Large Scale Unit Testing Algorithm v2

Chew, Kean Ho[1]

[1]ZORALab Enterprise
kean.ho.chew@zoralab.com

October, 2022, 1st Issue

1 Abstract

Working on unit testing software product in modern programming languages is getting more cumbersome as the software product is getting incrementally complex in a very rapid and demanding pace. Since year 2019, research efforts had been done to effectively deploy large scale testing specifically for Go Programming Language.

While the unit-testing algorithm is available in the past, it had quickly became outdated as new specialized techniques are developed to further enhance overall testing capabilities. This impedes one from building a more confident and battle-tested software product. Therefore, said algorithm has to be enhanced in order to cope with the latest update and shall be deployable across other programming languages.

This paper first revisits the past Large Scale Unit Testing for Go Programming Language Packages research paper for algorithm extractions. Then, the paper presents the algorithm enhancements, caveats, crucial lessons, and simultaneously deploying it to the Rust and TinyGo programming language as a 2nd and 3rd languages support.

Lastly, the paper concludes the enhanced large scale testing algorithm capable of future incremental improvement use not just for programming environment but a way of life.

2 Introduction

Working on unit testing software product in modern programming languages is getting more cumbersome as the software product is getting incrementally complex in a very rapid and demanding pace. Since year 2019, research efforts had been done to effectively deploy large scale testing specifically for Go Programming Language[1].

While the unit-testing algorithm is available in the past[1][2], it had quickly became outdated as new specialized techniques are developed[3] to further enhance overall testing capabilities. This impedes one from building a more confident and battle-tested software product. Therefore, said algorithm has to be enhanced in order to cope with the latest update and shall be deployable across other programming languages.

This paper first revisits the past Large Scale Unit Testing for Go Programming Language Packages research paper for algorithm extractions. Then, the paper presents the algorithm enhancements, caveats, crucial lessons, and simultaneously deploying it to the Rust and TinyGo programming language as a 2nd and 3rd languages support.

Lastly, the paper concludes the enhanced large scale testing algorithm capable of future incremental improvement use not just for programming environment but a way of life.

3 Background

This section covers the existing large scale unit testing algorithm based on the past researches[1][2]. It introduces the algorithm itself, how it approaches qualitative testing, test scopes, and its test developer experience to date. This shall provide a good and precise context and current state of development without requiring readers to spend large amount of resources to read through past research papers.

3.1 The Problems

The large scale unit testing algorithm was first developed for Go Programming Language in year 2019[1] after a background research in year 2018[2]. It was initially designed to solve the uncontrollable large and rapid growth of test codes where a simple but heavily tested software feature can easily scale to >1000 test cases in 1 development iteration, yielding at least 48229 lines of codes[1]. Without the algorithm, the test developer often confronts with unpredictable architectural code changes; extremely long, unmodifiable, and unmaintainable test codes; limited logging functionalities; frequent naming collisions; and coping with infrastructure differences[1].

3.2 The Algorithm

The algorithm is a simple simulation generator approach using factory design pattern[1] where in a Go package requires a minimum of:

1. A **testlibs_test.go** file that is responsible for generating the simulation parameters, values, and managing external libraries across all test suites or test cases including assertion[1]; and
2. A **Function_test.go** file that is responsible for test suite and test case of a public accessible Function (notice the title-case) [1]; and
3. A **scenarios_test.go** for generating each test cases' triggers for *testlibs_test.go* to generate a simulation environment using a mapped list of boolean string alongside test cases' report parameters like name for the entire package[1].

Each test file is properly isolating its roles and responsibilities accordingly in order to properly scale with maintainable sanity[1]. Moreover, all test files should and always comply to Go Programming Language's Effective Go standards without any special customizations or dependency[1][4].

3.3 Test Scope and Approaches

The algorithm is capable of facilitating a wide range of test approaches ranging from:

1. Standard Node CFG[1];
2. Edge CFG[1];
3. Condition CFG[1]; and
4. Boundary Value Analysis[1]

The algorithm recommends the use of table-driven test approach to systematically test across all function's boundaries limits[1]. This allows a developer to effectively test and guarantees a function behavior is working within a given scope when the development resources (e.g. time, knowledge, experience, software, and hardware) are severely limited[1].

If permitted by the development resources, developer can proceed to perform higher level testing with the same algorithm like integration testing and etc[1].

4 New Challenges

This section covers the use of the algorithm since its birth with new encountering and problems. It discusses each insights in details and why they matters in the algorithm enhancements.

4.1 New Test Facility

As time proceeds since year 2019, in year 2022, Go core developers releases a new test facility called "Go Fuzzing" that provides Open-Source Software Fuzz (OSS-Fuzz) fuzzy testing capabilities[3][5]. Figure 4.1.1 shows the example of implementing a fuzzing test approach[3].

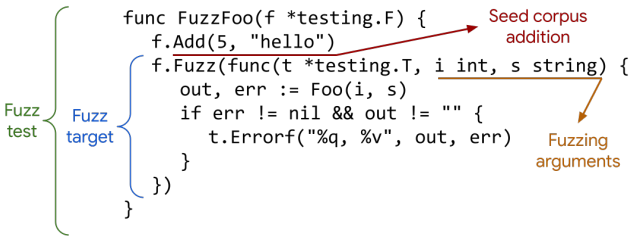


Figure 4.1.1 - new Fuzzing test approach in Go Programming Language[3]

The existing algorithm involving a *prepareTestHelper(t *testing.T)* register method prohibits its application to any new techniques[1]. Hence, the algorithm is not flexible enough where it contradicts its own advertised main advantage of being agile and nimble in testing businesses[1].

4.2 Resources Demanding and Unexportable Reports

Due to the nature of consolidating all test report data in a single file[1], rendering the report can sometimes crash a viewing operating system browser due to high memory and rendering computation demands. This is highly unfeasible and will acts as the algorithm application's upper-limiting factor[1]. Moreover, the overwhelming presentation of data at a time can make reader confused and difficult to digest.

4.3 Daunting Scrolling and Searches

Due to the consolidated test scenarios and test reports nature, it can very daunting to perform scrolling and searches although both are easily available to use[1]. It causes dependencies on external search tool in order to operate an upgrade to the existing test suites or test scenarios. This is not feasible for long run.

4.4 Assertion Nightmare

The existing algorithm offers a list of data verification and assertions that is capable of concluding a test case[1]. As such, the development of a new assertion function causes the entire simple test helper package transformed into a bloated data verification package. These data verification functions also have useful applications outside of testing environment like data sanitation. Therefore, it's better to provide just the missing test feature and functions while letting the test developer performs his/her own assertions outside of the test helper library.

4.5 Not Portable to Other Programming Languages

The current test helper library implementing the existing algorithm is very restricted to Go programming language due to over-reliance on Go's reflection package. Simple functions like *fmt.Printf* uses reflect for rendering the string output of unknown parameters in runtime[1]. Such reliance must be removed so that the algorithm can be applied to other programming languages without runtime features or other business paradigms. Moreover, opinionated tools like GolangCI-Lint and "standard" directory structure[6] complicates the algorithm portability.

5 Enhancements

This section covers the list of enhancements done to the algorithm presented in Section 3 with resolutions applied for solving all new challenges listed in Section 4. It explains its reasoning on why such features must be implemented and how they are being implemented.

5.1 Compartmentalized Test Suite

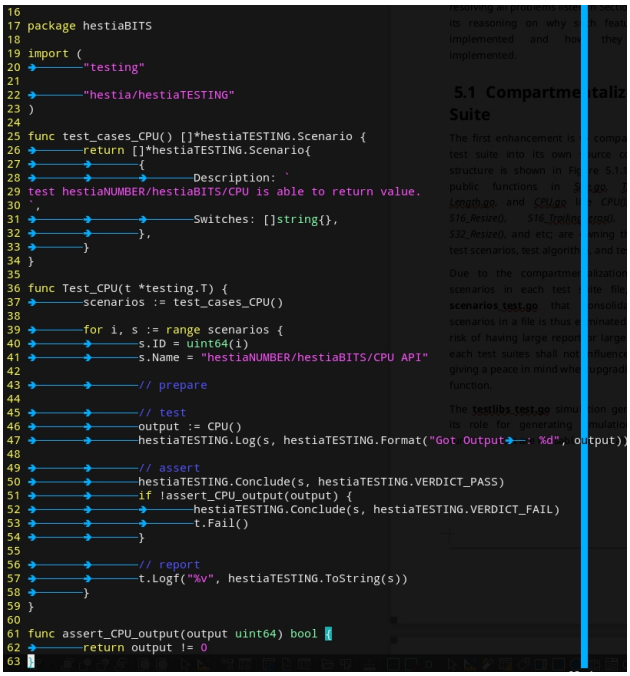
The first enhancement is to compartmentalize all test suite into its own source codes. The file structure is shown in Figure 5.1.1 where every public functions in *Size.go*, *TrailingZeros.go*, *Length.go*, and *CPU.go* like *CPU()*, *S16_Length()*, *S16_Resize()*, *S16_TrailingZeros()*, *S32_Length()*, *S32_Resize()*, and etc; are owning their respective test scenarios table list, test algorithm function(s), and test assertion functions. Figure 5.1.2 shows the test suite file content of such compartmentalization in **CPU_testing.go** for **CPU()** public function[7].

Due to the compartmentalization of the test scenarios in each test suite file, the original **scenarios_test.go** that consolidates all test scenarios in a single file is thus eliminated. This removed the risk of having massive-sized report or large-sized test file. Also, as each test suite is independent of another, it provides the test developer a peace in mind when upgrading a particular feature or function.

The **testlibs_test.go** simulation environment generator retains its role and existences for generating simulation values and functions that are reusable across all test suites[8].

```
u0:hestiaNUMBER$ tree hestiaBITS/
hestiaBITS/
├── Constants.go
├── CPU.go
├── CPU_test.go
├── Length.go
├── S16_Length_test.go
├── S16_Resize_test.go
├── S16_TrailingZeros_test.go
├── S32_Length_test.go
├── S32_Resize_test.go
├── S32_TrailingZeros_test.go
├── S64_Length_test.go
├── S64_Resize_test.go
├── S64_TrailingZeros_test.go
├── S8_Length_test.go
├── S8_Resize_test.go
├── S8_TrailingZeros_test.go
├── Size.go
├── testlibs_test.go
└── TrailingZeros.go
```

Figure 5.1.1 - Compartmentalized test suites



```
16
17 package hestiaBITS
18
19 import (
20 	"testing"
21
22 	"hestia/hestiaTESTING"
23 )
24
25 func test_cases_CPU() []*hestiaTESTING.Scenario {
26 	return []*hestiaTESTING.Scenario{
27 		{
28 			Description: `
29 test hestiaNUMBER/hestiaBITS/CPU is able to return value.
30 `,
31 			Switches: []string{},
32 		},
33 	}
34 }
35
36 func Test_CPU(t *testing.T) {
37 	scenarios := test_cases_CPU()
38
39 	for i, s := range scenarios {
40 		s.ID = uint64(i)
41 		s.Name = "hestiaNUMBER/hestiaBITS/CPU API"
42
43 		// prepare
44
45 		// test
46 		output := CPU()
47 		hestiaTESTING.Log(s, hestiaTESTING.Format("Got Output : %d", output))
48
49 		// assert
50 		hestiaTESTING.Conclude(s, hestiaTESTING.VERDICT_PASS)
51 		if !assert_CPU_output(output) {
52 			hestiaTESTING.Conclude(s, hestiaTESTING.VERDICT_FAIL)
53 			t.Fail()
54 		}
55
56 		// report
57 		t.Logf("%v", hestiaTESTING.ToString(s))
58 	}
59 }
60
61 func assert_CPU_output(output uint64) bool {
62 	return output != 0
63 }
```

Figure 5.1.2 - Compartmentalized test suite file containing its own test scenarios on the top, test algorithm in the middle, and test assertion at the bottom[7].

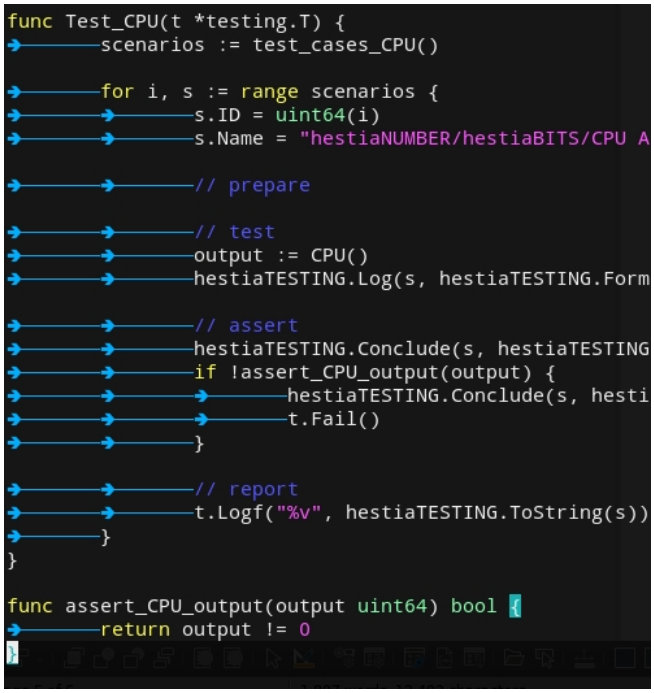
5.2 Data Type Assertion and Registration Function Removal

To remove unnecessary growth of the data validation assertion functions, the role of the algorithm is carefully re-examined and all unnecessary functions are rescinded.

It is vital to recognize that the algorithm's ultimate role is to only carefully process the state of the test case, organize the data, and present it into a necessary, very consistent, on-point, and inter-translatable report. Anything else shall be provided and operated by the programming language test infrastructure alone ranging from setting a conclusive verdict of a test case to data type assertions. In short, the enhanced algorithm must and shall not interfere and confuse developer.

Therefore, the algorithm is trimmed to only perform logging; providing simulation switches and test report parameters; and rendering the intended test reports. Figure 5.2.1 shows the new approach without assertions where the new algorithm shall not interfere with existing test infrastructure (e.g. *t.Fail()* is clearly stated in the assertion decision while *t.Log()* records the output of the test report rendered by the algorithm formatting function *hestiaTESTING.ToString(...)*[7].

Also, in order to future-proof any new test technique developed in the future like the 2022 Go Fuzzy test tool, the registration-like function is thus rescinded from the enhanced algorithm since assertion is no longer required. As such, with the enhanced algorithm capable of working independently from the test infrastructure, implementing Go Fuzzy test is no longer a blocking factor.



```
func Test_CPU(t *testing.T) {
	scenarios := test_cases_CPU()

	for i, s := range scenarios {
		s.ID = uint64(i)
		s.Name = "hestiaNUMBER/hestiaBITS/CPU A

		// prepare

		// test
		output := CPU()
		hestiaTESTING.Log(s, hestiaTESTING.Form

		// assert
		hestiaTESTING.Conclude(s, hestiaTESTING
		if !assert_CPU_output(output) {
			hestiaTESTING.Conclude(s, hesti
			t.Fail()
		}

		// report
		t.Logf("%v", hestiaTESTING.ToString(s))
	}
}

func assert_CPU_output(output uint64) bool {
	return output != 0
}
```

Figure 5.2.1 - Newer approach of using the test algorithm without interfering with test infrastructure conclusive function[7]

5.3 Direct Scenario Use

To eliminate a bunch of large code duplications due to the looping execution in a test suite such as test case's UID assertion and generations, test suite naming, and etc; with the new compartmentalized algorithm enhancement in Section 5.1 alongside the array nature of the test scenarios generator; both UID and test suite name can be safely and directly set in the test algorithm itself as shown in Figure 5.2.1 (*s.ID* and *s.Name*). The Scenario data structure can be directly used for facilitating the simulation parameters generator instead of having an intermediate translations. Figure 5.3.1 shows the direct use of Scenario data structure that only requires developer to fills in the test case description and its switches[7].

```
17 package hestiaBITS
18
19 import (
20 	"testing"
21
22 	"hestia/hestiaTESTING"
23 )
24
25 func test_cases_CPU() []*hestiaTESTING.Scenario {
26 	return []*hestiaTESTING.Scenario{
27 		{
28 			Description: `
29 test hestiaNUMBER/hestiaBITS/CPU is able to return
30 `,
31 			Switches: []string{},
32 		},
33 	}
34 }
```

Figure 5.3.1 - Using the Scenario data structure directly to generate the list of test cases[7]

5.4 Use Array Type for Switches

While attempting to port the enhanced algorithm to TinyGo Programming Language, dating to this paper, it appears that TinyGo had yet to implement some basic yet critical features such as but not limited to map abstract data type list range looping mechanism[9]. Apparently, manually implement such feature can be a daunting task so the next step is to replace it with a primitive string array data type for test Scenario's switches. Figure 5.4.1 shows an example of using string array data type for all the Switches in *S8_Length()* function test scenarios[10].

```
func test_cases_S8_Length() []*hestiaTESTING.S
	return []*hestiaTESTING.Scenario{
		{
			Description: `
test hestiaNUMBER/hestiaBITS/S8_Length is able
`,
			Switches: []string{
				cond_BITS_0,
			},
		}, {
			Description: `
test hestiaNUMBER/hestiaBITS/S8_Length is able
`,
			Switches: []string{
				cond_BITS_8,
			},
		},
	}
}
```

Figure 5.4.1 - Example of using string array for Scenario. Switches[10]

The advantage is that the switches are:

1. consistent and orderly rendered;
2. less complicated (due to the removal of additional boolean switch); and
3. straight to the point.

The disadvantage however, is that in order to scan for a particular condition, the array have to be looped from top to bottom for every queries[11]. This slows down the test executions but it shall not affect the actual software merchandise. However, it also means that the enhanced algorithm can have poor performance on interpretive programming languages such as but not limited to Ruby and Python.

To make querying a condition easier in this new string array list, a helper function like *HasCondition(...) bool* function can simplify the development experience as shown in Figure 5.4.2.

```
func assert_S8_Length_output(s *hestiaTESTING.Scenario,
	if hestiaTESTING.HasCondition(s, cond_BITS_8) {
		return output == uint8(value_BITS_8_COUN
	}

	if hestiaTESTING.HasCondition(s, cond_BITS_0) {
		return output == 0
	}

	return false
}
```

Figure 5.4.2 - Using HasCondition function to query specific string condition from the Scenario's Switches[10]

With the new data type for Switches, TinyGo is now capable of reusing the the enhanced algorithm test library vis-a-vis with the original Go.

5.5 Independent of Programming Language

In order to ensure the enhanced algorithm is portable to other programming languages or outside of the software industry, we have to make sure the algorithm itself does not rely on any programming languages' unique capabilities like Go's reflection and runtime features[11][12], TinyGo's LLVM optimizations capability[13], or Rust's macros[14]. This is the most difficult enhancement ever done since the test helper library ideally has to be completely independent from any dependency including the standard packages. Useful sensory or rendering functions are notoriously complicated to implement from scratch. For TinyGo and Go, it is very easy to invoke any reflection or runtime related functions when using any functions from Go's standard libraries. Hence, long term development efforts are required to ensure said goal is achieved.

As dated to this paper, the algorithm was successfully ported and implemented in TinyGo, Go, and Rust programming languages.

5.6 Export Capable Report Data

The last enhancement is enabling the capability of exporting the report data that are parse-able by common formats such as JSON, TOML, or YAML. Due to the strict enhancement in Section 5.5, the paper only implements TOML data format rendering function[7]. TOML was selected among others mainly because[20]:

1. It's very simple to implement without requiring a re-implementation of encoder and decoder just to validate the output[20]; and
2. Its string building algorithm is very simple to develop compared to the alternatives[20]; and
3. Its quotation escaping capability is simple enough for various possible string quoted values without requiring additional linters like its competitors[20].

Figure 5.6.1 shows the TOML rendering output that is parse-able by other software like documentation content management system[7].

```
u0:hestiaBITS$ go test -v .
=== RUN   Test_CPU
    CPU_test.go:57: [Result]
        ID = 0
        Verdict = 'PASSED'
        Name = '''
        hestiaNUMBER/hestiaBITS/CPU API
        '''
        Description = '''
        test hestiaNUMBER/hestiaBITS/CPU is able to return value.
        '''
        Switches = []

        [[Result.Log]]
        Value = '''
        Got Output      : 64
        '''
--- PASS: Test_CPU (0.00s)
=== RUN   Test_S16_Length
```

Figure 5.6.1 - test output rendered in TOML format[7]

6 Results

This section covers the enhanced algorithm deployment results across multiple programming languages in accordance to their specific language specialities. It demonstrates how to deploy the enhanced algorithms step-by-steps in an iterative manner.

6.1 Deployment in Go Programming Language

Just like its predecessor, the enhanced algorithm can be developed in the following sequences:

1 **Develop the test suite file first** – this identifies what you want to do and needed to be done. Among its components, in sequence:

1.1 **The test algorithm and assertion** – The main content of the test suite codes as shown in Figure 5.2.1.

1.2 **The test scenarios** – Then by observing the test subject and the algorithm, proceed to build the test scenarios list as shown in Figure 5.3.1.

2 **Develop the common testlibs_test.go test libraries** – this unifies common generator functions to reduce code duplications. Among the sub-components are usually:

2.1 **switch conditions** – Switches in its constant nature are kept here. These statements shall be human-readable, independent on its own context, self-explanatory, and should not rely on the scenario description for elaboration[8]. Figure 6.1.1 shows an example for listing out the string conditions.

```
16
17 package hestiaBITS
18
19 import (
20 ·····"hestia/hestiaTESTING"
21 )
22
23 // test conditions
24 const (
25 ·····cond_BITS_64 = "provide 64-bits value"
26 ·····cond_BITS_32 = "provide 32-bits value"
27 ·····cond_BITS_16 = "provide 16-bits value"
28 ·····cond_BITS_8  = "provide 8-bits value"
29 ·····cond_BITS_0  = "provide 0-bit value"
30
31 ·····cond_TO_BITS_1000 = "convert to 1000-bits value"
32 ·····cond_TO_BITS_64   = "convert to 64-bits value"
33 ·····cond_TO_BITS_35   = "convert to 35-bits value"
34 ·····cond_TO_BITS_32   = "convert to 32-bits value"
35 ·····cond_TO_BITS_22   = "convert to 22-bits value"
36 ·····cond_TO_BITS_16   = "convert to 16-bits value"
37 ·····cond_TO_BITS_12   = "convert to 12-bits value"
38 ·····cond_TO_BITS_8    = "convert to 8-bits value"
39 ·····cond_TO_BITS_5    = "convert to 5-bits value"
40 ·····cond_TO_BITS_0    = "convert to 0-bit value"
41
42 ·····cond_TO_UNSIGNED = "convert to unsigned value"
43 ·····cond_TO_SIGNED   = "convert to signed value"
44
45 ·····cond_NIL_INPUT = "provide nil input"
46 )
47
48 // test values
```

Figure 6.1.1 - commonly used test conditions[8]

2.2 **verifiable values used in assertion and generator functions** – Depending on the test nature, values that are used in generator and assertion functions can be kept here. There values shall be used at least twice across one or more test suites. Figure 6.1.2 shows an example for listing out common test values used in said manners.

```
48 // test values
49 const (
50 ·····value_MASKED_BITS_35 = 0x7FFFFFFFF
51 ·····value_MASKED_BITS_22 = 0x3FFFF
52 ·····value_MASKED_BITS_12 = 0xFFF
53 ·····value_MASKED_BITS_5  = 0x1F
54
55 ·····value_BITS_64_COUNT = 64
56 ·····value_BITS_32_COUNT = 32
57 ·····value_BITS_16_COUNT = 16
58 ·····value_BITS_8_COUNT  = 8
59 )
```

Figure 6.1.2 - commonly used test values[8]

2.3 **common generator functions** – common value generating functions used across multiple test suites shall be kept here. These functions shall be used by more than 2 test suites. Figure 6.1.3 shows an example listed out common generator functions used in said manners.

```
60
61 func create_sign(s *hestiaTESTING.Scenario) bool {
62 →	if hestiaTESTING.HasCondition(s, cond_TO_SIGNED) {
63 →	→	return true
64 →	}
65
66 →	if hestiaTESTING.HasCondition(s, cond_TO_UNSIGNED) {
67 →	→	return false
68 →	}
69
70 →	return false
71 }
72
73 func create_size(s *hestiaTESTING.Scenario) uint16 {
74 →	if hestiaTESTING.HasCondition(s, cond_TO_BITS_1000)
75 →	→	return 1000
76 →	}
77
78 →	if hestiaTESTING.HasCondition(s, cond_TO_BITS_64) {
79 →	→	return 64
80 →	}
81
82 →	if hestiaTESTING.HasCondition(s, cond_TO_BITS_35) {
83 →	→	return 35
84 →	}
85
86 →	if hestiaTESTING.HasCondition(s, cond_TO_BITS_32) {
87 →	→	return 32
88 →	}
89
90 →	if hestiaTESTING.HasCondition(s, cond_TO_BITS_22) {
91 →	→	return 22
92 →	}
```

Figure 6.1.3 - commonly used test functions[8]

3 **Observe the test report and improve iteratively** – Lastly, repeat step 1 to step 2 iteratively by observing the test reports and/or generate the report data file accordingly. Once satisfied, the developer can move on to the new test suite development. Figure 5.6.1 shows an example of the reporting in string format for a test report.

4 **Compile heatmap code coverage for effective and insightful testing** – the code coverage heatmap allows the developer to perform effective testing by insightful learning. This saves resources and perform pinpoint accuracy. Figure 6.1.4 shows an example of the heatmap code coverage that is testing against the test codes shown in Figure 5.2.1.

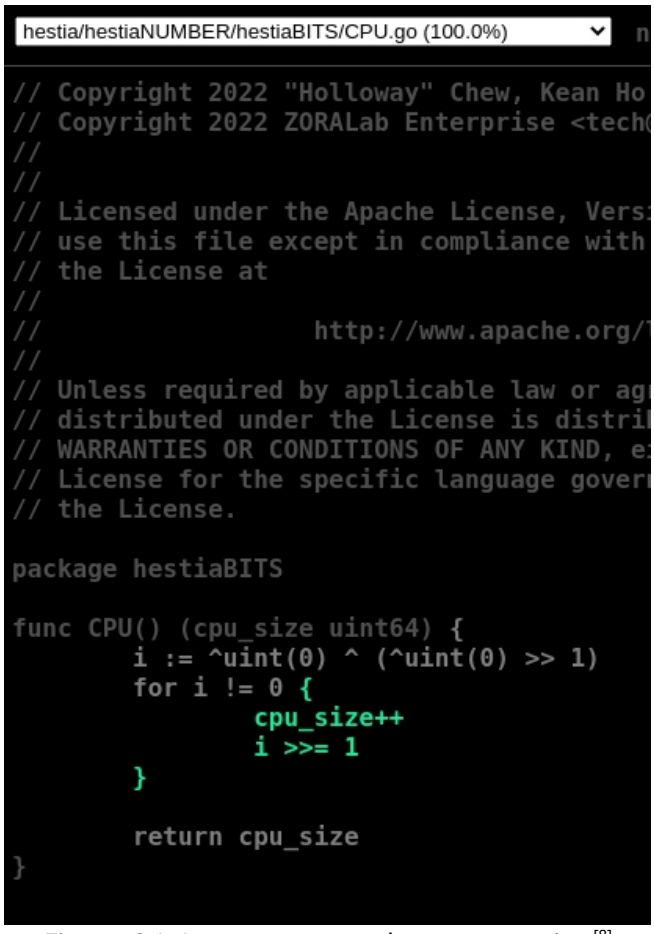


Figure 6.1.4 - test coverage heatmap testing[8]

6.2 TinyGo Deployment

Deployment for TinyGo is similar to Go as they share the same language. However, there are a few strict precautions due to the incomplete development nature of TinyGo compiler:

1 **Be careful with using functions and codes involving reflection** – Not all features in Go reflection package are readily available[9][21].

2 **No code coverage heat-map is made available** – Unlike Go compiler, TinyGo does not generate heat-map code coverage test report.

3 **Memory allocation warning** – Unlike Go, TinyGo can report out all its memory allocations which is something special to TinyGo alone[22]. This can compliment Go compiler to create a much simple functions with its algorithm much more portable to other languages.

6.3 Rust Programming Language Deployment

Deploying the enhanced algorithm into Rust Programming Language is different from deploying into Go and TinyGo mainly because Rust's test infrastructure is entirely different in nature. However, the deployment is still doable and is, in fact, a lot easier to deploy compared to Go and TinyGo.

6.3.1 Rust's Format! Macro

Unlike any other known programming languages, Rust uses *Format!* per-processing macro to perform string formatting instead of the conventional *Printf* formatting function as shown in Figure 6.3.1.1[15]. While the goal is for performance gain in the actual binary product via great use of macro, it does introduce a novel way of doing string formatting.

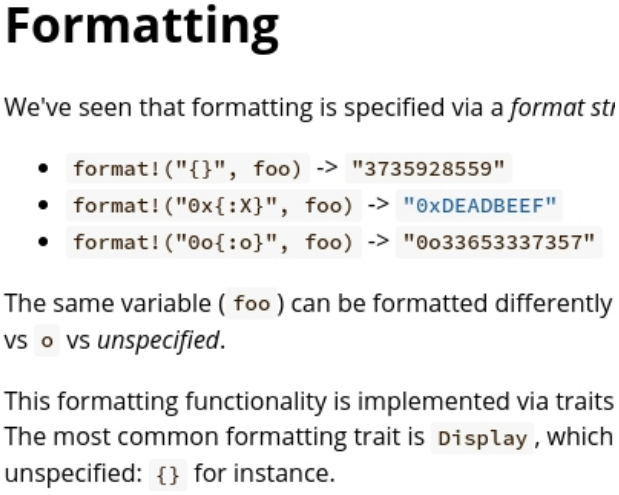


Figure 6.3.1.1 - Rust using *Format!* Macro to perform string formatting[15]

6.3.2 Rust's Test Functions

Unlike Go and TinyGo, Rust does not have a runtime feature to operate their test infrastructure or catching panics as shown in Figure 6.3.2.1[16]. Instead, Rust relies heavily on per-processor macro to indicate a test function can expect a panic via execution failure[16]. In short, the entire infrastructure relies solely on macro implementations[16][17]. Moreover, unlike Go or TinyGo, Rust's unit test function is ONLY meant for ONE (1) test case and not for a test suite with multiple test cases[16]. Therefore, the table-driven test methodology implementation can be quite awkward.

Fortunately, this problem can be solved by developing a macro function capable of per-process all unit-test functions of a given test suite on top of existing test infrastructure[17]. Unfortunately, due to Rust procedural macro not able to handle arithmetic counting at per-processing level (since it is only responsible for writing Rust codes), the UID data field in the Scenario has to be manually listed as shown in Figure 6.3.4.2.

```rust
pub fn divide_non_zero_result(a: u32, b: u32) -> u32 {
    if b == 0 {
        panic!("Divide-by-zero error");
    } else if a < b {
        panic!("Divide result is zero");
    }
    a / b
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_divide() {
        assert_eq!(divide_non_zero_result(10, 2), 5);
    }

    #[test]
    #[should_panic]
    fn test_any_panic() {
        divide_non_zero_result(1, 0);
    }

    #[test]
    #[should_panic(expected = "Divide result is zero")]
    fn test_specific_panic() {
        divide_non_zero_result(1, 10);
    }
}
```

Figure 6.3.2.1 - Rust using macros for unit testing and panic catching

6.3.3 Code Coverage Heatmap

Rust code coverage heatmap is awkwardly implemented as development work are still in progress. Currently, Rust depends on Mozilla's GRCOV cargo module to perform the necessary code coverage heatmap output[18]. Unlike Go, the setup for Rust' code coverage infrastructure is not as intuitive as Go since it is a 3rd-class citizen among its dependencies chart (gcov → cargo → rustc) versus 1st class citizen in Go (go).

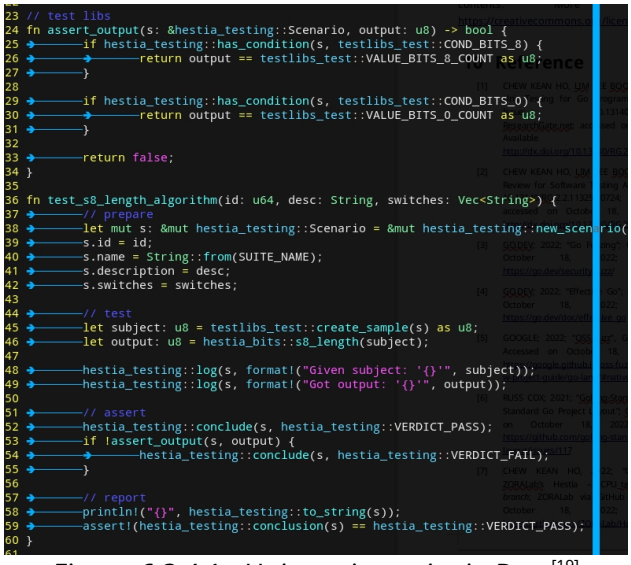
As dated to this paper, the authors failed to implement such feature numerously so it shall be left out from this paper. However, in the authors' opinions, it is believed that it is achievable and the authors have faith in Rust core team for making Rust's test infrastructure competitive to Go's test infrastructure.

6.3.4 Approaches

Implementing the enhanced algorithm in Rust is similar to Go as shown in Section 6.1 with slight differences against the test scenarios table list. The sequences are:

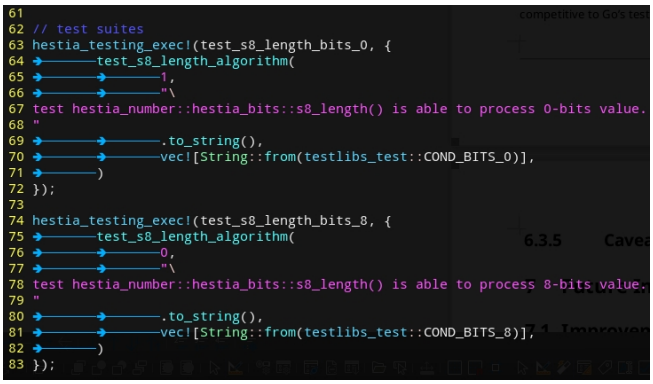
1 **Develop the test codes file first** – for identifying what is needed to be done and how the testing is done as shown in Figure 6.3.4.1[19]. In Rust, however, each test scenario has to be defined individually using the test function generator macro[17] directly into the test suite source code file as shown in Figure 6.3.4.2[19] instead of using a array listing as shown in Figure 5.4.1.

2



```rust
// test libs
fn assert_output(s: &hestia_testing::Scenario, output: u8) -> bool {
    if hestia_testing::has_condition(s, testlibs_test::COND_BITS_8) {
        return output == testlibs_test::VALUE_BITS_8_COUNT as u8;
    }

    if hestia_testing::has_condition(s, testlibs_test::COND_BITS_0) {
        return output == testlibs_test::VALUE_BITS_0_COUNT as u8;
    }

    return false;
}

fn test_s8_length_algorithm(id: u64, desc: String, switches: Vec<String>) {
    // prepare
    let mut s: &mut hestia_testing::Scenario = &mut hestia_testing::new_scenario(
    s.id = id;
    s.name = String::from(SUITE_NAME);
    s.description = desc;
    s.switches = switches;

    // test
    let subject: u8 = testlibs_test::create_sample(s) as u8;
    let output: u8 = hestia_bits::s8_length(subject);

    hestia_testing::log(s, format!("Given subject: '{}'", subject));
    hestia_testing::log(s, format!("Got output: '{}'", output));

    // assert
    hestia_testing::conclude(s, hestia_testing::VERDICT_PASS);
    if !assert_output(s, output) {
        hestia_testing::conclude(s, hestia_testing::VERDICT_FAIL);
    }

    // report
    println!("{}", hestia_testing::to_string(s));
    assert!(hestia_testing::conclusion(s) == hestia_testing::VERDICT_PASS);
}
```

Figure 6.3.4.1 - Unit testing suite in Rust[19]



```
61
62 // test suites
63 hestia_testing_exec!(test_s8_length_bits_0, {
64 →       test_s8_length_algorithm(
65 →       →       1,
66 →       →       "\
67 test hestia_number::hestia_bits::s8_length() is able to process 0-bits value.
68 "
69 →       →       .to_string(),
70 →       →       vec![String::from(testlibs_test::COND_BITS_0)],
71 →       )
72 });
73
74 hestia_testing_exec!(test_s8_length_bits_8, {
75 →       test_s8_length_algorithm(
76 →       →       0,
77 →       →       "\
78 test hestia_number::hestia_bits::s8_length() is able to process 8-bits value.
79 "
80 →       →       .to_string(),
81 →       →       vec![String::from(testlibs_test::COND_BITS_8)],
82 →       )
83 });
```

Figure 6.3.4.2 - Declaring each test scenario directly in Rust using the test function generator macro[19]

3 **Develop the common testlibs_test.rs test libraries** – for unifying common values and libraries. Similar to Go, all switches' conditions and commonly used test helper functions are defined here since they're used across many test suites as shown in Figure 6.3.4.3[19].

```
18
19 // test conditions
20 pub const COND_BITS_128: &str = "provide 128-bits value";
21 pub const COND_BITS_64: &str = "provide 64-bits value";
22 pub const COND_BITS_32: &str = "provide 32-bits value";
23 pub const COND_BITS_16: &str = "provide 16-bits value";
24 pub const COND_BITS_8: &str = "provide 8-bits value";
25 pub const COND_BITS_0: &str = "provide 0-bit value";
26
27 pub const COND_TO_BITS_1000: &str = "convert to 1000-bits va
28 pub const COND_TO_BITS_127: &str = "convert to 127-bits valu
29 pub const COND_TO_BITS_72: &str = "convert to 72-bits value
30 pub const COND_TO_BITS_64: &str = "convert to 64-bits value
31 pub const COND_TO_BITS_35: &str = "convert to 35-bits value
32 pub const COND_TO_BITS_32: &str = "convert to 32-bits value
33 pub const COND_TO_BITS_22: &str = "convert to 22-bits value
34 pub const COND_TO_BITS_16: &str = "convert to 16-bits value
35 pub const COND_TO_BITS_12: &str = "convert to 12-bits value
36 pub const COND_TO_BITS_8: &str = "convert to 8-bits value";
37 pub const COND_TO_BITS_5: &str = "convert to 5-bits value";
38 pub const COND_TO_BITS_0: &str = "convert to 0-bits value";
39
40 pub const COND_TO_UNSIGNED: &str = "convert to unsigned valu
41 pub const COND_TO_SIGNED: &str = "convert to signed value";
42
43 // test values
44 pub const VALUE_MASKED_BITS_72: u128 = 0xFFFFFFFFFFFFFFFFFF
45 pub const VALUE_MASKED_BITS_35: u128 = 0x7FFFFFFF;
46 pub const VALUE_MASKED_BITS_22: u128 = 0x3FFFFF;
47 pub const VALUE_MASKED_BITS_12: u128 = 0xFFF;
48 pub const VALUE_MASKED_BITS_5: u128 = 0x1F;
49
50 pub const VALUE_BITS_128_COUNT: u128 = 128;
51 pub const VALUE_BITS_64_COUNT: u128 = 64;
52 pub const VALUE_BITS_32_COUNT: u128 = 32;
53 pub const VALUE_BITS_16_COUNT: u128 = 16;
54 pub const VALUE_BITS_8_COUNT: u128 = 8;
55 pub const VALUE_BITS_0_COUNT: u128 = 0;
56
57 pub fn create_sign(s: &hestia_testing::Scenario) -> bool {
58 →       if hestia_testing::has_condition(s, COND_TO_SIGNED)
59 →       →       return true;
60 →       }
```

Figure 6.3.4.3 - Consolidating all commonly used values and helper functions in Rust' testlibs[19]

4 **Observe the test report and improve iteratively** – for improving the software product quality overtime as shown in Figure 6.3.4.4[19]. However, by default, Rust does not print out any reporting output onto the console. Therefore, the tester must issue the **-- --show-output** argument for the **cargo test** command (**$ cargo test -- --show-output**) in order to force it to render the output. The TOML format rendering of test report TOML format is unaffected.

```
---- hestia_number::hestia_bits::s8_length_test::tes

TEST REPORT

ID              : 0
Verdict         : PASSED
Name            : hestia_number::hestia_bits::s8_len
Description     :
test hestia_number::hestia_bits::s8_length() is able


Switches                :
[0]     provide 8-bits value



Log             :
[0]
Given subject: '128'
[1]
Got output: '8'

===[ END ]===
```

Figure 6.3.4.4 - Reading the test results from Rust Unit testing output[19]

5 **Compile heatmap code coverage for effective testing** – for pinpoint accuracy testing with minimal use of resources. The test efforts and development should be same as Go.

7 Future Improvements

This section covers all identified gaps for future improvements that can be done beyond this paper. It allows the algorithm to be further enhanced for effective and efficient adoption without much complexity.

7.1 Improvement for Rust Implementations

The implementation of the enhanced algorithm in Rust is a new venture compared to its predecessor where the translation is impossible. However, due to the limited experience with Rust programming language by this paper's authors, the authors believe that the algorithm can be further implemented effectively in the Rust programming language, allowing the interoperability between Rust, Go, and TinyGo.

7.2 Into Artificial Intelligence

The authors of this paper strongly believe that once the enhanced algorithm is implemented across many programming languages, the next step is to further enhance the algorithm for being useful in artificial intelligence developments and applications. Artificial intelligence is powerful and sophisticated enough to handle complex business problems in a very effective and efficient manner, strongly complimenting and potentially replacing some or most of the programming implementations in the future.

8 Conclusion

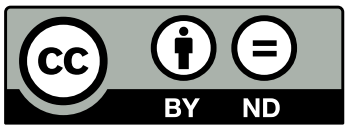
Working on unit testing software product in modern programming languages is getting more cumbersome as the software product is getting incrementally complex in a very rapid and demanding pace. While the unit-testing algorithm is made available, it had quickly became outdated as new specialized techniques were developed.

Among the enhancement did to the algorithms were the ability to compartmentalize and isolate all test suites from one another; the removal of assertion affecting the provided test infrastructure; the ability to use the test Scenario data structure directly for table-driven scenario definition list; the deployment of string array for switches; the capability of exporting test case's report data; and the portability across other programming languages.

The enhanced algorithm is also tested in other programming languages like TinyGo and Rust for assuring its advertised product advantage of being portable and flexible is confidently tested. Its implementations for Rust can be further improved and the authors are also looking forward to deploy the enhanced algorithm in the artificial intelligence sector. As of this paper, the authors concluded that the enhanced algorithm is successfully enhanced and is now named as "Large Scale Unit Testing Algorithm v2".

9 License

The paper is licensed under:



CC-BY-ND

This license lets you distribute; and build your work commercially and non-commercially upon the original contents as long as you credit the authors; and no remix, tweak, and edit upon the original contents. More info at: https://creativecommons.org/licenses/by-nd/4.0/

10 Acknowledgment

We would like to thank LIM LEE BOOI for her continuous constructive criticism of the manuscript despite all the hardships and contributing the development of this algorithm in the past.

ありがとうございました | Dankeschön | 谢谢 | Thank You

11 Reference

[1] CHEW KEAN HO, LIM LEE BOOI; 2019; “Large Scale Unit Testing for Go Programming Packages”; 1st Issue; 10.13140/RG.2.2.36308.76166; ResearchGate.net; accessed on October 18, 2022; Available at: http://dx.doi.org/10.13140/RG.2.2.36308.76166

[2] CHEW KEAN HO, LIM LEE BOOI; 2018; “Descriptive Review for Software Testing Algorithms”; 1st Issue; 10.13140/RG.2.2.11325.10724; Researchgate.net; accessed on October 18, 2022; Available at: http://dx.doi.org/10.13140/RG.2.2.11325.10724

[3] GO.DEV; 2022; “Go Fuzzing”; Google; accessed on October 18, 2022; Available at: https://go.dev/security/fuzz/

[4] GO.DEV; 2022; “Effective Go”; Google; accessed on October 18, 2022; Available at: https://go.dev/doc/effective_go

[5] GOOGLE; 2022; “OSS-Fuzz”, Google via GitHub.io; Accessed on October 18, 2022; Available at: https://google.github.io/oss-fuzz/getting-started/new-project-guide/go-lang/#native-go-fuzzing-support

[6] RUSS COX; 2021; “Golang-Standards: This Is Not A Standard Go Project Layout”; Github Inc.; Accessed on October 18, 2022; Available at: https://github.com/golang-standards/project-layout/issues/117

[7] CHEW KEAN HO, 2022; “GitHub Code Blob: ZORALab’s Hestia – CPU_test.go”; *Experimental branch*; ZORALab via GitHub Inc.; Accessed on October 18, 2022; Available at: https://github.com/ZORALab/Hestia/blob/experimental/hestiaGO/hestiaNUMBER/hestiaBITS/CPU_test.go

[8] CHEW KEAN HO, 2022; “GitHub Code Blob: ZORALab’s Hestia – testlibs_test.go”; *Experimental branch*; ZORALab via GitHub Inc.; Accessed on October 18, 2022; Available at: https://github.com/ZORALab/Hestia/blob/experimental/hestiaGO/hestiaNUMBER/hestiaBITS/testlibs_test.go

[9] KISHORE KONJETI, AYKE; 2022; “GitHub Issue: TinyGo.Org - panic: unimplemented: (reflect.Value).MapRange()”; TinyGo.org via GitHub Inc.; Accessed on October 18, 2022; Available at: https://github.com/tinygo-org/tinygo/issues/3104

[10] CHEW KEAN HO, 2022; “GitHub Code Blob: ZORALab’s Hestia – S8_Length_test.go”; *Experimental branch*; ZORALab via GitHub Inc.; Accessed on October 18, 2022; Available at: https://github.com/ZORALab/Hestia/blob/experimental/hestiaGO/hestiaNUMBER/hestiaBITS/S8_Length_test.go

[11] MICHAEL KNYSZEK; 2022; “Go Runtime: 4 Years Later”; *The Go Blog*; Google via Go.Dev; Accessed on October 18, 2022; Available at: https://go.dev/blog/go119runtime

[12] ROB PIKE; 2011; “The Laws of Reflection”; *The Go Blog*; Google via Go.Dev; Accessed on October 18, 2022; Available at: https://go.dev/blog/laws-of-reflection

[13] TINYGO.ORG; 2022; “Important Build Options”; *Documentations* > *References* > *Using TinyGo*; TinyGo.ORG; Accessed on October 18, 2022; Available at: https://tinygo.org/docs/reference/usage/important-options/

[14] STEVE KLABNIK, CAROL NICHOLS; 2022; “Macros”; *The Rust Programming Language Documentations*; Rust Team via rust-lang.org; Accessed on October 18, 2022; Available at: https://doc.rust-lang.org/book/ch19-06-macros.html

[15] RUST.ORG; 2022; “Rust by Example - Formatting”; The Rust Programming Language Documentations; Rust Team via rust-lang.org; Accessed on October 19, 2022; Available at: https://doc.rust-lang.org/rust-by-example/hello/print/fmt.html

[16] RUST.ORG; 2022; “Rust by Example – Unit Testing”; The Rust Programming Language Documentations; Rust Team via rust-lang.org; Accessed on October 19, 2022; Available at: https://doc.rust-lang.org/rust-by-example/testing/unit_testing.html

[17] CHEW KEAN HO, 2022; “GitHub Code Blob: ZORALab’s Hestia – execs.rs”; *Experimental branch*; ZORALab via GitHub Inc.; Accessed on October 19, 2022; Available at: https://github.com/ZORALab/Hestia/blob/experimental/hestiaRUST/hestia_testing/execs.rs

[18] MOZILLA, 2022; “GitHub: Mozilla’s GRCOV”; *master branch*; Mozilla via GitHub Inc.; Accessed on October 19, 2022; Available at: https://github.com/mozilla/grcov

[19] CHEW KEAN HO, 2022; “GitHub Code Blob: ZORALab’s Hestia – s8_length_test.rs”; *Experimental branch*; ZORALab via GitHub Inc.; Accessed on October 19, 2022; Available at: https://github.com/ZORALab/Hestia/blob/experimental/hestiaRUST/hestia_number/hestia_bits/s8_length_test.rs

[20] CHEW KEAN HO; 2022; “GitHub Commit: f0c5602c3d3479373f96b368a786d3af1341a791 : hestiaGO – purged hestiaTESTING toJSON and toYAML rendering functions”; *ZORALab’s Hestia Software*; ZORALab via GitHub Inc.; Accessed on October 19, 2022; Available at: https://github.com/ZORALab/Hestia/commit/f0c5602c3d3479373f96b368a786d3af1341a791

[21] TINYGO.ORG; 2022; “Packages Supported by Go”; *Documentations* > *References* > *Go Language Features*; TinyGo.ORG; Accessed on October 19, 2022; Available at: https://tinygo.org/docs/reference/usage/important-options/

[22] TINYGO.ORG; 2022; “MISC Build Options”; *Documentations* > *References* > *Using TinyGo*; TinyGo.ORG; Accessed on October 19, 2022; Available at: https://tinygo.org/docs/reference/usage/misc-options/