Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

# Extending a property-based testing tool with parallel and distributed execution

**Estudante:** Pablo Costas Sánchez
**Dirección:** Laura Milagros Castro Souto
Konstantinos Sagonas

A Coruña, setembro de 2020.

*To Sam and to Diego*

## Acknowledgements

First of all I would like to express my most sincere gratitude to Laura and Kostis, my supervisors of this project, as without them it would not have occurred. I would like to thank Laura for her kindness, support and for being ever so patient with me, always helping me out when in need and motivating me to push forward, whatever the circumstances. I would like to thank Kostis not only as his is the tool this project is about, but also for his good feedback and interest in our work, besides the help he has given during this year.

I would like to also thank my family and friends, those of old and those that I met during this degree, for putting up with me everyday and throughout these years.

Finally, I would like to thank Diego for always making me smile and for all the happiness that he has brought to my life.

## Abstract

Software testing plays an important role in software development, as it not only helps find bugs in the code, but also boosts the confidence of the developers that the program behaves correctly, besides reducing the cost of fixing such errors or flaws if done in early stages.

One of the most common methods of software testing is unit testing, which tests individual components of the software by asserting whether for cherry-picked test cases (i.e., for a given input), the component or unit produces the expected output. This approach to testing has however its downsides, as it is a tedious time-consuming activity, prone to errors of the developer, such as not covering every possible case.

Property-Based Testing is a method of testing that fixes the problems found in unit testing, for it uses properties, which are simply logical statements that capture partial correctness of the program, to generate random input to test whether the program satisfies those properties or not. However, while automation allows for the execution of many more tests, increasing their number also means longer test running times.

The main goal of this project is to extend PropEr, the most powerful property-based testing tool written in Erlang, with parallel and distributed execution and measure the obtained speedup from doing so.

## Resumo

Probar o noso código é unha das cousas máis importantes na disciplina do desenvolvemento do software, xa que non só nos axuda a encontrar erros no noso código, se non que tamén aumenta a confianza das desenvoldedoras e desenvoldedores de que o seu programa se comporta correctamente, ademais de reducir o custo de arranxar devanditos erros ou fallos se se fai dende o inicio.

Un dos métodos máis comúns para facer probas ao software son as denominadas probas de unidade, nas que para probar compoñentes individuáis do software, mírase se para casos específicos (ou sexa, para entradas concretas) o compoñente ou unidade produce a saída esperada. Esta forma de probar o código ten, porén, as súas desvantaxes, xa que é unha tarefa tediosa e pesada de facer que consume moito tempo, e a maiores é propensa e susceptíbel a erros das desenvoldedoras e desenvolvedores, coma non cubrir tódolos casos posibles.

As probas baseadas en propiedades son un método de probar software que soluciona estes problemas das probas de unidade, xa que no seu lugar empregan o concepto de propiedade, que é un predicado lóxico que captura a corrección parcial do programa. Estas propiedades son usadas para xerar entradas aleatorias para comprobar se o programa satisface as súas expectativas ou non. Porén aínda que a automatización permite realizar moitas máis probas, os tempos de execución tenden a incrementarse correlativamente.

O obxectivo principal deste proxecto é estender PropEr, a ferramenta de probas baseadas en propiedades máis potente escrita en Erlang, para permitir a súa execución paralela ou distribuída.

**Keywords:**

- Functional programming
- Property-Based Testing
- Erlang
- Concurrency
- Distributed execution
- Parallelization speedup

**Palabras chave:**

- Programación funcional
- Probas baseadas en propiedades
- Erlang
- Concurrencia
- Execución distribuída
- Aceleración por paralelización

# Contents

1 Introduction 1

1.1 Motivation . . . 1

1.2 Objectives . . . 2

1.3 Methodology . . . 3

1.4 Work plan . . . 4

1.4.1 Project cost . . . 5

1.5 Report layout . . . 6

2 Background 7

2.1 Erlang/OTP . . . 7

2.2 Property-Based Testing . . . 9

2.2.1 Unit Testing . . . 9

2.2.2 Property-Based Testing . . . 11

2.3 PropEr . . . 12

2.3.1 Stateless vs. stateful properties . . . 15

3 Development 17

3.1 Design . . . 17

3.1.1 Proposed design . . . 18

3.1.2 Test distribution among workers . . . 19

3.2 A PropEr study . . . 21

3.3 Implementation . . . 28

3.3.1 Preamble . . . 28

3.3.2 Project structure . . . 28

3.3.3 A PropEr implementation . . . 29

3.3.4 Problems faced during the Implementation stage . . . 36

3.3.5 Implementing the different strategies of test distribution . . . 37

**4 Testing and benchmarking** **39**
4.1 Testing the implementation . . . . . 39
4.2 Benchmarking . . . . . 40
4.2.1 Picking a strategy to split the tests . . . . . 41
4.2.2 Parallel execution benchmarking . . . . . 47
4.2.3 Distributed execution benchmarking . . . . . 51
4.3 Conclusions . . . . . 54

**5 Conclusions** **63**
5.1 Follow-up . . . . . 63
5.1.1 Impact on the code . . . . . 63
5.1.2 Lessons learned . . . . . 64
5.2 Future work . . . . . 65

**List of Acronyms** **69**

**Glossary** **71**

**Bibliography** **73**

# List of Figures

1.1 Iterations of the project . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

3.1 Even distribution of tests among the workers . . . . . . . . . . . . . . . . . . . 19
3.2 Uneven distribution of tests among the workers . . . . . . . . . . . . . . . . . 20
3.3 Batches distribution of tests among the workers . . . . . . . . . . . . . . . . . 20
3.4 *N*th sequence distribution of tests among the workers . . . . . . . . . . . . . 21
3.5 Tracing visualized using the *Event Tracer* . . . . . . . . . . . . . . . . . . . . 22
3.6 Exported definitions (i.e., functions and types) of the tool's main module . . . 23
3.7 Control flow of the tool originally . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.8 Control flow of the tool after implementing the design . . . . . . . . . . . . . 26

4.1 Reports of *Travis CI* in the project . . . . . . . . . . . . . . . . . . . . . . . . 40
4.4 Results of `compile_prop:compile()` . . . . . . . . . . . . . . . . . . . 56
4.7 Results of `beam_types_prop:associativity()` . . . . . . . . . . . 57
4.10 Results of `crypto_ng_api:prop__crypto_one_time()` . . . . . 58
4.13 Results of `crypto_ng_api:prop__crypto_init_update_final` 59
4.16 Results of `ssh_eqc_encode_decode:prop_ssh_decode_encode()` 60
4.19 Results of `shell_docs_prop:prop_render()` . . . . . . . . . . . . 61

# List of Tables

1.1 Breakdown by iteration of the effort of the project . . . . . 5
1.2 Total cost of the project . . . . . 5

4.1 Cowlib: `prop_str_huffman()` parallel benchmarks . . . . . 43
4.2 Kazoo: `prop_normalize()` parallel benchmarks . . . . . 44
4.3 Zotonic: `prop_s_utf8a()` parallel benchmarks . . . . . 45
4.4 Diffy: `prop_inner_diff()` parallel benchmarks . . . . . 46
4.5 Erlang/OTP: `compiler`, `compile()` parallel benchmarks . . . . . 48
4.6 Erlang/OTP: `compiler`, `associativity()` parallel benchmarks . . . . 48
4.7 Erlang/OTP: `crypto`, `prop__crypto_one_time()` parallel benchmarks . . . . . 49
4.8 Erlang/OTP: `crypto`, `prop__crypto_init_update_final()` parallel benchmarks . . . . . 49
4.9 Erlang/OTP: `ssh`, `prop_ssh_decode_encode()` parallel benchmarks 50
4.10 Erlang/OTP: `stdlib`, `prop_render()` parallel benchmarks . . . . . 51
4.11 Erlang/OTP: `compiler`, `compile()` distributed benchmarks . . . . . 52
4.12 Erlang/OTP: `compiler`, `associativity()` distributed benchmarks . . 52
4.13 Erlang/OTP: `crypto`, `prop__crypto_one_time()` distributed benchmarks . . . . . 52
4.14 Erlang/OTP: `crypto`, `prop__crypto_init_update_final()` distributed benchmarks . . . . . 53
4.15 Erlang/OTP: `ssh`, `prop_ssh_decode_encode()` distributed benchmarks . . . . . 54
4.16 Erlang/OTP: `stdlib`, `prop_render()` distributed benchmarks . . . . . 54

# Listings

2.1 Example of a *Hello World* program in Erlang . . . . . 8
2.2 Example of a concurrent program in Erlang . . . . . 8
2.3 Program that reverses the order of a list . . . . . 9
2.4 Example of unit testing in Erlang . . . . . 10
2.5 Example of a successful EUnit invocation . . . . . 10
2.6 Example of property-based testing in Erlang . . . . . 12
2.7 Example of a successful PropEr invocation . . . . . 13
2.8 Example of a failed PropEr invocation . . . . . 13
2.9 Example of property-based testing in Erlang . . . . . 14
2.10 Example of calling `proper:module/1` . . . . . 14
3.1 Exported definitions (i.e., functions and types) of the tool's main module . . . 23
3.2 Code of original `proper:inner_test/2` . . . . . 24
3.3 Simplified code of original `proper:perform/3` . . . . . 27
3.4 Directory structure of the tool . . . . . 28
3.5 Definition of the FORALL/3 macro . . . . . 30
3.6 Code of `proper:quickcheck/1,2` . . . . . 30
3.7 Code of `proper:test/2` . . . . . 31
3.8 Type definition of `proper:opts()` record . . . . . 32
3.9 Code of `proper:inner_test/2` with the modifications applied . . . . . 33
3.10 Code of `proper:perform_with_nodes/2` . . . . . 33
3.11 Code of `proper:start_node/1` . . . . . 35
3.12 Code of `proper:ensure_code_loaded/1` . . . . . 35
3.13 Code of `proper:maybe_load_binary/2` . . . . . 37
5.1 Comparison of LOC (Lines of Code) PropEr had before and after our work . . 64
5.2 Total of additions and deletions done throughout our work . . . . . 64

# Chapter 1
# Introduction

THROUGHOUT this first chapter of the report we will cover the motivations that pushed us to do this project, a short summary of the current situation and the objectives the final version of the application resulting from this project should satisfy.

## 1.1 Motivation

One of the most important parts of software development is testing. It has been stated multiple times that not only the cost of not testing your software increases over time, but it is also less expensive to find and fix problems in earlier stages of development than while in production or, in other words, that the cost of finding those defects exceeds that of developing the software originally. On the one hand, it boosts the developer's confidence in their code behaving the way it is supposed to; on the other hand, it can become a boring and tiresome task to cherry-pick the cases to test and, as Dijkstra said, "program testing can be used very effectively to show the presence of bugs but never to show their absence" [1]. Nonetheless it is a necessity that should not be avoided: tests are critical to program quality, and even more in long-lived projects.

All in all, even though software testing plays an important role in software development, in practice it is almost always lacking in some aspect: be it because it is non existent, either because when developing or maintaining the software it was not thought that necessary, or because although present, it is not able to properly accomplish its function as an assessment of the software's functionality.

Software testing is also a quite large area of software development, having numerous approaches (e.g., static testing, dynamic testing, passive testing, white-box testing, black-box testing) and levels of testing (i.e., unit testing, integration testing, system testing and acceptance testing), with unit testing being one of the most common methods used to perform the job.

This is where Property-Based Testing (PBT) [2] comes into play, as it is a type of testing which uses generated data as input for the tests, enabling the developer to instead of manually writing all the possible ways the code can be used, specify properties they want their program to satisfy. Then, through the use of a generative-testing tool (i.e., a property-based testing tool) of their liking (e.g., *QuickCheck* [2, 3], *PropEr* [4, 5], *test.check* [6], etc), randomized input data is generated and used to test whether the properties are satisfied by the code. Furthermore, the concept of a property is not a complex one: it is a logical statement that captures *partial* correctness of the code. In other words, a rule that dictates that as long as the input the code is given belongs in a certain set of values it should behave as expected (i.e., produce outputs that fulfill certain condition(s), present certain traits, or belong, in turn, to a certain range).

However, this type of testing has its downsides: scaling up the number of tests to a large number or checking if complex properties hold can take a quite significant amount of time.

There are other approaches to testing worth mentioning, such as formal verification and theorem proving, since both techniques are quite helpful in verifying the correctness of software. However, not only they suffer from having an excessively steep learning curve, but also from being hard to scale up to current software programs because of issues as the state explosion problem and the like.

Because of this and that with the ever-improving computer specifications over the years, not only concurrent software programs have become a trend but also distributed programs too (e.g., parallel computing, any kind of telecommunication software, network file systems, etc) because of their advantages of being more performant than their sequential counterparts, although at the cost of being harder to test; and the fact that most property-based testing tools run sequentially (PropEr can do parallel testing but it only works in certain types of tests and QuickCheck has an old driver that tries to expand it with parallel execution; however, no Property-Based Testing tool has an officially included concurrent execution for all of their runs), we thought that the execution times and performance had room for improvement by expanding one to allow parallel an distributed execution.

## 1.2 Objectives

The main goal of this bachelor thesis is to design how to extend a property-based testing tool to run tests in a parallel and distributed fashion, and to implement those changes in the tool with as little as possible impact in the existing codebase.

The property-based testing tool we will expand and patch is PropEr, which not only was inspired by QuickCheck, the original Property-Based Testing tool, but is also along with it the most powerful PBT tool to date. In addition, it is written in Erlang/OTP and is the most used within the community of the language to run property-based tests. We chose this tool as the popularity of it can help get feedback easily on the changes that will be done and because, as previously stated, it is built upon Erlang, an open-source battle-tested functional programming language (with a permissive license that allows us to work with it) for fault-tolerant and distributed systems, so we intend to bring these characteristics to this powerful testing tool to increase its power even more.

Hence, the objective of this project is to bring the aforementioned new features (parallel and distributed execution) to the tool using one of the language's strongest points: its actor model, which eases the job of writing concurrent software. Thanks to it and because of how time-consuming running many property-based tests can be, the extension of the tool will speedup the execution times of testing those properties by running them in parallel or distributed.

Once this primary objective is achieved we will run in a cluster this patched version of PropEr to validate and improve it, in addition to measure the obtainable speedup when running property-based tests of relevant projects (i.e., those used by the community), as some can take too long (e.g., Scalable Process Groups (spg) [7]).

Finally, once this project's work has been tested and proved working properly, the commits with the changes will be *Pull Requested* from the fork [8] where they have been published to the original repository [9], to allow the modifications to go *upstream*.

## 1.3 Methodology

This project's work and development has been approached following a modified version of the well-known Scrum methodology, adapted to a single person environment as it is originally team-oriented. It is an agile methodology built upon the concept of *sprints*, which are usually short iterations to be done within a fixed time period, allowing to develop a product in a series of incremental, more manageable, iterations that focus on certain aspects of the product and that always deliver a working product at their end. This approach to managing software development is also well-suited for situations of great uncertainty and risk, as it was this project.

As previously mentioned, although we used Scrum, some modifications were done to take into account that instead of a team developing a solution, the agile methodology was being used by an student and his two supervisors. Rather than having daily meetings we had weekly meetings were the objectives of each sprint and their progress was discussed. However, as one of the supervisors was remote and he could not attend those meetings, those were supplemented with being in constant communication with him, first by email and later on through Skype calls and chats.

Some of the methods we used from Scrum to approach the work were setting up a to-do list and keeping track of it as the sprints progressed to change the priorities of the tasks when deemed necessary, or besides the weekly meetings, having reviews of the progress and objectives with one of the supervisors, whom was none other than one of the original creators of PropEr.

## 1.4 Work plan

For the purpose of realizing this project, we divided it into self-contained iterations, each of them having their own gist to deal with. Those iterations are listed below and shown in 1.1:

- **It 1.** Study of the application and research its current architecture.
- **It 2.** Design an initial draft of a concurrent approach in Erlang.
- **It 3.** Locate the key components that should be changed or extended.
- **It 4.** Implementation of the parallel and distributed changes in the property-based testing tool.
- **It 5.** Testing and validation of our work by running benchmarks of projects with Property-Based Testing on both a cluster to test the distributed execution and in a powerful computer with a high number of cores to test the parallel execution.

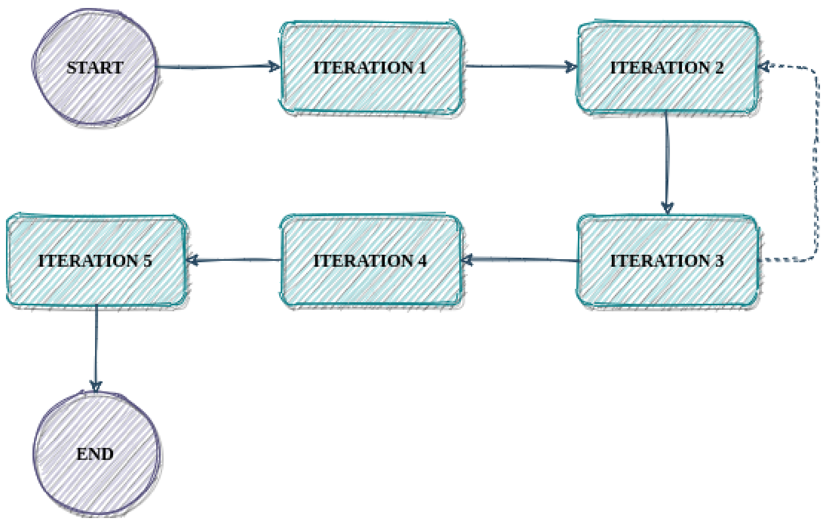


Figure 1.1: Iterations of the project

### 1.4.1 Project cost

As previously discussed, we followed the Scrum methodology with sprints of a defined duration of two weeks. By taking into account that and also the average work done per sprint, we have calculated the effort spent on each iteration, based on 1 hour/day averaged over 44 weeks, to be as follows:

| Iteration | Effort (hours) |
|---|---|
| **It 1.** Study and research of the tool | 45 |
| **It 2.** Design stage | 70 |
| **It 3.** Location of key components in the tool | 25 |
| **It 4.** Implementation of the design | 165 |
| **It 5.** Testing, validation and benchmarking | 145 |
| **Total** | 450 |

Table 1.1: Breakdown by iteration of the effort of the project

The total effort of this project therefore is 450 hours. However, it is important to note that the calculated effort is only from the student; the hours and effort from both supervisors have yet to be calculated. Based on the weekly meetings we had during the project, and adding a small bit more of time to take into consideration any interaction that was not during said meetings (i.e., Skype calls because of some questions, feedback, etc), each supervisor worked in the project around 2.5 hours per iterations, which adds to 12.5 hours of total work each supervisor through the iterations, or a total of 25 hours done by the two supervisors together for the duration of the project.

Then, based on the average salary for a Software Engineer in Spain, and estimating the hourly wage of each supervisor to be 60€/h, the cost of the student, both supervisors and therefore of the project can be calculated, and is shown in 1.2.

| Resource | Cost (€/h) | Hours | Total cost (€) |
|---|---|---|---|
| Student | 25 | 450 | 11250 |
| Supervisors | 40 | 25 | 1000 |
| | | | 12250 |

Table 1.2: Total cost of the project

As the rest of the tools used (e.g., Github, Travis CI) being free and because of both the CITIC lending us access to the cluster of machines used in the project, and the Uppsala University giving us a machine to test with too, the final budget of the project is 12250€.

## 1.5 Report layout

The rest of this report is structured as follows:

- **Chapter** 2 - ***Background***. In this chapter we will explain some notions and ideas needed to comprehend certain parts or aspects of this project's work.
- **Chapter** 3 - ***Development***. In this chapter we will cover how we set off to research the property-based testing tool to come up with a concurrent design for it and later implement it.
- **Chapter** 4 - ***Testing and benchmarking***. This chapter covers how we tested and validated the aforementioned implementation, as well as the benchmarks of its new execution times.
- **Chapter** 5 - ***Conclusions***. In this chapter we present the conclusions reached, along with possible future improvements that our work could benefit from.

# Chapter 2
# Background

In this chapter we will explain certain concepts and/or notions that the reader should know about to get a proper understanding of our work, such as what is all this fuss about Erlang and why it is an incredible language to write concurrent software, and a quick introduction into the world of Property-Based Testing.

## 2.1 Erlang/OTP

Erlang is a functional programming language, which means that as the rest of programming languages based on the functional paradigm, its programs are structured as a set of, usually pure, functions (i.e., functions that for a given input always return the same output), contrary to the more commonly used imperative paradigm, where commands or instructions are used to make up the programs. It was designed "from the bottom up to program concurrent, distributed, fault-tolerant, scalable, soft, real-time systems" [10] and developed by Ericsson; on the other hand, OTP is a collection of design principles and Erlang libraries to help develop these systems.

Some of Erlang features are that its data types (e.g., lists, functions, maps, strings, etc) are immutable, variables can only be bound once (i.e., they cannot have their value changed once assigned) and consequently any function defined in an Erlang program, also known as a module, will always produce a new copy of its output. It also makes use of the *pattern matching* mechanism, which in Erlang occurs when evaluating any function call, *receive*, *case*, *try* expressions and match operators (=) expressions, by matching the left-hand side of an expression against its right-hand side; if the matching succeeded any unbound variables became bound, if it failed a runtime error has been generated.

An example of a module that prints to the standard output "Hello, World!" is shown in 2.1 to give the reader a quick glance of what a simple program in Erlang would look like.

```erlang
-module(hello_word).
-export([greet/0]).

greet() -> io:format("Hello, World!~n").
```

Listing 2.1: Example of a *Hello World* program in Erlang

Nonetheless, the biggest strength and perhaps peculiarity of the language is its concurrency model, as Erlang uses the *Actor model* [11], modeling each actor as an Erlang process. These processes are not Operating System processes but rather are lightweight, fast to create and terminate, and isolated between themselves (i.e., they do not share memory) processes whose scheduling and mapping to actual OS processes is made very efficiently and transparently by Erlang's Virtual Machine, the BEAM, which also handles the distribution of Erlang itself across multiples machines through the use of nodes, of which we will talk about more in the next chapter.

These processes are created by calling `erlang:spawn/3`[1] function and are able to share information among themselves by sending messages. An example of a program that uses processes and message passing is shown in 2.2.

```erlang
-module(processes_example).
-export([start/0, add/3]).

add(X, Y, From) ->
    N = X + Y,
    From ! {addition, N, self()}.

start() ->
    Pid = spawn(?MODULE, add, [40, 2, self()]),
    receive
        {addition, N, Pid} -> N == 42
    end.
```

Listing 2.2: Example of a concurrent program in Erlang

For an easier understanding of the program, it has been broken down into its important bits:

1. First, a function that adds two numbers and sends the result back to the PID of another process, `processes_example:add/3`, is defined.

[1]In Erlang the number of arguments a function is passed to determines its *arity*. As functions are uniquely defined by the combination of their module name, function name and arity, they are are usually denoted as `module_name:function_name/arity`

2. Then, a process that executes that function is spawned at line 10, in this case to add 40 and 2 and send the result back to the caller of the program. The PID of the process is assigned to the Pid variable.

3. Finally, the result is sent back at line 7 and received and asserted during lines 11-13. We can be sure that it is the correct message as we *pattern matched* it against `{addition, N, Pid}` and since the `Pid` variable was already bound to the PID of the Erlang process that is executing the function, no other message will match except the one that was sent from the spawned process.

Some further examples of components from OTP that are worth highlighting are *gen_servers*, generic server processes with standard interfaces, and *supervisors*, processes capable of supervising and restarting if something goes wrong other *child processes* (which in turn can be either a supervisor or any other kind of process).

## 2.2 Property-Based Testing

To highlight the value that Property-Based Testing brings to the world of software testing, this section will start by summing up/talking about why software testing is important and how it is usually done.

### 2.2.1 Unit Testing

The most common method of software testing is *unit testing*, where individual units or components are validated by asserting whether for cherry-picked inputs the unit produces the expected output. These units are can be functions, modules, etc. The problem that comes with unit testing is

And so, if we wanted to test an Erlang program that reverses the order of a list, as shown in 2.3.

```erlang
-module(unit_testing_example).

reverse(L) -> reverse(L, []).

reverse([], Acc) -> Acc;
reverse([H|T], Acc) -> reverse(T, [H|Acc]).
```

Listing 2.3: Program that reverses the order of a list

One would write unit tests that for handpicked lists the developer decided to choose, the program will return them in reverse order (2.4).

```
-module(unit_testing_example).
-include_lib("eunit/include/eunit.hrl").

reverse(L) -> reverse(L, []).

reverse([], Acc) -> Acc;
reverse([H|T], Acc) -> reverse(T, [H|Acc]).

reverse_empty_list_test_() ->
    ?_assert([] == reverse([])).

reverse_1_element_test_() ->
    ?_assert([1] == reverse([1])).

reverse_2_elements_test_() ->
    ?_assert([2, 1] == reverse([1, 2])).

reverse_3_elements_test_() ->
    ?_assert([3, 2, 1] == reverse([1, 2, 3])).
```

Listing 2.4: Example of unit testing in Erlang

These unit tests, although grouped together into one function for this example, would be counted individually when running them with Erlang's unit testing framework, EUnit, and display the following (2.5) output.

```
======================== EUnit ========================
module 'unit_testing_example'
  unit_testing_example:10: reverse_empty_list_test_...ok
  unit_testing_example:13: reverse_1_element_test_...ok
  unit_testing_example:16: reverse_2_elements_test_...ok
  unit_testing_example:19: reverse_3_elements_test_...ok
  [done in 0.012 s]
=======================================================
  All 4 tests passed.
```

Listing 2.5: Example of a successful EUnit invocation

This example also helps to present and visualize the inherent problem of unit testing: as the test cases have to be cherry-picked, this not only becomes a time consuming task, but also a tiresome one to do. Furthermore, when writing a large number of test cases the chances of making a mistake can only go up and, to boot, one cannot be positive that all these tests demonstrate the correctness of the code.

### 2.2.2 Property-Based Testing

This is where Property-Based Testing shines, as the the idea behind this type of testing is a simple one: once properties we want our software to satisfy have been specified, the property-based testing tool of choice will automatically generate random inputs that test whether our software satisfies these previously mentioned properties or not.

In addition, properties are not a complex concept, but rather a simple one, as they are simply logical statements that capture *partial* correctness of the code. Coming back to the exampled used earlier to show what unit testing is about, one initial property[2] we could think of our reversing program should satisfy is that the reverse of a reversed list should be equal to the original list (2.2.2).

```
?FORALL(L, list(integer()), L == reverse(reverse(L))).
```

Or in other words, for all lists of integers L the reverse of its reverse order should be equal to itself. Nonetheless, we will break down this property so that the reader can understand what is happening when writing such line:

- When using PropEr, properties are checked by invoking the `?FORALL/3` macro[3]. This macro expects the following arguments, and in this particular order:

    1. The variable(s) where the values produced by the generator (i.e., the second argument) will be bound.
    2. The type generator to use when producing values for the variable(s).
    3. A boolean expression, be it either an already defined function or a definition of one, that must return *true* when the property holds and *false* otherwise.

To sum it up, a property is no other thing that a combination of a list of variables to use when writing it, a type generator to produce values for them and an expression using those variables to check whether the property holds or not.

This approach on how tests are defined shifts the focus of the developers and/or testers from thinking test cases and their expected output to finding the properties for the software, which yields deeper understanding of the software's behaviour and also works as a specification of it.

[2]Whilst we haven't covered yet PropEr itself, any examples of Property-Based Testing will be done with it, so this example has its syntax.

[3]A macro definition in Erlang can either be a constant or a function definition, which is expanded at compilation time.

## 2.3 PropEr

PropEr (*PROPerty-based testing tool for ERlang*) is the only existing property-based testing tool for Erlang with a free license that is still maintained. It was inspired by QuickCheck, the first property-based testing tool, and along with it its the most powerful tool to date. It was developed by Manolis Papadakis [12] (who mainly wrote the base system), Eirini Arvaniti [13] (who wrote the *stateful* code subsystem) and Kostis Sagonas, and has a long list of features, among them:

- It has two extra libraries to help deal with *stateful* code and its properties (i.e., properties that have to take into account that the code will not always return the same output, as it has state, and should test accordingly).
- As it is tightly integrated with Erlang's type language, the tool is able to:
  - Use custom defined types as generators instead of using PropEr's built-in ones.
  - Test functions automatically based on their specs[4] alone.
- Automatically *shrink* the input of a property that did not hold to the actual minimal input that would make it fail.
- It can narrow down the generated inputs so they are more relevant to the property when doing *Targeted Property-Based Testing*.

More information regarding the features the tool has or about other macros that have not been mentioned, as well as use cases, can be found in the the tool's official documentation [14].

Continuing with our example of unit testing the Erlang program that reverses a list from 2.4, we will instead use PropEr with the previously shown property (2.2.2) to demonstrate why and how Property-Based Testing is better. The code of the module with the property defined is shown in 2.6.

```erlang
-module(proper_testing_example).
-include_lib("proper/include/proper.hrl").

reverse(L) -> reverse(L, []).
reverse([], Acc) -> Acc;
reverse([H|T], Acc) -> reverse(T, [H|Acc]).

prop_reverse() ->
    ?FORALL(L, list(integer()), L == reverse(reverse(L))).
```

Listing 2.6: Example of property-based testing in Erlang

[4]Erlang's type language comes with a notation for declaring sets of Erlang terms to form a particular type and a specification system for functions based on these types and the built-in ones.

To start with, doing Property-Based Testing reduces the visual cluttering found of having a large number of unit tests, as the property to test whether the reverse function works is just one line long. Next, some of the best features from PropEr can only be seen when executed[5], so we will show a successful execution (2.7) of it and then we will change the property to a wrong one in order to demonstrate how PropEr behaves (2.8). Finally, we will add another property to further show how simple is to test with PropEr even in modules with multiple properties (2.9, 2.10).

Invoking PropEr is as simple as calling `proper:quickcheck/1` with the property function we want to to test.

```
1> proper:quickcheck(proper_testing_example:prop_reverse()).
..............................................................
..................................
OK: Passed 100 test(s).
```

Listing 2.7: Example of a successful PropEr invocation

During its execution PropEr will display a dot (.) for each passed test of the property. But what happens when PropEr finds that a property does not hold? In order to see that, we will change the old property (2.2.2) to a wrong one where we believe that the reverse of a list should be equal to the list itself, as show in 2.3.

```
?FORALL(L, list(integer()), L == reverse(L)).
```

If we call PropEr again, we will see fail, as shown in 2.8.

```
1> proper:quickcheck(proper_testing_example:prop_reverse()).
...!
Failed: After 4 test(s).
[-1,1]

Shrinking .(1 time(s))
[0,1]
```

Listing 2.8: Example of a failed PropEr invocation

What is interesting from this output, though, is that the moment PropEr found a failing case not only does it display it, but also it goes even further and by shrinking the input it was able to find the smallest possible input that would make the property not hold!

Now let us add two new properties and see how one can test a module that has multiple properties. To demonstrate this, the first property will check whether the reversed list has the same length as the original list or not, whereas the second property will check whether the

[5]PropEr can be executed two ways: one is to do it from inside Erlang's interactive shell [15] and the other is using *rebar3* proper plugin [16, 17]. We will be doing all our examples using the former for the sake of consistency.

reversed list and the original list are equal once both have been sorted. These new properties are shown in the final revision of the example module (2.9).

```erlang
-module(proper_testing_example).
-include_lib("proper/include/proper.hrl").

reverse(L) -> reverse(L, []).

reverse([], Acc) -> Acc;
reverse([H|T], Acc) -> reverse(T, [H|Acc]).

prop_reverse() ->
    ?FORALL(L, list(integer()), L =:= reverse(reverse(L))).

prop_length() ->
    ?FORALL(L, list(integer()), length(L) =:= length(reverse(L))).

prop_sort() ->
    ?FORALL(L, list(integer()),
        lists:sort(L) =:= lists:sort(reverse(L))).
```

Listing 2.9: Example of property-based testing in Erlang

This time, instead of *quickchecking* each property, we will use a function of PropEr that does that for us for every property found in a module, `proper:module/1` (2.10).

```
1> proper:module(proper_testing_example).
Testing proper_testing_example:prop_reverse/0
.................................................................
...................................
OK: Passed 100 test(s).

Testing proper_testing_example:prop_length/0
.................................................................
...................................
OK: Passed 100 test(s).

Testing proper_testing_example:prop_sort/0
.................................................................
...................................
OK: Passed 100 test(s).
```

Listing 2.10: Example of calling `proper:module/1`

Now, the reader might have noticed from comparing the output of EUnit with the output of PropEr, that one thing the former does that the latter does not is showing how long it took to run the test suite. Sure, one could measure the time that takes running all the properties in the modules and it will be quite fast, but the point to be made here is that even though each property could be run independently of the others, PropEr runs them sequentially. Furthermore, even EUnit allows executing tests in parallel, but, alas, PropEr has a limited scope of parallel execution to only stateful properties. Hence, this thesis' work.

### 2.3.1 Stateless vs. stateful properties

The reader might have noticed that, although *stateful* properties have been mentioned twice, we have yet to discuss them further, instead of simply name-dropping them. This short subsection will serve as an introduction to the concept of stateful properties, due to them being a key point of the design (3.1) and implementation (3.3) that will be discussed in the following chapter.

Stateful properties are those that test stateful systems, which are programs that are not purely functional[6] and have side-effects; in short, programs with state. Thus, the Erlang program example we have been using up to this point is a stateless one and its properties, shown earlier, have been stateless properties.

Akin to a stateless property, a stateful one has its own three main components: the system to test, a model representing what the system does, and a generator for commands to represent the execution flow of the system. With these three pieces, PropEr is able to test stateful codebases using state machines that have their state changed with the aforementioned generated commands while checking whether a series of preconditions and postconditions are satisfied for that command or not.

As adding an example of Stateful Property-Based Testing to this report and explaining it would take too many pages, we will instead point the reader to a great example of Stateful Property-Based Testing from the official website of the tool [18].

[6]Remember, pure functions are those that always produce the same output for a given input.

## Chapter 3

# Development

In this chapter we will talk about the initial design stage, how we approached PropEr and found the right places to make these improvements, and later implement them, and what challenges that brought and how we overcame them.

## 3.1 Design

One of the main goals we wanted to achieve during the design phase was to think of a simple yet efficient way of making the property-based testing tool work concurrently while having little impact on the existing codebase (i.e., changing as less code as possible). In order to do so in a feasible fashion, we had to take into account the following issues:

1. PropEr is able to deal with stateless and stateful testing. Since the latter is usually found in Erlang code in the form of *gen_servers*[1] and they can have unique identifiers for a single node[2], we needed to be able to ascertain whether a property was stateless or stateful before running its tests so that we could prepare in case of having to isolate the stateful ones among themselves.

2. If we make concurrent software, the chances of something going wrong increment. In the event of an error, we did not want to crash the whole Virtual Machine and therefore the runtime system running the property-based testing tool.

3. We also wanted to extend PropEr with both parallel and distributed execution, so we needed a practical form of doing so without having to duplicate code. Ideally, the distributed execution would be merely having to run the code in parallel but in a distributed fashion, not its own implementation.

[1]A gen_server is an Erlang process used to keep state, execute code asynchronously, etc, through the use of an standard set of functions.

[2]A node is an executing Erlang runtime system that has been named.

As a result of raising these concerns early during the Design stage, we had time to think them more carefully and come up with better ideas. We will discuss next the proposal we ended up coming with to solve this issues and which are its key points.

### 3.1.1 Proposed design

The most important facet is how we thought and designed the concurrency we wanted PropEr to have. We could have used a *gen_server* to handle performing the tests asynchronously and any error that could happen because of them, but as we have already mentioned this was not a valid solution as we did not like the idea of imposing on the users of PropEr not to use any process or gen_servers with the same unique identifier as ours (in short, we did not want to keep a name identifier for ourselves), and not having the imposition in the first place would mean that we could end up clashing with registered processes in codebases that had not looked into the source code of PropEr to notice the problem. Furthermore, even if we went along with the gen_server option, we would end up having to coordinate several of them when testing stateful properties (as we already discussed, stateful properties tests have to be cut off from each other).

Because of this and the fact that we could not add more dependencies to the project (and therefore use third party libraries) to help with this issue, what we ended up doing was working with the simplest concepts to do concurrent software in Erlang: spawning processes and message passing.

Originally, PropEr would execute a number of tests (by default 100) sequentially for each property, stopping only in the case of an error or a failed test and continuing otherwise. Our design proposes the following changes to the property-based testing tool:

- PropEr should spawn as many processes as the user specifies (from hereon they will be referred to as *workers*), assign to each of them a portion of the total of tests to run (more on this will be discussed later in 3.1.2 and in the Implementation section in 3.3.5), have the workers perform each their share in parallel and simply wait for all workers to finish testing before reporting the aggregated results.

- To solve the issue number 2, we settled on starting a dedicated node where the workers should be spawned thereafter. That way, in the case of a worker crash, only that node would crash. This idea was taken from *Nifty* [19], another project from one of this thesis' supervisors, whom insisted on having the concurrent model be as fault-tolerant as possible.

- In consequence of the previous point, and in order to address the issue number 1, the number of dedicated nodes that will be started will vary according to the type of property to test. Thus, stateless properties will spawn their workers on a single dedicated node started beforehand, whereas stateful properties will start as many nodes as workers will be created, and spawn those workers each in their own node, so that they are isolated them among themselves and to avoid any possible clash that otherwise could happen when testing stateful properties in a single node.

- Finally, we addressed the issue number 3 with this design decisions. With them, running PropEr in a distributed fashion means that we only need to know the machine where the node(s) and workers should be started and spawned, or a list of already started node(s) in other machines where the workers will be created.

### 3.1.2 Test distribution among workers

Because of the fact that we have to divide the workload among multiple workers, there was the need to think of a way of doing such distribution correctly, whilst taking into account that, as previously stated, PropEr generates random increasing inputs as the tests pass.

#### Even distribution

For example, a simple approach that one might think of is to just split the total number of tests evenly among the workers.

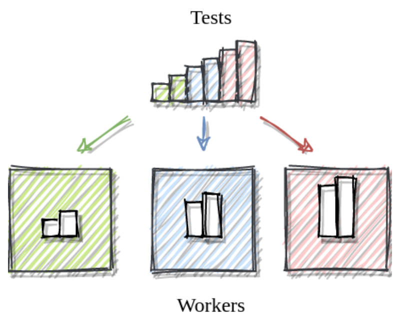


Figure 3.1: Even distribution of tests among the workers

The problem that this strategy entails is that, although there is an even distribution of the tests among the workers (as shown in 3.1), their actual load will be uneven — the last workers will run tests of larger size.

As we did not want to end up with an unbalanced workload among the workers, we devised three other different strategies, or approaches, to split the tests and give each worker their corresponding share.

### Uneven distribution

This strategy is the least different in concept to the first one introduced. It aims to fix the unbalanced workload problem the even distribution had by splitting the tests unevenly among the workers in such a way that takes into account that the last tests are of larger size and offsets this by giving more tests to the first workers (3.2).

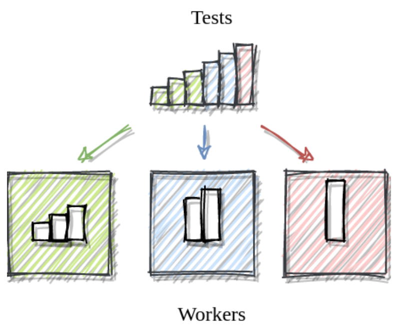


Figure 3.2: Uneven distribution of tests among the workers

### Batches distribution

One approach to keep the workload as balanced as possible is to produce batches of tests and send them to the workers during a certain number of rounds. For instance, if PropEr is to run *N* tests, with this approach the first batch would be made up of *N/k* tests, which then would be split evenly among the workers into their respective shares (*N/kn*) — if no test fails, the next batch should be sent to all workers; this should be continued for *k* rounds (3.3).

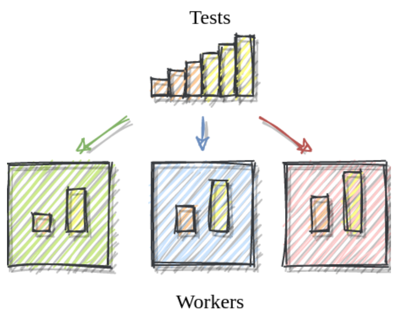


Figure 3.3: Batches distribution of tests among the workers

### *N*th sequence distribution

The last of the strategies we thought of is one that benefits from being an “embarrassingly parallel” one: instead of doing a single sequence of tests (i.e., having to run 100 tests is the same as doing a series of tests, starting at 0 and ending at 99), each worker only needs to run their own succession of test that is made up of selecting every $N$th test of the original sequence (3.4).

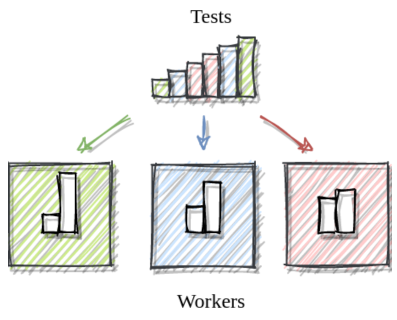


Figure 3.4: $N$th sequence distribution of tests among the workers

Since we wanted PropEr to have the best of the described approaches, for us to be able to determine which one was a better option we decided to implement all of the four strategies and benchmark their performance, but that will be discussed later in 3.3.5.

## 3.2 A PropEr study

In order to carry out the design we had in mind, there was the need to study thoroughly the tool in search of the key places where it would be feasible to start tinkering with. To do this, we first got the code from the original repository [9] and started researching it after we thought of an strategy to speed up this process: the quickest way to understand what code was of more importance was to execute PropEr in different environments and look at its control flow (i.e., the order in which the functions were being called).

And so, we set out to get as much information as possible in the shortest amount of time. To do so, we initially used one of the tools for tracing and investigating distributed systems that is included in Erlang's *Observer* application, *Trace Tool Builder*. This tool allows tracing applications during their runtime and, thanks to it, we were able to get a first glimpse into how the control flow if PropEr looked like, as shown in 3.5.

As we also needed to understand the tool and its code, we approached it by first reading its API documentation [14], where we discovered which module acted as the central component of the system, and we simply inspected those module's exported functions[3] (3.1) and started off reading from those functions more exhaustively and in-depth.

[3]In Erlang, only the exported functions are visible from outside that module. In other words, exported functions act as a public API of their module.

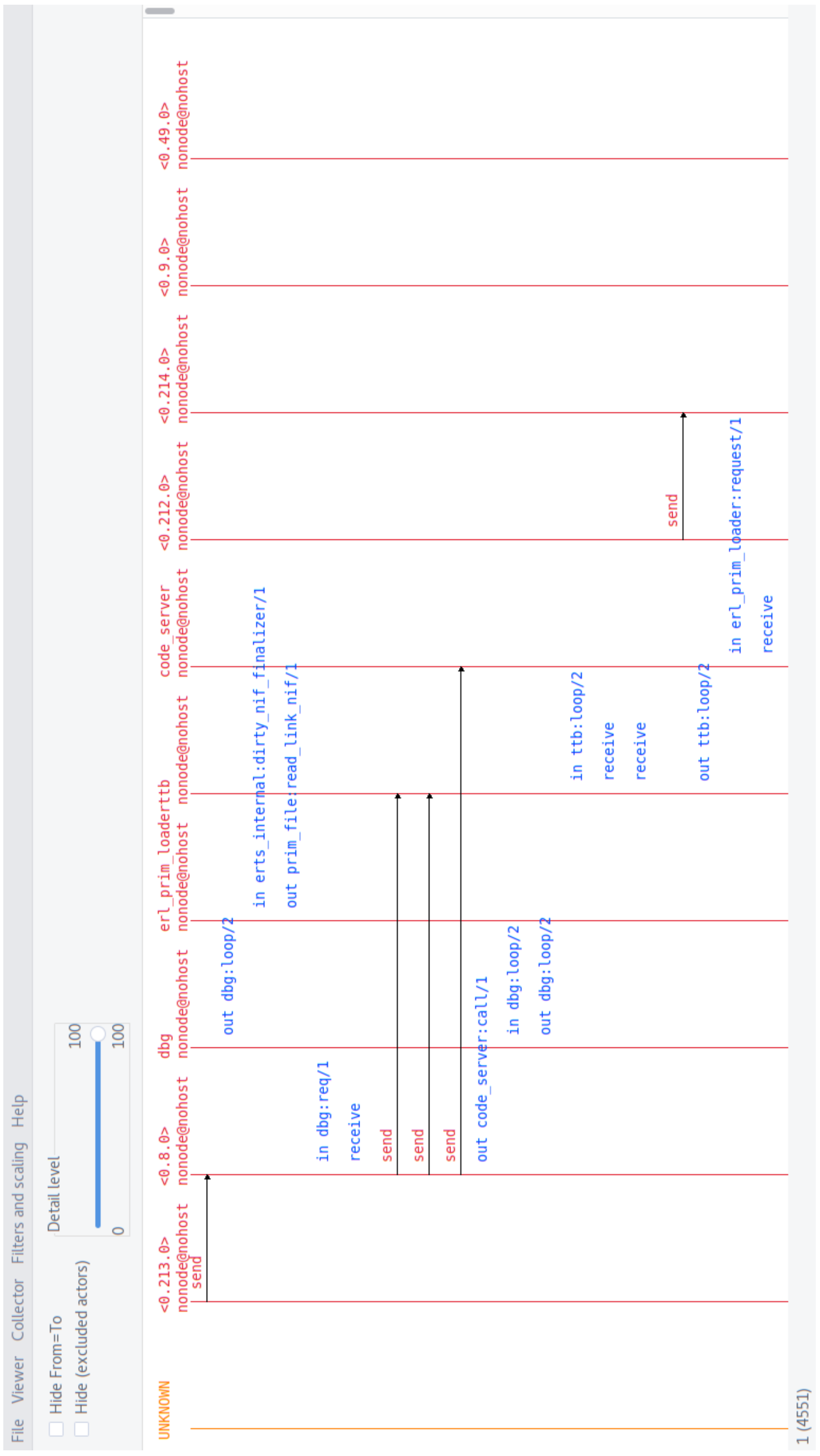
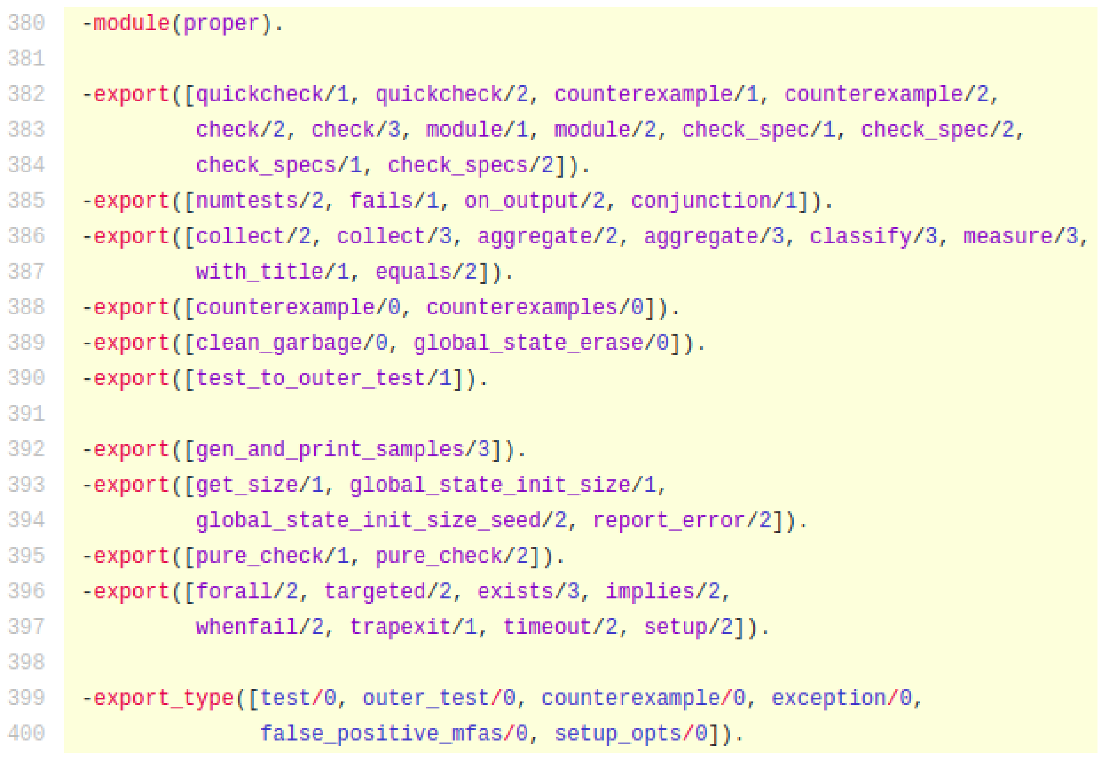


Figure 3.5: Tracing visualized using the *Event Tracer*



```erlang
-module(proper).

-export([quickcheck/1, quickcheck/2, counterexample/1, counterexample/2,
         check/2, check/3, module/1, module/2, check_spec/1, check_spec/2,
         check_specs/1, check_specs/2]).
-export([numtests/2, fails/1, on_output/2, conjunction/1]).
-export([collect/2, collect/3, aggregate/2, aggregate/3, classify/3, measure/3,
         with_title/1, equals/2]).
-export([counterexample/0, counterexamples/0]).
-export([clean_garbage/0, global_state_erase/0]).
-export([test_to_outer_test/1]).

-export([gen_and_print_samples/3]).
-export([get_size/1, global_state_init_size/1,
         global_state_init_size_seed/2, report_error/2]).
-export([pure_check/1, pure_check/2]).
-export([forall/2, targeted/2, exists/3, implies/2,
         whenfail/2, trapexit/1, timeout/2, setup/2]).

-export_type([test/0, outer_test/0, counterexample/0, exception/0,
              false_positive_mfas/0, setup_opts/0]).
```

Figure 3.6: Exported definitions (i.e., functions and types) of the tool's main module

```erlang
-module(proper).

-export([quickcheck/1, quickcheck/2, counterexample/1,
         counterexample/2, check/2, check/3, module/1, module/2,
         check_spec/1, check_spec/2, check_specs/1, check_specs/2]).
-export([numtests/2, fails/1, on_output/2, conjunction/1]).
-export([collect/2, collect/3, aggregate/2, aggregate/3,
     classify/3, measure/3,
    with_title/1, equals/2]).
-export([counterexample/0, counterexamples/0]).
-export([clean_garbage/0, global_state_erase/0]).
-export([test_to_outer_test/1]).
-export([gen_and_print_samples/3]).
-export([get_size/1, global_state_init_size/1,
    global_state_init_size_seed/2, report_error/2]).
-export([pure_check/1, pure_check/2]).
-export([forall/2, targeted/2, exists/3, implies/2,
          whenfail/2, trapexit/1, timeout/2, setup/2]).
-export_type([test/0, outer_test/0, counterexample/0, exception/0,
         false_positive_mfas/0, setup_opts/0]).
```

Listing 3.1: Exported definitions (i.e., functions and types) of the tool's main module

From the list of exported functions is shown in 3.6, the most important functions that can be found are `quickcheck/1,2` and `module/1,2`, with the former being the entry point to PropEr when calling a `?FORALL/3` property macro, and the latter a function that invokes the former with each property that exists in a given module. The reader might notice that both functions have a similar function that instead has arity 2, those simply take as their second argument a list of user-defined options if present; as they end up converging with their matching function in the end, it does not matter which of the two arities we decide to pick as our starting point.

For an easier comparison of the tool's control flow, before (3.7) and after (3.8) the design was implemented, the annotated sequence diagrams are shown one after the other.

As seen in the old one, the most interesting key points of PropEr to start redesigning with concurrency in mind were `proper:inner_test/2` and `proper:perform/3`. The former is the last function to be executed before performing the tests and the one to report the result gotten from them; whereas the latter is the main function related to the execution of the tests, counting how many tests have been done and are left to do, if a test needs to be tried again, etc. Their code is shown in 3.2 and 3.3, respectively.

```
inner_test(RawTest, Opts) ->
    %% Get the relevant user (or default) options
    #opts{numtests = NumTests, long_result = Long,
            output_fun = Print} = Opts,
    %% Get the test to run
    Test = cook_test(RawTest, Opts),
    %% Perform it and get the results
    ImmResult = perform(NumTests, Test, Opts),
    Print("~n", []),
    %% Report them
    report_imm_result(ImmResult, Opts),
    {ShortResult,LongResult} = get_result(ImmResult, Test, Opts),
    case Long of
  true  -> LongResult;
  false -> ShortResult
    end.
```

Listing 3.2: Code of original `proper:inner_test/2`

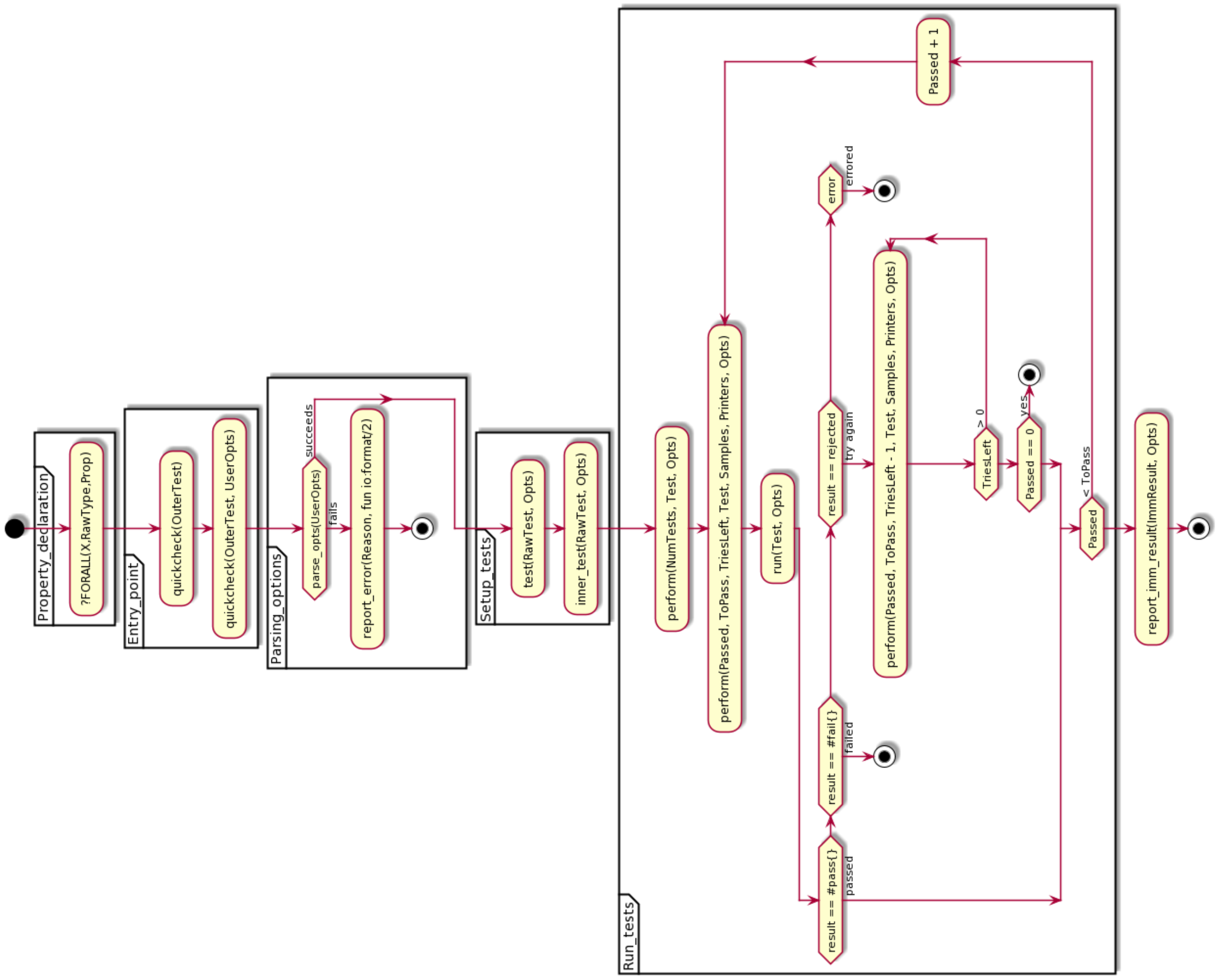


Figure 3.7: Control flow of the tool originally

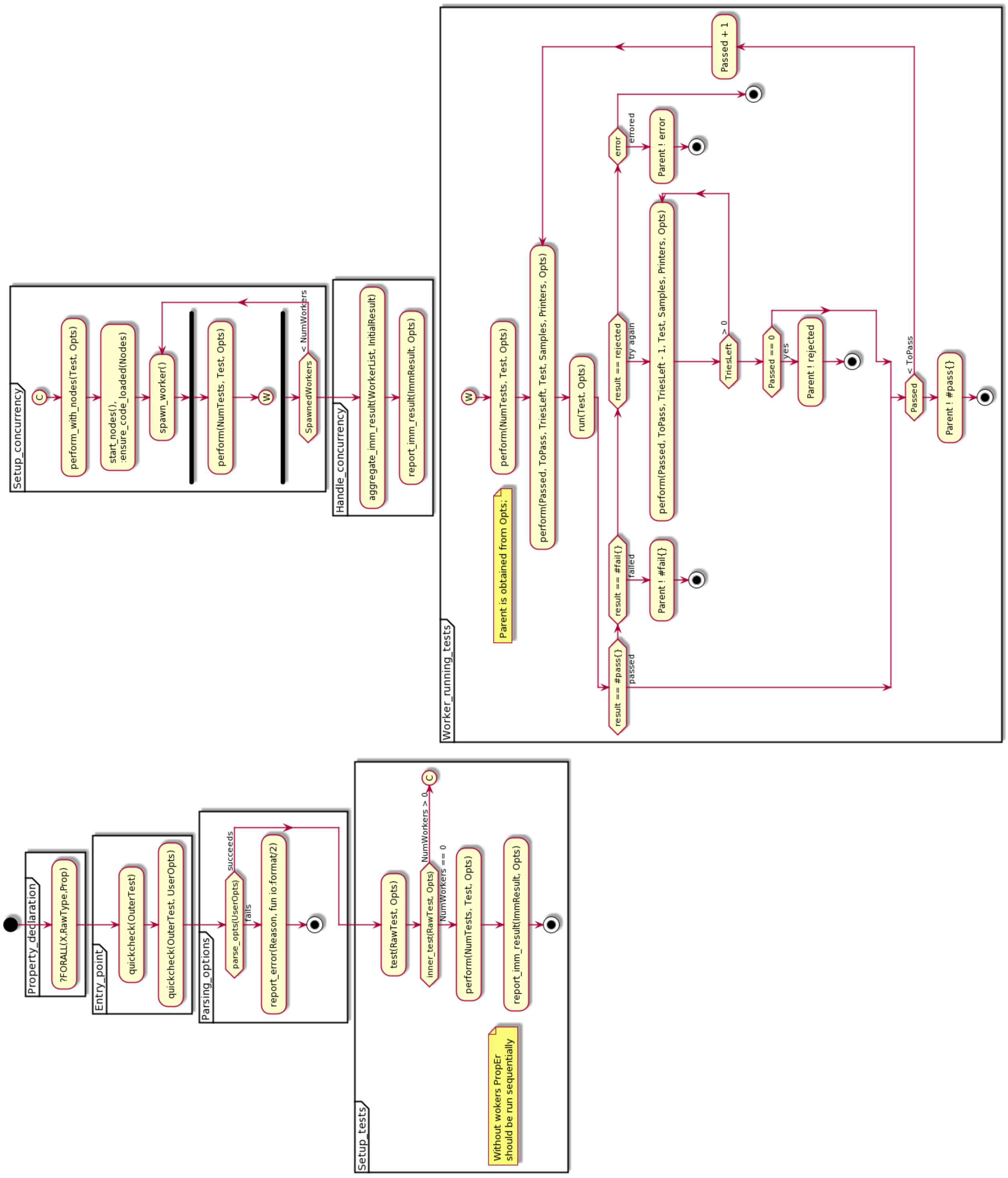

Property_declaration
?FORALL(X,RawType,Prop)
Entry_point
quickcheck(OuterTest)
quickcheck(OuterTest, UserOpts)
Parsing_options
parse_opts(UserOpts)
succeeds
fails
report_error(Reason, fun io:format/2)
Setup_tests
test(RawTest, Opts)
inner_test(RawTest, Opts)
NumWorkers > 0
NumWorkers == 0
perform(NumTests, Test, Opts)
report_imm_result(ImmResult, Opts)
Without wokers PropEr should be run sequentially
Setup_concurrency
perform_with_nodes(Test, Opts)
start_nodes(), :ensure_code_loaded(Nodes)
spawn_worker()
perform(NumTests, Test, Opts)
SpawnedWorkers
< NumWorkers
Handle_concurrency
aggregate_imm_result(WorkerList, InitialResult)
report_imm_result(ImmResult, Opts)
Worker_running_tests
Parent is obtained from Opts;
perform(NumTests, Test, Opts)
perform(Passed, ToPass, TriesLeft, Test, Samples, Printers, Opts)
run(Test, Opts)
result == #pass{}
passed
result == #fail{}
failed
Parent ! #fail{}
result == rejected
try again
perform(Passed, ToPass, TriesLeft - 1, Test, Samples, Printers, Opts)
error
errored
Parent ! error
TriesLeft
> 0
Passed == 0
yes
Parent ! rejected
Passed
< ToPass
Passed + 1
Parent ! #pass{}

Figure 3.8: Control flow of the tool after implementing the design

```
perform(NumTests, Test, Opts) ->
    perform(0, NumTests, ?MAX_TRIES_FACTOR * NumTests, Test, none,
            none, Opts).

%% No retries left for a test, base case
perform(Passed, _ToPass, 0, _Test, Samples, Printers, _Opts) ->
    case Passed of
        0 -> {error, cant_satisfy};
        _ -> #pass{samples = Samples, printers = Printers,
                   performed = Passed, actions = []}
    end;
%% All tests have passed, base case
perform(ToPass, ToPass, _TriesLeft, _Test, Samples, Printers,
        _Opts) ->
    #pass{samples = Samples, printers = Printers,
            performed = ToPass, actions = []};
%% Recursive case
perform(Passed, ToPass, TriesLeft, Test, Samples, Printers,
        #opts{output_fun = Print} = Opts) ->
    case run(Test, Opts) of
        %% Test passed, do another one
        #pass{reason = true_prop, ...} ->
            perform(Passed + 1, ToPass, TriesLeft - 1,
                    Test, ..., Opts);
        %% Test failed, stop and return the fail
        #fail{} = FailResult ->
            FailResult#fail{performed = Passed + 1};
        %% Test was rejected, try again
        {error, rejected} ->
            perform(Passed, ToPass, TriesLeft - 1, Test,
                Samples, Printers, Opts);
        %% From hereon, return the error from running the test
        {error, Reason} = Error when Reason =:= arity_limit
                             orelse Reason =:= non_boolean_result
                             orelse Reason =:= type_mismatch ->
            Error;
        {error, {cant_generate,_MFAs}} = Error ->
            Error;
        {error, {typeserver,_SubReason}} = Error ->
            Error;
        Other ->
            {error, {unexpected,Other}}
    end.
```

Listing 3.3: Simplified code of original `proper:perform/3`

Most of the code of `proper:perform/3,4` has been simplified, as the internal workings of the function is not that relevant to making the property-based testing tool concurrent or fitting to leave written on this report; in short, we only care about the results obtained from running a test.

At first, our idea was to make `proper:perform/3,4` concurrent, but in order for this to be feasible we would end up creating as many processes as tests are to be ran. In one of the discussions on this topic with Kostis, he suggested to start redesigning from the other previously mentioned function, `proper:inner_test/2`, as at that level of code we have more control over the way the tests are executed.

## 3.3 Implementation

The final revision of our work can be found in a fork of the original project (both on Github) at the following link:

`https://github.com/pablocostass/proper`

### 3.3.1 Preamble

In this section we will talk about how we implemented the previously discussed design to extend the property-based testing tool, PropEr, with parallel and distributed execution while maintaining the existing functionality of the tool and following the best of coding practices, such as *Test Driven Development* and type checking of functions, besides following the directions of one of the tool's creators and maintainers.

Some of the points of the implementation will not be explained thoroughly as this report has a maximum length allowed and many concepts from Erlang and from the tool itself should be known beforehand in order to have a proper understanding of all the code, were it not to be shown simplified or highlighted in snippets.

### 3.3.2 Project structure

As this is a fork, it has to follow the same license as the original repository does (GPL-3.0 [20]) and an equal project structure, which is shown in 3.4

```
$ tree -d
.
├── doc
├── examples
├── examples_test
├── include
├── scripts
├── src
```

```
└── test

7 directories
```

Listing 3.4: Directory structure of the tool

The project structure follows Erlang's *Directory Structure Guidelines for a Development Environment* [21]. As specified in those guidelines the project has the following directories, some of which are required and some are optional:

- `doc` (*optional*). Contains the documentation of the tool.
- `include` (*optional*). Contains public code that should be available in other applications, such as the macro definitions of the tool (e.g., ?FORALL/3).
- `src` (*required*). Contains the Erlang source code of the tool itself.
- `test` (*optional*). Contains the files regarding the tests of the tool.

The other two directories that have not been listed above, `examples` and `examples_test`, are exclusive of this project and not related to the aforementioned guidelines. The former contains files with examples of properties and the latter has a test suite to run them.

Furthermore, the project also has a few files that can be found on the top level directory that are related to building and testing utilities for the project: a configuration file for *rebar3* [16], the most widely adopted building tool in the Erlang community and a *Makefile* to help compile, run type checking on the code and test the property-based tool itself using the aforementioned building tool.

### 3.3.3 A PropEr implementation

We set off the implementation stage with the goal of first achieving a parallel execution of the tool and then expanding it to a distributed execution also. Since the tool has received a lot of support and feedback from the community and its creators and maintainers throughout the years, one of the advantages our work has over other more traditional bachelor's thesis is that we have most of the groundwork already laid for us and we can make use of it to speed up some tasks.

As we already discussed in the design section, to make PropEr into a concurrent tool we need to change its control flow from `proper:inner_test/2` onwards. To help the reader properly perceive the changes that we have done, we will again walk through how the tool used to work (although this time faster) and why that function is important.

Properties are defined with PropEr's ?FORALL/3 macro (3.5)

```erlang
-define(FORALL(X,RawType,Prop),
        proper:forall(RawType, fun(X) -> Prop end)).
```

Listing 3.5: Definition of the FORALL/3 macro

and they are parsed by the property-based testing tool either by calling `proper:quickcheck/1,2` when one wants to test one specific property, or `proper:module/1,2` when one wants to test all properties found inside a module. Given that the latter calls the former for each property found in the module, and since both functions end up converging at some point of the tool's control flow, we will skip the code of `proper:module/1,2` as it is a bit more complex and redundant for the purpose of this section.

The usual entry point to the tool's functionality is, as previously stated, one of the two functions. They both accept a list of user defined options that will be used to either override the default values the tool has (e.g., the default number of tests to be run is 100, but if one wants to instead do a thousand tests that can be changed with the *{numtests,1000}* option) or to add some that change how the tool behaves (e.g., if one wishes to not see a dot for every passed test or an exclamation mark with failed ones; in short, the output of running the tests, and only cares about the final result the *quiet* option can be added; or when there is no need to shrink on the failing cases the *noshrink* option can be used). The code for `proper:quickcheck/1` and `proper:quickcheck/2` is shown in 3.6 to illustrate how the entry point works.

```erlang
%% @doc Runs PropEr on the property `OuterTest'.
-spec quickcheck(outer_test()) -> result().
quickcheck(OuterTest) -> quickcheck(OuterTest, []).

%% @doc Same as {@link quickcheck/1}, but also accepts a list of
    options.
-spec quickcheck(outer_test(), user_opts()) -> result().
quickcheck(OuterTest, UserOpts) ->
    try parse_opts(UserOpts) of
    ImmOpts ->
        {Test,Opts} = peel_test(OuterTest, ImmOpts),
        test({test,Test}, Opts)
    catch
    throw:{Err,_Opt} = Reason when Err =:= erroneous_option;
                                   Err =:= unrecognized_option ->
        report_error(Reason, fun io:format/2),
        {error, Reason}
    end.
```

Listing 3.6: Code of `proper:quickcheck/1,2`

The function with arity 1 assumes that the user defined list of options is empty and calls its analogous function of arity 2. None of this function's code is of real significance whatsoever, since it is quite trivial if one were to simplify the functions it invokes within its body: at line 9 the tool tries to parse the aforementioned user defined list of options and:

- If the parsing goes well and the list is valid, the tool then gets the property that needs to be tested at line 11 and goes on to prepare to test it in the following line.
- If, however, the list of options could not be parsed, the tool will throw and error and report the user of the failure.

The last function to be called on a successful parse, `proper:test/2`, is a setup function where the tool's internal values needed for testing are initialized before running the tests and erased afterwards, as shown in 3.7.

```
1  -spec test(raw_test(), opts()) -> result().
2  test(RawTest, Opts) ->
3      %% Initialize the state of the tool
4      global_state_init(Opts),
5      %% Set up the test
6      Finalizers = setup_test(Opts),
7      Result = inner_test(RawTest, Opts),
8      %% Clean the test
9      ok = finalize_test(Finalizers),
10     %% Erase the state of the tool
11     global_state_erase(),
12     Result.
```

Listing 3.7: Code of `proper:test/2`

Finally, as the reader might have noticed, we have found `proper:inner_test/2` at line 5; at last we know where it is called during the tool's control flow! It its also worth to note that, from what has been shown throughout the past code listings, not that much code have been executed to get to this point. Now we can start explaining the changes done to the tool without worrying the reader might feel that we have missed out something while illustrating what the tool has done till this point of execution.[4]

As we discussed earlier in the Design section (3.1), our proposal had four key changes needed to make the property-based testing tool into a concurrent one:

- It has to use processes to run the multiple tests at the same time.
- It should use dedicated nodes to improve the fault-tolerance in the case of an unexpected error or crash.

[4]For an easier comparison and to help the reader recall the details of how the function used to be, it was shown in 3.2.

- The number of said nodes depends on the type of property to test (*stateless* vs. *stateful*).
- The distributed execution should just be carrying out the parallel one but on a cluster of well-known beforehand machines or nodes.

We will now show and explain how we addressed all these four key points with both the modifications that were carried out in the established code (i.e., already shown) and the newly implemented functions needed to make the concurrent design work.

The tool manages all the information it needs of a property that is about to test, or in the middle of testing, through a combination of the boolean function that represents the property and, mainly, a record of the options to use during that testing. For us to integrate our design easily into the tool as it was, we decided to modify said record definition to add two new fields, *num_workers* and *parent*; the former to handle the number of processes or workers to make use of and the latter to store the PID of the main process that will take care of aggregating the results obtained from the workers. This newly updated record definition is shown in 3.8.

```
-record(opts, {
    output_fun       = fun io:format/2 :: output_fun(),
    long_result      = false           :: boolean(),
    numtests         = 100             :: pos_integer(),
    search_steps     = 1000            :: pos_integer(),
    search_strategy  = proper_sa       :: proper_target:strategy(),
    start_size       = 1               :: proper_gen:size(),
    seed             = os:timestamp()  :: proper_gen:seed(),
    max_size         = 42              :: proper_gen:size(),
    max_shrinks      = 500             :: non_neg_integer(),
    noshrink         = false           :: boolean(),
    constraint_tries = 50              :: pos_integer(),
    expect_fail      = false           :: boolean(),
    any_type       :: {'type', proper_types:type()} | 'undefined',
    spec_timeout     = infinity        :: timeout(),
    skip_mfas        = []              :: [mfa()],
    false_positive_mfas                :: false_positive_mfas(),
    setup_funs       = []              :: [setup_fun()],
    num_workers      = 1               :: non_neg_integer(),
    parent           = self()          :: pid(),
    nocolors         = false           :: boolean()
    }).
-type opts() :: #opts{}.
```

Listing 3.8: Type definition of `proper:opts()` record

Then, as `proper:inner_test/2` uses that record to get certain bits of information, like the number of tests to be ran, how to print the output obtained, etc; we decided to modify it to check whether the number of workers is not zero, and therefore is going to be executed concurrently, or is indeed zero and should be executed as usual. The modified code is shown in 3.9.

```erlang
-spec inner_test(raw_test(), opts()) -> result().
inner_test(RawTest, Opts) ->
    #opts{numtests = NumTests, long_result = Long,
            output_fun = Print, num_workers = NumWorkers} = Opts,
    Test = cook_test(RawTest, Opts),
    ImmResult = case NumWorkers > 0 of
        %% Non-zero number of workers, perform concurrently
        true ->
            case NumWorkers > NumTests of
                %% Weird case of having more workers than tests.
                %% Use only NumTests workers in this situation.
                true ->
                    perform_with_nodes(Test,
                        Opts#opts{num_workers = NumTests});
                %% Use the specified number of workers.
                false -> perform_with_nodes(Test, Opts)
            end;
        %% 0 is the default value, perform as usual.
        false -> perform(NumTests, Test, Opts)
    end,
    Print("~n", []),
    report_imm_result(ImmResult, Opts),
    {ShortResult,LongResult} = get_result(ImmResult, Test, Opts),
    case Long of
    true  -> LongResult;
    false -> ShortResult
    end.
```

Listing 3.9: Code of `proper:inner_test/2` with the modifications applied

This function goes hand in hand with a new one that handles everything related to setting up the dedicated nodes, splitting the tests among the workers and stopping the nodes at the end, when running the tool in a concurrent fashion, `proper:perform_with_nodes/2` (3.10).

```erlang
-spec perform_with_nodes(test(), opts()) -> imm_result().
perform_with_nodes(Test, #opts{numtests = NumTests,
        num_workers = NumWorkers} = Opts) ->
    %% Split the tests among the workers
    TestsPerWorker = tests_per_worker(NumTests, NumWorkers),
    NodeList =
    case property_type(Test) of
        {kind, Type} when Type =:= constructed; Type =:= wrapper ->
```

```
9              %% Stateful property. Start as many nodes as workers are
10             Nodes = start_nodes(NumWorkers),
11             ensure_code_loaded(Nodes),
12             lists:zip(Nodes, TestsPerWorker);
13         _ ->
14             % Stateless property. Start a single node
15             [Node] = start_nodes(1),
16             ensure_code_loaded([Node]),
17             lists:map(fun(N) -> {Node, N} end, TestsPerWorker)
18     end,
19     %% Disable displaying log erros to the standard output
20     ok = ?disable_logging(),
21     %% Handle maybe starting the coverage tool of Erlang
22     {ok, _} = maybe_start_cover_server(NodeList),
23     SpawnFun = fun({Node, {Start, ToPass}}) ->
24         spawn_link_migrate(Node,
25                 fun() -> perform(Start, ToPass, Test, Opts) end)
26     end,
27     %% Create the workers
28     WorkerList = lists:map(SpawnFun, NodeList),
29     InitialResult = #pass{samples = [], printers = [],
30                         actions = []},
31     %% Wait for the results to be aggregated
32     AggregatedImmResult = aggregate_imm_result(WorkerList,
33                             InitialResult),
34     %% Handle maybe stopping the coverage tool of Erlang
35     ok = maybe_stop_cover_server(NodeList),
36     %% Stop the dedicated nodes
37     ok = stop_nodes(),
38     %% Return the aggregated results
39     AggregatedImmResult.
```

Listing 3.10: Code of `proper:perform_with_nodes/2`

In those 37 lines of code a lot of things are happening, so we will break this function down into smaller chunks of its logic, highlighting the most important ones that made the concurrency possible.

1. At line 5 the tests are distributed among the workers. This is done following one of the previously mentioned strategies (3.3.5).

2. A different number of dedicated nodes are started with `proper:start_nodes/1` (either at line 11 or 16). That function simply starts as many nodes as needed calling the following function:

```erlang
-spec start_node(node()) -> node().
start_node(SlaveName) ->
    %% Ensure the Erlang Port Mapper Daemon is started
    [] = os:cmd("epmd -daemon"),
    HostName = list_to_atom(net_adm:localhost()),
    %% Start the main node, needed to start the other node(s)
    _ = net_kernel:start([proper_master, shortnames]),
    %% Start the worker node
    case slave:start_link(HostName, SlaveName) of
        {ok, Node} -> Node;
        {error, {already_running, Node}} -> Node
    end.

```

Listing 3.11: Code of `proper:start_node/1`

As a result of nodes in Erlang being runtime systems, we have to also ensure the relevant code is loaded after the nodes have been started, or otherwise we will not be able to spawn workers in them as they will not be able to execute the functions from the modules being tested. That is done with the `proper:ensure_code_loaded/1` function.

```erlang
-spec ensure_code_loaded([node()]) -> 'ok'.
ensure_code_loaded(Nodes) ->
    %% Get all the files that need to be loaded from the current
    directory
    Files = filelib:wildcard("**/*.beam"),
    Modules =
        [erlang:list_to_atom(filename:basename(File, ".beam"))
            || File <- Files],
    %% Call the functions that ensure all modules are available on
    the nodes
    [maybe_load_binary(Nodes, Module) || Module <- Modules],
    [rpc:multicall(Nodes, code, add_patha, [Path])
        || Path <- code:get_path()]
    _ = rpc:multicall(Nodes, code, ensure_modules_loaded,
            [Modules]),
    ok.

```

Listing 3.12: Code of `proper:ensure_code_loaded/1`

3. After the workers have been spawned, the main process of the tool has to start aggregating the results sent back from each worker, which is done by calling

`proper:aggregate_imm_result/2`. This function relies on *pattern matching* the message passing specified in the workers to combine those individual results into one that has the format expected by the tool when reporting it.

4. Finally, the nodes are stopped when `proper:stop_nodes()` is called, which uses functions already defined in Erlang to stop each node.

### 3.3.4 Problems faced during the Implementation stage

We thought it would be of significance to discuss some of the issues and/or technical challenges we faced while implementing the design in the property-based testing tool, as some of them amounted for quite a lot of effort into finding their solution or fix and are worth a mention.

The biggest problem we had to face was properly setting up the dedicated node(s) where the workers are going to be spawned. When Erlang runs a program or its interactive shell is started, what is happening behind the scenes is that an Erlang node is booted up, although locally and hidden by default, which runs the ERTS. For an ERTS to work in a distributed fashion it first has to start a named node, as those are the only ones that can communicate with other nodes. Thankfully, as nodes are a key point in making distributed applications in Erlang, the programming language has built-in functions that help manage distributed applications (as shown in 3.11) and even modules in OTP to easily create and use *main/worker* architectures in our programs.

However, for one to be able to smoothly use nodes certain criteria has to be met:

1. The nodes cannot differ in their Erlang version, as the distribution mechanism is not backwards compatible. There is, however, an option that enables the compatibility mode so that nodes that are within two releases (e.g., Erlang/OTP 19 and Erlang/OTP 21) can connect and communicate.

2. As Erlang is a programming built to face and solve concurrent problems, although each node can have a different version of a module loaded, if they want to share functions from that module among themselves, they must all have the same version loaded.

For this thesis' work, the first condition was not an issue, as we were always using the same version of Erlang across all nodes and, given that this condition is well-known, users of the tool should not have any problem complying with it either.

The second condition, however, was a hardship we had to overcome due to two reasons: first, we had to discern which were the relevant modules (i.e., modules related to the application that has the properties we are about to test) that should be loaded across the nodes;

second, we had to load those modules in the proper fashion, as otherwise Erlang would not recognize their functions and would crash.

To solve the both issues, we implemented `proper:ensure_code_loaded/1` (3.12) that, together with `proper:maybe_load_binary/2` (3.13), finds all modules that have been loaded by PropEr (and therefore the ones we need) and loads them in the nodes.

```erlang
-spec maybe_load_binary([node()], module()) -> 'ok' | 'error'.
maybe_load_binary(Nodes, Module) ->
    %% We check whether the module was preloaded or cover_compiled.
    %% We ignore such modules.
    case code:is_loaded(Module) of
        {file, Loaded} when is_list(Loaded) ->
            case code:get_object_code(Module) of
                {Module, Binary, Filename} ->
                    %% Load the binary code of the module
                    %% in all the nodes.
                    _ = rpc:multicall(Nodes, code, load_binary,
                                [Module, Filename, Binary]),
                    ok;
                error -> error
            end;
        _ -> ok
    end.
```

Listing 3.13: Code of `proper:maybe_load_binary/2`

Finally, a small yet important issue we found in the Implementation stage was that, although the tool is capable of growing the size of the random generated input used when testing, it has no way of starting at a given size, meaning that we had to develop ourselves a way of doing so. We came up with a function that is returns the expected size for the *N*th test in an execution. That way, we were able to make the workers start at a given test number with the correct size.

### 3.3.5 Implementing the different strategies of test distribution

As we did previously mention in the Design section, we thought of four different ways, or strategies, of dividing the tests and its workload among the workers, and decided to implement each of them so that we could pick the best of the strategies after comparing their benchmarks.

Although all four strategies were implemented as described previously, we ran into a few issues with two of them. We will briefly cover each of the strategies, or distributions, below:

- Even strategy (3.1). This was the easiest to implement, as the only edge case to take into account is when the number of workers is odd.

- *N*th strategy (3.4). This strategy benefits from being an "embarrassingly parallel" one: if the tool has to run 100 tests, that means there is a sequence of test numbers from 0 to 99 to be executed. Hence, *N* workers simply have to select (or compute) every *N*th tests from that sequence.
- Batches strategy (3.3). While testing this implementation we found out that it had two main problems, both related to using a large numbers of workers: there was a problem of latency due to sending more messages than in the other strategies and with this strategy, the tool usually hit its limit of retries of a rejected test.
- Uneven strategy (3.2). Because of the lack of time, this strategy was sadly implemented in a rushed way and it was not be feasible to really use this strategy in later stages of this work.

As a result of this, the benchmarking comparison was done between the *even strategy* and the *Nth strategy*, leaving the rest to be revisited in the future.

Chapter 4

# Testing and benchmarking

In past chapters, we mentioned that one of the main motivations behind this thesis' work was to improve the times of a property-based testing tool by extending it with parallel and distributed testing. Throughout this chapter we will explain how we tested this revamping of the property-based testing tool and later measured its execution times, with properties from different relevant projects of the Erlang community, as otherwise we could not detect under which circumstances the speedups happened and if they did occur.

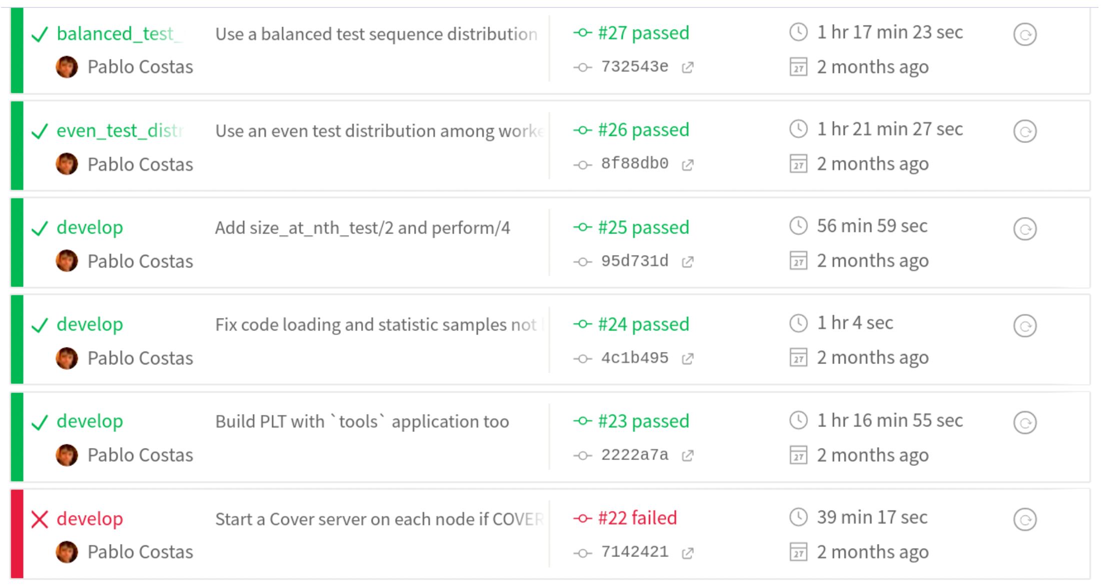
## 4.1 Testing the implementation

Due to the fact that we were working on an already established project, old enough to be considered pretty matured, and that we were expanding it with new ways of executing the tool, we had an extensive test suite to make use of. that would ensure that the tool did not break any of its existing features because of any of our modifications.

Thanks to that we were always sure whether, over the many iterations the implementation took place, that the changes done to the codebase did not break any of its existing features and functionalities or if it indeed break some along the way, in which case a fix would be made. This was easily checked, as the project used *Travis CI* to do *Continuous Integration*, and by simply enabling it in our fork we would benefit from it too. After each push to the repository, Travis would start an automated build to download the latest changes, build the project and run the tool's test suite (4.1).

Testing the parallel execution was as simple as running the test suite of the tool while using workers. Although our patched tool was tested with different values for the number of workers, the biggest indicator of it working properly was that with one (i.e., a worker does the job for the tool) and two workers (i.e., two workers share the work) the test suite passed.

Furthermore, as one of our objectives was to built the distributed execution upon the parallel one, as long as we tested the latter we could be sure the former worked.



| | | | | |
|---|---|---|---|---|
| ✓ balanced_test_<br>Pablo Costas | Use a balanced test sequence distribution | #27 passed<br>732543e | 1 hr 17 min 23 sec<br>2 months ago | |
| ✓ even_test_distr<br>Pablo Costas | Use an even test distribution among work | #26 passed<br>8f88db0 | 1 hr 21 min 27 sec<br>2 months ago | |
| ✓ develop<br>Pablo Costas | Add size_at_nth_test/2 and perform/4 | #25 passed<br>95d731d | 56 min 59 sec<br>2 months ago | |
| ✓ develop<br>Pablo Costas | Fix code loading and statistic samples not | #24 passed<br>4c1b495 | 1 hr 4 sec<br>2 months ago | |
| ✓ develop<br>Pablo Costas | Build PLT with `tools` application too | #23 passed<br>2222a7a | 1 hr 16 min 55 sec<br>2 months ago | |
| ✗ develop<br>Pablo Costas | Start a Cover server on each node if COVER | #22 failed<br>7142421 | 39 min 17 sec<br>2 months ago | |

Figure 4.1: Reports of *Travis CI* in the project

## 4.2 Benchmarking

In the next subsections we will first talk about the results of benchmarking the strategies to decide which one was a better fit to use through the rest of the work, secondly, about the those from benchmarking both the parallel and the distributed execution and, finally, about the conclusions we reached from examining those results.

However, before doing that, we think it is important to establish how we carried out said benchmarks and under which execution environments were they done.

The specs of the systems that were used to benchmark the paralle and distributed executions are the following:

- For the parallel benchmarks, we used a single machine from the Uppsala University. It has 128 GB of RAM, runs Debian 4.19 and has 64 cores with AMD Opteron(TM) Processor 6276 (2.3 GHz).

- For the distributed benchmarks, we used a cluster from the CITIC. Each of the 10 machines from the cluster has 8 GB of RAM, runs Ubuntu 18.04.1 LTS and has 4 cores with Intel Xeon E5-2650 v4 (2.20GHz).

- In both environments, we decided to use the same Erlang/OTP version[1], in order to

[1]The sole exception to this was when running properties from tests in Erlang/OTP itself, as we first had to compile the programming language from source and use that version to run the tests. In those cases, the version used was *Erlang/OTP 24*.

reduce uncertainty that changes within the implementation of the language could produce. The version we used was *Erlang/OTP 21.2.6*.

In addition, we thought it would be more relevant to benchmark specific properties than whole modules, as with the former it is possible to cherry-pick cases that were of interest, either because of their time complexity or because of their computation cost; whereas with the latter only the total speedup could be noted.

Nonetheless, both stateful and stateless properties have been benchmarked, as we were interested in observing the cost of having to isolate the workers (when testing the former kind of properties) or not (with the latter kind of properties).

Furthermore, as we knew from the beginning of the existence of the overhead derived from having to start dedicated nodes, we were able to set up a modified environment that took into account it from the get-go. In this environment we always first performed a light round of one standard execution of the tool (i.e., 100 tests) at each number of workers we wanted to test with (from 0 to 64 in the parallel machine and from 0 to 8 in the distributed cluster) so that the nodes would have the code loaded beforehand. Nevertheless, we still measured the time it took to handle the logic of the nodes, as although the overhead was reduced it was still of significance to note.

### 4.2.1 Picking a strategy to split the tests

As previously stated, only two (even distribution, 3.1, and *N*th sequence distribution, 3.4) of the total of the four strategies were used and had their benchmarks compared with. In addition, we picked the best strategy based on the parallel execution benchmarks, due to the distributed execution being built upon the parallel one.s

The properties we decided to benchmark with will always be the same throughout the following sections, and they all come from either notable or relevant projects of the Erlang community. Every benchmark that will be shown has been obtained the same way: first, we ran the light round to initialize the nodes and have the relevant code loaded in them, as we previously mentioned; then, we ran for every combination of number of tests (usually from 100 to 1000000) and number of workers shown a total of ten executions and found the median value of the execution times at each pair of values.

The benchmarks will be grouped by the project they are originated from and its type of property will be noted (*stateful* vs. *stateless*), as its an important factor. The original implementation of PropEr (in other words, PropEr with zero workers being used) will considered the baseline for the benchmarks. Furthermore, some time disparities will be seen in those baselines when comparing the time of base PropEr between strategies; this is due to the machine where the parallel benchmarks were done not having all of its resources available for us as it was being used by other researchers of Uppsala University. All measurements are in seconds.

### Cowlib

*Cowlib* [22] is a support library for manipulating web protocols, optimized for completeness rather than speed, which provides functionalities such as parsing and building messages for multiple Web protocols (i.e., HTTP/1.1, HTTP/2, Websockets).

We used from this project an stateful property that checks whether it is possible for a binary to be encoded and decoded with *Huffman Coding* [23]. The benchmark of the property, `cow_hpack:prop_str_huffman()`, is shown in 4.1.

| NumTests | NumWorkers | Median execution time (even strat.) | Median execution (*N*th strat.) |
|---|---|---|---|
| 100 | 0 | 0.0043085 | 0.0043485 |
| 100 | 1 | 1.1962975 | 1.1904545 |
| 100 | 2 | 1.215874 | 1.190575 |
| 100 | 4 | 1.2535485 | 1.2407425 |
| 100 | 8 | 1.2596395 | 1.2637315 |
| 100 | 16 | 1.312265 | 1.3470175 |
| 100 | 32 | 1.434364 | 1.4816925 |
| 100 | 64 | 0.7276165 | 0.75407 |
| 1000 | 0 | 0.04491 | 0.0452655 |
| 1000 | 1 | 0.218274 | 0.20159 |
| 1000 | 2 | 0.1971315 | 0.2035755 |
| 1000 | 4 | 0.478623 | 0.1987635 |
| 1000 | 8 | 0.3809485 | 0.244365 |
| 1000 | 16 | 0.4095365 | 0.3248635 |
| 1000 | 32 | 0.5010915 | 0.441245 |
| 1000 | 64 | 0.765738 | 0.761062 |
| 10000 | 0 | 0.449596 | 0.4540915 |
| 10000 | 1 | 0.8153415 | 0.8395805 |
| 10000 | 2 | 0.4802005 | 0.513207 |
| 10000 | 4 | 2.9868485 | 0.3788485 |
| 10000 | 8 | 1.626953 | 0.310093 |
| 10000 | 16 | 1.249897 | 0.3778095 |
| 10000 | 32 | 1.0791045 | 0.5139375 |
| 10000 | 64 | 1.158012 | 0.776513 |

| | | | |
|---|---|---|---|
| 100000 | 0 | 4.4230875 | 4.5214735 |
| 100000 | 1 | 4.4421215 | 4.718555 |
| 100000 | 2 | 2.9091925 | 2.4595815 |
| 100000 | 4 | 27.4280975 | 1.3656275 |
| 100000 | 8 | 13.594567 | 0.83403 |
| 100000 | 16 | 7.390055 | 0.741856 |
| 100000 | 32 | 5.200581 | 0.7272785 |
| 100000 | 64 | 3.9837955 | 0.9227425 |
| 1000000 | 0 | 42.527384 | 45.471653 |
| 1000000 | 1 | 44.112267 | 51.263478 |
| 1000000 | 2 | 27.835295 | 25.8550235 |
| 1000000 | 4 | 17.0330655 | 13.2980945 |
| 1000000 | 8 | 9.190302 | 6.840301 |
| 1000000 | 16 | 4.89988 | 3.893725 |
| 1000000 | 32 | 3.160499 | 3.338172 |
| 1000000 | 64 | 2.662097 | 2.816228 |

Table 4.1: Cowlib: `prop_str_huffman()` parallel benchmarks

### Kazoo

*Kazoo* [24] is one of the most powerful open-source *VoIP* available, designed to provide robust telecommunication services.

Despite the fact that this project has multiple properties written for PropEr, we could not make use of most of them for benchmarking due to how the project itself is structured, making it really hard to load all the relevant code in the dedicated nodes, and because of some of those properties testing the architecture of the project (i.e., they test how the software behaves in regards to the nodes they use), meaning that those properties could not be used either. Nonetheless, we found an interesting stateless property, `knm_converters_tests:prop_normalize()`, that checks whether a pair of telephone numbers within a range can be normalized or not (4.2).

| NumTests | NumWorkers | Median execution time (even strat.) | Median execution (*N*th strat.) |
|---|---|---|---|
| 100 | 0 | 1.2298875 | 1.316535 |
| 100 | 1 | 1.2063575 | 1.355374 |
| 100 | 2 | 1.237179 | 1.0168675 |
| 100 | 4 | 1.2145395 | 1.047321 |

| | | | |
|---|---|---|---|
| 100 | 8 | 1.2179455 | 1.060538 |
| 100 | 16 | 1.2148005 | 1.1055665 |
| 100 | 32 | 1.2582235 | 1.2755245 |
| 100 | 64 | 1.3320655 | 1.085785 |
| 1000 | 0 | 11.237414 | 13.3373095 |
| 1000 | 1 | 11.9686205 | 9.57503 |
| 1000 | 2 | 11.985622 | 9.8005445 |
| 1000 | 4 | 12.2558495 | 10.143767 |
| 1000 | 8 | 12.394612 | 10.251728 |
| 1000 | 16 | 12.268519 | 9.8840315 |
| 1000 | 32 | 12.509876 | 9.34092 |
| 1000 | 64 | 8.874254 | 10.6230795 |
| 10000 | 0 | 108.6293275 | 133.44307 |
| 10000 | 1 | 105.9389425 | 101.571297 |
| 10000 | 2 | 120.5917215 | 100.7109155 |
| 10000 | 4 | 119.7413025 | 98.762332 |
| 10000 | 8 | 122.252236 | 98.6406455 |
| 10000 | 16 | 121.5295695 | 98.6961375 |
| 10000 | 32 | 122.8359415 | 123.5200135 |
| 10000 | 64 | 81.589644 | 108.9131675 |

Table 4.2: Kazoo: `prop_normalize()` parallel benchmarks

### Zotonic

*Zotonic* [25] is an open-source Erlang Web Framework and content management system, capable of providing flexible high speed applications.

It is a project made of various components, spawning over many repositories. The one of interest for our work, however, is its standard library, as it has properties to check whether it is capable of properly sanitize strings in UTF8. Again, we picked the one that took the most to execute, `z_string_sanitize_utf8_test:prop_s_utf8a()` and show only that one (4.3).

| NumTests | NumWorkers | Median execution time (even strat.) | Median execution (*N*th strat.) |
|---|---|---|---|
| 100 | 0 | 0.041248 | 0.0396525 |
| 100 | 1 | 0.998514 | 0.962967 |
| 100 | 2 | 0.995777 | 0.9567575 |

| | | | |
|---|---|---|---|
| 100 | 4 | 1.004187 | 0.959927 |
| 100 | 8 | 1.0137185 | 0.9836805 |
| 100 | 16 | 1.06197 | 1.0513905 |
| 100 | 32 | 1.2017605 | 1.1950655 |
| 100 | 64 | 0.6283005 | 0.644169 |
| 1000 | 0 | 0.414754 | 0.4121415 |
| 1000 | 1 | 0.569497 | 0.5867375 |
| 1000 | 2 | 0.501841 | 0.3746395 |
| 1000 | 4 | 0.3786765 | 0.27011 |
| 1000 | 8 | 0.3194805 | 0.2498845 |
| 1000 | 16 | 0.3214465 | 0.287736 |
| 1000 | 32 | 0.4200385 | 0.4035595 |
| 1000 | 64 | 0.6683985 | 0.651667 |
| 10000 | 0 | 4.309788 | 4.2923995 |
| 10000 | 1 | 10.36273 | 10.3820285 |
| 10000 | 2 | 5.681023 | 5.3670255 |
| 10000 | 4 | 2.9683095 | 2.7579885 |
| 10000 | 8 | 1.6316345 | 1.527821 |
| 10000 | 16 | 1.0153945 | 1.0510645 |
| 10000 | 32 | 0.9882455 | 0.957311 |
| 10000 | 64 | 1.0766025 | 1.0587965 |

Table 4.3: Zotonic: `prop_s_utf8a()` parallel benchmarks

### Diffy

*Diffy* [26] is a project which implements in Erlang the *Diff, Match and Patch* [27] library and is used by bigger and well-known projects, such as Zotonic itself.

Even though it is a small project, it has a few properties that test its implementation. However, we decided to look for the property that took the longest to run and use that one to benchmark for this report; that property, `diffy_tests:prop_inner_diff()`, is an stateless one that checks whether the different algorithm is capable of properly *diffing* the innermost part of an HTML document after having it modified (4.4).

| NumTests | NumWorkers | Median execution time (even strat.) | Median execution ($N$th strat.) |
|---|---|---|---|
| 1000 | 0 | 3.2038845 | 3.193476 |
| 1000 | 1 | 3.313632 | 3.378731 |

| | | | |
|---|---|---|---|
| 1000 | 2 | 1.7992555 | 1.7900135 |
| 1000 | 4 | 0.9959045 | 1.0011105 |
| 1000 | 8 | 0.6224575 | 0.634341 |
| 1000 | 16 | 0.4903785 | 0.5155705 |
| 1000 | 32 | 0.4742175 | 0.4984285 |
| 1000 | 64 | 0.591725 | 0.607396 |
| 10000 | 0 | 31.958289 | 31.9596165 |
| 10000 | 1 | 32.2683375 | 32.513183 |
| 10000 | 2 | 16.86818 | 16.7938955 |
| 10000 | 4 | 8.861493 | 8.8009345 |
| 10000 | 8 | 4.989744 | 4.8861055 |
| 10000 | 16 | 3.462057 | 3.3754105 |
| 10000 | 32 | 3.180591 | 2.973362 |
| 10000 | 64 | 3.0812605 | 2.8868445 |
| 100000 | 0 | 313.1051505 | 318.925156 |
| 100000 | 1 | 320.2353155 | 319.443231 |
| 100000 | 2 | 165.54899 | 162.098418 |
| 100000 | 4 | 86.138611 | 84.323321 |
| 100000 | 8 | 47.026769 | 46.433824 |
| 100000 | 16 | 31.986817 | 31.893608 |
| 100000 | 32 | 27.8032935 | 27.516021 |
| 100000 | 64 | 26.035407 | 25.8266085 |

Table 4.4: Diffy: `prop_inner_diff()` parallel benchmarks

### Best strategy

Although we did more benchmarks, they have been left out of the report as otherwise we would have too many pages wasted on only showing benchmarks, whereas it would be more relevant to note what can be learnt from those that have been shown.

First of all, parallelization will not always bring a speedup, as seen in both of the strategies at test runs that already took a short moment to finish (usually when using a small number of tests). However, for test runs that would normally (i.e., with the original implementation of PropEr) take five seconds or more, the benefits from parallelizing the tests can be perceived.

Whilst this is true for both stateless and stateful properties, this time barrier is lower on the former type of properties than in the latter type, as those have to isolate the workers in their own dedicated nodes to deal with stateful systems, instead of simply spawning more workers in the existing node to increment the parallelism.

All in all, between the two strategies that have been benchmarked (even distribution and *N*th sequence distribution), despite the fact that both have some hiccups at some number of workers from time to time (probably related to the latency produced from having to send too many messages), we think that the latter is a better fit for the rest of the benchmarks than the former, as it handles better spreading the workload among the workers.

### 4.2.2 Parallel execution benchmarking

Due to time restraints at this point of our work, and because of the general lack of both projects using PropEr and properties that were of interest those projects we found, we will hereafter only show benchmarks of properties from different test suites of Erlang/OTP itself, as we consider it the biggest existing open-source project actually doing Property-Based Testing in Erlang. Erlang/OTP has in some of its applications, or components, a test suite of property-based tests, of which a few can be run with the tool that this work extended, PropEr, whereas others can only be run with the paid license of EQC.

Out of the total of six applications (`compiler`, `crypto`, `ftp`, `ssh`, `ssl`, `stdlib`) that had property-based tests, just four of them (`compiler`, `crypto`, `ssh`, `stdlib`) had tests that could be run and used to benchmark.

#### compiler

This compiler of Erlang/OTP itself has in its test suite a single stateful property, `compile_prop:compile()`, to checks whether it is possible to compile generated Erlang abstract code with no errors or not (4.5) and a series of properties to check whether some logical properties are always held for the existing types in the BEAM or not. Again, we picked the most time consuming property, `beam_types_prop:associativity()`, from the series of properties to show in the report (4.6).

| Module:Property | NumTests | NumWorkers | Median execution |
|---|---|---|---|
| compile_prop:compile() | 1000 | 0 | 9.1924835 |
| compile_prop:compile() | 1000 | 1 | 9.238612 |
| compile_prop:compile() | 1000 | 2 | 4.8878485 |
| compile_prop:compile() | 1000 | 4 | 2.659909 |
| compile_prop:compile() | 1000 | 8 | 1.580098 |
| compile_prop:compile() | 1000 | 16 | 1.0493715 |
| compile_prop:compile() | 1000 | 32 | 1.000874 |
| compile_prop:compile() | 1000 | 64 | 0.780156 |

| | | | |
|---|---|---|---|
| compile_prop:compile() | 10000 | 0 | 91.5118675 |
| compile_prop:compile() | 10000 | 1 | 89.170889 |
| compile_prop:compile() | 10000 | 2 | 45.227779 |
| compile_prop:compile() | 10000 | 4 | 23.1776705 |
| compile_prop:compile() | 10000 | 8 | 11.8982525 |
| compile_prop:compile() | 10000 | 16 | 6.2006945 |
| compile_prop:compile() | 10000 | 32 | 4.612698 |
| compile_prop:compile() | 10000 | 64 | 3.5742405 |

Table 4.5: Erlang/OTP: `compiler`, `compile()` parallel benchmarks

| Module:Property | NumTests | NumWorkers | Median execution |
|---|---|---|---|
| beam_types_prop:associativity() | 5000 | 0 | 13.022068 |
| beam_types_prop:associativity() | 5000 | 1 | 13.2373845 |
| beam_types_prop:associativity() | 5000 | 2 | 6.6338075 |
| beam_types_prop:associativity() | 5000 | 4 | 3.4362205 |
| beam_types_prop:associativity() | 5000 | 8 | 1.870995 |
| beam_types_prop:associativity() | 5000 | 16 | 1.122578 |
| beam_types_prop:associativity() | 5000 | 32 | 0.8995795 |
| beam_types_prop:associativity() | 5000 | 64 | 0.9441975 |

Table 4.6: Erlang/OTP: `compiler`, `associativity()` parallel benchmarks

### crypto

As Erlang/OTP supports multiple cryptography ciphers and has support for the *OpenSSL cryptolib*, and more importantly recently changed its API to deal with changes in the aforementioned cryptolib, we think the properties from this application might be interesting to benchmark. Both properties, `crypto_ng_api:prop__crypto_one_time()` (4.7) and `crypto_ng_api:prop__crypto_init_update_final()` (4.8), are stateful and test the new implementation of the API.

| Module:Property | NumTests | NumWorkers | Median execution |
|---|---|---|---|
| crypto_ng_api:prop__crypto_one_time() | 10000 | 0 | 110.628972 |
| crypto_ng_api:prop__crypto_one_time() | 10000 | 1 | 145.8276025 |
| crypto_ng_api:prop__crypto_one_time() | 10000 | 2 | 81.5819725 |
| crypto_ng_api:prop__crypto_one_time() | 10000 | 4 | 41.218044 |

| | | | |
|---|---|---|---|
| crypto_ng_api:prop__crypto_one_time() | 10000 | 8 | 21.464494 |
| crypto_ng_api:prop__crypto_one_time() | 10000 | 16 | 11.1920615 |
| crypto_ng_api:prop__crypto_one_time() | 10000 | 32 | 9.095284 |
| crypto_ng_api:prop__crypto_one_time() | 10000 | 64 | 7.3540885 |

Table 4.7: Erlang/OTP: `crypto`, `prop__crypto_one_time()` parallel benchmarks

| Module:Property | NumTests | NumWorkers | Median execution |
|---|---|---|---|
| crypto_ng_api:prop__crypto_init_update_final() | 10000 | 0 | 110.2280835 |
| crypto_ng_api:prop__crypto_init_update_final() | 10000 | 1 | 167.8912475 |
| crypto_ng_api:prop__crypto_init_update_final() | 10000 | 2 | 83.8977575 |
| crypto_ng_api:prop__crypto_init_update_final() | 10000 | 4 | 42.5350435 |
| crypto_ng_api:prop__crypto_init_update_final() | 10000 | 8 | 22.124254 |
| crypto_ng_api:prop__crypto_init_update_final() | 10000 | 16 | 11.623337 |
| crypto_ng_api:prop__crypto_init_update_final() | 10000 | 32 | 8.7799665 |
| crypto_ng_api:prop__crypto_init_update_final() | 10000 | 64 | 7.294942 |

Table 4.8: Erlang/OTP: `crypto`, `prop__crypto_init_update_final()` parallel benchmarks

### ssh

At the risk of repeating ourselves, Erlang/OTP is a functional programming language built to solve concurrency related issues while providing a robust platform to develop and work on. Since most Erlang applications are distributed ones and those run on nodes, either on one machine or server, or in multiple ones, it has a built-in ssh application included to help start clients or daemons and run commands in a shell on a remote server with the language itself.

This component of Erlang/OTP has only two properties to check whether it is possible to properly encode, or first decode and then encode, a ssh message. Once acain, in this report we will include only the latter of the properties, `ssh_eqc_encode_decode:prop_ssh_decode_encode()`, as it was the most time consuming of the two (although in this case it was pretty fast by itself and we had to scale the number of tests up to get proper benchmarks).

| Module:Property | NumTests | NumWorkers | Median execution |
|---|---|---|---|
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100 | 0 | 0.0888925 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100 | 1 | 0.430613 |

| | | | |
|---|---|---|---|
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100 | 2 | 0.3949945 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100 | 4 | 0.3877615 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100 | 8 | 0.416598 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100 | 16 | 0.44926 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100 | 32 | 0.5559945 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100 | 64 | 0.4801235 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 1000 | 0 | 0.848077 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 1000 | 1 | 0.9098715 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 1000 | 2 | 0.510886 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 1000 | 4 | 0.319352 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 1000 | 8 | 0.2444555 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 1000 | 16 | 0.2411055 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 1000 | 32 | 0.323419 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 1000 | 64 | 0.5012535 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 10000 | 0 | 8.4215655 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 10000 | 1 | 8.4747755 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 10000 | 2 | 4.2756365 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 10000 | 4 | 2.2221815 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 10000 | 8 | 1.2049885 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 10000 | 16 | 0.8060915 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 10000 | 32 | 0.7260185 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 10000 | 64 | 0.8326165 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100000 | 0 | 84.384597 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100000 | 1 | 83.621504 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100000 | 2 | 41.946111 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100000 | 4 | 21.035486 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100000 | 8 | 10.8567325 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100000 | 16 | 5.6160635 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100000 | 32 | 4.5863795 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100000 | 64 | 3.750699 |

Table 4.9: Erlang/OTP: `ssh`, `prop_ssh_decode_encode()` parallel benchmarks

#### stdlib

In Erlang, although it is possible and there is an official API to document modules and functions with notations, said documentation could only be checked in the form of HTML documents. Recently, however, Erlang/OTP has been changed to include the documentation of a module in its compiled file in a standardized way, so that this information can be retrieved and displayed in more ways (i.e., from the interactive shell). As a result of that, the standard library added to its test suite a property, `shell_docs_prop:prop_render()`, that checks whether it is possible to render, validate and normalize the documentation of a module with the new format.

| Module:Property | NumTests | NumWorkers | Median execution |
|---|---|---|---|
| shell_docs_prop:prop_render() | 10000 | 0 | 60.293814 |
| shell_docs_prop:prop_render() | 10000 | 1 | 104.6819455 |
| shell_docs_prop:prop_render() | 10000 | 2 | 53.814188 |
| shell_docs_prop:prop_render() | 10000 | 4 | 29.4466085 |
| shell_docs_prop:prop_render() | 10000 | 8 | 18.669691 |
| shell_docs_prop:prop_render() | 10000 | 16 | 9.155277 |
| shell_docs_prop:prop_render() | 10000 | 32 | 5.844506 |
| shell_docs_prop:prop_render() | 10000 | 64 | 4.1181045 |

Table 4.10: Erlang/OTP: `stdlib`, `prop_render()` parallel benchmarks

### 4.2.3 Distributed execution benchmarking

Because we will be benchmarking the same properties as in the previous section, where we showed the parallel benchmarks (4.2.2), we will simply show hereafter the tables of benchmarks and we will talk about the results of both benchmarks in the Conclusions section of this chapter (4.3).

Also, we consider important to note that the following tables will have less workers per table (i.e., less rows) than in the previous section, as sadly we could only run in the cluster the tool with up to 16 workers, so the rows related to 32 and 64 workers have been left out.

#### compiler

| Module:Property | NumTests | NumWorkers | Median execution |
|---|---|---|---|
| compile_prop:compile() | 1000 | 0 | 3.531327 |
| compile_prop:compile() | 1000 | 1 | 3.550182 |

| | | | |
|---|---|---|---|
| compile_prop:compile() | 1000 | 2 | 1.884106 |
| compile_prop:compile() | 1000 | 4 | 1.0266975 |
| compile_prop:compile() | 1000 | 8 | 0.6090585 |
| compile_prop:compile() | 1000 | 16 | 0.4183475 |
| compile_prop:compile() | 10000 | 0 | 35.642739 |
| compile_prop:compile() | 10000 | 1 | 35.058796 |
| compile_prop:compile() | 10000 | 2 | 18.3370905 |
| compile_prop:compile() | 10000 | 4 | 9.2465115 |
| compile_prop:compile() | 10000 | 8 | 4.835463 |
| compile_prop:compile() | 10000 | 16 | 2.492885 |

Table 4.11: Erlang/OTP: `compiler`, `compile()` distributed benchmarks

| Module:Property | NumTests | NumWorkers | Median execution |
|---|---|---|---|
| beam_types_prop:associativity() | 5000 | 0 | 4.3008585 |
| beam_types_prop:associativity() | 5000 | 1 | 4.4070315 |
| beam_types_prop:associativity() | 5000 | 2 | 2.319693 |
| beam_types_prop:associativity() | 5000 | 4 | 1.2771485 |
| beam_types_prop:associativity() | 5000 | 8 | 0.7332075 |
| beam_types_prop:associativity() | 5000 | 16 | 0.4896175 |

Table 4.12: Erlang/OTP: `compiler`, `associativity()` distributed benchmarks

**crypto**

| Module:Property | NumTests | NumWorkers | Median execution |
|---|---|---|---|
| crypto_ng_api:prop__crypto_one_time() | 10000 | 0 | 29.0995205 |
| crypto_ng_api:prop__crypto_one_time() | 10000 | 1 | 40.7774245 |
| crypto_ng_api:prop__crypto_one_time() | 10000 | 2 | 21.125296 |
| crypto_ng_api:prop__crypto_one_time() | 10000 | 4 | 10.7539585 |
| crypto_ng_api:prop__crypto_one_time() | 10000 | 8 | 5.7919015 |
| crypto_ng_api:prop__crypto_one_time() | 10000 | 16 | 3.08204 |

Table 4.13: Erlang/OTP: `crypto`, `prop__crypto_one_time()` distributed benchmarks

| Module:Property | NumTests | NumWorkers | Median execution |
|---|---|---|---|
| crypto_ng_api:prop__crypto_init_update_final() | 10000 | 0 | 29.67034 |
| crypto_ng_api:prop__crypto_init_update_final() | 10000 | 1 | 41.0459865 |
| crypto_ng_api:prop__crypto_init_update_final() | 10000 | 2 | 21.029765 |
| crypto_ng_api:prop__crypto_init_update_final() | 10000 | 4 | 10.8909075 |
| crypto_ng_api:prop__crypto_init_update_final() | 10000 | 8 | 5.779673 |
| crypto_ng_api:prop__crypto_init_update_final() | 10000 | 16 | 3.072786 |

Table 4.14: Erlang/OTP: `crypto`, `prop__crypto_init_update_final()` distributed benchmarks

**ssh**

| Module:Property | NumTests | NumWorkers | Median execution |
|---|---|---|---|
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100 | 0 | 0.024615 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100 | 1 | 0.1149505 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100 | 2 | 0.111211 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100 | 4 | 0.108683 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100 | 8 | 0.115139 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100 | 16 | 0.1590255 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 1000 | 0 | 0.251863 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 1000 | 1 | 0.3445005 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 1000 | 2 | 0.2239095 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 1000 | 4 | 0.169825 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 1000 | 8 | 0.1619355 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 1000 | 16 | 0.1821265 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 10000 | 0 | 2.453757 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 10000 | 1 | 2.5533595 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 10000 | 2 | 1.3547285 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 10000 | 4 | 0.7534805 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 10000 | 8 | 0.463339 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 10000 | 16 | 0.3546005 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100000 | 0 | 24.4840895 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100000 | 1 | 24.416444 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100000 | 2 | 12.6548485 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100000 | 4 | 6.49339 |

| | | | |
|---|---|---|---|
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100000 | 8 | 3.4009055 |
| ssh_eqc_encode_decode:prop_ssh_decode_encode() | 100000 | 16 | 1.808103 |

Table 4.15: Erlang/OTP: `ssh`, `prop_ssh_decode_encode()` distributed benchmarks

**stdlib**

| Module:Property | NumTests | NumWorkers | Median execution |
|---|---|---|---|
| shell_docs_prop:prop_render() | 10000 | 0 | 18.6674395 |
| shell_docs_prop:prop_render() | 10000 | 1 | 28.6606325 |
| shell_docs_prop:prop_render() | 10000 | 2 | 15.1546485 |
| shell_docs_prop:prop_render() | 10000 | 4 | 8.2777155 |
| shell_docs_prop:prop_render() | 10000 | 8 | 4.481394 |
| shell_docs_prop:prop_render() | 10000 | 16 | 3.5812895 |

Table 4.16: Erlang/OTP: `stdlib`, `prop_render()` distributed benchmarks

## 4.3 Conclusions

There are a few conclusions to draw from looking at both benchmarks and taking into account the cost of implementing each new way of execution.

To start with, we want to state that although the reader might feel inclined to compare each benchmark against the other, that should not be done as the systems used to benchmark were radically different from each other. Given that the machine where the parallel benchmarks were done was always being used by more researchers, it is normal for its baseline results (sequential PropEr) to be slower when running a certain number of tests compared with those results from the baseline of the distributed benchmarks, where the machine was always completely free of any other program consuming resources. In short, both settings are valid and we will be simply commenting on the results we obtained from each.

Secondly, as we will be showing diagrams with the speedup obtained throughout the different properties we benchmarked, we wanted to explain what that value can mean:

- If the speedup value obtained is between 0 and 1, the parallelization did not help to shorten the execution times, meaning that the program runs in about the same time than its sequential version.

- If the speedup value, however, is between 0 and *N*, with *N* being the number CPUs used, the parallel program runs faster than its sequential counterpart. Ideally we would want to have a speedup value of N, as that would mean that the work has been evenly distributed among the CPUs.

- Finally, if the speedup value happens to be bigger than *N*, which can happen in rare situations, it is a *superlinear speedup* and it means that program has been sped up by more than the rise of CPUs.

Both settings did show the advantages of turning the property-based testing tool into a concurrent one. However, the benefits are greater in the distributed execution than in the parallel execution because we have a bigger total number of resources for the program to make use of (in other words, more machines instead of one) and there is no machine handling that many nodes at the same time; whereas in the parallel version, the more workers you use the more stress you put in your single machine.

Furthermore, in both settings we can notice that running a single worker might be worse (i.e., a spike can happen) than running the sequential tool, as there is an extra latency produced from handling the workers and nodes, and even more so when testing stateful properties.

Finally, as seen in the tables throughout the benchmarks, concurrency does indeed speed up the program, but first the tests to run should take a bit of time for those benefits to appear. In other words, if your property is being tested 100 times and it takes 1 second to do so, it is really hard to shorten such an execution time; whereas if your running those 100 tests takes 4-5 seconds, at some point of increasing the number of workers, the testing will be speeded up.

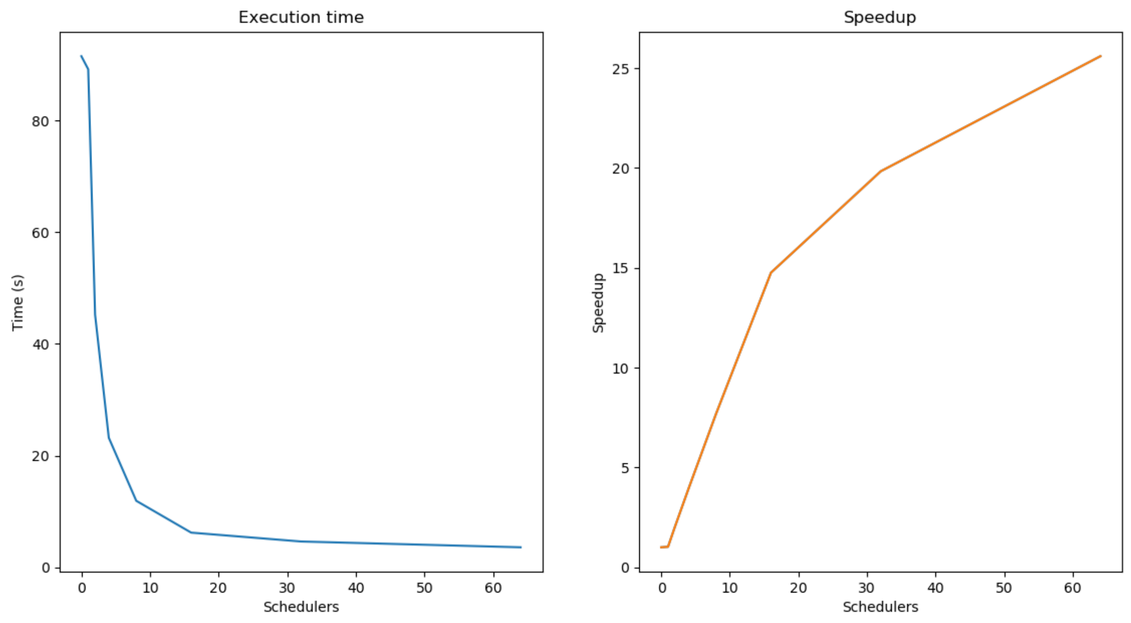


Figure 4.2: Parallel execution

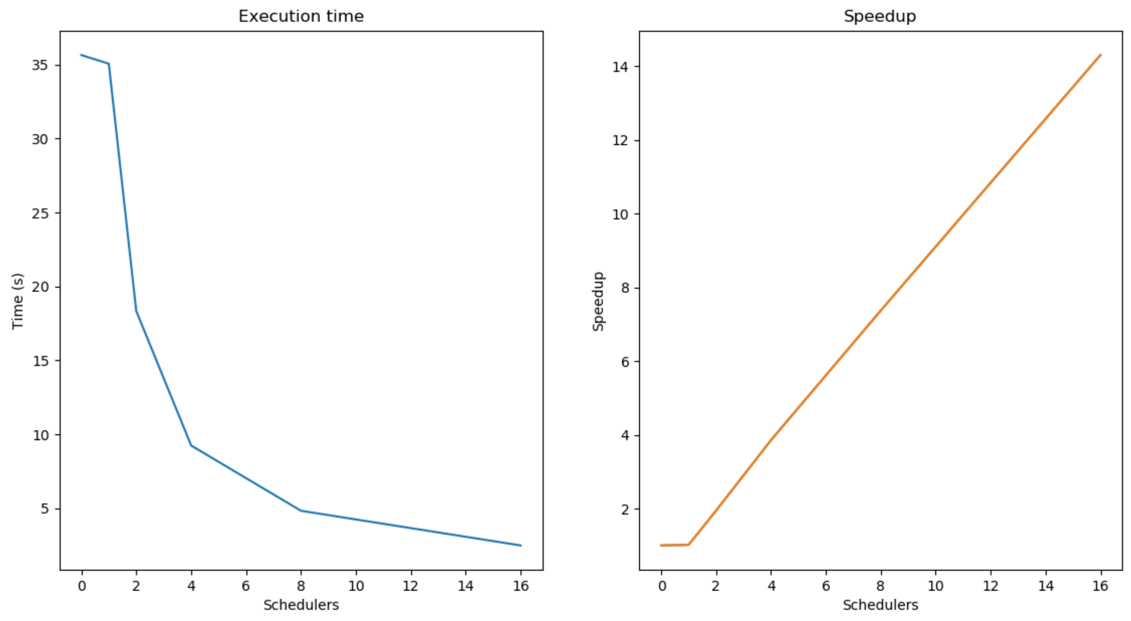


Figure 4.3: Distributed execution

Figure 4.4: Results of `compile_prop:compile()`

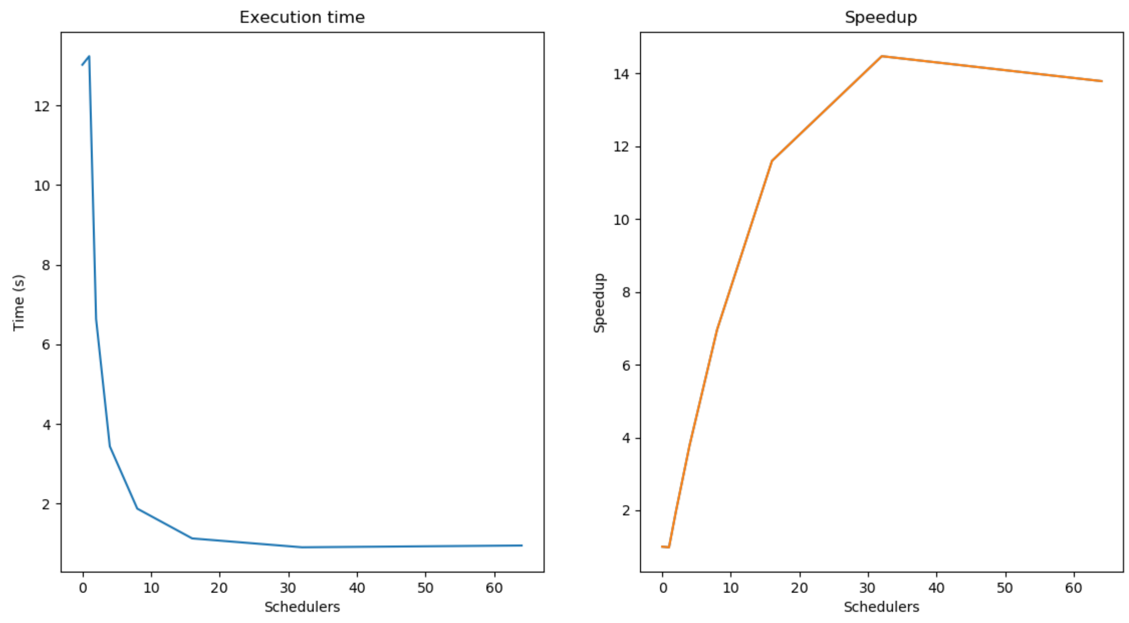


Figure 4.5: Parallel execution

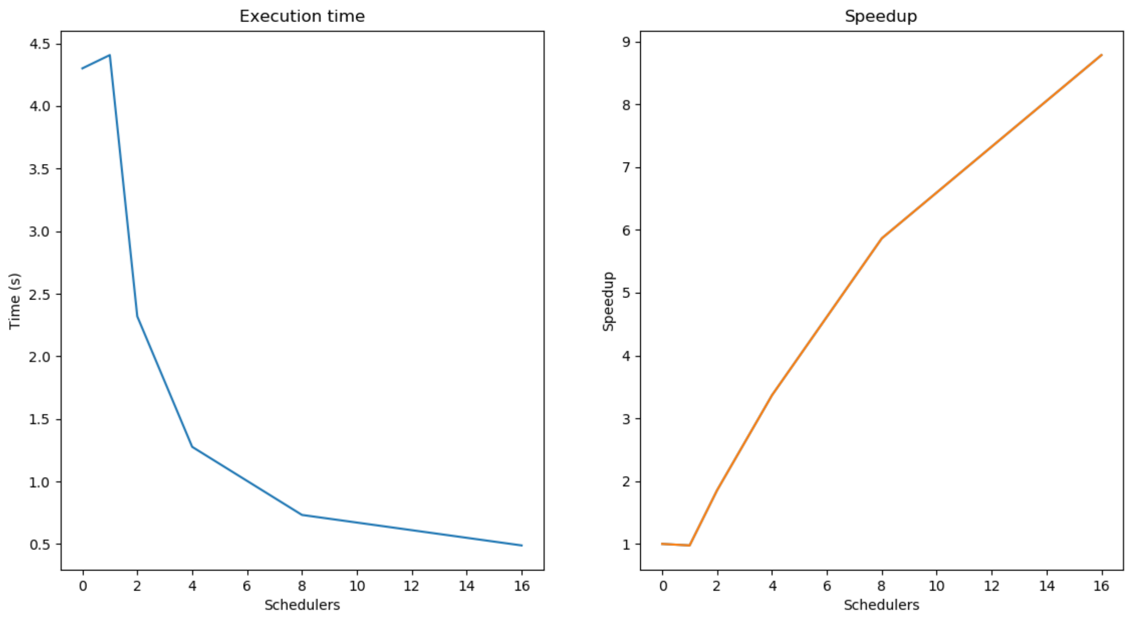


Figure 4.6: Distributed execution

Figure 4.7: Results of `beam_types_prop:associativity()`

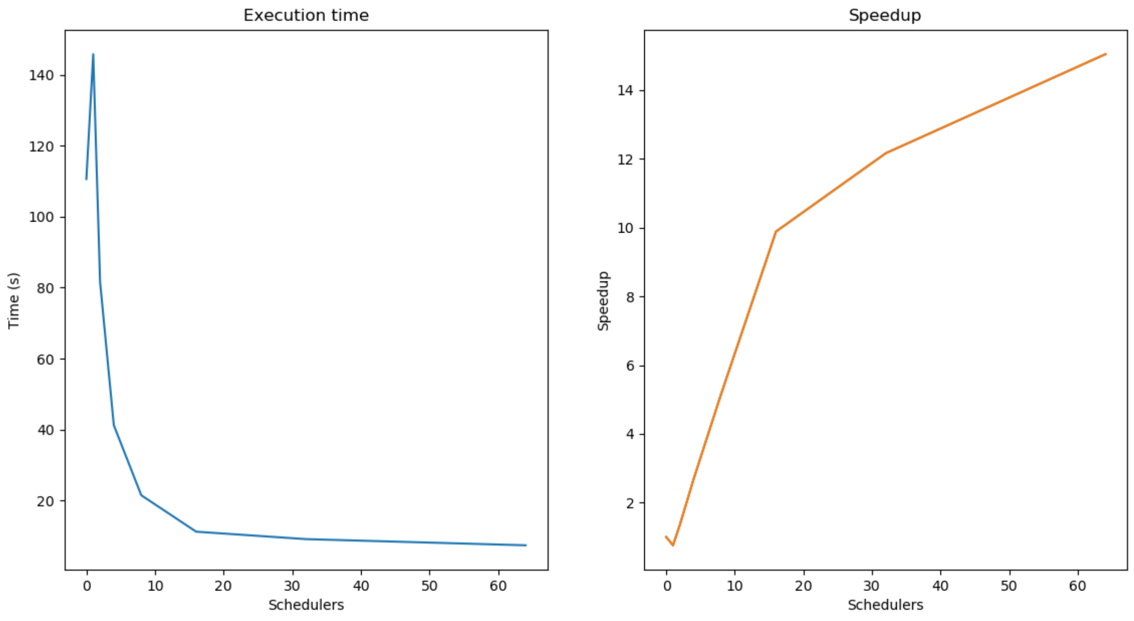


Figure 4.8: Parallel execution

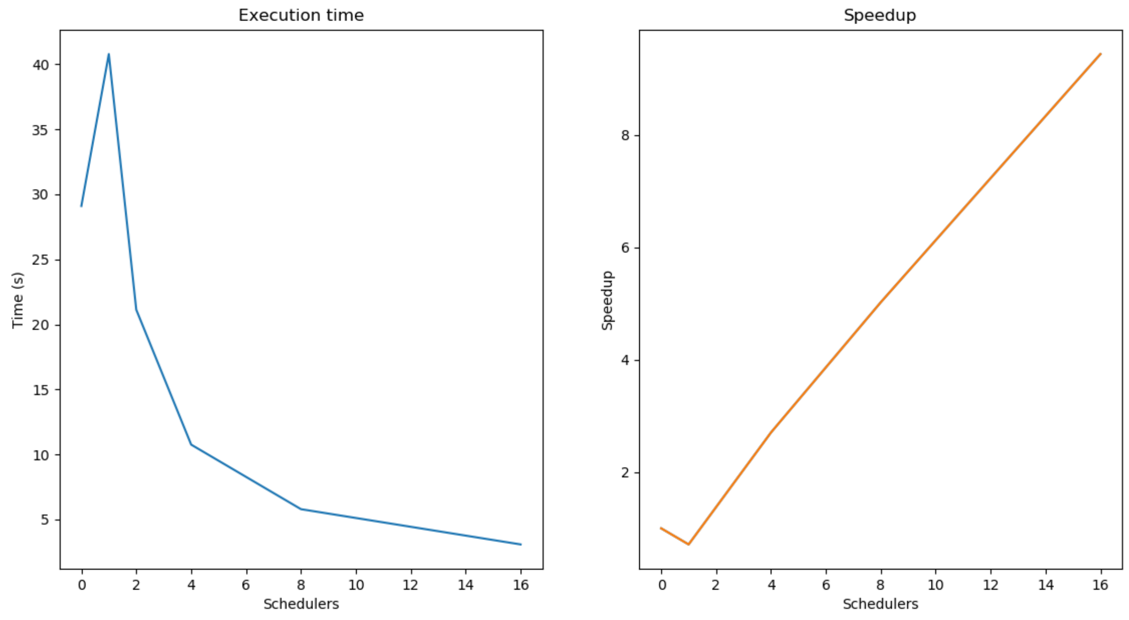


Figure 4.9: Distributed execution

Figure 4.10: Results of `crypto_ng_api:prop__crypto_one_time()`

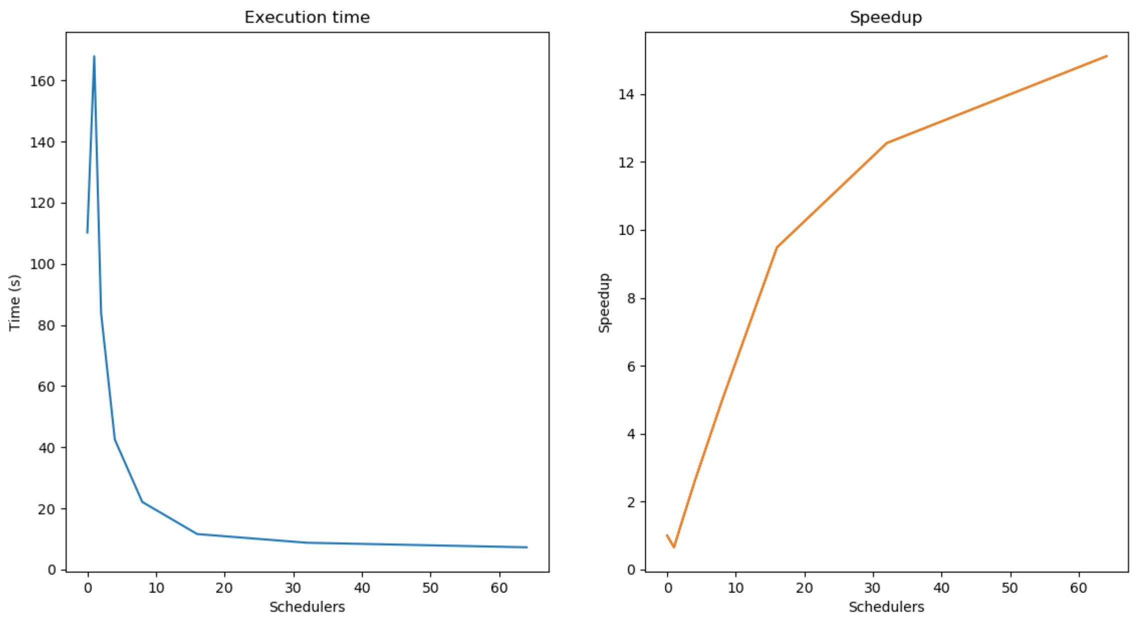


Figure 4.11: Parallel execution

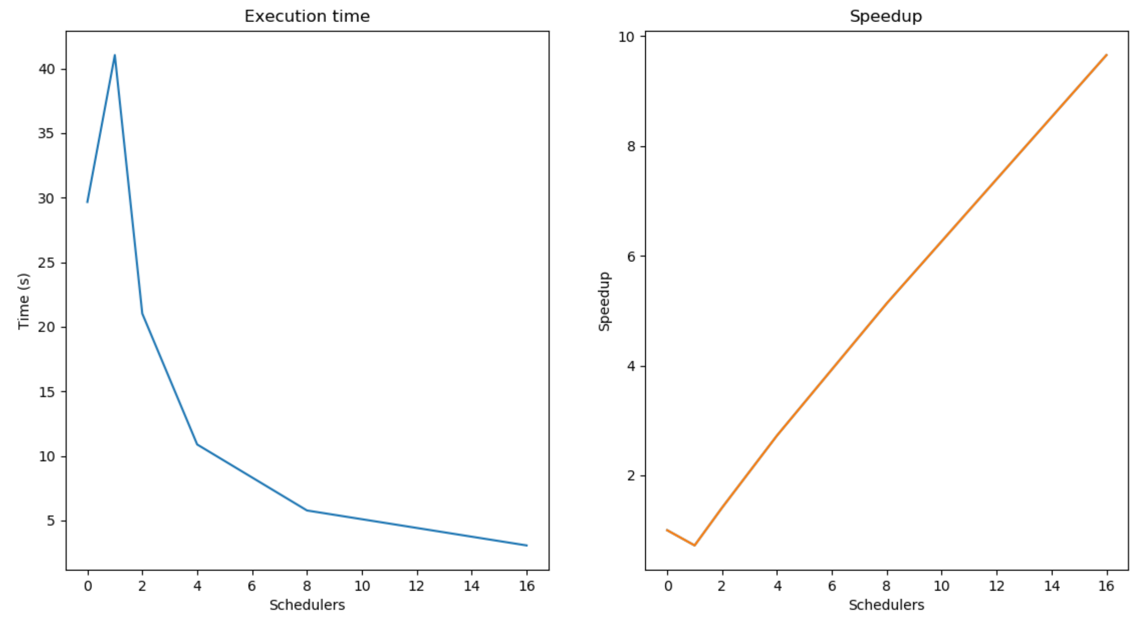


Figure 4.12: Distributed execution

Figure 4.13: Results of `crypto_ng_api:prop__crypto_init_update_final`

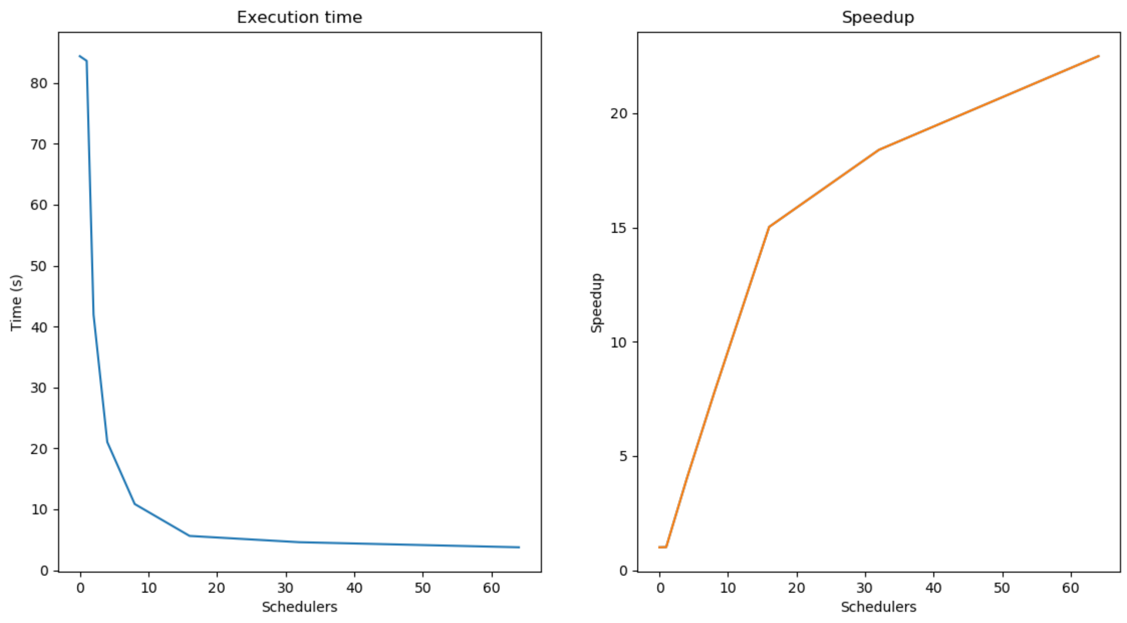


Figure 4.14: Parallel execution

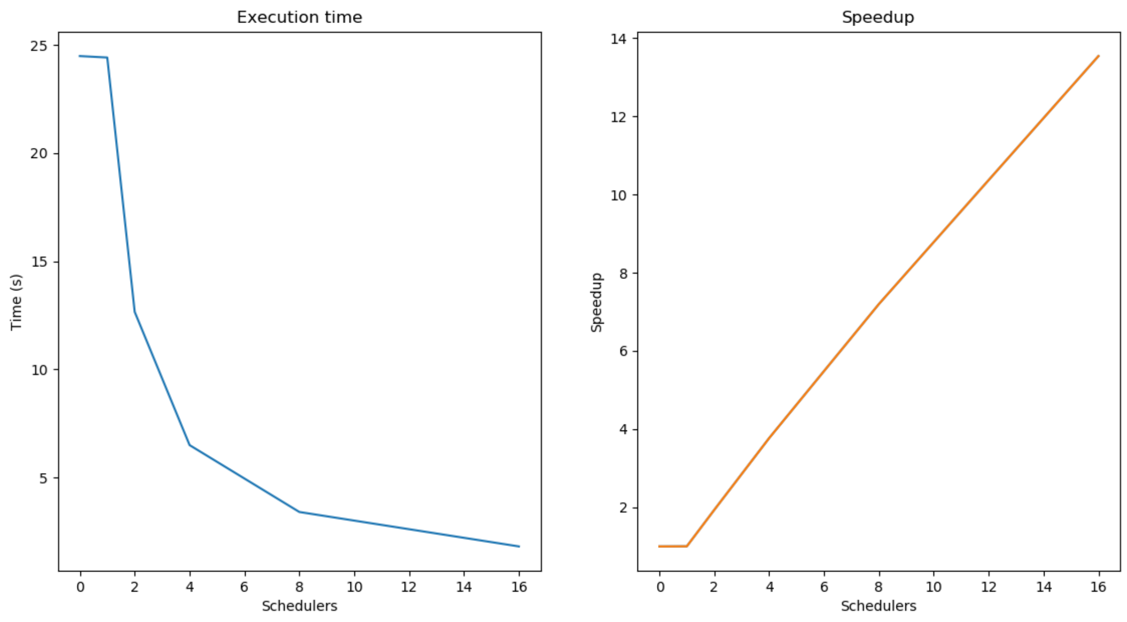


Figure 4.15: Distributed execution

Figure 4.16: Results of `ssh_eqc_encode_decode:prop_ssh_decode_encode()`

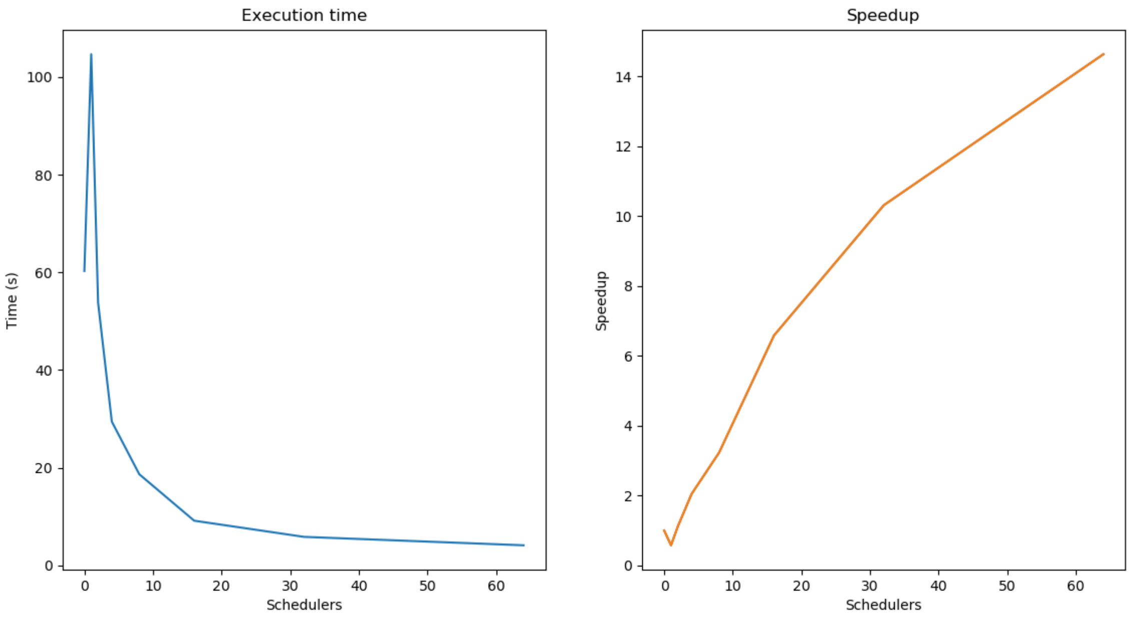


Figure 4.17: Parallel execution

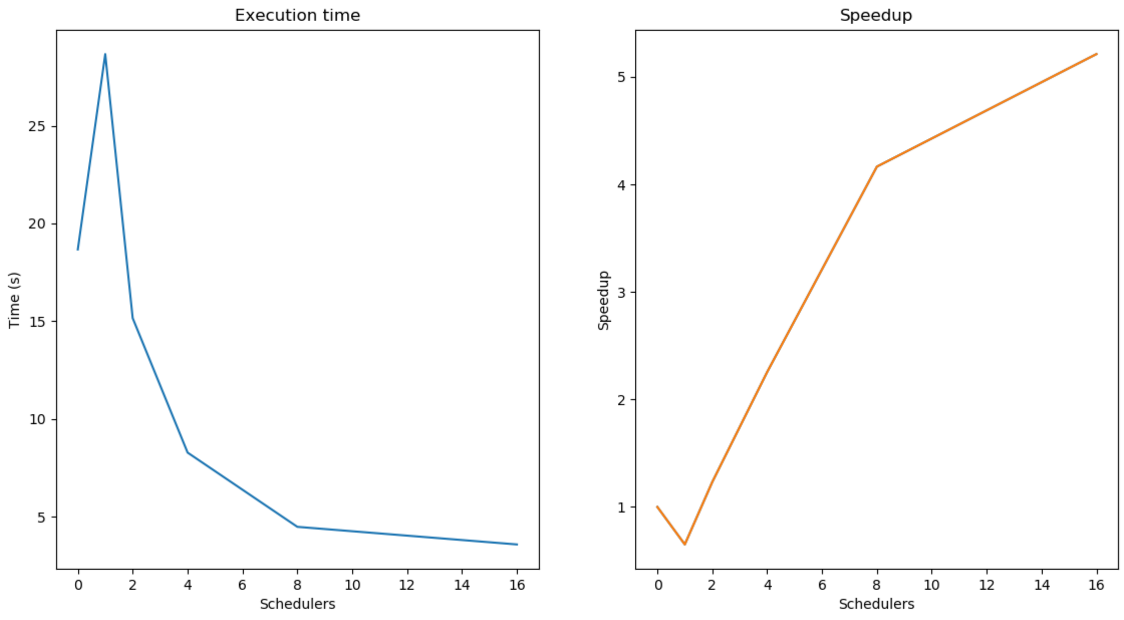


Figure 4.18: Distributed execution

Figure 4.19: Results of `shell_docs_prop:prop_render()`

Chapter 5

# Conclusions

In this final chapter of the report, we will first contrast the proposed objectives we had set for the project with the reached ones at the end. We will also talk about the lessons learned from carrying out this project and we will end up by talking about future possible improvements that could be done to the tool.

## 5.1 Follow-up

The main goal of this project was to extend the most powerful property-based testing tool for Erlang with parallel and distributed execution. We wanted to make use of the full potential of the programming language the tool was written in and for, to easily modify the existing codebase with as little impact as possible (and while following the best of practices) to bring a new usable version of the tool with both new manners of execution. Finally, we wanted to measure the obtainable speedup of our patched version of the tool by running it both in a cluster to test the distributed execution and in a single powerful machine to test the parallel execution.

Both implementations were done successfully and the tool can now be executed in a parallel or distributed fashion. The benchmarks could be improved upon, but the lack of time at the end of the project and the general absence of well-known projects, or at least projects used by the Erlang community, that had property-based tests usable by PropEr was a big decisive factor in the end.

### 5.1.1 Impact on the code

The property-based testing tool we extended in this project, PropEr, consisted of 12729 lines of Erlang code (as shown in 5.1) in the commit we started working on. After our patches, and also because of the upstream repository having some updates of their own that were added to our version too, the final number of lines of Erlang code the tool has is 14007. It is important to note, however, that the real number of changes done to the codebase is 490 insertions and 23 deletions (as shown in the diff of 5.2).

This means that, in order to bring parallel and distributed execution to the tool, the total number of code changes we did was just 513, which is around 3.66% of the total code of the tool.

```
$ cloc proper_before/
--------------------------------------------------------------------
  Language       files        blank         comment          code
--------------------------------------------------------------------
  Erlang          68           2294           4494         127297
  #     elided output as it is not relevant to the report     #

$ cloc proper_after/
--------------------------------------------------------------------
  Language       files        blank         comment          code
--------------------------------------------------------------------
  Erlang          77           2638           5080         14007
  #     elided output as it is not relevant to the report     #
```

Listing 5.1: Comparison of LOC (Lines of Code) PropEr had before and after our work

```
$ git diff upstream/master --stat
 Makefile                     |   2 +-
 include/proper_internal.hrl |  15 +++++
 src/proper.erl               | 476 ++++++++++++++++++++++++++++++++
                                    ++++++++++++++++++++++++++++++++
                                    ++++++++++++++++++++++++++++++++
                                    ++++++++++++++++++++++++++++++++
                                    ++----
 test/proper_specs_tests.erl |   4 +-
 test/proper_tests.erl       |  16 +++--
 5 files changed, 490 insertions(+), 23 deletions(-)
```

Listing 5.2: Total of additions and deletions done throughout our work

### 5.1.2 Lessons learned

Since the property-based testing tool we decided to extend is one that works and is very much cared about by its maintainers and users, this project meant to the student developing with the utmost care and their first look into an open-source application in Erlang used by many professionals.

Thus, this project helped the student to further understand and practise not only their knowledge of Erlang itself, but also what it meant to make and use a concurrent application and how to modify an existing one to make it parallel or distributed. This was done by reading papers related to the subject of making an Erlang program more fault-tolerant [28] and the benefits of the parallelization of an existing Erlang program [29]. Moreover, due to the relevance of the application in the area of Property-Based Testing, the student was able to deepen their knowledge of this approach to testing.

Furthermore, this project helped the student understand the importance of following a good methodology, as otherwise it is easy to lose track of everything and focusing into short sprints helps breaking down the project into smaller bits. In addition, the help and guidance of both supervisors has been a critical point of the project, and the student has not only learned that the knowledge from those with years of experience can be very useful when in need of help, but that there is no shame in doing so and that knowing how to communicate clearly is vital.

Finally, developing such an interesting project related to a field of software that is usually not very liked, testing, has helped the student realize the importance and advantages of having good test suites and knowing different methods of testing.

## 5.2 Future work

As future work for this project, there are a few things left to do:

- Revise the two strategies to split the tests that could not be benchmarked due to implementation issues.
- Clean up the code and apply any suggestions from the feedback of the creator of the tool before creating a Pull Request to the original repository to have the changes of this work go upstream.
- The experience from carrying out this project and the results of obtained from it will be also turned into a scientific paper that will be published later.

# Appendices

# List of Acronyms

|  |  |
|---|---|
| **BEAM** | Bogdan/Björn’s Erlang Abstract Machine. 8, 47 |
| **EQC** | Erlang Quickcheck. 47 |
| **ERTS** | Erlang RunTime System. 36 |
| **OTP** | Open Telecom Platform. 7, 36 |
| **PBT** | Property-Based Testing. 2 |
| **PID** | process identifier. 8 |
| **spg** | Scalable Process Groups. 3 |

# Glossary

**upstream** Original repository from where the code has been forked.. 3

# Bibliography

[1] E. W. Dijkstra, "On the reliability of programs (EWD303)." [Online]. Available: http://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD303.html

[2] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, vol. 46, 01 2000. [Online]. Available: https://doi.org/10.1145/1988042.1988046

[3] ——. (2000) Quickcheck. [Online]. Available: https://hackage.haskell.org/package/QuickCheck

[4] M. Papadakis and K. Sagonas, "A PropEr Integration of Types and Function Specifications with Property-Based Testing," in *Proceedings of the 2011 ACM SIGPLAN Erlang Workshop*. New York, NY: ACM Press, Sep. 2011, pp. 39–50. [Online]. Available: https://doi.org/10.1145/2034654.2034663

[5] PropEr: A QuickCheck-Inspired Property-Based Testing Tool for Erlang. [Online]. Available: https://proper-testing.github.io

[6] R. Hickey and R. Drape. (2014) test.check. [Online]. Available: https://github.com/clojure/test.check

[7] M. Fedorov. (2019) Scalable Process Groups. [Online]. Available: https://github.com/max-au/spg

[8] Fork of PropEr in GitHub. [Online]. Available: https://github.com/pablocostass/proper

[9] PropEr repository in GitHub. [Online]. Available: https://github.com/proper-testing/proper

[10] J. Armstrong, *Software for a Concurrent World*, 2nd ed. Pragmatic Bookshelf, 2013.

[11] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, p. 235–245. [Online]. Available: https://dl.acm.org/doi/10.5555/1624775.1624804

[12] M. Papadakis, "Automatic Random Testing of Function Properties from Specifications," Diploma thesis, National Technical University of Athens, School of Electrical and Computer Engineering, Oct. 2010. [Online]. Available: https://proper-testing.github.io/papers/manolis-thesis.pdf

[13] E. Arvaniti, "Automated Random Model-Based Testing of Stateful Systems," Diploma thesis, National Technical University of Athens, School of Electrical and Computer Engineering, Jul. 2011. [Online]. Available: https://proper-testing.github.io/papers/eirini-thesis.pdf

[14] Proper's documentation. [Online]. Available: https://proper-testing.github.io/apidocs/

[15] Getting Started with Erlang. [Online]. Available: http://erlang.org/documentation/doc-5.3/doc/getting_started/getting_started.html

[16] rebar3. [Online]. Available: https://www.rebar3.org/

[17] F. Hebert. rebar3 proper. [Online]. Available: https://github.com/ferd/rebar3_proper

[18] A PropEr statem tutorial. [Online]. Available: https://proper-testing.github.io/tutorials/PropEr_testing_of_generic_servers.html

[19] Nifty - nif interface generator. [Online]. Available: http://parapluu.github.io/nifty/

[20] PropEr's licence: GPL-3.0. [Online]. Available: https://github.com/proper-testing/proper/blob/master/COPYING

[21] Erlang's Directory Structure Guidelines for a Development Environment. [Online]. Available: https://erlang.org/doc/design_principles/applications.html#directory-structure-guidelines-for-a-development-environment

[22] Cowlib. [Online]. Available: https://github.com/ninenines/cowlib

[23] V. Raghunathan. Huffman Coding (ECE264). [Online]. Available: https://engineering.purdue.edu/ece264/17au/hw/HW13?alt=huffman

[24] Kazoo. [Online]. Available: https://www.2600hz.org/

[25] Zotonic. [Online]. Available: http://zotonic.com/

[26] Diffy. [Online]. Available: https://github.com/mmzeeman/diffy

[27] N. Fraser. (2006, Apr.) Diff strategies. [Online]. Available: https://neil.fraser.name/writing/diff/

[28] A. Löscher and K. Sagonas, "The Nifty Way to Call Hell from Heaven," in *Trends in Functional Programming*. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–11. [Online]. Available: https://doi.org/10.1145/2975969.2975970

[29] S. Aronis and K. Sagonas, "On Using Erlang for Parallelization," in *Trends in Functional Programming*, H.-W. Loidl and R. Peña, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 295–310. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-40447-4_19