Towards a Container-based Approach for Non-functional Testing of Code Generators

Mohamed Boussaa Olivier Barais
Benoit Baudry
Diverse team INRIA, Rennes, France
{mohamed.boussaa, olivier.barais,
benoit.baudry}@inria.fr

Gerson Sunyé
AtlanMod team INRIA, Nantes, France
gerson.sunye@inria.fr

Abstract

The intensive use of domain specific languages (DSL) and generative programming techniques provides an elegant engineering solution to deal with the heterogeneity of platforms or technological stacks. The use of DSLs also leads to the creation of numerous code generators and compilers that will automatically translate high-level system specifications into multi-target executable code and scripts. However, the use of code generators may induce different bugs and inconsistencies within generated software artifacts. Although software designers provide generally high-level test suites to verify the functional outcome of generated code, it remains challenging and tedious to verify the behavior of produced code in terms of non-functional properties.

This paper describes a runtime monitoring infrastructure based on system containers, as execution platforms, to evaluate the consistency and coherence of generated code regarding the non-functional requirements. This approach provides a fine-grained understanding of resource consumption and analysis of components behavior. We evaluate our approach by analyzing the non-functional properties of HAXE, a popular high-level programming language that involves a set of cross-platform code generators able to compile to different target platforms. Our experimental results show that our approach is able to detect some non-functional inconsistencies within HAXE code generators.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors compilers, code generation

General Terms Generative Programming, Testing, Components

Keywords code quality, non-functional properties, code generator, testing

1. Introduction

Nowadays, the intensive use of generative programming techniques has become a common practice for software development since it reduces the development and maintenance effort by developing at a higher-level of abstraction through the use of domain-specific languages [2] (DSLs). DSLs, as opposed to general-purpose languages, are software languages that focus on specific problem domains. Thus, the realization of model-driven software development for a specific domain requires the creation of effective code generators and compilers for these DSLs. Code-generation environments automate and systematize the process of building families of software systems which can clearly reduce the effort of software implementation. Indeed, a code generator is needed to transform manually designed models (source code programs represented in a graphical/textual modeling language) to general purpose programming languages such as C, Java, C++, etc. In turn, generated code is transformed into machine code (binaries) using a set of specific compilers. These compilers serve as a basis to target different ranges of platforms.

However, code generators can be difficult to understand since they involve a set of complex and heterogeneous technologies which make the task of performing design, implementation, and testing very hard and time-consuming [6, 8]. Code generators have to respect different requirements which preserve software reliability and quality. In fact, faulty code generators can generate defective software artifacts which range from uncompilable or semantically dysfunctional code that causes serious damage to the target platform, to non-functional bugs which lead to poor-quality code that can affect system reliability and performance (e.g., high resource usage, high execution time, etc.).

Generally, in order to check the correctness of code generation process, developers often use to define (at design time) a set of test cases that verify the functional outcome

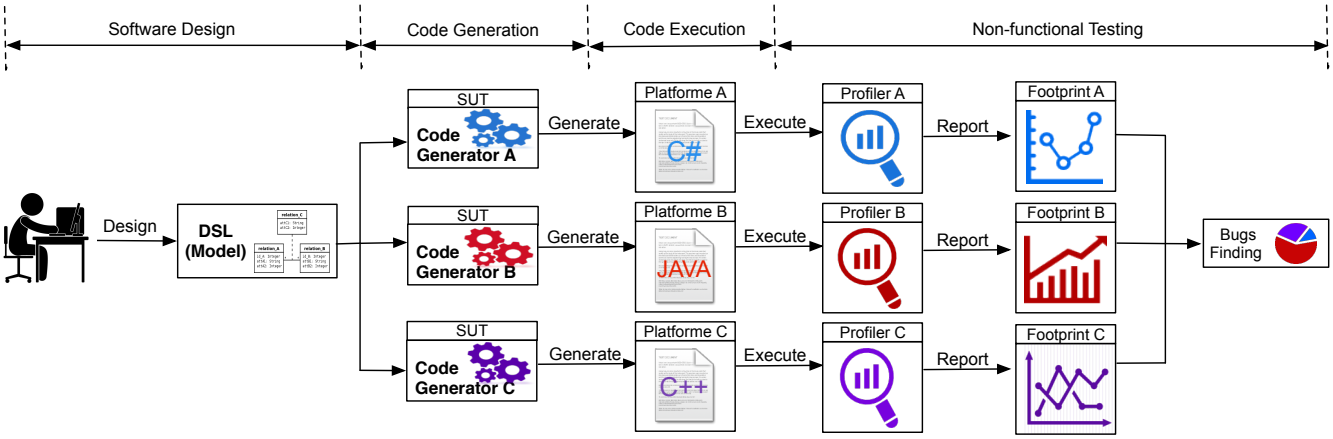

Figure 1. An overall overview of the different processes involved to ensure the code generation and non-functional testing of produced code from design time to run time.

of generated code. After code generation, test suites are executed within each target platform. This may lead to either a correct behavior (i.e., expected output) or a failure (i.e., crashes, bugs).

On the other hand, for non-functional testing of code generators, developers need to deploy and execute software artifacts within different execution platforms. Then, they have to collect and visualize information about the performance and efficiency of generated code. Finally, they report issues related to the code generation process such as incorrect typing, memory management leaks, etc. To do so, developers generally use several platform-specific profilers, trackers, and monitoring tools [4, 7] in order to find some inconsistencies or bugs during code execution. Ensuring the code quality of generated code can refer to several non-functional properties such as code size, resource or energy consumption, execution time, among others [13]. Therefore, we believe that testing the non-functional properties of code generators remains challenging and time-consuming task because developers have to analyze and verify code for each target platform using platform-dependent tools.

This paper describes a runtime monitoring infrastructure based on system containers, as execution platforms, to evaluate the consistency and coherence of generated code regarding the non-functional properties. This approach provides a fine-grained understanding of resource consumption and analysis of components behavior. We evaluate our approach by analyzing the non-functional properties of HAXE, a popular high-level programming language that involves a set of cross-platform code generators able to compile to different target platforms. Our experimental results show that our approach is able to detect non-functional inconsistencies within HAXE code generators.

In this paper, we make the following contributions:

- We propose a micro-service infrastructure to ensure the deployment and monitoring of generated code. In this paper, we focus more on the relationship between runtime execution of generated code and resource consumption profiles (CPU and memory usage).
- 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 infrastructure is efficient and effective.

The paper is organized as follows. Section II describes the background and motivation behind this work. We present in Section III our infrastructure for non-functional testing of code generators using micro-services. The evaluation and results of our experiments are discussed in Section IV. Finally, related work, concluding remarks and future work are provided in Sections V and VI.

2. Background and Motivation

A reliable and accepted way to increase confidence in the correct functioning of code generators is to validate and check the functionality of generated code, which is common practice for compiler validation and testing. Therefore, developers try to check the syntactic and semantic correctness of generated code by means of different techniques such as static analysis, test suites, etc., and ensure that the code is behaving correctly. In model-based testing for example [9, 15], testing code generators focuses on testing the generated code against its design. Thus, the model and the generated code are executed in parallel, by means of simulations, with the same set of test suites. Afterwards, the two outputs are compared with respect to certain acceptance criteria. Test cases, in this case, can be designed to maximize the code or model coverage [16].

Another important aspect of code generator's testing is to test the non-functional properties of produced code. Proving that the generated code is functionally correct is not enough to claim the effectiveness of the code generator under test. In

fact, code generators have to respect different requirements which preserve software reliability and quality [5]. A non-efficient code generator might generate defective software artifacts (code smells) that violates common software engineering practices. Thus, poor-quality code can affect system reliability and performance (e.g., high resource usage, low execution speed, etc.).

Figure 1 shows an overall overview of the different processes that are involved together to ensure the code generation and non-functional testing of produced code from design time to run time. We distinguish 4 major steps: the software design using high-level system specifications, code generation by means of code generators, code execution, and non-functional testing of generated code.

Firstly, software developers have to define, at design time, software's behavior using a high-level abstract language, generally DSLs. Afterwards, developers can use platform-specific code generators to ease the software development and generate automatically code that targets different languages and platforms. We depict, in Figure 1, three code generators capable to generate code in three software programming languages (JAVA, C# and C++). In this step, code generators transform the previously designed model to produce, as a consequence, software artifacts for the target platform. Transformations from model to code within each code generator might be different and may integrate different transformation rules. As an example, we distinguish model-to-model transformations languages such as ATL [10] and template-based model-to-text transformation languages such as Acceleo [11] to translate high-level system specifications into executable code and scripts [1, 3]. The main task of code generators is to transform models to general-purpose and platform-dependent languages. In the next step, generated software artifacts (e.g., JAVA, C#, C++, etc.) are compiled, deployed and executed across different target platforms (e.g., Android, ARM/Linux, JVM, x86/Linux, etc.). Thus, several code compilers are needed to transform source code to machine code (binaries) in order to get executed. Finally, to perform the non-functional testing of generated code, developers have to collect and visualize information about the performance and efficiency of running code across the different platforms. Therefore, they generally use several platform-specific profilers, trackers, instrumenting and monitoring tools in order to find some inconsistencies or bugs during code execution [4, 7]. Ensuring the code quality of generated code can refer to several non-functional properties such as code size, resource or energy consumption, execution time, among others [13]. Finding inconsistencies within code generators involves analyzing and inspecting the code and that, for each execution platform. For example, one of the methods to handle that is to analyze the memory footprint of software execution and find memory leaks. Developers then, can inspect the execution trace and find some parts of the code-base that have triggered this issue. Such non-functional error could occur when the code generator produces code that presents for example: incorrect typing, faulty memory management, code-smells, etc. Therefore, software testers generally use to report statistics about the performance of generated code in order to fix, refactor, and optimize the code generation process.

In short, then, we believe that testing the non-functional properties of code generators remains challenging and time-consuming task because developers have to analyze and verify code for each target platform using platform-dependent tools which makes the task of maintaining code generators very tedious. The heterogeneity of platforms and the diversity of target software languages increase the need of supporting tools that can evaluate the consistency and coherence of generated code regarding the non-functional properties. This paper describes a new approach, based on micro-services as execution platforms, to automate and ease the non-functional testing of code generators. This runtime monitoring infrastructure provides a fine-grained understanding of resource consumption and analysis of generated code's behavior.

3. An Infrastructure for Non-functional Testing Using System Containers

In general, there are many non-functional requirements that can be evaluated by software testers such as performance (execution time), code quality, robustness, resource usage, etc. In this paper, we focus on the non-functional properties related to the performance and efficiency of generated code in terms of resource consumption (memory and CPU).

In fact, to assess the performance/non-functional properties of generated code many system configurations (i.e., execution environments) must be considered. Running different applications (i.e., generated code) with different configurations on one single machine is complicated because a single system has limited resources and this can lead to performance regressions. Moreover, each execution environment comes with a collection of appropriate tools such as compilers, code generators, debuggers, profilers, etc.

Therefore, we need to deploy the test harness, i. e.the produced binaries, on an elastic infrastructure that that provides to compiler user facilities to ensure the deployment and monitoring of generated code in different environment settings.

Monitoring information should also be provided to inform about resource utilization required/needed and to automate the resource management of deployed components.

For this purpose, we propose a testing infrastructure based on System Container techniques such as Docker[1] environment. This framework automates the deployment and execution of applications inside software containers by allowing multiple program configurations to run autonomously

[1] `https://www.docker.com`

on different servers (i. e., a cloud servers). It also provides a distributed environment where system storage and resources can be finely managed and limited according to the needs.

Consequently, we rely on this component-based infrastructure and benefit from all its advantages to automatically:

1. Deploy the generated code within a set of containers
2. Execute the produces binaries in an isolated environment
3. Monitor service containers
4. Gather performance metrics (CPU, Memory, I/O, etc.)

Thus, we integrate a collection of components to define the adequate infrastructure for testing and monitoring of code generators. In the following sections, we describe the deployment and testing architecture of generated code within containers.

3.1 System Containers as Execution platforms

Before starting to monitor and test applications, we have to deploy generated code on different components to ease containers provisioning and profiling. We aim to use Docker Linux containers to monitor the execution of different generated artifacts in terms of resource usage. Docker is an open source engine that automates the deployment of any application as a lightweight, portable, and self-sufficient container that runs virtually on a host machine. Using Docker, we can define preconfigured applications and servers to host as virtual images. We can also define the way the service should be deployed in the host machine using configuration files called Dockerfiles. We use the Docker Hub[2] for building, saving, and managing all our Docker images. We can then instantiate different containers from these Docker images.

Therefore, each generated code is executed individually inside an isolated Linux container. By doing so, we ensure that each executed program runs in isolation without being affected by the host machine or any other processes. Moreover, since a container is cheap to create, we are able to create too many containers as long as we have new programs to execute. Since each program execution requires a new container to be created, it is crucial to remove and kill containers that have finished their job to eliminate the load on the system. In fact, containers/softwares are running sequentially without defining any resource constraints. So once 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.

In short, the main advantages of this approach are:

- The use of containers induces less performance overhead and resource isolation compared to using a full stack virtualization solution [14]. Indeed, instrumentation and monitoring tools for memory profiling like Valgrind [12] can induce too much overhead.
- Thanks to the use of Dockerfiles, the proposed framework can be easily configured by software testers in order to define the code generators under test (e. g., code generator version, dependencies, etc.), the host IP and OS, the DSL design, the optimization options, etc. Thus, we can use the same configured Docker image to execute different instances of generated code. For hardware architecture, containers share the same platform architecture as the host machine (e.g., x86, x64, ARM, etc.).
- Docker uses Linux control groups (cgroups) to group processes running in the container. This allows us to manage the resources of a group of processes, which is very valuable. This approach increases the flexibility when we want to manage resources, since we can manage every group individually. For example, if we would evaluate the non-functional requirements of generated code within a resource-constraint environment, we can easily request and limit resources within the execution container according to the needs.
- 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 easily gathered and managed by other containers (i. e., for storage purpose, visualization)

3.2 Runtime Testing Components

In order to test our running applications within Docker containers, we aim to use a set of Docker components to ease the extraction of non-functional properties related to resource usage.

3.2.1 Monitoring Component

This container provides an understanding of the resource usage and performance characteristics of our running containers. Generally, Docker containers rely on cgroups file systems to expose a lot of metrics about accumulated CPU cycles, memory, block I/O usage, etc. Therefore, our monitoring component automates the extraction of runtime performance metrics stored in cgroups files. For example, we access live resource consumption of each container available at the cgroup 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). This component will automate the process of service discovery and metrics aggregation for each new container. Thus, instead of gathering manually metrics located in cgroups file systems, it extracts automatically the runtime resource usage statistics relative to the running component (i.e., the generated code that is running within a container). We note that resource usage information is collected in raw data. This process may induce a little overhead because it does very fine-grained accounting of resource usage on running container. Fortunately, this may not affect the gathered performance values since we run only

[2] `https://hub.docker.com/`

one version of generated code within each container. To ease the monitoring process, we integrate cAdvisor, a Container Advisor[3]. It is a tool that monitors service containers at runtime.

However, cAdvisor monitors and aggregates live data over only 60 seconds interval. Therefore, we would like to record all data over time since container's creation. This is useful to run queries and define non-functional metrics from historical data. Thereby, to make gathered data truly valuable for resource usage monitoring, it becomes necessary to log it into a database at runtime. Thus, we link our monitoring component to a back-end database component.

3.2.2 Back-end Database Component

This component represents a time-series database back-end. It is plugged with the previously described monitoring component to save the non-functional data for long-term retention, analytics and visualization.

During the execution of generated code, resource usage stats are continuously sent to this component. When a container is killed, we are able to access to its relative resource usage metrics through the database. We choose a time series database because we are collecting time series data that correspond to the resource utilization profiles of programs execution.

We use InfluxDB[4], an open source distributed time-series database as a back-end to record data. InfluxDB allows the user to execute SQL-like queries on the database. 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'
```

To give an idea about the data gathered from the monitoring component and stored in the time-series database, we describe in Table 1 these collected metrics:

Metric	Description
Name	Container Name
T	Elapsed time since container's creation
Network	Stats for network bytes and packets in an out of the container
Disk IO	Disk I/O stats
Memory	Memory usage
CPU	CPU usage

Table 1. Resource usage metrics recorded in InfluxDB

Apart from that, our framework provides also information about the size of generated binaries and the compilation time needed to produced code. For instance, resource usage statistics are collected and stored using these two components. It is relevant to show resource usage profiles of running programs overtime. To do so, we present a front-end visualization component for performance profiling.

3.2.3 Front-end Visualization Component

Once we gather and store resource usage data, the next step is visualizing them. That is the role of the visualization component. It will be the endpoint component that we use to visualize the recorded data. Therefore, we provide a dashboard to run queries and view different profiles of resource consumption of running components through web UI. Thereby, we can compare visually the profiles of resource consumption among containers. Moreover, we use this component to export the data currently being viewed into static CSV document. So, we can perform statistical analysis on this data to detect inconsistencies or performance anomalies. To do so, we choose Grafana[5], a time-series visualization tool available for Docker.

3.3 Wrapping Everything Together: Architecture Overview

To summarize, we present, in Figure 2, an overall overview of the different components involved in our approach to perform the non-functional testing of code generators.

First, instead of configuring all code generators under test (GUTs) within the same host machine (as we presented in the motivation section), we wrap each GUT within a container. To do so, we create a new configuration image for each GUT (i.e., the Docker image) where we install all the libraries, compilers, and dependencies needed to ensure the code generation and compilation. Thereby, the GUT produce code within multiple instances of preconfigured Docker images. Afterwards, a new instance of the container is created to enable the execution of generated code in a isolated and configured environment. In this step, new configurations are needed to run each job without any issue (i.e., define the execution environment within the container, the host machine, the target platform, limit the resources, etc.). For our case, a job represents a generated and compiled program using one of the GUTs. Meanwhile, we start our runtime testing components (e.g., cAdvisor, InfluxDB and Grafana). The monitoring component collects usage statistics of all running containers and save them at runtime in the time series database component. The visualization component comes later to allow end users to define performance metrics and draw up charts. The use of the front-end visualization component is optional and we can directly access to information stored in the database through REST API calls. Thus, we can observe later informations about the resource usage of generated code and detect inconsistencies in the generated code by GUTs.

[3] https://github.com/google/cadvisor

[4] https://github.com/influxdata/influxdb

[5] https://github.com/grafana/grafana

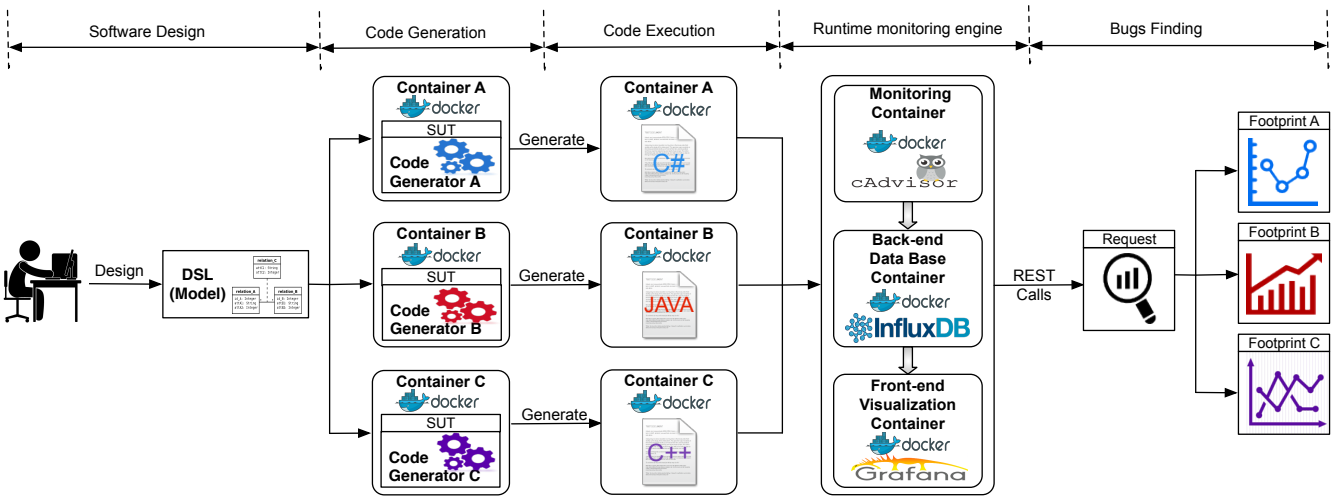

Figure 2. An overall overview of the different processes involved to ensure the code generation and non-functional testing of produced code from design time to runtime.

Remark. *We would notice that this testing infrastructure can be generalized and adapted to other case studies other than code generators. Using system containers, any software application/generated code can be easily deployed within containers (i.e., by configuring the container image). It will be later executed and monitored using our runtime monitoring engine.*

4. Evaluation

So far, we have presented a sound procedure and automated component-based framework for extracting the non-functional properties of generated code. In this section, 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[6].

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 and its code generators.

Haxe 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, multi-platform compilers, and different native libraries. The Haxe language is a high-level programming language which is strictly typed. This language supports both functional programming and object-oriented programming paradigms. It has a common type hierarchy, making certain API available on every targeted platform. Moreover, Haxe comes with a set of compilers that translate manually-written code (in Haxe language) to different target languages and platforms. Haxe code can be compiled for applications running on desktop, mobile and web platforms. Compilers ensure the correctness of user code in terms of syntax and type safety. Haxe comes also with a set of standard libraries that can be used on all supported targets and platform-specific libraries for each of them.

The process of code transformation and generation can be descibed as following: Haxe compilers analyzes 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 target platform/language.

Haxe offers the option of choosing which platform to target for each program using a command-line tool. Moreover, some optimizations and debugging information can be enabled through CLI but in our experiments, we did not turned on any further options.

4.1.2 Cross-platform Benchmarking

One way to prove the effectiveness of our approach is to create benchmarks. Thus, we use Haxe language, code generators, and a set of libraries to build a cross-platform benchmark based on this language. In fact, the provided benchmark is composed of a collection of cross-platform libraries that can be compiled to different targets[7]. In these experiments, we are considering five targets: Java, JS, C++, CS, and PHP. 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 covers the different features of the API. They can be executed within the target platform to test the functional behavior of generated programs. We use these test

[6] `https://testingcodegenerators.wordpress.com/`

[7] `http://thx-lib.org/`

suites then, to generate a load and stress the target library. This can be useful to study the impact of this load on the resource usage of the system. In fact, we have removed all the tests that fail to compile to our five targets and we kept only test suites that are functionally correct. Thus, we run each test suite 1000 times to get comparable values in terms of resource usage. Table 2 describes the Haxe libraries that we have selected from this benchmark to evaluate our approach.

Library	#TestSuites	Description
Color	19	Color conversion from/to any color 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	3	Library for lightweight promises and futures
Culture	4	Localization library for Haxe
Math	3	Generation of random values

Table 2. Description of selected benchmark libraries

4.1.3 Evaluation Metrics Used

We use to evaluate the efficiency of generated code using the following non-functional metrics:

-*Memory Usage (MU)*: It corresponds to the maximum memory consumption of the running container under test. Memory usage is measured in bytes.

-*Execution Time (T)*: Program execution time is measured in seconds.

4.1.4 Setting up Infrastructure

To assess our approach, we configure our previously proposed container-based infrastructure for non-functional testing of code generators in order to run experiments on the Haxe case study. Figure 3 shows a big picture of the testing and monitoring infrastructure considered in these experiments.

First, we create a new Docker image in where we install the Haxe code generators and compilers (through the configuration file "Dockerfile"). Then a new instance of that image is created. It takes as an input the Haxe library we would to test and the list of test suites (step 1). It produces as an output the source code and binaries that have to be executed. These files are saved in a shared repository. In Docker environment, this repository is called Data Volume. A data volume is a specially-designated directory within containers that share data with the host machine. So, when we execute the generated test suites, we provide a shared volume with the host machine so that, binaries can be executed in the execution container (Step 2). In fact, for the code execution we created, as well, a new Docker image in where we install all execution tools and environments such as php interpreter, NodeJS, etc.

In the meantime, while running test suites inside the container, we collect runtime resource usage data using cAdvisor (step 3). The cAdvisor Docker image does not need any configuration on the host machine. We have just to run it on our host machine. It will then have access to resource usage and performance characteristics of all running containers. This image uses the cgroups mechanism described previously to collect, aggregate, process, and export ephemeral real-time information about running containers. Then, it reports all statistics via web UI (*http://localhost:8080*) to view live resource consumption of each container. cAdvisor has been widely used in different projects such as Heapster[8] and Google Cloud Platform[9]. In this experiment, we choose to gather information about the memory usage of running container. Afterwards, we record these data into a new time-series database using our InfluxDB back-end container (step 4). Thus, we define its corresponding ip port into the monitoring component so that, container statistics are sent over TCP port (e.g., *8083*) exposed by the database component.

Next, we run Grafana and we link it to InfluxDB by setting up the data source port 8086 so that, it can easily request data from the database. We recall that InfluxDB also provides a web UI to query the database and show graphs (step 5). But, Grafana let us display live results over time in much pretty looking graphs. Same as InfluxDB, we use SQL queries to extract non-functional metrics from the database for visualization and analysis (step 6). In our experiment, we are gathering the maximum memory usage values without presenting the graphs of resource usage profiles.

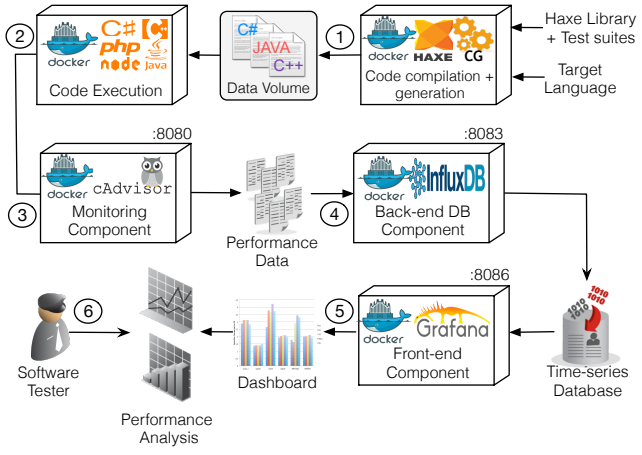

Figure 3. Comparison of average memory consumption and execution time of FFmpeg containers compiled with standard GCC optimization options

To obtain comparable and reproducible results, we use the same hardware across all experiments: an AMD A10-7700K

[8] `https://github.com/kubernetes/heapster`

[9] `https://cloud.google.com/`

	JAVA			JS			CPP			CS			PHP		
	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
Color	2.72	0,69	28,58	1,67	0,52	16,66	3,52	0,37	48,31	3,81	0,52	45,97	23,16	0,61	279,27
Core	0,82	0,57	8,71	0,86	0,51	14,01	0,61	0,42	8,11	0,98	0,50	21,60	49,52	0,47	2479,07
Hxmath	1,62	0,66	3,38	1,1	0,54	2,04	0,81	0,39	1,42	1,64	0,56	4,01	26,53	1,06	73,44
Format	2,19	0,71	5,03	2,23	0,60	5,402	2,57	0,37	7,67	4,97	0,54	12,38	93,53	1,11	220,75
Promise	0,89	0,63	1,72	1,44	0,54	2,09	0,69	0,39	1,12	0,99	0,48	1,71	11,96	1,13	31,18
Culture	0,64	0,55	0,82	0,50	0,46	0,57	0,44	0,37	0,62	0,50	0,43	0,65	1,46	0,61	3,79
Math	5,26	0,98	12,51	1,66	0,75	3.10	6,30	0,93	16,29	5,75	0,94	14,14	523,17	55,45	1448,9

Table 3. Execution time stats

	JAVA			JS			CPP			CS			PHP		
	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
Color	115,98	0,53	1362,55	62,33	0,70	900,70	169,90	0,16	2275,49	87,51	0,42	1283,3	173,69	0,81	2189,85
Core	21,94	0,52	1057,32	2,19	0,64	59,96	1,89	0,29	35,97	2,38	0,59	42,92	2,49	0,46	32,91
Hxmath	121,47	0,70	389,72	37,23	1,37	111,68	96,15	1,20	296,43	50,19	0,81	156,40	472,26	1,11	1192,98
Format	214,04	0,67	685,90	38,93	2,42	92,34	70,29	1,12	204,85	31,11	1,08	69,16	177,17	0,58	385,11
Promise	32,85	1,23	128,29	31,65	0,54	55,29	0,93	0,45	1,73	12,85	0,87	34,49	57,96	0,65	134,20
Culture	2,20	1,15	4,04	1,12	0,70	1,59	1,68	1,35	2,02	3,34	0,55	11,20	10,11	1,11	36,32
Math	353,18	1,21	831,44	165,71	0,93	493,66	597,02	1,61	1492,97	312,37	3,37	806,33	1448,45	626,83	3088,15

Table 4. Memory stats

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.

4.2 Experimental Results

The experimental consisted of comparing the execution time and memory consumption on the following targets: Java, JS, C++, CS, and PHP.

4.3 Discussions

5. Related Work

6. Conclusion and Future Work

Acknowledgments

This work was funded by the European Union Seventh Framework Program (FP7/2007-2013) under grant agreement n611337, HEADS project (www.heads-project.eu)

References

[1] A. Bragança and R. J. Machado. Transformation patterns for multi-staged model driven software development. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 329–338. IEEE, 2008.

[2] M. Brambilla, J. Cabot, and M. Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[3] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003.

[4] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on software Engineering*, 30(12):859–872, 2004.

[5] M. Demertzi, M. Annavaram, and M. Hall. Analyzing the effects of compiler optimizations on application reliability. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 184–193. IEEE, 2011.

[6] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[7] V. Guana and E. Stroulia. Chaintracker, a model-transformation trace analysis tool for code-generation environments. In *ICMT*, pages 146–153. Springer, 2014.

[8] V. Guana and E. Stroulia. How do developers solve software-engineering tasks on model-based code generators? an empirical study design. In *First International Workshop on Human Factors in Modeling (HuFaMo 2015). CEUR-WS*, pages 33–38, 2015.

[9] S. Jörges and B. Steffen. Back-to-back testing of model-based code generators. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 425–444. Springer, 2014.

[10] F. Jouault and I. Kurtev. Transforming models with atl. In *International Conference on Model Driven Engineering Languages and Systems*, pages 128–138. Springer, 2005.

[11] J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire. Acceleo user guide. *See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf*, 2, 2006.

[12] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[13] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *International Symposium on Code Generation and Optimiza-*

tion (CGO'06), pages 12–pp. IEEE, 2006.

[14] C. C. Spoiala, A. Calinciuc, C. O. Turcu, and C. Filote. Performance comparison of a webrtc server on docker versus virtual machine. In *2016 International Conference on Development and Application Systems (DAS)*, pages 295–298. IEEE, 2016.

[15] I. Stuermer, M. Conrad, H. Doerr, and P. Pepper. Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering*, 33(9):622, 2007.

[16] I. Stürmer, D. Weinberg, and M. Conrad. Overview of existing safeguarding techniques for automatically generated code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–6. ACM, 2005.