An Infrastructure for Testing Code Generators Using System Containers

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 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 language that involves a set of cross-platform code generators able to compile to different targets. Our initial 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

In model-driven software engineering, 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 (Brambilla et al. 2012) (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[ref]. Code generators have to respect different requirements which preserve software reliability and quality. In fact, a defective code generator 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, low execution speed, etc.).

Generally, in order to check the correctness of code generation process, developers often use to define (at design

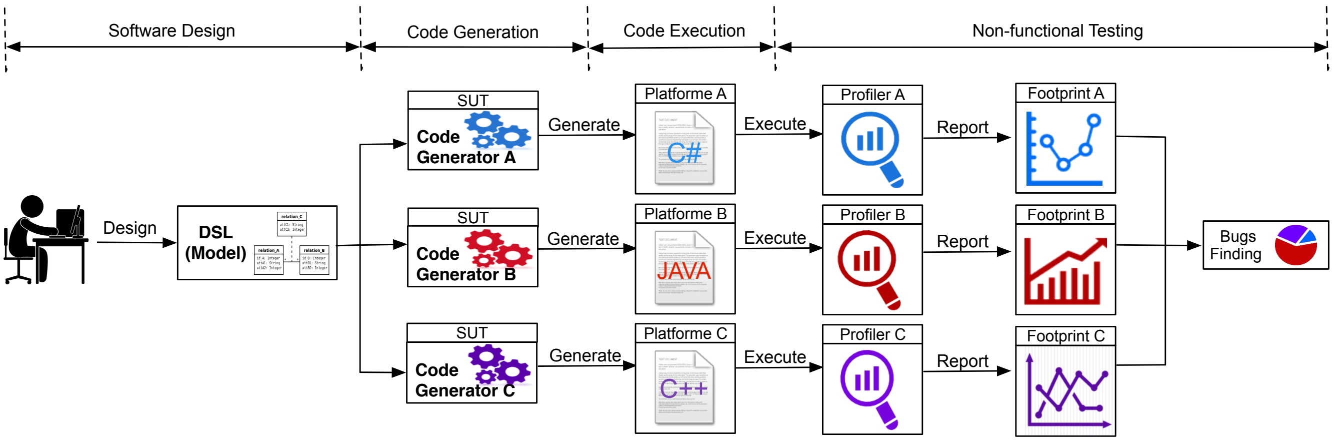

Figure 1. An overall overview of the different processes involved to ensure the code generation and non-functional testing of produced code from design time to runtime.

time) a set of test cases that verify the functional outcome 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). 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 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 [ref]. 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 language that involves a set of cross-platform code generators able to compile to different targets. Our initial 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 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, 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. [ref back to back testing/systematic]

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. 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.

So first, software developers have to create, at design time, new models that describe software's behavior using a high-level design 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, the code generator takes as an input the previously designed model and produce, as a consequence, software artifacts for the desired 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[ref] and template-based model-to-text transformation languages such as Acceleo [ref] to translate high-level system specifications into executable code and scripts [ref WS]. 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.) with their generated test suites 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. 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. Finding inconsistencies within code generators involves analyzing and inspecting the code and that, for each execution platform. For example, one of the solutions is to analyze the memory footprint of software execution and find memory leaks. Developers then, can inspect the source code 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 Docker Containers

In general, to assess the performance/non-functional properties of generated code many system configurations, i.e., execution environment configurations, must be considered. Running different applications 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. //provides preconfigured virtual server images, storage, and resources that may be provisioned by //testers. Docker will automate the deployment of applications inside software containers. It will simplify the creation of highly distributed systems by allowing multiple applications to run autonomously on a server (i. e., a cloud server). Docker provides a platform as a service (PaaS) style of deployment for

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

software programs. Consequently, we rely on this technology and benefit from all its advantages to:

1. Deploy generated code within Docker containers
2. Automate optimization sequences generation
3. Monitor service containers
4. Gather performance metrics (CPU, Memory, I/O, etc.)

We integrate a collection of Docker 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 Docker.

3.1 System Container as Deployment Environment

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 produced binaries by GCC compilers in term 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. To achieve that, Docker uses the Linux container technology. The main advantages that Docker offers compared to using a full stack virtualization solution is less performance overhead and resource isolation.

Using Docker, we can define preconfigured applications and servers to host. We can also define the way the service should be deployed in the host machine. As properties, we can define the OS where the service has to run, dependencies, etc. Once Docker images are defined, we can instantiate different containers. A simple way to define Docker images is to use Dockerfiles. Docker can build images automatically by reading the instructions from a Dockerfile. Therefore, for our experiments we describe a Dockerfile that defines the target compiler to test, as well the container OS. The same Docker image will be used then, to execute different instances of generated code. Basically, each container deploys an optimized version of the input source code program.

Docker uses as well 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.

Therefore, to run our experiments, each optimized program 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/programs are running sequentially without defining any constraints on resource utilization for each container. 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 the performance trade-offs.

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.

3.2.1 Monitoring Component

This container will provide us 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 using cgroups. For example, we access to 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 so that, instead of gathering manually metrics located in cgroups file systems, it will extract automatically runtime resource usage statistics relative to running components. 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 one generated program by GCC within each container.

To ease the monitoring process, we use google containers called cAdvisor as Container Advisor[2]. It is a tool developed by Google to monitor their infrastructure. cAdvisor Docker image does not need any configuration on the host machine. We have just to run it on our Docker host. It will then have access to the resource usage and performance characteristics of all running containers. This image uses the cgroups mechanism described above 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 in different projects such as Heapster[3] and Google Cloud Platform[4].

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

[3] https://github.com/kubernetes/heapster

[4] https://cloud.google.com/

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 resources usage monitoring, it becomes necessary to log it in a database at runtime. Thus, we link our monitoring component to a back-end database for better understanding of non-functional properties.

3.2.2 Back-end Database Component

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

During the execution of generated code, resource usage stats will be continuously sent into this component. When a container is killed, all statistics will be deleted afterward. We choose a time series database because we are collecting time series data that corresponds to the resource utilization profile of generated code execution.

We use InfluxDB[5], 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 average memory usage of container ”$generated\_code\_v1$” for each 2s since container has started:

```
select mean(memory_usage) from stats where
container_name='generated_code_v1' group by
time(2s)
```

To give an idea about data stored in InfluxDB. The following table describes the different stored metrics:

Name	Name of the container
Ts	Starting time of the container in epoch time
Network	Stats for network bytes and packets in an out of the container
Disk IO	Disk I/O stats
Memory	Memory usage
CPU	Cumulative CPU usage

Table 1. Resource usage metrics recorded in InfluxDB

For instance, we set the back-end container of our component-based infrastructure. It would be nice to pull all pieces together to view resource consumption graphs within a complete dashboard. It is relevant to show performance profiles of memory and CPU consumption for example of our running applications overtime. To do that, 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. Thereby, we can perform statistical analysis on this data to detect inconsistencies or performance anomalies. An overview of the monitoring dashboard is shown in Figure 3.

To do so, we choose Grafana[6], one of the best time-series metric visualization tools available for Docker. It is considered as a web application running within a container. We run Grafana within a container 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. But, Grafana will let us to 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.

3.3 Wrapping Everything Together: Architecture Overview

To summarize, we present, as shown in Figure 1, an overall overview of the different components involved in our Docker monitoring infrastructure.

Our testing infrastructure will run different jobs within Docker containers. First, we generate and run different versions of code using our target compiler. To do so, we run multiple instances of our preconfigured Docker image that corresponds to specific code generator (e.g, GCC compiler). Each container will execute a specific job. For our case, a job represents a program compiled with new optimization sequence generated by NS. In the 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.

4. Evaluation

zSo 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

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

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

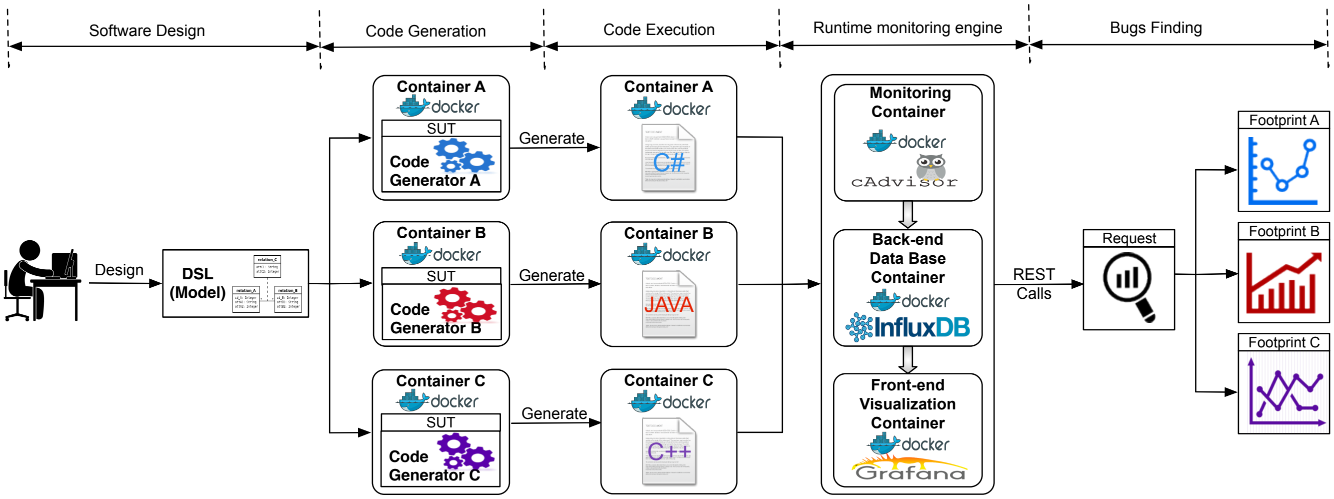

Figure 2. An overall overview of the different processes involved to ensure the code generation and non-functional testing of produced code from design time to runtime.

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 Generator Under Test: HAXE compilers

In order to test the applicability of our approach for non-functional testing of generated code based on system containers, 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++ and provides experimental support for C# and Java. It can also generate bytecode that can be executed on Haxe VM called NekoVM.

Haxe involves many features: the Haxe language, multi-platform compilers, and different 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 source code 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 parsed into a typed structure and type-checked, resulting 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 which compiles Haxe code into source code or bytecode of a target language or platform. 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 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.

-*CPU Usage (CU)*: It corresponds to the CPU usage. The CPU consumption is measured in seconds, as the CPU time.

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

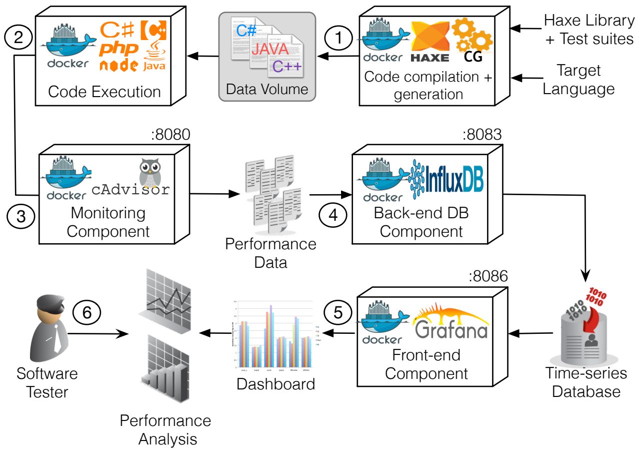

Figure 3. Comparison of average memory consumption and execution time of FFmpeg containers compiled with standard GCC optimization options

4.1.3 Setting up Infrastructure

4.2 Experimental Results

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

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