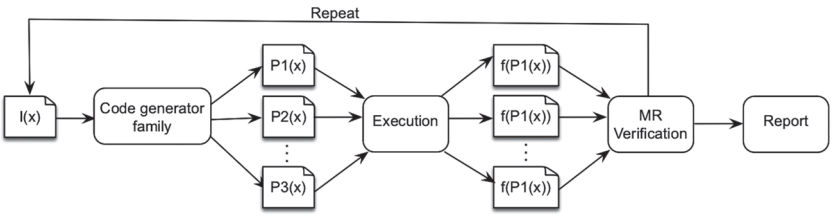


Figure 2. The metamorphic testing approach for automatic detection of code generator inconsistencies.

To define the R_2 relation, because we are comparing equivalent implementations of the same program written in different languages, we assume that the variation of memory usage and execution time of test suites execution across the different versions is more or less the same. Obviously, we are expecting to get a variation between different executions because we are comparing the execution time and memory usage of test suites that are written in different languages and executed using different technologies (e.g., interpreters for PHP, JVM for Java, etc.). This observation is also based on initial experiments, where we evaluate the resource usage/execution time of several test suites across a set of equivalent versions generated from the same code generator family (presented in details in Section 4). As a consequence, we use the notation $\Delta\{f(P_1(t_i)), f(P_2(t_i))\}$ to designate the variation of memory usage or execution time of test suite execution t_i across two versions of generated code P_1 and P_2 written in different languages. We suppose that this variation should not exceed a certain threshold value T, otherwise, we raise a code generator inconsistency. As a consequence, we define a code generator inconsistency as a generated code that exhibits an unexpected behaviour in terms of performance or resource usage compared with all equivalent implementations in the same code generator family.

Based on this intuition, the MR can be represented as

$$\begin{aligned} &P_1(t_i) \equiv P_2(t_i) \equiv \ldots \equiv P_n(t_i) \Rightarrow \\ &\Delta\{f(P_1(t_i)), f(P_2(t_i)), \ldots, f(P_n(t_i))\} < T \quad (n \geqslant 2). \end{aligned} \tag{2}$$

This MR is equivalent to say that if a set of functionally equivalent programs are generated using the same code generator family ($(P_1(t_i), P_2(t_i), \ldots, P_n(t_i))$, and with the same input test suite t_i, then the comparison of their non-functional outputs ($f(P_1(t_i)), f(P_2(t_i)),\ldots, f(P_n(t_i))$) defined by the variation Δ should not exceed a specific threshold value T.

The generated code that violates this metamorphic property represents an inconsistency, and its corresponding code generator is considered as defective.

3.2.4. Metamorphic testing. So far, we have defined the MR necessary for inconsistencies detection. We describe now our automatic metamorphic testing approach based on this relation (Steps 2, 3 and 4). Figure 2 shows the approach overview. The code generator family takes the same input program I and generate a set of equivalent test programs ($P_1, P_2, \ldots, P_n$). This corresponds to Step 2. In our MT adaptation, follow-up test cases represent the equivalent test programs that are automatically generated using a code generator family. Test suites are also generated automatically because we suppose that they are already defined at design time. In fact, the same test suite (test cases + input data values) is passed to all generated programs. Then, generated programs and their corresponding test suites are executed (Step 3). Afterwards, we measure the memory usage or execution time of these generated programs ($f(P_1(t_i)), f(P_2(t_i)),\ldots, f(P_n(t_i))$). Finally, the execution results are compared and verified using the MR defined earlier (Step 4). In this process, inconsistencies are reported when one of the follow-up equivalent test programs exhibits an unexpected behaviour (i.e., high variation), violating the MR.

3.2.5. Variation threshold. One of the questions that may be raised when applying our MT approach is how we can find the right variation threshold T from which an inconsistency is detected? Answering this question is very important to prove the effectiveness of our MT approach. To do so,

we conduct a statistical analysis in order to find an accurate threshold value T. Before that, the non-functional outputs need to be scaled to make them suitable for the statistical methods employed by our methodology. Thus, we describe first our process for data preparation.

Data preparation As depicted in Table I, each program comes with a set of test suites $(t_1, t_2, \ldots, t_m)$. Evaluating a test suite requires the calculation of the memory usage or execution time $f(P_1(t_i))$, $f(P_2(t_i)), \ldots, f(P_n(t_i))$, where $(1 \leq i \leq m)$ for all target software platforms. Thus, the obtained results represent a matrix where columns indicate the non-functional value (raw data) for each target software platform and rows indicate the corresponding test suite.

The non-functional data should be converted into a format that can be understood by our statistical methods. One way to compare these non-functional outputs is to study the factor differences. In other words, we would evaluate for each target platform the number of times (the factor) that a test suite takes to run compared with a reference execution. The reference execution corresponds to the minimum obtained non-functional value of t_i execution across the n target platforms. The resulting factor is the ratio between the actual non-functional value and the minimum value obtained among the n versions. The following equation is applied for each cell in order to transform our data:

$$F(f(P_j(t_i))) = \frac{f(P_j(t_i))}{Min(f(P_1(t_i)), \ldots, f(P_n(t_i)))}. \tag{3}$$

The reference execution will automatically get a score value $F = 1$. The maximum value is the one leading to the maximum deviation from the reference execution. For example, let P_1 be the generated program in Java. If the execution time needed to run t_1 yields to the minimum value $f(P_1(t_1))$ compared with other versions, then $f(P_1(t_1))$ will get a factor value F equal to 1 and the other versions will be divided by $f(P_1(t_1))$ to get the corresponding factor values compared with Java.

Statistical analysis. In our MT approach, an inconsistency is a resource usage/performance variation that exceeds a specific threshold value T. We propose the use of two popular variation analysis methods [33]: PCA and range charts (R-Chart). Table II gives an overview of these two statistical methods. The key objective of these methods is to evaluate the memory usage and performance variation and consequently defining an appropriate T value for our MR.

R-Chart. In this approach, the variation evaluation between the different versions is determined by comparing the non-functional measurements based on a statistical quality control technique called R-Chart or range chart [33]. R-Chart is used to analyse the variation within processes. It is designed to detect changes in variation over time and to evaluate the consistency of process variation. R-Chart uses control limits (LCL and UCL) to represent the limits of variation that should be expected from a process. LCL denotes the lower control limit and UCL denotes the upper control limit.

Table I. Results of test suites execution.

	Target Platform 1	Target Platform 2	...	Target platform n
t_1	$f(P_1(t_1))$	$f(P_2(t_1))$	...	$f(P_n(t_1))$
t_2	$f(P_1(t_2))$	$f(P_2(t_2))$	...	$f(P_n(t_2))$
...	...	...	...	...
t_m	$f(P_1(t_m))$	$f(P_2(t_m))$	...	$f(P_n(t_m))$

Table II. Variation analysis approaches.

Technique	**Method**
R-Chart	Define T as a variation between an upper and lower control limit
PCA	A cutoff value of the PC score distances defines the T

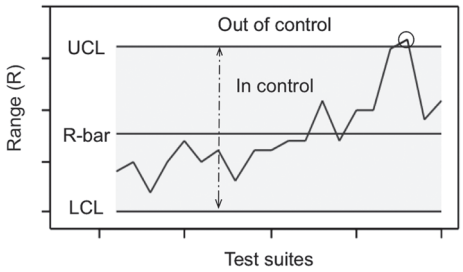


Figure 3. The R-Chart process.

By definition, when a process is within the control limits, any variation is considered as normal. It is said that the process is in control. Outside limit variations, however, it is considered as deviation, and the R-Chart is considered as out of control that means the process variation is not stable. Thus, it tells that there is an inconsistency leading to this high variation deviation (see Figure 3).

In our case, a process represents the n non-functional outputs obtained after the execution of a test suite t_i. As we defined the MR, the variation within a single process has to be lower than a threshold T. According to R-Charts, this variation must be between the LCL and UCL.

Therefore, for each test suite, we calculate the range R corresponding to the difference between the maximum and minimum non-functional outputs across all target platforms as

$$\begin{aligned} R(t_i) = &\, Max(f(P_1(t_i)), \ldots, f(P_n(t_i))) \\ &- Min(f(P_1(t_i)), \ldots, f(P_n(t_i))). \end{aligned} \tag{4}$$

R quantifies the variation results when running the same test suite t_i across different program versions. To determine whether the variation is in control or not, we need to determine the control limit values. UCL and LCL reflect the actual amount of variation that is observed. Both metrics are a function of R-bar ($\bar{R}$). $\bar{R}$ is the average of R for all test suites. The UCL and LCL are calculated as follows:

$$\begin{aligned} UCL &= D_4 \bar{R}, \\ LCL &= D_3 \bar{R}, \end{aligned} \tag{5}$$

where D_4, D_3, are control chart constants that depend on the number of variables inside each process (see constants values[‡‡]).

For example, for a family composed of less than seven code generators, the D_3 value is equal to 0, and as a consequence, $LCL = 0$. In this case, the UCL represents the threshold value T from which we detect a high deviation from $\bar{R}$, leading to an inconsistency. As we stated earlier, the UCL is a function of $\bar{R}$, and $\bar{R}$ is a function of range differences. So the UCL value (or T) is sensitive to new test suites. So when a new test suite is executed, the T value is updated, and the variation is evaluated with the new threshold value.

We present in the following an alternative statistical approach to analyse the variation of all our data.

Principal component analysis. With a large number of program versions, the matrix of non-functional data (Table I) may be too large to study and interpret the variation properly. There would be too many pairwise correlations between the different versions to consider and the variation is impossible to display (graphically) when test suites are executed in more than three target software platforms. With 12 variables, for example, there will be more than 200 three-dimensional scatter plots to be designed to study the variation and correlations. To interpret the data in a more meaningful form, it is therefore necessary to reduce the number of variables composing our variability model.

[‡‡]http://www.bessegato.com.br/UFJF/resources/table_of_control_chart_constants_old.pdf

Principal component analysis[§§] is a multivariate statistical approach that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components (PCs). It can be applied when data are collected on a large number of variables from a single observation. Thus, we apply the PCA approach to our case study because our dimension space, as it is presented in Table I, is composed of a set of processes (test suites) where n variables (*e.g.*, target programming languages) are composing each observation. The variability within our model is correlated to these n variables representing the test suites running on n target platforms.

The main objective of applying PCA is to reduce the dimensionality of the original data and explain the maximum amount of variance with the fewest number of principal components. To do so, PCA is concerned with summarizing the variance–covariance matrix. It involves computing the eigenvectors and eigenvalues of the variance–covariance matrix. The eigenvectors are used to project the data from n dimensions down to a lower dimensional representation. The eigenvalues give the variance of the data in the direction of the eigenvector. The first principal component is calculated such that it accounts for the greatest possible variance in the data set. The second principal component is calculated in the same way, with the condition that it is uncorrelated with (i.e., perpendicular to) the first principal component and that it accounts for the next highest variance. PCA uses many data transformations and statistical concepts. We are not interested in studying all the mathematical aspects of PCA. Thus, we use an existing R package[¶¶] to transform and reduce our data into two PCs in order to visualize the variation of all our data points in a two-dimensional space.

Our intuition behind the PCA approach is to conduct a general and complete analysis of variation in order to find extreme variation points at the boundaries of the multivariate data. These extreme points represent, from a statistical perspective, *outliers*. Following our MT approach, these points correspond to the inconsistencies (or deviations) we would detect. Outliers have an important influence over the PCs. An outlier is defined as an observation which does not follow the model followed by the majority of data. One way to detect outliers is to use a metric called score distance (SD). SD measures the dispersion of the observations within the PCA space. It thus measures how far an observation lies from the rest of the data within the PCA subspace. SD measures the statistical distance from a PC score to the centre of scores. For an observation x_i, the score distance is defined as

$$SD_i = \sqrt{\sum_{j=1}^{a} \frac{t_{ij}^2}{\lambda_j}}, \tag{6}$$

where a is the number of PCs forming the PCA space, t_{ij} are the elements of the score matrix obtained after running PCA, and λ_j is the variance of the j^{th} PC that corresponds to the j^{th} eigenvalue. In order to find the outliers, we compute the 97.5% quantile Q of the chi-square distribution as a cutoff value of the SD ($\sqrt{\chi^2_{a,0.975}}$). It corresponds to a confidence ellipse that covers 97.5% of the data points. According to the table of the chi-square distribution[‖‖], this value is equal to $\sqrt{7.38} = 2.71$. Any sample whose SD is larger than the cutoff value is identified as an outlier (or inconsistency). This cutoff value represents the variation threshold T we would define for our MR using the PCA approach.

In short, PCA and R-Chart represent two statistical methods that help us to evaluate the variation of our output data and to define the threshold value from which an inconsistency is detected. On the one hand, the R-Chart method evaluates the variation using the range difference between the data and defines the T as a control limit. PCA represents an alternative that enables us to reduce the variability space and then analyse the variation of all data graphically in a two-dimensional space. The T is based on an outlier detection method that is applied on a multivariate data matrix. The two methods are complementary, and the main objective of applying both of them is to evaluate the accuracy of our approach, verifying if we can detect the same inconsistencies.

[§§] https://en.wikipedia.org/wiki/Principal_component_analysis

[¶¶] http://factominer.free.fr/

[‖‖] https://store.fmi.uni-sofia.bg/fmi/statist/education/Virtual_Labs/tables/tables3.html

We move now to present the evaluation of our approach.

4. EVALUATION

So far, we have presented an automatic approach for detecting inconsistencies within code generator families. So we shape our goal as this research question:

RQ: *How effective is our metamorphic testing approach to automatically detect inconsistencies in code generator families?*

To answer this question, we evaluate the implementation of our approach by explaining the design of our empirical study and the different methods we used to assess the effectiveness of our approach. The experimental material is available for replication purposes[***].

4.1. Experimental setup

4.1.1. Code generators under test: Haxe compilers. In order to test the applicability of our approach, we conduct experiments on a popular high-level programming language called Haxe[†††] [15] and its code generators.

Haxe comes with a set of compilers that translate the manually written code (in Haxe language) to different target languages and platforms.

The process of code transformation and generation can be described as follows: Haxe compilers analyse the source code written in Haxe language. Then, the code is checked and parsed into a typed structure, resulting in a typed abstract syntax tree (AST). This AST is optimized and transformed afterwards to produce source code for the target platform/language. Haxe offers the option of choosing which platform to target for each program using command-line options. Moreover, some optimizations and debugging information can be enabled through command-line interface, but in our experiments, we did not turn on any further options.

The Haxe code generators constitute the code generator family we would evaluate in this work.

4.1.2. Cross-platform benchmark. One way to prove the effectiveness of our approach is to create benchmarks. Thus, we use the Haxe language and its code generators to build a cross-platform benchmark. The proposed benchmark is composed of a collection of cross-platform libraries that can be compiled to different targets. In these experiments, we consider a code generator family composed of five target Haxe compilers: Java, JS, C++, CS and PHP. To select the cross-platform libraries, we explore github and we use the Haxe library repository[‡‡‡]. So we select seven libraries that provide a set of test suites with high code coverage scores.

In fact, each Haxe library comes with an API and a set of test suites. These tests, written in Haxe, represent a set of unit tests that cover the different functions of the API. The main task of these tests is to check the correct functional behaviour of generated programs. To prepare our benchmark, we remove all tests that fail to compile to the five targets (i.e., errors, crashes and failures), and we keep only those who are functionally correct in order to focus on the non-functional properties. Moreover, we add manually new test cases to some libraries in order to extend the number of test suites. The number of test suites depends on the library size.

We use then these test suites to transform the functional tests into stress tests. We run each test suite 1 K times to get comparable values in terms of performance and resource usage. Table III describes the Haxe libraries that we have selected in this benchmark to evaluate our approach and the number of test suites used per benchmark. In total, we depict 95 test suites to run across the five target software platforms.

4.1.3. Evaluation metrics used. We evaluate the efficiency of generated code using the following non-functional metrics:

[***] https://testingcodegenerators.wordpress.com/
[†††] http://haxe.org/
[‡‡‡] https://lib.haxe.org/all/

Table III. Description of selected benchmark libraries.

Library	#TestSuites	Description
Colour	19	Colour conversion from/to any colour space
Core	51	Provides extensions to many types
Hxmath	6	A 2D/3D math library
Format	4	Format library such as dates, number formats
Promise	5	Library for lightweight promises and futures
Culture	5	Localization library for Haxe
Math	5	Generation of random values
Total	95	

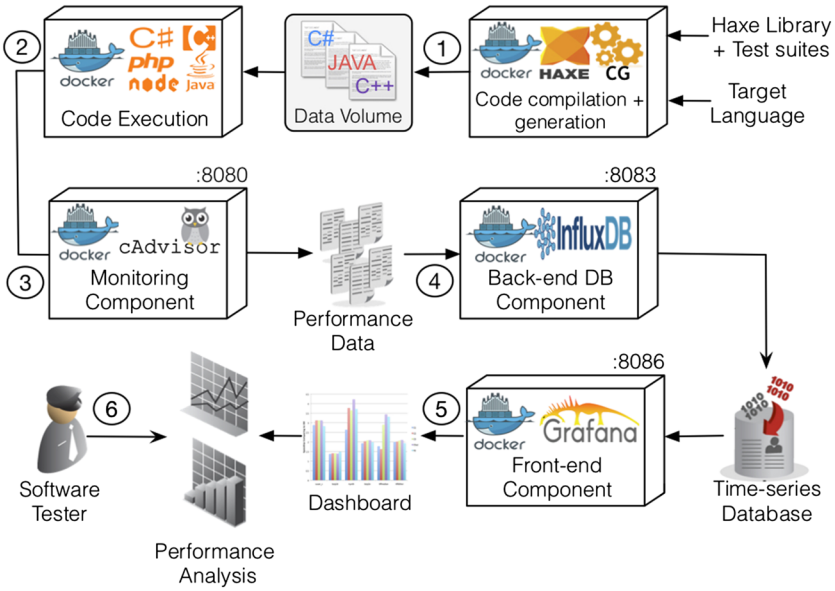


Figure 4. Infrastructure settings for running experiments.

- *Memory usage*: It corresponds to the maximum memory consumption of the running test suite. Memory usage is measured in MB.
- *Execution time*: The execution time of test suites is measured in seconds.

We recall that our testing infrastructure is able to evaluate other non-functional properties such as code generation time, compilation time, code size and CPU usage. We choose to focus, in this experiment, on the performance (i.e., execution time) and resource usage (i.e., memory usage). Collecting resource usage metrics is ensured by our monitoring infrastructure, presented in Section 3.1.

4.1.4. Setting up infrastructure. To assess our approach, we configure our previously proposed container-based infrastructure in order to run experiments on the Haxe case study. Figure 4 shows a big picture of the testing infrastructure considered in these experiments.

First, a first component is created in where we install the Haxe code generators and compilers. It takes as an input the Haxe library we would evaluate and the list of test suites (Step 1). It produces as an output the source code files relative to the target software platform. Afterwards, generated files are compiled (if needed) and automatically executed within the execution container (Step 2). This component is a preconfigured container instance where we install the required execution environments such as *interpreters* (for PHP), *node* (for JS), *mono* (for C#), etc. In the meantime, while running test suites inside the container, we collect the runtime resource usage data (Step 3). To do so,

we integrate cAdvisor, a Container Advisor[§§§] widely used in different projects such as Heapster[¶¶¶] and Google Cloud Platform[‖‖‖]. cAdvisor monitors the running containers at runtime and collects resource usage information stored in Cgroups files. To save this data, we use InfluxDB[****], an open source distributed time series database available for Docker as a back-end (Step 4). InfluxDB allows us to execute SQL-like queries on the database or to directly run HTTP requests to extract data. We also provide a dashboard to run queries and view different resource consumption profiles of running containers, through a Web UI. As a visualization container, we use Grafana[††††], a time series visualization tool available for Docker that displays live results over time in much pretty looking graphs. This component is not used during these experiments. Instead, we directly extract the resource usage information from InfluxDB. Finally, in Step 5, we apply our metamorphic approach to analyse the non-functional data and detect code generator inconsistencies.

We use the same hardware across all experiments: an AMD A10-7700 K APU Radeon(TM) R7 Graphics processor with 4 CPU cores (2.0 GHz), running Linux with a 64-bit kernel and 16 GB of system memory. We run the experiments on top of a private data centre that provides a bare-metal installation of Docker. On a single machine, containers are running sequentially, and we pin p cores and n Gbytes of memory for each container[‡‡‡‡]. Once the execution is done, resources reserved for the container are automatically released to enable spawning next containers. Therefore, the host machine will not suffer too much from performance trade-offs.

4.2. Experimental methodology and results

In the following paragraphs, we report the methodology we used to answer RQ and the results of our experiments.

4.2.1. Method. We now conduct experiments based on the new created benchmark libraries. The goal of running these experiments is to observe and compare the behaviour of generated code using the defined MR in order to detect code generator inconsistencies.

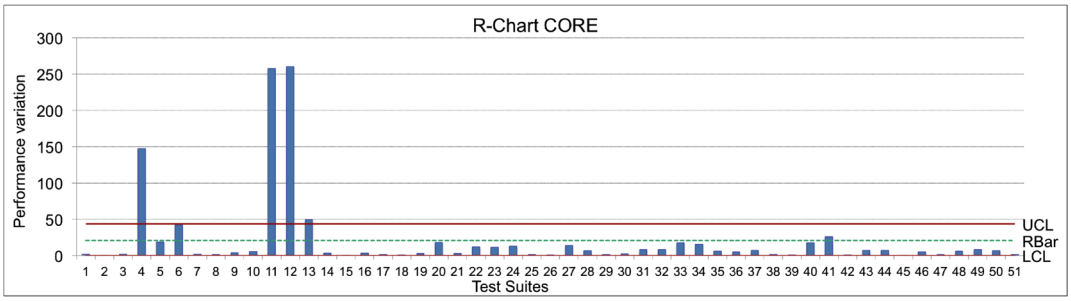
Therefore, we set up, first, our container-based infrastructure as it is presented in Section 3.1 in order to generate, execute and collect the memory usage of our test suites. Afterwards, we prepare and normalize the gathered data to make it valuable for statistical analysis. Then, we conduct the R-Chart and PCA analysis as described in Section 3.2.5 in order to analyse the performance and resource usage variations. This will lead us to define an appropriate formula of the MR, used to automatically detect inconsistencies within Haxe code generators (Section 3.2.4). Finally, we report the inconsistencies we have detected.

4.2.2. Results.

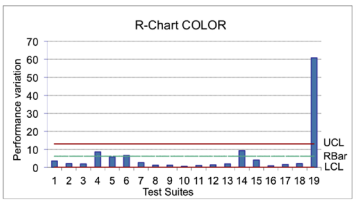
R-Chart results The results of R-Charts for the seven benchmark programs relative to the performance and resource usage variations are reported in Figures 5 and 6. In Figure 5, we report the performance variation corresponding to the range difference R between the maximum and minimum execution time of each test suite across the five targets (Java, JS, C++, C# and PHP).

The LCL in our experiments is always equal to 0 because the D_3 constant value as defined in Equation (5), is equal to zero according to the R-Chart constants table. In fact, the D_3 constant changes depending on the number of subgroups. In our experiments, our data record is composed of five subgroups corresponding to the five target programming languages. The central line (in green) corresponds to $\bar{R}$. This value changes from one benchmark to another depending on the average of R for all test suites in the benchmark. As a consequence, UCL, which is a function of $\bar{R}$, changes

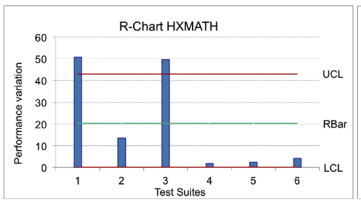
[§§§] https://github.com/google/cadvisor
[¶¶¶] https://github.com/kubernetes/heapster
[‖‖‖] https://cloud.google.com/
[****] https://github.com/influxdata/influxdb
[††††] https://github.com/grafana/grafana
[‡‡‡‡] p and n can be configured
http://www.bessegato.com.br/UFJF/resources/table_of_control_chart_constants_old.pdf



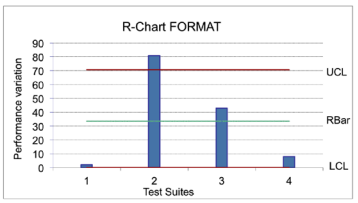
(A) R-Chart of the Core benchmark program



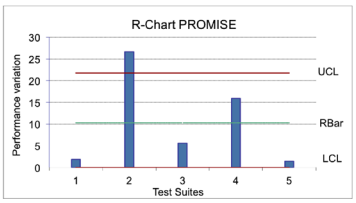
R-Chart of the Color benchmark program



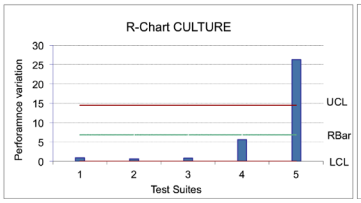
(C) R-Chart of the Hxmath benchmark program



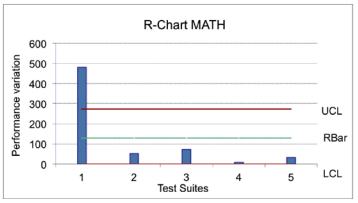
(D) R-Chart of the Format benchmark program



(E) R-Chart of the Promise benchmark program



(F) R-Chart of the Culture benchmark program



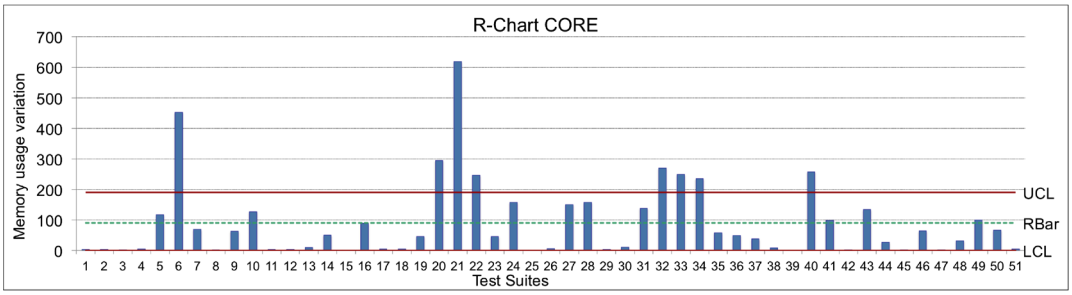
(G) R-Chart of the Math benchmark program

Figure 5. Performance variation of test suites across the different Haxe benchmarks. (a) R-Chart of the Core benchmark program. (b) R-Chart of the Colour benchmark program. (c) R-Chart of the Hxmath benchmark program. (d) R-Chart of the Format benchmark program. (e) R-Chart of the Promise benchmark program. (f) R-Chart of the Culture benchmark program. (g) R-Chart of the Math benchmark program.

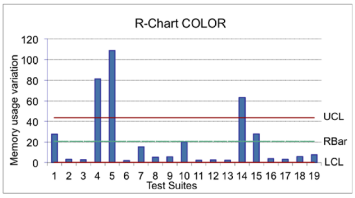
dynamically as long as we add new test suites to the experiments. UCL is equal to $D_4 * \bar{R}$, where $D_4 = 2.114$ according to the R-Chart constants table. We note that the LCL parameter value is appropriate to each benchmark program. We made the threshold values specific to each benchmark because we believe that the variation is highly dependent on the application domain and on the program under test. The R-Charts used for visualizing the memory usage variation follow the same concept as we have just been describing for performance variation.

Results in Figure 5, show that most of the performance variations are in the interval $[0, UCL]$, which corresponds to the *in-control* variation zone as it is described in Section 3.2.5. However, we remark for several test suites that the performance variation becomes relatively high (higher than the UCL value of the corresponding benchmark program). We detect 11 among the 95 performance variations lying in the *out of control* variation zone. For the other test suites, the variation is even less than the total average variations $\bar{R}$. There are only seven test suites among the remaining 84 ones, where the variation lies in the interval $[\bar{R}, UCL]$. This variation is high, but we are not detecting it as a performance deviation because according to the R-Chart, variation in this zone is still *in control*. The 11 performance deviations we have detected can be explained by the fact that the execution time of one or more test suites varies considerably from one language to another. This argues the idea that the code generator has produced suspect code behaviour, which led to a high performance variation. We provide later further explanation of the source of such variation.

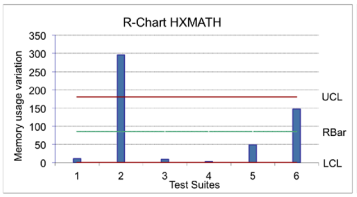
Similarly, Figure 6 resumes the comparison results of test suites execution regarding the memory usage. The variations in this experiment are more important than previous results. This can be argued by the fact that the memory utilization and allocation patterns are different from one language to another. Nevertheless, we can recognize some points, where the variation is extremely high. Thus, we detect 15 among 95 test suites that exceed the corresponding UCL value. When the variation is below UCL, we detect 14 among the 80 remaining test suites where the variation lies in the interval



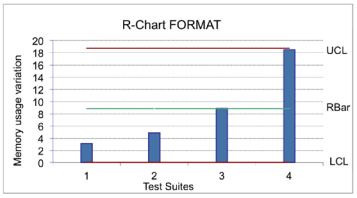
(a) R-Chart of the Core benchmark program



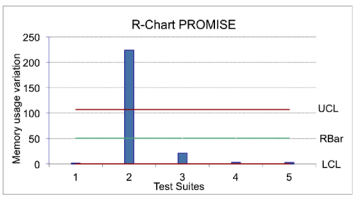
(b) R-Chart of the Color benchmark program



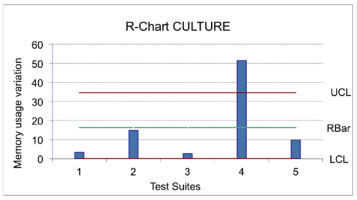
(c) R-Chart of the Hxmath benchmark program



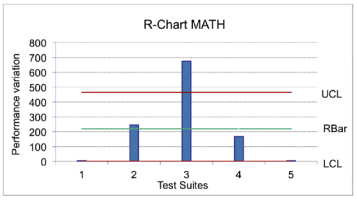
(d) R-Chart of the Format benchmark program



(e) R-Chart of the Promise benchmark program



(f) R-Chart of the Culture benchmark program



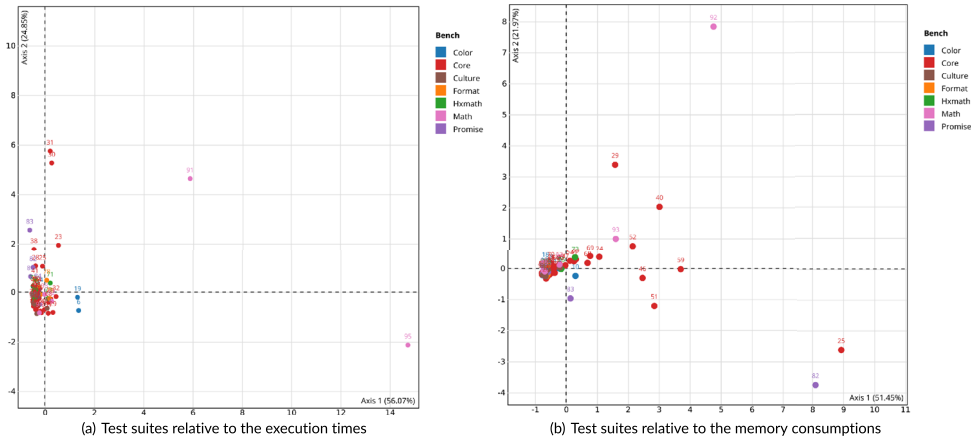
(g) R-Chart of the Math benchmark program

Figure 6. Memory usage variation of test suites across the different Haxe benchmarks. (a) R-Chart of the Core benchmark program. (b) R-Chart of the Colour benchmark program. (c) R-Chart of the Hxmath benchmark program. (d) R-Chart of the Format benchmark program. (e) R-Chart of the Promise benchmark program. (f) R-Chart of the Culture benchmark program. (g) R-Chart of the Math benchmark program.

$[\bar{R}, UCL]$, which is relatively high. One of the reasons that caused this variation may occur when the test suite executes some parts of the code (in a specific language) that strangely consume a lot of resources. This may not be the case when the variation is lower than $\bar{R}$ for example.

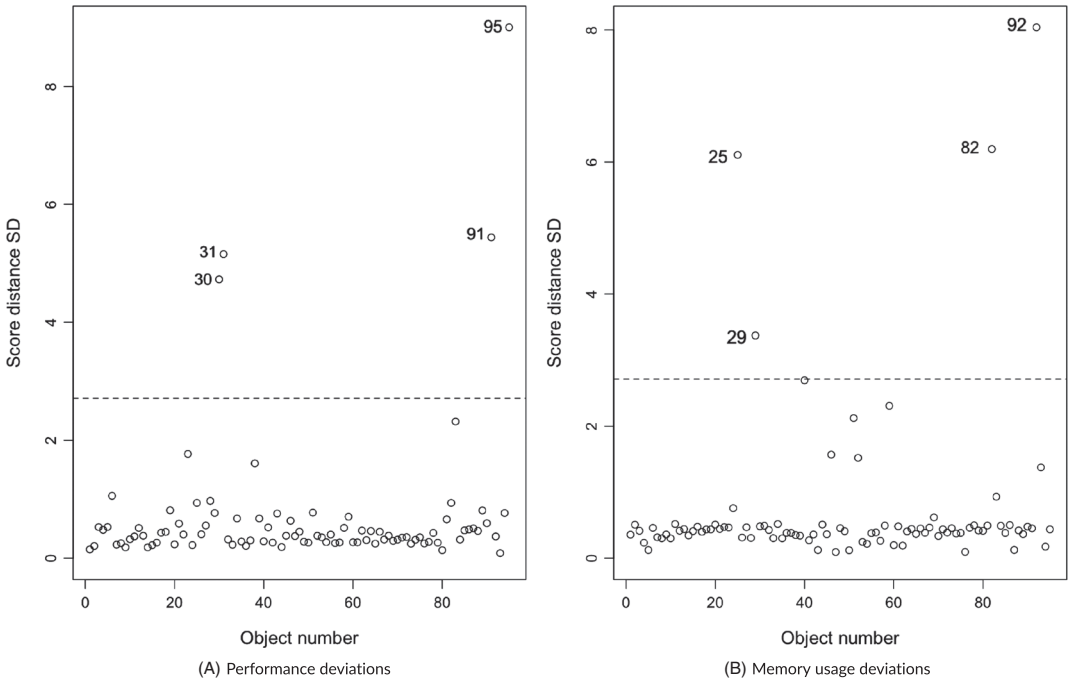
To resume, we have detected 11 extreme performance variations and 15 extreme memory usage variations among the 95 executed test suites. We assume then that defective code generators, in identified points, represent a threat for software quality since the generated code has shown symptoms of poor quality design.

PCA results We apply the PCA approach as an alternative to the R-Chart approach. Figure 7 shows the dispersion of our data points in the PC subspace. PC1 and PC2 represent the directions of our two first principal components, having the highest orthogonal variations. Our data points represent the performance variation (Figure 7a) and the memory usage variation (Figure 7b) of the 95 test suites we have executed. Variation points are coloured according to the benchmark program they belong to (displayed in the figure legend). At the first glance, we can clearly see that the variation points are situated in the same area except some points that lie far from this data cluster. In Figure 7a, the pink points corresponding to the *Math* benchmark show visually the largest variation. The three *Core* test suites (in red), which are identified as performance deviations in R-Chart, show also a deviation in the PCA scatter plot. Point 91 relative to the *Math* benchmark is deviating from the cloud point. However, in the R-Chart diagram, it is not detected as a performance deviation (see the test suite 3 of Figure 5g). In fact, this test suite takes more than 80 times to run. Compared with other test suites, the performance variation does not exceed 80. In effect, PCA performs a complete analysis of the whole data we have collected in all benchmarks. Thus, variations are displayed with respect to all test suites variations in all benchmarks. The variation evaluation is not limited within



(a) Test suites relative to the execution times (b) Test suites relative to the memory consumptions

Figure 7. PCAs showing the dispersion of our data over the PC subspace. (a) Test suites relative to the execution times. (b) Test suites relative to the memory consumptions.



(A) Performance deviations (B) Memory usage deviations

Figure 8. Diagnostic plots using score distance SD. The horizontal line indicates the critical value separating regular observations from outliers (97.5%). (a) Performance deviations. (b) Memory usage deviations.

the benchmark program as we used to do using R-Charts. We report the same results in Figure 7b about the memory usage variation in the PCA.

To confirm this observation, we present in Figure 8 the results of our outliers detection approach. We identify four inconsistencies (or outliers) in each diagnostic plot. Inconsistencies in Figure 8a are relative to the performance deviations. Points 30 and 31 correspond to the test suites 12 and 11 in benchmark *Core* of Figure 5a. Points 91 and 95 correspond to the test suites 3 and 1 in benchmark *Math* of Figure 5g. For memory usage variation, we detect points 25, 29, 82 and 92 that corresponds relatively to the test suites 21 and 6 of benchmark *Core*, 2 of benchmark *Promise*, and 3 of benchmark *Math*. We can clearly see that this technique helps to identify extreme-value outliers, which are mostly covered by the R-Chart approach. We used 97.5% quantile of the chi-square distribution to define the cutoff value that is commonly used in the literature [34, 35].

Table IV. Raw data values of test suites that led to the highest variation in terms of execution time.

Benchmark	Test Suite	Java	JS	CPP	CS	PHP	UCL(R)	Defective CG
Colour	TS19	1.90	1	2.37	3.31	61.84	13.08	PHP
Core	TS4	1	1.59	1.67	2.78	148.20	43.62	PHP
	TS11	1.14	2.71	1	3.63	258.94		PHP
	TS12	1.28	2.94	1	3.36	261.36		PHP
	TS13	1	1.05	1.86	2.39	50.30		PHP
Hxmath	TS1	2.38	1.43	1	2.82	51.72	42.97	PHP
	TS3	2.14	1.10	1	2.25	50.56		PHP
Format	TS2	1.16	1.27	1	3.35	81.85	70.66	PHP
Promise	TS2	1.52	1.85	1	1.51	27.67	21.76	PHP
Culture	TS5	1.62	1	1.27	2.02	27.29	14.47	PHP
Math	TS1	4.15	1	5.41	4.70	481.68	273.24	PHP

Table V. Raw data values of test suites that led to the highest variation in terms of memory usage.

Benchmark	Test suite	Java	JS	CPP	CS	PHP	UCL	Defective CG
Colour	TS4	1	2.29	1.47	3.59	82.46	43.53	PHP
	TS5	1	3.08	1.83	4.53	109.69		PHP
	TS14	1	1.32	1.00	2.03	64.45		PHP
Core	TS6	250.77	71.71	1	69.90	454.15	190.03	PHP & Java
	TS20	2.31	1.34	1	3.27	296.10		PHP
	TS21	11.90	1	14.63	36.18	620.22		PHP
	TS22	1	2.70	1.74	4.69	247.32		PHP
	TS32	270.78	2.27	1	5.61	153.37		Java
	TS33	1.82	1.12	1	54.19	250.35		PHP
	TS34	1	1.17	1.48	3.90	236.97		PHP
	TS40	160.84	1.10	1	49.43	259.20		PHP
Hxmath	TS2	1	1.16	1.91	2.82	296.16	181.11	PHP
Promise	TS2	214.53	92.45	1	57.68	224.41	106.82	PHP & Java
Culture	TS4	2.75	1.01	2.52	1	52.47	34.63	PHP
Math	TS3	1.29	1	1.72	3.60	675.00	464.80	PHP

Detected inconsistencies Now that we have observed the performance and memory usage variations of test suites execution, we can analyse the extreme points we have previously detected in order to understand the source of such deviation. For that reason, we present in Tables IV and V the raw data values of these test suites leading to an extreme variation in terms of execution time and memory usage. We report the inconsistencies gathered from the first approach, R-Chart.

Table IV shows the execution time factor of each test suite execution in a specific target language. This factor is scaled with respect to the the lowest execution time among the five targets. We also report the defined UCL value per benchmark. In the last column, we report the code generator that caused such large deviation. To do so, we designate by *defective CG*, the code generator that led to a performance variation higher than the UCL value. We can clearly see that the PHP code has a singular behaviour regarding the performance with a factor ranging from x27.29 for test suite 5 in benchmark *Culture* (*Culture_TS5*) to x481.7 for *Math_TS1*. For example, if *Math_TS1* takes 1 minute to run in JS, the same test suite in PHP will take around 8 h to run, which is a very large gap. The highest detected factor for other languages is x5.41 that is not negligible, but it represents a small deviation compared with PHP deviations. While it is true that we are comparing different versions of generated code, it was expected to get some variations while running test suites in terms of execution time. However, in the case of PHP code generator, it is far to be a simple variation, but it is a code generator inconsistency that led to such performance regression.

Meanwhile, we gather information about the points that led to the highest variation in terms of memory usage. Table V shows these results. Again, we can identify a singular behaviour of the PHP code regarding the memory usage with a factor ranging from x52.47 to x675. For other test suite versions, the factor varies from x1 to x160.84. We observe as well a singular behaviour of

the Java code for *Core_TS6*, *Core_TS32* and *Promise_TS2*, yielding to a variation higher than the UCL. These results prove that the PHP and Java code generators are not always efficient and they constitute a threat for the generated software in terms of memory usage.

To give more insights about the source of this issue, we provide in the following further analysis of these inconsistencies.

4.2.3. Analysis. These inconsistencies need to be fixed by code generator experts in order to enhance the quality of generated code (e.g., PHP code). Because we are proposing a black-box testing approach, our solution is not able to provide more accurate and detailed information about the part of code causing these performance issues, which is one of the limitations of our testing approach.

Therefore, to understand this unexpected behaviour of the PHP code when applying the test suite *Core_TS4,* for example, we looked (manually) into the PHP code corresponding to this test suite. We observe the intensive use of "arrays" in most of the functions executed by the test suite. In fact, native arrays in PHP are allocated dynamically that leads to a slower write time because the memory locations needed to hold the new data is not already allocated. Thus, slow writing speed damages the performance of PHP code and impacts consequently the memory usage. As an alternative, PHP library has introduced much more advanced functions such as array_fill and specialized abstract types such as "SplFixedArray"[§§§§] to overcome this limitation. For example, "SplFixedArray" pre-allocates the necessary memory and allows a faster array implementation, thereby solving the issue of slower write times. To follow our intuition, we change these two parts in the generated code, and we measure again the execution time and memory usage of the corresponding test suite. As a results, we improve the PHP code speed with a factor x5 that is very valuable. We also reduce the memory usage by a factor of x2.

Following these interesting results, we reported the test suites triggering inconsistencies and their corresponding benchmarks to the Haxe community in order to investigate the issue and make updates to the PHP code generator. Consequently, they have recently released a new version of the PHP code generator[¶¶¶¶] with many performance improvements, especially for arrays. Following our suggested improvements, they introduced advanced functions (*array_fill*[‖‖‖‖]) for arrays initialization in the PHP code generator in order to improve its performance.

In short, the lack of use of specific types, in native PHP standard library, by the PHP code generator such as *SplFixedArray,* shows a real impact on the non-functional behaviour of generated code. Obviously, the types used during code generation are not the best ones. In contrast, selecting carefully the adequate types and functions to generate code can lead to performance improvement.

4.3. Application of our approach to other case studies

To evaluate the effectiveness of our approach, we apply the same methodology to another case study (i.e., another code generator family), ThingML. As discussed in the motivation Section 2.1, ThingML is a popular domain-specific language designed to generate code to different target software platforms (e.g., resource-constrained devices). Similarly to Haxe, ThingML experts use a set of test cases, written in ThingML, to verify the correct functional behaviour of their code generators. They run these tests when a new complier is made available or updated. We count 120 available functional tests, that compile to at least 4 targets. The list of test cases they use to run is available here[*****]. We benefit then from existing ThingML sample projects and test cases to reproduce our metamorphic testing approach to this case study. We select three target compilers C, JAVA and JavaScript. We run the 120 test cases 1 K times, and we gather the memory usage relative to each test case execution for the target platform. The statistical method applied in this experiment is the PCA. Our outlier-based method is also applied to identify test cases triggering inconsistencies.

[§§§§] http://php.net/manual/fr/class.splfixedarray.php

[¶¶¶¶] https://github.com/HaxeFoundation/haxe/releases/tag/3.4.0

[‖‖‖‖] https://github.com/HaxeFoundation/haxe/blob/f375ec955b41550546e494e9f79a5deefa1b96ac/std/php7/Global.hx#L226

[*****] https://github.com/TelluIoT/ThingML/tree/master/testJar

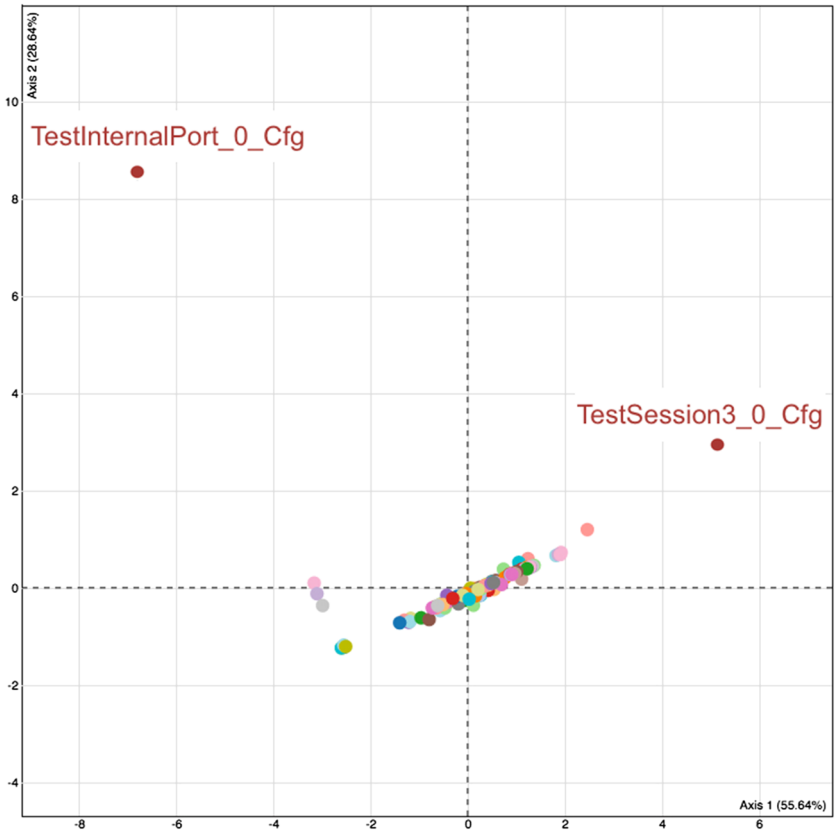


Figure 9. PCA showing the dispersion of ThingML test cases relative to the memory usage over the PC subspace: We identify two detected inconsistencies *TestInternalPort_0_Cfg* and *TestSession3_0_Cfg*.

The results of running this experiment is depicted in Figure 9. It shows the dispersion of 120 ThingML test cases relative to the memory usage over the PC subspace. The detected inconsistencies are also shown in the same figure relative to tests: *TestInternalPort_0_Cfg* and *TestSession3_0_Cfg*. These two test cases indicate an abnormal variation of memory usage compared with other points in the cluster. This result shows the effectiveness of our approach to identify inconsistencies in other case studies, involving a family of code generators. As we did with Haxe community, we reported this two inconsistencies to ThingML experts in order to investigate the cause of such high variation in memory usage for those two test cases.

Key findings for RQ

- Our approach is able to automatically detect, among 95 executed test suites, 11 performance and 15 memory usage inconsistencies, violating the metamorphic relation for Haxe code generators.
- The analysis of test suites triggering the inconsistencies shows that there exist potential issues in some code generators, affecting the quality of delivered software.
- Our approach can be easily applied to other case studies, involving a family of code generators such as ThingML.

4.4. Threats to validity

We resume, in the following paragraphs, external and internal threats that can be raised.

External validity refers to the generalizability of our findings. In this study, we perform experiments on Haxe and on a set of test suite selected from Github and from the Haxe community. For instance, we have no guarantee that these libraries cover all Haxe language features. Consequently, we cannot guarantee that our approach is able to find all code generator issues unless we develop a

more comprehensive test suite (e.g., based on code or model coverage). Moreover, in case all code generators are defective or "nonmature", the variation of performance or resource usage for each test suite can be very unstable. We cannot confirm in this case the source of such high variation.

Internal validity is concerned with the use of a container-based approach. The proposed virtualized infrastructure requires the Linux kernel to run. Even if it exists, emulators such as Qemu[†††††] that allow to reflect the behaviour of heterogeneous hardware, the chosen infrastructure has not been evaluated to test that generated code that target heterogeneous hardware machines. In addition, even though system containers are known to be lightweight and less resource-intensive compared with full stack virtualization, we would validate the reliability of our approach by comparing the results with a nonvirtualized approach in order to see the impact of using containers on the accuracy of results.

5. RELATED WORK

5.1. Automatic code generators testing approaches

Most of the previous work on code generator testing focuses on checking the correct functional behaviour of generated code [7, 36–41].Most of these research efforts rely on the comparison of the model execution to the generated code execution. This is known in the software testing community as equivalence, comparative or back-to-back testing approach [9, 10].

For instance, Stuermer et al. [7] present a systematic test approach for model-based code generators. They investigate the impact of optimization rules for model-based code generation by comparing the output of the code execution with the output of the model execution. If these outputs are equivalent, it is assumed that the code generator works as expected. They evaluate the effectiveness of their approach by means of optimizations performed by the TargetLink code generator. They use Simulink as a simulation environment of models. In the work of Jorges et al. [39], authors present a testing approach of the Genesys code generator framework that tests the translation performed by a code generator from a semantic perspective rather than just checking for syntactic correctness of the generation result. Basically, Genesys realizes back-to-back testing by executing both the source model as well as the generated code on top of different target platforms. Both executions produce traces and execution footprints that are then compared. The limitation of these testing approaches is that they are applicable only when the input model/source code is executable. Compared with our proposal, we rather propose an approach that test generators at the source code level regardless of the input model/source code execution.

Previous work on non-functional testing of code generators focuses on comparing, as oracle, the non-functional properties of handwritten code to automatically generated code [19, 20]. As an example, Strekelj et al. [42] implemented a simple 2D game in both, the Haxe programming language and the target programming language, and evaluated the difference in performance between the two versions of code. They showed that the generated code through Haxe has better performance than the handwritten one.

In the work of Ajwad et al. [43], authors compare some non-functional properties of two code generators, the TargetLink code generator and the Real-Time Workshop Embedded Coder. They also compare theses properties to manually written code. The metrics used for comparison are ROM and RAM memory usage, execution speed, readability and traceability. Many test cases are executed to see if the controller behaves as expected. The comparison results show that the generated code by TargetLink is more efficient than the manually written code and the other generated code in terms of memory and execution time. They also show that the generated code can be easily traced and edited.

Cross-platform mobile development has been also part of the non-functional testing goals because many code generators are increasingly used in industry for automatic cross-platform development. For instance, Pazirandeh et al. [44] and Hartmann et al. [45] compare the performance of a set of cross-platform code generators to present the most efficient tools.

[†††††] https://goo.gl/SxKG1e

5.2. Applications of metamorphic testing

The metamorphic testing method has been proposed by Chen et al. [46] to alleviate the oracle problem. In a similar case as for code generators, MT was recently applied to compiler testing. Le et al. [30] present an approach called equivalence modulo inputs (EMI) testing. The idea of this approach is to pass different program versions (with same behaviour) to the compiler in order to inspect the output similarity after code compilation and execution. The authors propose to create equivalent versions of a program by profiling its execution and pruning unexecuted code. Once a program and its equivalent variant are constructed, both are used as input to the compiler under test and then, inconsistencies in their results are checked. The metamorphic relation represents, in this case, the comparison between the functional output of the program and its variants. This method has detected 147 confirmed bugs in two open source C compilers, GCC and LLVM. A closely related idea was presented by Tao et al. [31] to test the semantic-soundness property of compilers. They use three different techniques in generating equivalent source code programs and then test the mutants with the original programs, such as replacing an expression with an equivalent one. Empirical results show that their approach is able to detect real issues in GCC and ICC compilers. A metamorphic approach has also been used to test GLSL compilers via opaque value injection [29].

In general, the application of the metamorphic approach to the compiler testing problem is similar to our approach. Instead of deriving program variants (follow-up programs) from original programs under test, we benefit from the existence of code generator families to generate equivalent implementations of the same program (instead of applying program transformations) and then, we define a metamorphic relation relative to the non-functional properties instead of a simple comparison of the functional outputs.

Metamorphic testing has been applied to numerous other problems presented by Segura et al. [47] such as detection of inconsistencies in online web search applications, solving complex numerical problems, testing image processing programs, etc.

For performance assessment, MT was also used by Segura et al. [48, 49] to reveal performance failures. MRs are used as fitness functions to guide the search-based algorithms in the context of the automated analysis of feature models.

5.3. Container-based testing approaches

The container technology is widely used in order to create a portable, consistent operating environment for development, deployment and testing in the cloud [25, 50]. For example, Marinescu et al. [51] have used Docker as technological basis in their repository analysis framework Covrig to conduct a large-scale and safe inspection of the revision history from six selected Git code repositories. For their analysis, they run each version of a system in isolation and collect static and dynamic software metrics, using a lightweight container environment that can be deployed on a cluster of local or cloud machines. According to the authors, the use of Docker as a solution to automatically deploy and execute the different program reversions has clearly facilitated the testing process. Another Docker-based approach is presented in the BenchFlow2 project that focuses on benchmarking BPMN 2.0 engines [52]. This project is dedicated to the performance testing of workflow engines. In this work, Ferme et al. present a framework for automatic and reliable calculation of performance metrics for BPMN 2.0 Workflow Management Systems (WfMSs). BenchFlow exploits Docker as a containerization technology, to enable the automatic deployment and configuration of the WfMSs. Thanks to Docker, BenchFlow automatically collects all the data needed to compute the performance metrics and to check the correct execution of the tests (metrics related the RAM/CPU usage and execution time). Hamdy et al. [53] propose Pons, a web based tool for the distribution of pre-release mobile applications for the purpose of manual testing. Pons facilitates building, running and manually testing of Android applications directly in the browser. Based on Docker technology, this tool gets the developers and end users engaged in testing the applications in one place, alleviating the tester's burden of installing and maintaining testing environments and providing a platform for developers to rapidly iterate on the software and integrate changes over time.

Resource usage monitoring of containers has been also applied to solve several research problems. As an example, Kookarinrat et al. [54] have investigated the problem of auto-sharding in

NoSQL databases using a container-based infrastructure for runtime monitoring. Therefore, authors analysed and evaluated the variation of a shard key's choices on the DB performance. They simulated an environment using Docker containers and measured the read/write performance of variety of keys. Inside each container, they executed write/read queries into the MongoDB database and used Docker stats to automatically retrieve information about the memory and CPU usage. Sun et al. [55] present a tool to test, optimize and automate cloud resource allocation decisions to meet QoS goals for web applications. Their infrastructure relies on Docker to gather information about the resource usage of deployed web servers. Containers' monitoring has been applied in other research efforts related especially to cloud computing and virtualization [25, 56].

6. CONCLUSION

In this paper, we have described a metamorphic testing approach for automatic detection of code generator inconsistencies in terms of non-functional properties (i.e., resource usage and performance). We detect inconsistencies when the variation of performance or resource usage of test suites across the different targets exceeds a specific threshold value. We apply two statistical methods (i.e., principal component analysis and range charts) in order to evaluate the data variation. We also described a container-based testing environment for deploying, executing and monitoring the resource usage of generated code in multiple target software platforms. The experimental results show that our approach is able to detect, among 95 executed test suites, 11 performance and 15 memory usage inconsistencies, violating the metamorphic relation for Haxe code generators. We also applied our approach to the ThingML case study, showing the effectiveness of the proposed solution to automatically detect real issues in code generator families.

As a current work, we are discussing with the Haxe and ThingML communities in order to expand our testing approach, introducing new target software platforms to test and create new benchmark programs with high-quality test suites. As a future work, we aim to provide, in addition to our black-box approach, a traceability method that can be applied to track the inconsistency, at the source code level. Thus, test suites triggering inconsistencies can be investigated in-depth in order to identify the source of the inconsistency (e.g., parts of the code that affect software performance) and to fix the issues (e.g., transformation rules, templates, etc). We intend also to deploy tests on many nodes in the cloud using multiple containers in order to speed up the time required to run experiments. Finally, we may evaluate the impact of the new code generator improvements (i.e., running the same experiments with new code generator versions) and check if the fixes have eliminated the previously identified inconsistencies.

REFERENCES

1. Betz T, Cabac L, Güttler M. Improving the development tool chain in the context of petri net-based software development. *PNSE*: Newcastle, UK, 2011; 167–178.
2. Czarnecki K, Eisenecker UW. *Generative programming*, 2000. Edited by G. Goos, J. Hartmanis, and J. van Leeuwen, 15.
3. France R, Rumpe B. Model-driven development of complex software: a research roadmap. In *2007 future of software engineering*. IEEE Computer Society: Minneapolis, MN, USA, 2007; 37–54.
4. Guana V, Stroulia E. How do developers solve software-engineering tasks on model-based code generators? An empirical study design. *First International Workshop on Human Factors in Modeling (HUFAMO 2015). CEUR-WS*: Ottawa, Canada, 2015; 33–38.
5. Delgado N, Gates AQ, Roach S. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering* 2004; **30**(12):859–872.
6. Guana V, Stroulia E. Chaintracker, a model-transformation trace analysis tool for code-generation environments. *7th International Conference on Model Transformation (ICMT14)*: Springer: York, UK, 2014; 146–153.
7. Stuermer I, Conrad M, Doerr H, Pepper P. Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering* 2007; **33**(9):622.
8. Yang X, Chen Y, Eide E, Regehr J. Finding and understanding bugs in c compilers. *ACM SIGPLAN Notices*, Vol. 46: ACM: San Jose, CA, USA, 2011; 283–294.
9. McKeeman WM. Differential testing for software. *Digital Technical Journal* 1998; **10**(1):100–107.
10. Vouk MA. Back-to-back testing. *Information and software technology* 1990; **32**(1):34–45.

11. Boussaa M, Barais O, Baudry B, Sunyé G. Automatic non-functional testing of code generators families. *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*: ACM: Amsterdam, Netherlands, 2016; 202–212.
12. Boussaa M. Automatic non-functional testing and tuning of configurable generators. *Ph.D. Thesis*, Université Rennes 1, 2017.
13. Chae W, Blume M. Building a family of compilers. *Software Product Line Conference, 2008. SPLC'08. 12th International*, IEEE: Limerick, Ireland, 2008; 307–316.
14. Fumero JJ, Remmelg T, Steuwer M, Dubach C. Runtime code generation and data management for heterogeneous computing in java. *Proceedings of the principles and practices of programming on the java platform*: ACM: Melbourne, FL, USA, 2015; 16–26.
15. Dasnois B. *Haxe 2 Beginner's Guide*. Packt Publishing Ltd, 2011.
16. Fleurey F, Morin B, Solberg A, Barais O. Mde to manage communications with and between resource-constrained systems. *International conference on model driven engineering languages and systems*: Springer: Wellington, New Zealand, 2011; 349–363.
17. Rastogi A, Swamy N, Fournet C, Bierman G, Vekris P. Safe & efficient gradual typing for typescript. *ACM SIGPLAN Notices*, Vol. 50: ACM: Mumbai, India, 2015; 167–180.
18. Barr ET, Harman M, McMinn P, Shahbaz M, Yoo S. The oracle problem in software testing: a survey. *IEEE Transactions on Software Engineering* 2015; **41**(5):507–525.
19. Richard-Foy J, Barais O, Jézéquel J-M. Efficient high-level abstractions for web programming. *ACM SIGPLAN Notices*, Vol. 49: ACM: Indianapolis, IN, USA, 2013; 53–60.
20. Stepasyuk S, Paunov Y. *Evaluating the haxe programming language-performance comparison between haxe and platform-specific languages*, 2015.
21. Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, Vol. 42: ACM: San Diego, California, USA, 2007; 89–100.
22. Soltesz S, Pötzl H, Fiuczynski ME, Bavier A, Peterson L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *ACM SIGOPS Operating Systems Review*, Vol. 41: ACM: Lisbon, Portugal, 2007; 275–287.
23. Spoiala CC, Calinciuc A, Turcu CO, Filote C. Performance comparison of a webrtc server on docker versus virtual machine. *2016 International Conference on Development and Application Systems (DAS)*: IEEE: Suceava, Romania, 2016; 295–298.
24. Merkel D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014; **2014**(239):2.
25. Peinl R, Holzschuher F, Pfitzer F. Docker cluster management for the cloud-survey results and own solution. *Journal of Grid Computing* 2016; **14**(2):265–282.
26. Harman M, McMinn P, Shahbaz M, Yoo S. A comprehensive survey of trends in oracles for software testing. *Technical Report Tech. Rep. CS-13-01*, University of Sheffield, Department of Computer Science: Sheffield, UK, 2013.
27. Chen TY, Cheung SC, Yiu SM. Metamorphic testing: a new approach for generating next test cases. *Technical Report Technical Report HKUST-CS98-01*, Department of Computer Science, Hong Kong University of Science and Technology: Hong Kong, 1998.
28. Zhou ZQ, Huang D, Tse T, Yang Z, Huang H, Chen T. Metamorphic testing and its applications. *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*: Xian, China, 2004; 346–351.
29. Donaldson AF, Lascu A. Metamorphic testing for (graphics) compilers. *Proceedings of the 1st international workshop on metamorphic testing*: ACM: Austin, TX, USA, 2016; 44–47.
30. Le V, Afshari M, Su Z. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices*, Vol. 49: ACM: Edinburgh, UK, 2014; 216–226.
31. Tao Q, Wu W, Zhao C, Shen W. An automatic testing approach for compiler based on metamorphic testing technique. *2010 17th Asia Pacific Software Engineering Conference (APSEC)*: IEEE: Sydney, NSW, Australia, 2010; 270–279.
32. Chan W, Chen TY, Lu H, Tse T, Yau SS. Integration testing of context-sensitive middleware-based applications: a metamorphic approach. *International Journal of Software Engineering and Knowledge Engineering* 2006; **16**(5):677–703.
33. Malik H, Hemmati H, Hassan AE. Automatic detection of performance deviations in the load testing of large scale systems. *Proceedings of the 2013 International Conference on Software Engineering*: IEEE Press: San Francisco, CA, USA, 2013; 1012–1021.
34. Enot DP, Lin W, Beckmann M, Parker D, Overy DP, Draper J. Preprocessing, classification modeling and feature selection using flow injection electrospray mass spectrometry metabolite fingerprint data. *Nature Protocols* 2008; **3**(3):446–470.
35. Hubert M, Rousseeuw P, Verdonck T. Robust PCA for skewed data and its outlier map. *Computational Statistics & Data Analysis* 2009; **53**(6):2264–2274.
36. Burnard A, Rover L. Verifying and validating automatically generated code. *Proc. of International Automotive Conference (IAC)*: Citeseer: Stuttgart, Germany, 2004; 71–78.
37. Conrad M. Testing-based translation validation of generated code in the context of IEC 61508. *Formal Methods in System Design* 2009; **35**(3):389–401.

38. Conrad TE, Maier-Komor T, Sandmann G, Pomeroy M. Code generation verification–assessing numerical equivalence between simulink models and generated code. *4th Conference Simulation and Testing in Algorithm and Software Development for Automobile Electronics*: Berlin, Germany, 2010.
39. Jörges S, Steffen B. Back-to-back testing of model-based code generators. *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Springer: Corfu, Greece, 2014; 425–444.
40. Sturmer I, Conrad M. Test suite design for code generation tools. *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings*: IEEE: Montreal, Que., Canada, Canada, 2003; 286–290.
41. Zelenov SV, Silakov DV, Petrenko AK, Conrad M, Fey I. Automatic test generation for model-based code generators. *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*: Paphos, Cyprus, 2006; 75–81.
42. Štrekelj D, Leventić H, Galić I. Performance overhead of haxe programming language for cross-platform game development. *International Journal of Electrical and Computer Engineering Systems* 2015; **6**(1):9–13.
43. Ajwad N. Evaluation of automatic code generation tools. *MSc Theses*, Department of Automatic Control, 2007.
44. Pazirandeh A, Vorobyeva E. *Evaluation of cross-platform tools for mobile development*, 2015.
45. Hartmann G, Stead G, DeGani A. Cross-platform mobile development. *Mobile Learning Environment, Cambridge* 2011; **16**(9):158–171.
46. Chen TY, Huang D, Tse T, Zhou ZQ. Case studies on the selection of useful relations in metamorphic testing. *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*: Polytechnic University of Madrid: Madrid, Spain, 2004; 569–583.
47. Segura S, Fraser G, Sanchez AB, Ruiz-Cortés A. A survey on metamorphic testing. *IEEE Transactions on software engineering* 2016; **42**(9):805–824.
48. Segura S, Troya J, Durán A, Ruiz-Cortés A. Performance metamorphic testing: motivation and challenges. *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track*: IEEE Press: Buenos Aires, Argentina, 2017; 7–10.
49. Segura S, Troya J, Durán A, Ruiz-Cortés A. Performance metamorphic testing: a proof of concept. *Information and Software Technology* 2018; **98**:1–4.
50. Li L, Tang T, Chou W. A rest service framework for fine-grained resource management in container-based cloud. *2015 IEEE 8th international conference on cloud computing*: IEEE: New York, NY, USA, 2015; 645–652.
51. Marinescu P, Hosek P, Cadar C. Covrig: A framework for the analysis of code, test, and coverage evolution in real software. *Proceedings of the 2014 international symposium on software testing and analysis*: ACM: San Jose, CA, USA, 2014; 93–104.
52. Ferme V, Ivanchikj A, Pautasso C. A framework for benchmarking bpmn 2.0 workflow management systems. *International conference on business process management*: Springer, 2015; 251–259.
53. Hamdy A, Ibrahim O, Hazem A. A web based framework for pre-release testing of mobile applications. *MATEC Web of Conferences*, Vol. 76: EDP Sciences: Corfu, Greece, 2016; 4041.
54. Kookarinrat P, Temtanapat Y. Analysis of range-based key properties for sharded cluster of mongodb. *2015 2nd International Conference on Information Science and Security (ICISS)*: IEEE: Seoul, South Korea, 2015; 1–4.
55. Sun Y, White J, Eade S, Schmidt DC. Roar: a qos-oriented modeling framework for automated cloud resource allocation and optimization. *Journal of Systems and Software* 2016; **116**:146–161.
56. Medel V, Rana O, Arronategui U, Bañares JÁ. Modelling performance & resource management in kubernetes. *Proceedings of the 9th International Conference on Utility and Cloud Computing*: ACM: Shanghai, China, 2016; 257–262.