TESTed: one judge to rule them all

Niko Strijbol
Supervisors: Peter Dawyndt, Bart Mesuere

Abstract

Writing a programming exercise is a time-consuming activity. When the goal is using the same exercise in multiple programming languages, the required time grows fast. One needs to manually translate the exercise to each language one wishes to support, although most exercises do not make use of language specific constructs. The translation is not intellectually stimulating work: it is an almost mechanical job. This article introduces TESTed, a prototype of an exercise framework in which a test plan is written in a language-independent format. This allows evaluating submissions in multiple programming languages for the same exercise. TESTed also integrates with Dodona, an online exercise platform. First the inner workings of the framework are described, followed by a discussion of the current limitations and future work.

1. Introduction

Technology is becoming increasingly important in our society: digital solutions are used to solve tasks and problems in more and more domains. As a result of this evolution, students have to become familiar with computational thinking: translating problems from the real world into problems understood by a computer [1]. Computational thinking is broader than just programming, yet programming is a very good way to train students in computational thinking. However, learning to program is often perceived as difficult by students [2]. As solving exercises is beneficial to that learning process, it is important to provide students with exercises that are high in both quality and quantity. This requirement imposes two challenges on educators:

- Educators have to write suitable exercises: they need to take into account what concepts the students are familiar with, how long it takes to solve an exercise, etc.
- The students' submissions must be provided with high quality feedback. This feedback allows students to learn from their mistakes and improve their programming skills (and by proxy their computational thinking skills).

A significant part of programming exercises are for novices: they are intended to teach how to program, while the language details are not very important. Another type of exercise are algorithmic exercises, in which the algorithm is the most important aspect. Both of these types of exercises are prime candidates for being available in multiple programming languages. However, making an exercise available in multiple programming languages adds a non-negligible time cost to the already time-consuming and labour-intensive process of writing exercises. The exercise must be translated manually to each language one wishes to support. This translation work is not intellectually stimulating or interesting: in most cases, it is a fairly mechanical translation. The TESTedframework, introduced in Section 3, is an answer to the question "is it feasible to create a framework, in which an exercise is written once, but is solvable in multiple programming languages?".

2. Dodona

To cope with the second challenge, providing good feedback, educators often use a platform in which submissions for exercises are evaluated automatically. One such platform is Dodona[1], an online platform for programming exercises. It supports multiple programming languages and provides real-time automatic feedback for submissions. Dodona is freely available for schools and is used by multiple courses at Ghent University.

Dodona is a modular system: the programming language specific code for evaluating a submission (the *judge*) is separated from the platform. This judge is run in a Docker container and communicates with Dodona via a JSON interface, using `stdin` for the input and `stdout` for the output. The input are mainly configuration parameters, such as memory and time limits, in addition to the submission and the exercise code. The output of the Docker container is the result of the evaluation. Naturally, Dodona supports determining the correctness of a submission. However, the judge system is a flexible one: other types of feedback are possible. For example, some existing judges run a linter on the submissions, which can contain useful hints for students to improve the quality of their code.

TESTed, introduced in the next section, is implemented as a judge and thus fully integrates with Dodona. As the Dodona JSON interfaces are sufficiently flexible and generic, we believe it is independently usable as well.

[1] `https://dodona.ugent.be/`

3. TESTed

TESTed is a prototype of a framework for writing programming exercises and evaluating submissions for those exercises in multiple programming languages. The framework can be split into three distinct parts:

- The *test plan* and serialization format. The test plan is a specification on how to evaluate a submission for a given exercise. Combined with the serialization format, it makes writing language-independent exercises possible.
- The *core*, which takes care of handling input and output (in the Dodona JSON format), generating test code from the test plan (using the language configurations), executing said code, and evaluating the results of said execution.
- The *language configurations*, which are the subsystem responsible for translating the test plan into actual code. By separating the language configurations from the main logic, it is easy to add support for new programming languages in TESTed.

3.1. *The test plan*

The test plan is a programming language independent format that specifies how a submission for an exercise must be evaluated. It contains elements such as the different tests, the inputs, the expected outputs, etc. A concrete example of a simple test plan is included in Listing 3.1. The structure of the test plan is heavily inspired by the Dodona JSON format and consists of a hierarchy of the following elements:

Tabs A tab is the top-level grouping mechanism for the tests. One test plan can consist of multiple tabs, which are shown as distinct tabs in the Dodona user interface.

Contexts Each tab consists of one or more contexts. A context is an independent execution and evaluation of a submission. Each context is run in a new subprocess.

Testcase A context consists of one or more testcases. A testcase is the evaluation of one input and the resulting outputs. We distinguish two types of testcases:

Context testcase The testcase containing the call to the main function (or script execution). A context can have at most one context testcase.

Normal testcase Testcases with other inputs, such as function calls or assignments.

Test Finally, a testcase contains multiple tests, each for a different output channel. Currently, there are tests for `stdout`, `stderr`, return values, exceptions and exit codes. The testcase contains the input, while the tests contain the different expected outputs.

```json
{
  "tabs": [
    {
      "name": "Tab",
      "contexts": [
        {
          "context_testcase": {
            "input": {
              "main_call": true,
              "stdin": {
                "type": "text",
                "data": "input-1"
              }
            },
            "output": {
              "stdout": {
                "type": "text",
                "data": "input-1"
              }
            }
          },
          "testcases": [
            {
              "input": {
                "type": "function",
                "name": "echo",
                "arguments": [
                  {
                    "type": "integer",
                    "data": 50
                  }
                ]
              },
              "output": {
                "result": {
                  "value": {
                    "type": "integer",
                    "data": 50
                  }
                }
              }
            }
          ]
        }
      ]
    }
  ]
}
```

Listing 1: An example of a test plan for a fictional exercise. A solution for this exercise should print `stdin` to `stdout` when executed, but also contain a `echo` function, which returns its argument. Both scenario's are tested in the same context in this test plan. In real usage, these would probably be separate contexts.

An important part of the test plan that deserves further attention is the input. As we've mentioned, each testcase has a single input. There are basically two types of input: `stdin` and a statement. Since our goal is not to create a universal programming language, statements have intentionally been kept simple. A statement is either an expression or an assignment. An assignment must be understood as giving a name to the result of an expression. An expression can be a function call, an identifier (referring to a variable previously created using an assignment) or a literal value. The arguments of a function call are also expressions.

Additionally, TESTed defines a serialization format for values. This format consists of two pieces: the data type of a value and the encoding of the value. Since the serialization format is defined in JSON, as is the test plan, the encoding is simply a JSON type. For example, numerical values are encoded as a JSON number. The data type of the value is complexer: since we support multiple programming languages, we must support generic data types. To this end, TESTed defines two kinds of data types:

- Basic types, which include integral numbers, rational numbers, booleans, strings, sequences, sets and maps. These are abstract types, and we don't concern ourselves with implementation details for these types. For example, all integer types in C (`int`, `long`, etc.) map to the same integral number type in the serialization format.
- Advanced types, which are more detailed (`int64`) or programming language specific. Here we do concern ourselves with implementation details.

The advanced types are associated with a basic type, which acts as a fallback. If a programming language does not support a certain advanced type, the basic fallback type will be used. For example, a `tuple` in Python will be considered equal to an `array` in Java, since they both have the basic type `sequence`. Programming languages can also indicate that there is no support for certain types (e.g. no support for sets in C), in which case the exercise will not be solvable in that programming language if the exercise uses that data type.

The serialization format is used for both encoding return values and as the format for literal types in the test plan. However, it is important to note that this is not a formal type system. While a formal type system's main goal is describing which categories of data a construct may be or receive, the type system in the serialization format is used to describe concrete values. For example, the test plan does not specify "a function with return type `int64`". Instead, it specifies "a function whose expected return value is 16, which is of type `int64`". As such, TESTed does not use nor need techniques normally associated with a type system, such as a formal type checker.

3.2. *Running an evaluation*

A first part of the core of TESTed is responsible for running the evaluation of a submission for an exercise. Each evaluation runs trough a list of steps, as illustrated by Figure 1 and described below:

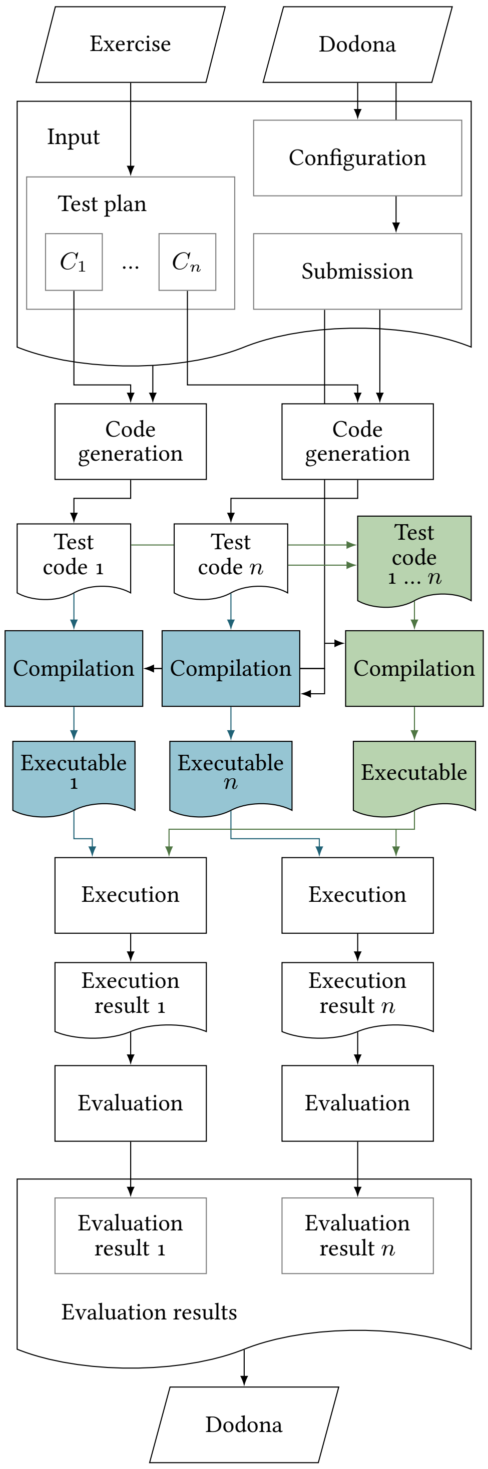

Figure 1: A flow chart depicting the various steps of an evaluation in TESTed. C_1 and C_n stand for context 1 and context n of the test plan. The two colours indicate the different possible paths for the compilation step: blue for context compilation and green for batch compilation.

1. The Docker container for TESTed is started by Dodona, and the submission and configuration are made available to TESTed.
2. The test plan is checked to verify that the exercise is solvable in the programming language of the submission.
3. For each context in the test plan, the test code is generated.
4. The test code is compiled in one of two ways:

   Batch compilation The test code of every context is bundled and compiled together in one compilation. This mode results in one executable.

   Context compilation The test code for each context is compiled separately. For n contexts, there will be n compilations, resulting in n executables.

   This compilation step is, of course, optional. Note that we use the term executable to denote the output of the compilation step, although the output is not always an executable, as is the case in Java.
5. The result of the previous step (the compilation results or the test code if there is no compilation) is executed. Each context is executed in a new subprocess, to combat sharing information between contexts.
6. The collected results of the execution are then evaluated, which is described in Section 3.3.
7. As a final step, TESTed sends the evaluation results to Dodona.

One might ask why we need two compilation modes. The answer is performance. Since the evaluation of a submission happens in real-time, it is desirable to keep the evaluation short. One bottleneck is the compilation step, which can be quite slow in languages such as Java or Haskell. It is common to have test plans of 50 contexts or more, which does mean there would be 50 or more compilations. As a solution, we don't compile each context independently: instead we compile all contexts in one go. Note that the contexts are still executed independently.

3.3. Evaluating the results

A second part of the core is evaluating the results of the previous execution step. As we've mentioned, each output channel is denoted by a different test in the test plan. Currently, TESTed has support for following output channels: `stdout`, `stderr`, exceptions, return values, created files and the exit code. In most cases, the test plan would specify how a channel should be evaluated for each relevant output channel. If an output channel is not relevant, TESTed provides sane defaults (e.g. not specifying `stderr` means there should be no output on `stderr`; if there is output, the submission is considered wrong).

Generally, there are three ways in which an output channel can be evaluated:

Language-specific evaluation The evaluation code is included in the test code generated for the context and is executed directly in the same subprocess as the test code. As a result, the evaluation code must be written in the same programming language as the submission. This mode is intended to evaluate programming language specific aspects.

Programmed evaluation The evaluation code is executed separately from the test code, in a different process. The results of the execution pass through TESTed: they are first serialized in the test process and deserialized in the evaluation process. This means the programming language of the submission and the evaluation code can differ. For example, the evaluation code can be written in Python, and used to evaluate the results of submissions in Java, JavaScript, Haskell, etc.

Generic evaluation TESTed has built-in support for simple evaluations, like comparing a produced value against an expected value contained in the test plan. For example, if a function call with argument a should result in value b, there is no need to write evaluation code. There is support for evaluating textual results (`stdout`, `stderr`), return values, exceptions and the exit code. The evaluator for values is intelligent and takes the data types into account. If the test plan specifies that the return value should be a tuple, TESTed will apply strict comparisons in language supporting tuples (Python and Haskell), but loose comparisons in other languages. This means that for Python and Haskell solutions, only tuples will be accepted. In other languages, all data types with the corresponding basic type will be accepted (such as arrays and lists).

Not all modes are available for all output channels. For example, the language-specific evaluation mode is only available for return values and exceptions. It is assumed that textual output (e.g. `stout`) is not programming language specific.

4. Configuring programming languages

Support for a programming language in TESTed consists of three parts:

1. A configuration file
2. A configuration class
3. Templates

The configuration file is used to record properties of the programming language, such as the file extension or which data structures are supported.

The configuration class handles the language-specific aspects of the compilation and execution step. TESTed expects a command to perform during those steps (e.g. for C, the command would along the lines of `gcc -std=c11 file1.c file2.c`).

The third component are the templates. TESTed uses the Mako templating system [3] to generate the test code and translate language independent concepts from the test plan into actual code (such as literal values). The templating system works similar to the one used by webapps (such as ERB in Rails, Blade in Laravel or EEX in Phoenix), but generates code instead of HTML/JSON.

Since the test plan is language independent, existing exercise require no change to work with a newly added programming language.

5. Future work

In this section, we briefly discuss the current limitations of the prototype and identify area's where future improvements are possible.

5.1. *Test plan*

A first area in which future improvements are possible is the test plan. The test plan currently does not support dynamic scheduling of contexts: the contexts to execute are decided before the execution happens. The test plan can be extended to allow some form of decision, for example deciding if a context should run depending on the result of the previous context. Related are repeated executions of the same context. This is useful for exercises with randomness, where the output is not deterministic. For example, generating a list of random numbers with a minimum and maximum constraint would need multiple runs to validate that random numbers are used.

Another aspect of the test plan are the statements and expressions. While these are currently intentionally simple, this does impose some limitations. For example, an expression with mathematical operators (`5 + 5`) is not possible. The expressions and statements could be extended to provide more capabilities. It is also worth investigating if, when more capabilities are added, it might be worth switching to an existing language and transpile this language to the various programming languages TESTed supports.

The test plan is also fairly verbose. We believe this could be solved by introducing a preprocessing step, that translates a more compact format to the test plan. A promising upcoming format is [4]. We also envision different formats for different exercise types. For example, an IO exercise and exercise with function calls have different requirements.

5.2. *Programming paradigms*

Secondly, TESTed does not translate programming paradigms. For example, a given exercise might be solved using two top-level functions in Python, but would often be solved with an object in Java. Functional languages also have different paradigms compared to object-oriented languages. It might be worth researching common patterns and their equivalent in other programming languages and providing translation for those common patterns.

5.3. *Performance*

Lastly, while performance is not bad, there is room for improvement. One area in particular are the programmed evaluations. Each evaluation is currently compiled and executed, even though it is common for multiple contexts to use the same evaluation code.

6. Conclusion

We have presented TESTed, a judge for the Dodona platform, capable of evaluating submissions in multiple programming languages for the same exercise. At the moment, TESTed supports Python, Java, Haskell, C and JavaScript. While there still some limitations on the type of exercises TESTed can evaluate, it can already be used for a variety of exercises.

References

[1] B. Bastiaensen, J. De Craemer, Zo denkt een computer, Vlaamse Overheid, 2017.

[2] A. Luxton-Reilly, Simon, I. Albluwi, B. A. Becker, M. Giannakos, A. N. Kumar, L. Ott, J. Paterson, M. J. Scott, J. Sheard, C. Szabo, Introductory programming: A systematic literature review, in: Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2018 Companion, Association for Computing Machinery, New York, NY, USA, 2018, p. 55–106. `doi:10.1145/3293881.3295779`.

[3] M. Bayer, G. T. Dairiki, P. Jenvey, D. Peckam, A. Ronacher, B. Bangert, B. Trofatter, Mako templates for python.
URL `https://www.makotemplates.org/`

[4] P. Conrad, C. Bart, S. Edwards, Programming exercise markup language.
URL `https://cssplice.github.io/peml/`