"HotCI: An automated tool for reliable software upgrade/downgrade in Erlang OTP"

Zenon, Alexandre

ABSTRACT

Erlang/OTP is a powerful programming language for building fault-tolerant and scalable applications. One of its key features is the ability to perform hot code upgrades, which allow developers to update code without interrupting the running system. However, this feature is often underutilized because of the perceived complexities in comprehension, implementation, and testing. This thesis aims to solve this issue by introducing a new Continuous Integration/Continuous Delivery (CI/CD) tool called HotCI. HotCI provides a robust way to reliably and semi-automatically test the correctness of hot code upgrades and downgrades. It also offers additional benefits such as static analysis, unit tests, automatic generation of the files needed for the upgrade and release build.

CITE THIS VERSION

UCLouvain

epl

École polytechnique de Louvain

HotCI

An automated tool for reliable software upgrade/downgrade in Erlang OTP

Author: **Alexandre Zenon**
Supervisors: **Peer Stritzinger, Peter Van Roy**
Readers: **Julien Lienard, Pierre Martou**
Academic year 2023–2024
Master [120] in Computer Science

Abstract

Erlang/OTP is a powerful programming language for building fault-tolerant and scalable applications. One of its key features is the ability to perform hot code upgrades, which allow developers to update code without interrupting the running system. However, this feature is often underutilized because of the perceived complexities in comprehension, implementation, and testing. This thesis aims to solve this issue by introducing a new Continuous Integration/Continuous Delivery (CI/CD) tool called HotCI.

HotCI provides a robust way to reliably and semi-automatically test the correctness of hot code upgrades and downgrades. It also offers additional benefits such as static analysis, unit tests, automatic generation of the files needed for the upgrade and release build.

Dedication

À Freddy, mon papy, avec qui j'aurais aimé célébrer la fin de mes études.

Acknowledgements

The completion of this Master's thesis was made possible by the invaluable support and contributions of many individuals. I extend my sincere gratitude to all those who offered their expertise, guidance, and encouragement throughout this process.

I am deeply grateful to my family and Alicé for their unwavering support and encouragement throughout my academic journey.

I extend my sincere gratitude to my supervisors for their guidance, insightful feedback, and the opportunity to explore such an interesting research topic.

I would also like to thank Vladislav for his meticulous peer reviews.

Finally, I want to express my appreciation to the Erlang community for their warm welcome, invaluable assistance in understanding the complexities of Erlang/OTP, and their valuable insights.

Contents

1 Introduction **12**
- 1.1 Context . . . 12
- 1.2 Problems . . . 12
- 1.3 Contributions . . . 13
  - 1.3.1 Documentation for Erlang/OTP novices . . . 13
  - 1.3.2 Automatic generation of files related to DSU . . . 14
  - 1.3.3 Semi-automated testing for Erlang/OTP release upgrade and downgrade . . . 14
- 1.4 Roadmap . . . 14

2 Introduction to Erlang/OTP releases concepts **16**
- 2.1 Introduction . . . 16
- 2.2 Dynamic code change . . . 17
  - 2.2.1 The code_change function . . . 19
- 2.3 Project structure . . . 20
  - 2.3.1 Modules . . . 21
  - 2.3.2 Application . . . 21
  - 2.3.3 Release . . . 22
- 2.4 Conclusion . . . 23

3 Building a release in Erlang/OTP **24**
- 3.1 Introduction . . . 24
- 3.2 Manually building a release . . . 24
  - 3.2.1 Before starting . . . 24
  - 3.2.2 Writing an application resource file . . . 25
  - 3.2.3 Writing a release resource file . . . 26
  - 3.2.4 Generating a boot script . . . 27
  - 3.2.5 Creating a release package . . . 28
  - 3.2.6 Starting the release . . . 28
  - 3.2.7 Conclusion . . . 29
- 3.3 Building a release with Rebar3 . . . 29

3.3.1 Before starting . . . 29
3.3.2 Writing an application resource file . . . 29
3.3.3 Writing a rebar.config file . . . 29
3.3.4 Creating a release package . . . 30
3.3.5 Starting the release . . . 30
3.3.6 Conclusion . . . 31
3.4 Conclusion . . . 31

4 Building a hot code upgrade in Erlang/OTP 32
4.1 Introduction . . . 32
4.2 Creating a release including Pixelwar version 0.2.0 . . . 33
4.2.1 Building the release . . . 33
4.2.2 Running the release . . . 33
4.3 Updating the release to include Pixelwar version 0.3.0 . . . 34
4.3.1 Applying modifications to the Pixelwar application . . . 34
4.3.2 Generating a relup for the release . . . 36
4.4 Executing the upgrade of the release . . . 37
4.4.1 Packing the new version of the release . . . 37
4.4.2 Moving the new release to the running release's folder . . . 37
4.4.3 Upgrading the release . . . 37
4.4.4 Testing that the upgrade has been applied . . . 37
4.5 Conclusion . . . 38

5 HotCI 39
5.1 Introduction . . . 39
5.2 The Dandelion project . . . 40
5.3 Workflows . . . 40
5.3.1 First workflow: Erlang-CI . . . 40
5.3.2 Second workflow: Relup-CI . . . 44
5.3.3 Third workflow: Publish-tarball . . . 51
5.4 Usage example . . . 55
5.4.1 Introduction . . . 55
5.4.2 Creating a new project . . . 55
5.4.3 Creating a first version of the release . . . 55
5.4.4 Releasing the first version . . . 58
5.4.5 Creating a second version of the release . . . 58
5.4.6 Modifying the upgrade_downgrade_SUITE . . . 59
5.4.7 Releasing the second version . . . 61
5.4.8 Conclusion . . . 61
5.5 Limitations . . . 62
5.5.1 GitHub Action . . . 62

5.5.2 Maintaining the GitHub template . . . 63
5.6 Conclusion . . . 63

6 Evaluation **64**
6.1 Introduction . . . 64
6.2 Gathered feedback . . . 64
6.2.1 Likert scale questions . . . 65
6.2.2 Open-ended questions . . . 67
6.3 Integration within the community . . . 68
6.4 Execution time . . . 68
6.5 Conclusion . . . 69

7 Conclusion **72**
7.1 Introduction . . . 72
7.2 Future works . . . 72
7.2.1 Distributed systems . . . 72
7.2.2 Integrating test reports into GitHub Pages . . . 73
7.2.3 Accelerating testing and reducing cost with caching . . . 73
7.2.4 Fostering an improved collaboration with the Erlang community . . . 73

A Pixelwar **77**
A.1 Subset of version 1.0.0 . . . 77
A.2 Version 0.2.0 . . . 80
A.3 Version 0.3.0 . . . 85

B Upgrade downgrade test suites **94**
B.1 First Version . . . 94
B.2 Second version . . . 95
B.3 Third version . . . 97

C HotCI **100**
C.1 Benchmark . . . 117

D Dandelion **121**

List of Figures

2.1 Fully qualified call with a single version in the process 17
2.2 Fully qualified call with two versions in the process 18
2.3 Not fully qualified call with two versions in the system 19
2.4 Tree representation of the project structure 20

3.1 Initial file structure for section 3.2.1 25

5.1 Execution of the erlang-ci workflow 43
5.2 Execution of the relup-ci workflow 45
5.3 Execution of the upgrade downgrade test suite 48
5.4 Execution of the publish-tarball workflow 52
5.5 Using webhooks to be notified when a new GitHub release is published 54
5.6 Upgrading the release when the webhook is triggered 54
5.7 Directory structure after merging the user's repository based on the HotCI template and Pixelwar 0.2.0 57
5.8 HotCI's ceremony 62

6.1 Boxplot of erlang-ci and relup-ci's execution time 70
6.2 Line Chart of erlang-ci and relup-ci's execution time 71

A.1 Directory structure of the subset of Pixelwar version 1.0.0 77
A.2 Directory structure of Pixelwar version 0.2.0 81
A.3 Directory structure of Pixelwar version 0.3.0 86

C.1 Directory structure of HotCI 100

List of Tables

6.1 User satisfaction towards different aspects of HotCI 66
6.2 User's agreement towards general statements about HotCI 66

Listings

2.1 Upgrading to the new version with a fully qualified call 17
2.2 Keeping the old version with a not fully qualified call 18
2.3 code_change example for a gen_server 20
2.4 Example of an Erlang/OTP module 21
2.5 Example of an application resource file 21
2.6 Example of an appup file using high-level instructions 22
2.7 Example of a release resource file 23
2.8 Relup file structure 23
3.1 Application resource file for the Pixelwar application 25
3.2 Compiling and running the Pixelwar application 26
3.3 Finding versions of the required applications and the ERTS 26
3.4 Release resource file for the Pixelwar application 27
3.5 Generating a boot script 27
3.6 Launching the release with the boot script and testing some functions 27
3.7 Packing the release 28
3.8 Deploying, starting and interacting with the release 28
3.9 A minimal rebar.config file for Pixelwar 30
3.10 Deploying, starting and interacting with the release 30
4.1 Modifying the release's state 33
4.2 -vsn directive present in the pixelwar_matrix_serv module 34
4.3 code_change implementation to transition from 0.2.0 to 0.3.0 . . . 35
4.4 The matrix record and its equivalent tuple representation 35
4.5 Pixelwar's appup to transition between 0.2.0 and 0.3.0 36
4.6 Output after applying the upgrade 38
5.1 Runtime error missed by Dialyzer because of its optimism 41
5.2 Modifying the before_upgrade_case 59
5.3 Modifying the after_upgrade_case 60
5.4 Modifying the before_downgrade_case 60
5.5 Modifying the after_downgrade_case 60
A.1 matrix.hrl 77
A.2 pixelwar_app.erl 77

A.3 pixelwar_matrix_serv.erl . . . 78
A.4 pixelwar_matrix.erl . . . 79
A.5 pixelwar_sup.erl . . . 79
A.6 pixelwar.app.src . . . 80
A.7 rebar.config . . . 81
A.8 pixelwar_app.erl . . . 82
A.9 pixelwar_matrix_serv.erl . . . 82
A.10 pixelwar_sup.erl . . . 83
A.11 pixelwar.app.src . . . 84
A.12 pixelwar_serv_SUITE.erl . . . 84
A.13 rebar.config . . . 86
A.14 matrix.hrl . . . 87
A.15 pixelwar_app.erl . . . 87
A.16 pixelwar_matrix_serv.erl . . . 87
A.17 pixelwar_matrix.erl . . . 88
A.18 pixelwar_sup.erl . . . 89
A.19 pixelwar.app.src . . . 90
A.20 pixelwar.appup.src . . . 90
A.21 pixelwar_serv_SUITE.erl . . . 90
A.22 pixelwar_matrix_SUITE.erl . . . 91
B.1 Excerpt from the first Github workflow . . . 94
B.2 Testing the state of a release from bash . . . 94
B.3 A generic python module for interfacing between Robot Framework and an OTP release . . . 95
B.4 An application specific python module for interfacing with our release 96
B.5 A Robot Framework test suite using the python modules . . . 96
B.6 Upgrade/downgrade CT test suite . . . 97
C.1 publish-ct-results/action.yml . . . 101
C.2 setup-beam/action.yml . . . 101
C.3 erlang-ci.yml . . . 101
C.4 relup-ci.yml . . . 102
C.5 publish-tarball.yml . . . 104
C.6 check_versions . . . 105
C.7 get_release_name . . . 107
C.8 upgrade_downgrade_SUITE.erl . . . 107
C.9 LICENSE.md . . . 110
C.10 README.md . . . 113
C.11 rebar.config . . . 116
C.12 Python script used to generate figure 6.1 and 6.2 . . . 118
D.1 check_versions . . . 121

Disclaimer

Grammar and spelling were refined through a combination of peer review with a fellow student and friend, and AI tools such as Google's Gemini Large Language Model.

Chapter 1

Introduction

1.1 Context

To initiate our discussion, let us define what is a Dynamic Software Update (DSU), a procedure also known as *Hot Code Upgrade*, *Hot Code Change*, or *Hot Code Swapping*.

The DSU mechanism allows for the system to be updated without having to stop and while remaining operational.

This ensures uninterrupted service for end-users, making it an invaluable mechanism for a wide array of systems, including internet-based applications, distributed systems, operating systems, or databases[9].

It is worth noting that various techniques and algorithms exist to facilitate such updates. However, this thesis does not delve into them as it focuses on Erlang/OTP and relies on the abstractions it provides.

Although Erlang/OTP is capable of performing a DSU, most developers either abstain from utilizing this functionality or caution against its use due to perceived complexities in comprehension, implementation, and testing.

It is regrettable that this situation has arisen, considering the substantial resources and expertise dedicated to creating a system with such capabilities and clear benefits to many domains in computer science.

1.2 Problems

To begin, let us familiarize ourselves with the problems faced by developers within the realm of Erlang/OTP, releases and DSU.

The first hurdle lies in the *system's complexity*. Despite Erlang/OTP's robustness, comprehending all the intricacies behind certain concepts proves challenging.

This complexity becomes evident in online forums or discussion channels, where novices often struggle to grasp, for example, the concept of releases and experienced developers admit that they have never used them even though they have a decade of experience with Erlang/OTP[12].

The second challenge lies in the *complexity of implementing a hot code upgrade/downgrade*. This process necessitates meticulously crafting multiple files that detail the sequence and method for updating various Erlang/OTP applications and modules, as well as correctly defining a state transition function. This function includes Erlang code to transform the state of the first version to that of the second version. Both of these requirements are highly error-prone due to their manual nature.

The complexity of the system and the implementation lead to the observation that Erlang/OTP does not meet the *ease-of-use* property[20]. The *ease-of-use* property suggests that, generally, the simpler the creation and application of an update, the less chance there is of making an error. However, here, since the system is complex and many steps in specifying an update are manual, the likelihood of making a mistake is significantly increased.

The last difficulty lies in *asserting the correctness* of a hot code upgrade. Successfully performing a DSU does not guarantee its *correctness*. An application may manage to update without crashing but may end up in an invalid state, a fact not lost on wary programmers who consequently approach DSU cautiously.

This problem highlights the necessity of verifying the *correctness* of updates, a task made challenging by the inability to completely automate the writing of such tests. For instance, transforming the state from a version to another might involve complex business logic that a generic program for generating tests might not be able to infer. The *correctness* property[20] states that a dynamic update should not lead the system to an incorrect state.

This limitation leads to the necessity of creating manual tests, though these tests can then be integrated into automated suites for regular execution, enhancing confidence in code and configuration changes.

1.3 Contributions

1.3.1 Documentation for Erlang/OTP novices

This thesis introduces various concepts related to Erlang/OTP release by providing concrete examples and step-by-step instructions to build the different components required for their creation.

These concepts include hot code change, the creation of releases, both manually and automatically, as well as the creation of updates. For the automated creation

of a release, Rebar3[22], the official build tool for Erlang is used.

Given that readers of this thesis may not be familiar with Erlang/OTP, the goal of this introduction is to ensure that even someone with little or no experience can understand the different steps involved in applying and creating a DSU for an Erlang/OTP release.

1.3.2 Automatic generation of files related to DSU

HotCI, the tool developed for this thesis, leverages Rebar3 and its appup plugin to automate the creation of the *appups*, *relups*, and the *boot script* files which are used to describe how to apply an update. Normally, writing *appups* is done manually. However, as a beginner, it is often hard to decide what instructions to include inside an *appup* file. By automating their generation, HotCI improves the *ease-of-use* of Erlang/OTP's release system and reduces the potential for errors.

One of our hopes regarding this contribution is that simplifying the use of the release system may lead to its democratization.

1.3.3 Semi-automated testing for Erlang/OTP release upgrade and downgrade

HotCI's biggest contribution is a semi-automated testing process for hot code upgrades and downgrades. This process involves launching an Erlang/OTP release, using a separate Erlang process to interact with and modify its state, executing the upgrade, verifying that the state is correct, downgrading the release and asserting that the final state is as expected. A comprehensive and detailed explanation of this methodology can be found in chapter 5.

The robust method provided by HotCI for testing upgrades/downgrades is crucial as it enhances their *correctness*, giving developers greater confidence in the updates they create.

Previously, without HotCI, assessing the *correctness* of updates was a difficult and cumbersome process. This led to the previously mentioned apprehension and underutilization of the upgrade/downgrade feature.

By providing better tools for testing hot code upgrades and downgrades, the hope is, once again, that this functionality will become more commonly used by the community.

1.4 Roadmap

This Master's thesis is structured into two main parts, each serving a distinct purpose.

The first part encompasses chapters 2, 3 and 4. It serves as a foundational knowledge base, introducing Erlang/OTP concepts essential for comprehending the subjects discussed in the subsequent section. This segment represents the compilation of all gathered information necessary for developing HotCI.

Chapter 2 introduces various Erlang/OTP concepts and mechanisms essential for understanding how HotCI operates.

Chapter 3 digs into the process of building an Erlang/OTP release, both manually and with the aid of state-of-the-art build tools. Understanding the structure of a release is crucial for subsequent discussions.

Chapter 4 provides details on how to write and perform release upgrades and downgrades.

The second part comprises chapters 5, 6, and 7 focusing on HotCI, the CI/CD tool developed for this thesis.

Chapter 5 describes the HotCI tool, covering its development, design, objectives, and usage.

Chapter 6 provides an evaluation of HotCI. It gauges overall user satisfaction, gathers feedback, measures community reception, and analyzes execution durations.

Chapter 7 serves as the conclusion of this thesis and outlines future work planned for this tool.

Chapter 2

Introduction to Erlang/OTP releases concepts

2.1 Introduction

Erlang/OTP stands apart from many programming environments due to its powerful mechanisms for seamless code updates in live systems. However, mastering the specific concepts of the Erlang/OTP release system like dynamic code changes, applications, and releases can present a steep learning curve. This chapter serves as a primer, demistifying these concepts which are essential to understand the creation and upgrade of a release as well as the contributions brought by HotCI.

Section 2.2 begins by introducing the concept of dynamic code changes, explaining how Erlang/OTP allows for hot code upgrades by maintaining two versions of each module's code and allowing processes to transition between them. It also details the `code_change` function, a key mechanism for managing state transitions during code upgrades and downgrades of generic behaviors.

Section 2.3 details the project structure of Erlang/OTP projects, starting with modules, the basic units of code organization. It then discusses applications, collections of modules that can be started or stopped as a unit, and releases, complete systems comprising applications. Finally, it touches upon appup and relup files, which provide instructions for upgrading and downgrading applications and releases, respectively.

By the end of this chapter, readers will have a grasp of the foundational concepts underpinning Erlang/OTP's releases and their hot code upgrade capabilities.

2.2 Dynamic code change

The dynamic code change mechanism serves as the foundation of hot code upgrades in Erlang/OTP.

A comprehensive explanation of the dynamic code change mechanism is provided in Armstrong's thesis [10]. Here, a simplified overview is offered.

The Erlang system maintains two versions of each module's code. In cases where only one version of the code exists, a single version is retained.

Upon loading a new version of the code, processes have the option to carry on with either the old version or transition to the new version. This decision is achieved by invoking functions using their fully qualified names, i.e., `module:function()`, rather than solely referencing their name, `function()`.

The subsequent examples are drawn from [10]

Listing 2.1: Upgrading to the new version with a fully qualified call

```
1 -module(m).
2 ...
3 loop(Data, F) ->
4     receive
5         {From, Q} ->
6             {Reply, Data1} = F(Q, Data),
7             m:loop(data1, F)
8     end.
```

Initially, with a single version of the code present, the process is not updated and its code remains unchanged. Figure 2.1 illustrates this situation.

However, upon loading another version of the module, the process upgrades to the newer version due to the invocation of the loop function with its fully qualified name on line 7. (See Figure 2.2)

Figure 2.1: Fully qualified call with a single version in the process

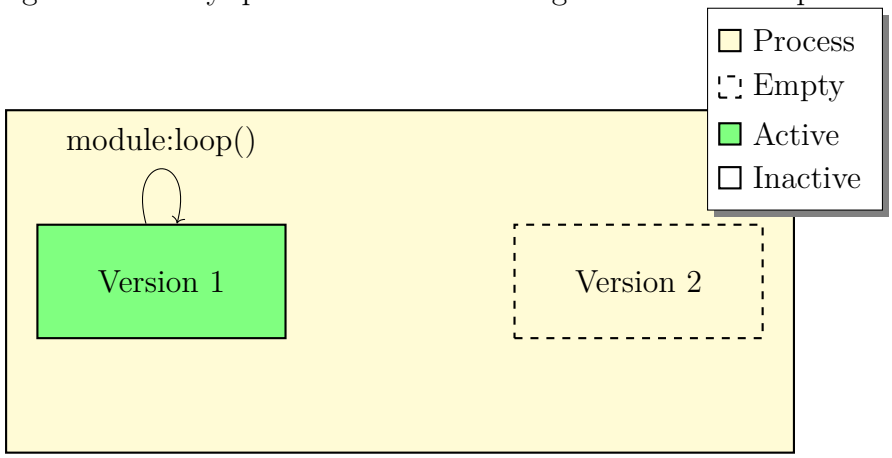
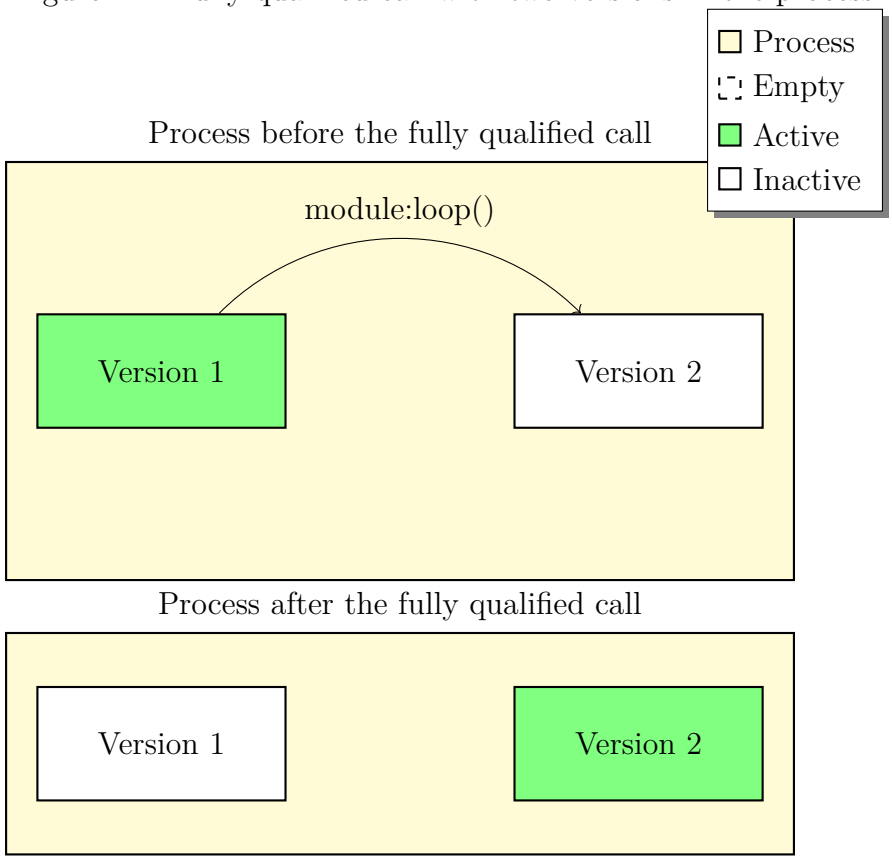


Figure 2.2: Fully qualified call with two versions in the process

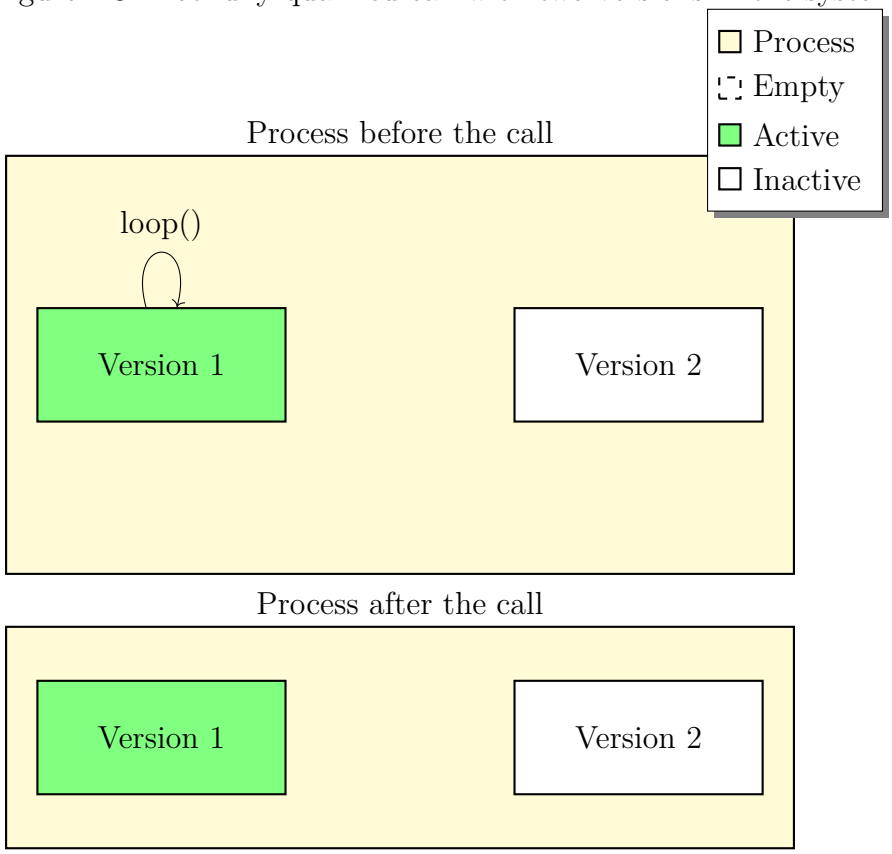


Listing 2.2: Keeping the old version with a not fully qualified call

```
1 -module(m).
2 ...
3 loop(Data, F) ->
4     receive
5         {From, Q} ->
6             {Reply, Data1} = F(Q, Data),
7             loop(data1, F)
8     end.
```

In this second code example and Figure 2.3, the process remains unaffected as the call to loop is unqualified.

Figure 2.3: Not fully qualified call with two versions in the system



It should be noted that ensuring compatibility between the new and old functions, in this case the `loop` function, is the responsibility of the developer, a task which is prone to errors.

2.2.1 The code_change function

Within Erlang/OTP's generic behaviors, dynamic code changes are facilitated by the `code_change()` function. This function typically accepts three arguments, though some behaviors may require more.

The first argument, `OldVSN`, is the version to upgrade from or the version to downgrade to. For upgrades, any data type is valid. Downgrades necessitate a tuple in the format `{down, Term}`.

The `State` argument, second in order, represents the current state of the behavior. Its purpose is to enable the conversion of state data from its definition in the current version to its definition in the target version.

The third argument, `Extra`, allows for the inclusion of supplementary information if needed.

Upon successful execution, this function must return a tuple of the form `{ok, State}`, where `State` is the transformed state. If an error arises, the function should return `{error, Reason}`, triggering a rollback to the previously active version.

The following example illustrates an implementation of the `code_change` function within a `gen_server`.

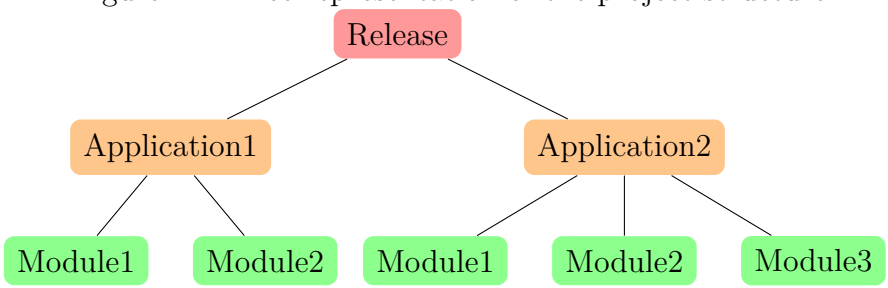
Listing 2.3: code_change example for a gen_server

```erlang
% Upgrade from 0.2.0 to the current version
code_change("0.2.0", State, _Extra) ->
    % Apply modification to the state if necessary
    % ...
    {ok, NewState};

% Downgrade from the current version to the 0.2.0 version
code_change({down, "0.2.0"}, State, _Extra) ->
    % Apply modification to the state if necessary
    % ...
    {ok, NewState};

% Default case
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

2.3 Project structure

The following subsections introduce the project structure, illustrated in Figure 2.4, employed in Erlang/OTP, progressing from the lower levels to the higher levels.

Figure 2.4: Tree representation of the project structure



2.3.1 Modules

Erlang/OTP code is organized into modules. Each module comprises a sequence of attributes and function declarations, each terminated by a period. An attribute specifies a particular property of a module.

The following example is drawn from [6].

Listing 2.4: Example of an Erlang/OTP module

```
1 -module(m).           % module attribute
2 -export([fact/1]).    % module attribute
3
4 fact(N) when N>0 ->   % beginning of function declaration
5     N * fact(N-1);    %  |
6 fact(0) ->            %  |
7     1.                % end of function declaration
```

2.3.2 Application

An application constitutes a collection of modules that can be initiated or terminated as a cohesive unit and can also be repurposed in other systems.

It is worth noting that Erlang applications are sometimes likened to libraries in other languages, potentially offering a helpful analogy for understanding their structure and function.

The definition of an application is encapsulated within an application resource file.

Listing 2.5: Example of an application resource file

```
1  {application, pixelwar, [
2      % A one-line description of the application.
3      {description, "An OTP application"},
4      % Version of the application.
5      {vsn, "0.3.0"},
6      % All names of registered processes started in this application.
7      {registered, []},
8      % Specifies the application callback module and a start argument
9      {mod, {pixelwar_app, []}},
10     % All applications that must be started before this application
       is started.
11     {applications, [
12         kernel, % All applications depend on kernel and stdlib
13         stdlib
14     ]},
15     % Configuration parameters used by the application.
16     {env, []},
17     % All Modules introduced by this application
```

```
18      {modules, [pixelwar_app, pixelwar_matrix_serv, pixelwar_matrix,
        pixelwar_sup]},
19      {licenses, ["Apache-2.0"]},
20      {links, []}
21  ]}.
```

Appup

An appup file serves as a recipe for handling the upgrade or downgrade of an application within a running release.

It provides detailed instructions for transitioning from a specific version to another, encompassing directives for both upgrading and downgrading scenarios. These instructions dictate which modules require reloading, which applications should be upgraded/downgraded, and other pertinent details necessary for a seamless transition.

Listing 2.6: Example of an appup file using high-level instructions

```
1  {"0.3.0", % New version
2      [{"0.2.0", [ % Upgrade from
3          {add_module, pixelwar_matrix},
4          {update, pixelwar_matrix_serv, {advanced, []}}
5      ]}],
6      [{"0.2.0", [ % Downgrade to
7          {delete_module, pixelwar_matrix},
8          {update, pixelwar_matrix_serv, {advanced, []}}
9      ]}]
10 }.
```

The instructions contained within the appup file are typically categorized into two distinct levels: high-level and low-level. High-level instructions are automatically translated into their low-level counterparts during processing. It is strongly advised that developers exclusively utilize high-level instructions when composing an appup file.

Conventionally, appups are manually crafted by developers. Resources such as the appup cookbook [2] can aid developers in the creation of appups. Nevertheless, due to the manual nature of the process and the absence of a validation phase, it is susceptible to errors.

One proposed solution to mitigate this issue is the implementation of tools to automate the generation of appups. Such tools will be investigated in chapter 5.

2.3.3 Release

A release is a complete system composed of a set of custom applications and a subset of Erlang/OTP applications from the standard library.

Releases are defined by a release resource file. It is required from the developer to define the name and version of the release, which Erlang runtime system (ERTS) version is used and which applications are part of it:

Listing 2.7: Example of a release resource file

```
{release, {"pixelwar", "1.0.0"}, {erts, "14.0"},
    [
        {sasl, "4.2.1"}, {pixelwar, "1.0.0"},
        {stdlib, "5.0"}, {kernel, "9.0"}
    ]
}.
```

Relup

A relup file closely resembles an appup file and functions as a descriptor outlining the steps for upgrading or downgrading a release within a running system.

The format of a relup file mirrors that of appup files. However, it includes a translation of all the high-level instructions from all the appup files into low-level instructions, arranged in an order defined by the boot script associated with the release.

The generation of a relup file is automated by invoking the `systools:make_relup` function.

Listing 2.8: Relup file structure

```
{Vsn,
  [{UpFromVsn, Descr, Instructions}, ...],
  [{DownToVsn, Descr, Instructions}, ...]}.
```

2.4 Conclusion

In conclusion, this chapter has established a foundational understanding of the core Erlang/OTP concepts essential for creating and upgrading releases.

Building upon this knowledge, the subsequent chapter will explore the practical aspects of release creation, exploring both manual and automated approaches, and showcasing the complexity of this task.

Chapter 3

Building a release in Erlang/OTP

3.1 Introduction

This chapter provides the necessary knowledge to construct releases within Erlang/OTP, a required step before undertaking hot code upgrades which are one of the main concerns of HotCI.

Section 3.2 offers a manual, step-by-step guide to constructing a release, providing insights into the underlying mechanisms. It details the creation of application and release resource files, the generation of boot scripts, and the packaging and deployment of the release.

Section 3.3 introduces Rebar3, a state-of-the-art build tool that simplifies the release creation process. It highlights Rebar3's key features and demonstrates its extensive integration within HotCI.

By the end of this chapter, readers will be equipped to build a release, either manually or using Rebar3.

3.2 Manually building a release

3.2.1 Before starting

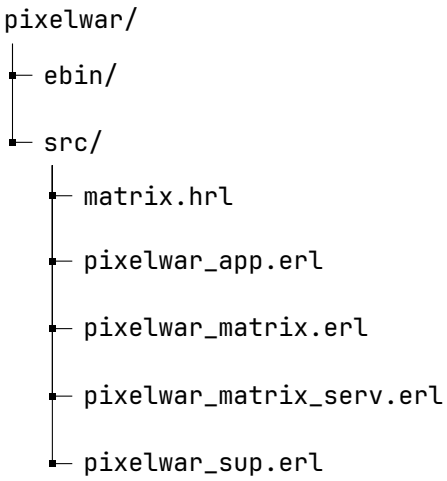
This section draws inspiration from both the *OTP releases by hand* [23] and *Building Erlang applications the hard way* [19] blog posts, adapting their instruction to a subset of the Pixelwar application.

Pixelwar, an Erlang/OTP release and application that takes inspiration from the Reddit r/places web page (a collaborative pixel art game where users paint on a shared digital canvas), serves as a testing ground to ensure the functionality of HotCI.

The specific functionality of each module included in Pixelwar is not pertinent to this discussion; therefore, further elaboration will be omitted. However, for the sake of replicability, the source code is available under Appendix A.1.

The following sections assume that the following file structure is used:

Figure 3.1: Initial file structure for section 3.2.1



3.2.2 Writing an application resource file

Once the logic has been separated into individual modules, the initial step in creating an application involves writing an application resource file. Below is an example of such a file for the Pixelwar application:

Listing 3.1: Application resource file for the Pixelwar application

```
1  % file: pixelwar/src/pixelwar.app.src
2
3  {application, pixelwar, [
4      {description, "Yet another r/place clone"},
5      {vsn, "1.0.0"},
6      {registered, []},
7      {mod, {pixelwar_app, []}},
8      {applications, [kernel, stdlib]},
9      {env, []},
10     {modules, [pixelwar_app, pixelwar_matrix_serv, pixelwar_matrix,
       pixelwar_sup]}
11 ]}.
```

Next, the modules are compiled, and their binaries, along with the application resource file, are relocated to the `ebin` folder. The functionality of the application is then verified by starting the application and listing all active applications.

Listing 3.2: Compiling and running the Pixelwar application

```
1  % dir: pixelwar/
2
3  erlc -o ebin src/*.erl
4  cp src/pixelwar.app.src ebin/pixelwar.app
5  erl -pa ebin
6
7  1> ok = application:start(pixelwar).
8  ok
9  2> application:which_applications().
10 [{pixelwar,"Yet another r/place clone","1.0.0"},
11  {stdlib,"ERTS  CXC 138 10","5.0"},
12  {kernel,"ERTS  CXC 138 10","9.0"}]
```

The function call at line 7 returns successfully with an `ok` value, while the call at line 9 showcases the various applications included in our application resource file, including the Pixelwar application, as anticipated.

3.2.3 Writing a release resource file

With the application resource file written, the next step is to create a release including this application.

In addition to the Pixelwar application, it is imperative to include other essential applications such as `Kernel`, `Stdlib`, and `SASL`. Kernel and Stdlib are mandatory dependencies as they form the foundation upon which all applications rely. SASL is also required as it manages release handling processes.

To determine the versions of these various modules and the version of the ERTS, the following commands can be executed.

Listing 3.3: Finding versions of the required applications and the ERTS

```
1  % dir : pixelwar/
2
3  3> application:start(sasl).
4  ok
5  4> application:which_applications().
6  [{sasl,"SASL  CXC 138 11","4.2.1"},
7   {pixelwar,"Yet another r/place clone","1.0.0"},
8   {stdlib,"ERTS  CXC 138 10","5.0"},
9   {kernel,"ERTS  CXC 138 10","9.0"}]
10 5> erlang:system_info(version).
11 "14.0"
```

Once all this information has been gathered, a `pixelwar.rel` file, containing these applications along with their respective exact versions, can be written.

Listing 3.4: Release resource file for the Pixelwar application

```
% file: pixelwar/ebin/pixelwar.rel

{release, {"pixelwar", "1.0.0"}, {erts, "14.0"},
    [
        {sasl, "4.2.1"}, {pixelwar, "1.0.0"},
        {stdlib, "5.0"}, {kernel, "9.0"}
    ]
}.
```

3.2.4 Generating a boot script

The boot script describes how the Erlang runtime system (ERTS) is started. It contains instructions on which code to load and which processes and applications to start[7].

To generate the boot script, the `systools` module and the `make_script` function are used. The `make_script` function takes two arguments: the name of the script and an array of options. In this instance, the `path` option is included to instruct the `make_script` function to generate a script from the files located in the current directory.

Listing 3.5: Generating a boot script

```
% dir: pixelwar/ebin

erl

1> systools:make_script("pixelwar", [{path, ["."]}]).
ok
```

After generating the script, the next step is to proceed with launching the release and interacting with it to ensure that it is functioning as intended.

Listing 3.6: Launching the release with the boot script and testing some functions

```
% dir: pixelwar/ebin

erl -pa . -boot pixelwar

1> {ok, Pid} = pixelwar_sup:add_matrix("SomeName").
{ok,<0.100.0>}
2> pixelwar_matrix_serv:set_element("SomeName", {42, 42, 2}).
ok
3> pixelwar_matrix_serv:get_state("SomeName").
<<42,0,42,0,2,0>>
```

3.2.5 Creating a release package

First, a folder in which the release will reside is created. Subsequently, a tarball containing the release is generated using the `make_tar` function from the `systools` library.

The `make_tar` function takes two arguments: the name of the tarball and a list of options. In this case, the `erts`, `path`, and `outdir` options are used. The `erts` option specifies the location of the ERTS installed on the machine, while the path option indicates the source location of the release files. Finally, the `outdir` option determines the destination directory for the generated tarball.

Listing 3.7: Packing the release

```
% dir: pixelwar/

mkdir -p _rel/pixelwar
cd ebin
erl

1> R = code:root_dir().
"/path/to/erlang"
2> systools:make_tar("pixelwar", [{erts, R}, {path, ["."]}, {outdir,
    "../_rel/pixelwar"}]).
ok
```

3.2.6 Starting the release

Now that the release is packaged, it is possible to emulate uploading the release to a server by unpacking the release into the */tmp* folder and, then, starting it.

Unpacking and starting the release can be done with the following steps:

Listing 3.8: Deploying, starting and interacting with the release

```
% dir: pixelwar/_rel/pixelwar

mkdir /tmp/pixelwar
tar -xf pixelwar.tar.gz -C /tmp/pixelwar
cd /tmp/pixelwar
./erts-14.0/bin/erl -boot releases/1.0.0/start

1> {ok, Pid} = pixelwar_sup:add_matrix("SomeName").
{ok,<0.100.0>}
2> pixelwar_matrix_serv:set_element("SomeName", {42, 42, 2}).
ok
3> pixelwar_matrix_serv:get_state("SomeName").
<<42,0,42,0,2,0>>
```

Every function call behaves as before. We can then conclude that this last step is successful.

3.2.7 Conclusion

This section has provided a detailed walkthrough of the manual crafting process involved in creating a release for the Pixelwar application.

The essential steps have been introduced, from writing application and release resource files to generating boot files, creating release packages and simulating the deployment and start of the release.

This hands-on approach provides a deeper understanding of the release creation process and demonstrates that creating a release is not a straightforward task, highlighting the need for an accessible build tool.

This tool is named Rebar3 and its functionalities will be introduced in the following section.

3.3 Building a release with Rebar3

3.3.1 Before starting

This sections starts with the identical file structure as the one depicted in Figure 3.1.

3.3.2 Writing an application resource file

The inclusion of the application resource file remains necessary. The existing application resource file, from subsection 3.2.2 can be retained and placed at the same location as before.

Afterward, as in subsection 3.2.2, we proceed to compile our modules. This operation is easily accomplished with Rebar3 by entering the following command into the terminal:

```
1 rebar3 compile
```

3.3.3 Writing a rebar.config file

A minimal `rebar.config` file appears as follows. It is worth noting that there is no need to include kernel or stdlib in the dependencies, as Rebar3 recognizes that both kernel and stdlib are necessary for all applications.

Listing 3.9: A minimal rebar.config file for Pixelwar

```
% file: pixelwar/rebar.config

{relx, [{release,
        {pixelwar, "0.1.0"}, % {Name of the release, Version}
        [pixelwar,sasl]}] % [dependencies]
}.
```

To generate the release, simply enter this command:

```
rebar3 release
```

3.3.4 Creating a release package

Creating a release package is as simple as generating a release. It is done by executing the following command:

```
rebar3 tar
```

The release package is now built and accessible at:
`pixelwar/_build/default/rel/pixelwar/pixelwar-0.1.0.tar.gz`

3.3.5 Starting the release

The last step, like with the manual building of a release, is to emulate the deployment and the start of the release.

Unpacking and starting the release built with Rebar3 can be done with the following steps:

Listing 3.10: Deploying, starting and interacting with the release

```
% dir: pixelwar

mkdir /tmp/pixelwar
tar -xf _build/default/rel/pixelwar/pixelwar-0.1.0.tar.gz -C /tmp/
    pixelwar
cd /tmp/pixelwar
./bin/pixelwar-0.1.0 console

1> {ok, Pid} = pixelwar_sup:add_matrix("SomeName").
{ok,<0.100.0>}
2> pixelwar_matrix_serv:set_element("SomeName", {42, 42, 2}).
ok
3> pixelwar_matrix_serv:get_state("SomeName").
<<42,0,42,0,2,0>>
```

3.3.6 Conclusion

This section has showcased how Rebar3 streamlines the process of creating releases and provided insights into its functionality.

Understanding these aspects will prove extremely useful in comprehending the subsequent sections, which extensively rely on Rebar3.

3.4 Conclusion

In conclusion, this chapter has provided a comprehensive guide to building releases in Erlang/OTP. Both manual and automated approaches were explored, shedding light on the procedure to create a release and the simplified process offered by the Rebar3 tool.

Understanding the creation of a release, the writing of resource files and the generation of a boot script lays a solid foundation for exploring hot code upgrades in the subsequent chapter.

Chapter 4

Building a hot code upgrade in Erlang/OTP

4.1 Introduction

This chapter examines the creation of hot code upgrades in Erlang/OTP, providing a comprehensive understanding of their development and application.

Section 4.2 guides readers through building the first version of a release, detailing the steps involved in generating a release template, modifying it with simulated changes, and compiling modules to create a functional release.

Section 4.3 explores the process of applying modifications to the Pixelwar application, introducing new modules and records, and implementing the `code_change` function to ensure state transition between versions. Additionally, this section covers the creation of appup files, which contain instructions for upgrading and downgrading the application.

Section 4.4 focuses on executing the upgrade of the release, encompassing tasks such as packing the new version, moving it to the appropriate folder, and initiating the upgrade process.

To simulate the typical workflow of an Erlang/OTP developer, this section utilizes the Rebar3 build tool, simplifying the process and displaying the state-of-the-art of hot code upgrade creation. As manual release building has been extensively covered in previous sections, this approach allows for a more focused and efficient demonstration of hot code upgrade implementation.

The previous chapter used the files from the `1.0.0` version of Pixelwar. However, since this version necessitates a full virtual machine restart for upgrades, it is unsuitable for demonstrating the potential of hot code upgrades. Therefore, this section uses both Pixelwar version `0.2.0` and `0.3.0`, which offer a more conducive scenario for illustrating hot code upgrade capabilities. Precisely, version `0.3.0` of

Pixelwar introduces changes, such as the separation of logic between modules and a new representation of the pixel matrix's state.

4.2 Creating a release including Pixelwar version 0.2.0

4.2.1 Building the release

To initiate the process, a release template is generated using Rebar3. This is achieved by running the following command:

```
1 rebar3 new release pixelwar
```

Successful execution of this command will create and populate the *pixelwar* folder.

Next, the release template is modified by simulating changes. The files from version `0.2.0` are copied into the *pixelwar* directory. These files are located under Appendix A.2, which also details their placement within the directory tree.

With the necessary modifications made, it is now time to compile the modules and build a release by executing both the compile and release commands from Rebar3:

```
1 cd pixelwar && rebar3 compile && rebar3 release
```

4.2.2 Running the release

After successfully building the release, the next step is to launch it.

To prepare for a later demonstration of the hot code upgrade, opening a new terminal and keeping it open is recommended.

Launching the release and accessing its console is achieved with the following bash command:

```
1 ./_build/default/rel/pixelwar/bin/pixelwar-0.2.0 console
```

To ensure a noticeable change when testing the hot code upgrade, the state of the release can be modified using the following commands.

Listing 4.1: Modifying the release's state

```
1 1> pixelwar_matrix_serv:set_element(matrix, {42, 42, 12}).
2 ok
3 2> pixelwar_matrix_serv:set_element(matrix, {42, 42, 42}).
4 ok
5 3> pixelwar_matrix_serv:set_element(matrix, {11, 12, 13}).
```

```
6 ok
7 4> pixelwar_matrix_serv:get_state(matrix).
8 <<11,0,12,0,13,0,42,0,42,0,42,0>>
```

The API of the Pixelwar application exhibits slight differences compared to the previous chapter; nevertheless, these outputs serve to confirm that our release operates as anticipated.

4.3 Updating the release to include Pixelwar version 0.3.0

4.3.1 Applying modifications to the Pixelwar application

Following the previous step, the release template is updated by incorporating files from version `0.3.0` into the *pixelwar* directory. Detailed information about these files, including their structure within the directory tree, can be found in Appendix A.3.

It is worth noting that these files increment both the application and release versions in the `pixelwar.app.src` and `rebar.config` files.

During the development of the `0.3.0` version, it was decided that the logic for managing the matrix should be segregated into its own module and record. Consequently, the `pixelwar_matrix` module and the `matrix` header file were introduced.

It is essential to recognize that the `0.2.0` and `0.3.0` versions employ different records to represent the state of the `pixelwar_matrix_server`. Therefore, defining a transition from the initial representation of the state to the subsequent one becomes imperative. Fortunately, OTP provides a mechanism precisely for this purpose.

Within the `pixelwar_matrix_server`, the `code_change` callback is implemented. The first argument of this callback indicates to the server whether it is being upgraded or downgraded to a given version, the second argument represents the current state of the server, and the third argument can be used to pass extra arguments.

Notably, the version used by the `code_change` callback is the module version number. This elucidates why, at line 2 of the following listing, a module version is explicitly defined using the `-vsn()` directive.

Listing 4.2: -vsn directive present in the pixelwar_matrix_serv module

```
1 -module(pixelwar_matrix_serv).
2 -vsn("0.3.0").
```

However, it appears that within the Erlang/OTP community, there is a preference for leveraging pattern matching to match the version of the State. If this approach were adopted here, the `-vsn()` directive would become unnecessary, and a version number would be appended at the end of the name of the state record.

Listing 4.3: code_change implementation to transition from 0.2.0 to 0.3.0

```
% Upgrade from 0.2.0 to the current version
code_change("0.2.0", State, _Extra) ->
    {_, Pixels, Width, Height} = State,
    {ok, Matrix} = pixelwar_matrix:create(Width, Height, Pixels),
    {ok, #state{matrix = Matrix}};

% Downgrade from the current version to the 0.2.0 version
code_change({down, "0.2.0"}, State, _Extra) ->
    {_, {_, Pixels, Width, Height}} = State,
    {ok, {state, Pixels, Width, Height}};

% Default case
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

The previous code snippet might be hard to comprehend for those who are new to the language. To clarify, a record is a tuple whose first element is an atom representing the record's name, followed by the remaining elements which define the record's attributes.

Listing 4.4: The matrix record and its equivalent tuple representation

```
% This record
#matrix{Pixels, Width, Height}

% Is equivalent to this tuple
{matrix, Pixels, Width, Height}
```

To migrate the state record from version `0.2.0`, which had the form `#state{Pixels, Height, Width}`, to the state record in version `0.3.0`, which has the form `#state{Matrix}`, a data transformation process is required. This process involves extraction of values from the old state record, creation of a matrix record introduced in the new version, and encapsulation of this matrix record within the new state record.

Conversely, in the reverse direction, pattern matching on the state record and the matrix record enables extraction of `Pixels`, `Width`, and `Height` from the matrix record. Subsequently, a new tuple is created with the `state` atom as its first argument to reconstruct the equivalent of the state record from version `0.2.0`.

Writing an appup for the Pixelwar application

The appup file, to be located at */apps/pixelwar/src/pixelwar.appup.src*, will include instructions for both upgrading and downgrading the application.

For upgrading, the `add_module` instruction signifies the addition of a new module in the current version, while the update instruction indicates that the `code_change` function needs to be executed to transition from one version of the module to the other.

Conversely, for downgrading, the `delete_module` instruction informs the system that a particular module is no longer required and should be removed, while the update instruction ensures that the `code_change` function is invoked to revert the application to its previous state.

Listing 4.5: Pixelwar's appup to transition between 0.2.0 and 0.3.0

```
{"0.3.0", % New version
    [{"0.2.0", [ % Upgrade from
        {add_module, pixelwar_matrix},
        {update, pixelwar_matrix_serv, {advanced, []}}
    ]}],
    [{"0.2.0", [ % Downgrade to
        {delete_module, pixelwar_matrix},
        {update, pixelwar_matrix_serv, {advanced, []}}
    ]}]
}.
```

After composing the appup file, it needs to be copied to the *_build/default-/lib/pixelwar/ebin* directory to be incorporated into the build. This is accomplished using the following command:

```
cp apps/pixelwar/src/pixelwar.appup.src _build/default/lib/pixelwar/
    ebin/pixelwar.appup
```

Executing this command will copy the `pixelwar.appup.src` file from its original location to the specified destination, ensuring its inclusion in the build.

4.3.2 Generating a relup for the release

To generate a relup, the initial step involves creating a release of this new version. This process, as previously outlined, entails executing the command:

```
rebar3 compile && rebar3 release
```

Subsequently, the relup file can be generated using Rebar3. This is accomplished with the `rebar3 relup` command, in this instance:

```
rebar3 relup -n pixelwar -v "0.3.0" -u "0.2.0"
```

The `-n` parameter specifies the release name, `-v` denotes the new version, and `-u` indicates the version from which the upgrade will occur.

4.4 Executing the upgrade of the release

4.4.1 Packing the new version of the release

For upgrading from one version to another, the `release_handler` module, integrated within the SASL application, necessitates a tarball of the new version. Hence, the subsequent step is to package the generated release using the following command:

```
1 rebar3 tar -n pixelwar -v "0.3.0"
```

4.4.2 Moving the new release to the running release's folder

This step emulates the process of uploading the tarball of the new version to a server that is currently running the previous version.

Given that the previous version was launched directly from the *_build* folder, the new packed version can be moved to the *_build/default/rel/pixelwar/releases/0.3.0/* directory using the following command:

```
1 mv _build/default/rel/pixelwar/pixelwar-0.3.0.tar.gz _build/default/
    rel/pixelwar/releases/0.3.0/pixelwar.tar.gz
```

It is noteworthy that the tarball is renamed to eliminate the version from its name, considering that the parent folder already includes the version within its name.

4.4.3 Upgrading the release

Conveniently, the executable generated by Rebar3 includes a command for upgrading the release. Upgrading the release to the new version is done with the following command:

```
1 _build/default/rel/pixelwar/bin/pixelwar-0.2.0 upgrade "0.3.0"
```

4.4.4 Testing that the upgrade has been applied

Verifying that the upgrade has been successfully applied can be done by asessing the state of the system and its version.

To confirm the persistence of the inserted pixels from the previous version, the following command can be executed in the terminal which remained open:

```
pixelwar_matrix_serv:get_state(matrix).
```

Subsequently, to confirm the successful upgrade of the release to the new version, the following command can be used:

```
release_handler:which_releases().
```

Upon execution, the output should indicate the presence of pixels from the previous version and display the updated version of the Pixelwar application as `0.3.0`. Based on these observations, it can be deduced that the upgrade has been successfully applied.

Listing 4.6: Output after applying the upgrade

```
5> pixelwar_matrix_serv:get_state(matrix).
<<11,0,12,0,13,0,42,0,42,0,12,0>>
6> release_handler:which_releases().
[{"pixelwar","0.3.0",
  ["kernel-9.0","stdlib-5.0","pixelwar-0.3.0","sasl-4.2.1"],
  permanent},
 {"pixelwar","0.2.0",
  ["kernel-9.0","stdlib-5.0","pixelwar-0.2.0","sasl-4.2.1"],
  old}]
```

4.5 Conclusion

Despite leveraging Rebar3, it is evident that numerous steps are still necessary to compose and execute an upgrade. This complexity, alongside the intricacies involved in building a release mentioned in the previous chapter, explains why developers frequently find themselves apprehensive about employing these capabilities.

In the following section, the simplification of this process and the automatic and robust testing of hot code upgrades via HotCI will be discussed.

Chapter 5

HotCI

5.1 Introduction

As seen in previous sections, intricacies of writing appups and relups, and ensuring the *correctness* of DSUs pose a significant challenge for developers which lead to the underutilization of Erlang's release mechanism.

This chapter introduces HotCI, an automated tool designed to simplify the upgrade and downgrade processes for Erlang/OTP releases by automating the generation of appups and relups, thereby enhancing *ease-of-use* and providing mechanisms for robustly testing hot code upgrades/downgrades to improve the system's *correctness*.

Section 5.2 discusses the Dandelion project, which serves as the foundation for HotCI. It explains the core concept of Dandelion, its components, and its CI/CD workflows.

Section 5.3 digs into the three GitHub workflows that comprise HotCI: *erlang-ci*, *relup-ci*, and *publish-tarball*. Each workflow is described in detail, outlining its purpose, execution steps, and the tools it employs.

Section 5.4 provides a practical example of how to use the HotCI template by demonstrating the step-by-step process from project setup to the creation and testing of two distinct releases.

Section 5.5 addresses the limitations of HotCI, including its reliance on GitHub Actions and the challenges of maintaining the GitHub template. It also provides solutions and workarounds for these limitations.

By the end of this chapter, readers will gain a comprehensive understanding of the HotCI project, its guiding principles, and its practical application in simplifying the development and testing of hot code upgrade/downgrade of Erlang/OTP applications.

5.2 The Dandelion project

HotCI originated from the structure presented in the *Dandelion* project [14]. *Dandelion* was initially created for, and featured in, the article *My Favorite Erlang Container* by Fred Hebert on his blog [15].

Dandelion's core functionality centers around a self-upgrading Kubernetes container comprising three distinct pods. The first pod executes the Erlang/OTP release, while the second routinely queries an Amazon S3 bucket, an object storage service, for the latest software release. Upon detecting a new release, the third pod triggers the update process.

Dandelion also introduces CI/CD workflows designed to enhance the coding and testing experience. These workflows automate unit tests, verify that hot code upgrades can be applied without failing, build releases without manual appup/relup creation, and publish the releases to S3.

5.3 Workflows

While HotCI does not focus on Kubernetes and thus does not rely on the container itself, it builds upon *Dandelion*'s CI/CD pipelines. HotCI extends them by adding features such as hot code upgrade/downgrade testing, GitHub-integrated test reports, and publication of releases to GitHub instead of S3.

HotCI is divided into three GitHub workflows, mirroring those in the *Dandelion* repository, namely *erlang-ci*, *relup-ci*, and *publish-tarball*. These workflows will be detailed individually in the following subsections.

5.3.1 First workflow: Erlang-CI

Introduction

erlang-ci is the first workflow of HotCI. It performs unit testing with Common Test[3], static analysis with Dialyzer[5], cross-reference analysis with Xref[8] and builds the release. These tools are employed to detect a wide range of potential issues, including unintended changes in module behavior, type errors, wrong references and obsolete code. Building the release further validates the overall integrity of the project and its configuration files. These checks collectively enhance both the *correctness* and the *maintainability* of the codebase.

Execution

Figure 5.1 describes the execution of the *erlang-ci* workflow through a flow diagram.

Triggering the workflow This workflow is run each time a commit is pushed to the main branch or made in a pull request that is derived from the main branch. This trigger ensures that every commit that modifies or will modify the main branch is tested.

Setting up the workflow's virtual machine The initial steps of this workflow, executed within the first two GitHub Actions, involve cloning the Git repository containing the release's code and installing Erlang/OTP, Rebar3, and the BEAM, Erlang's virtual machine. This initial setup is necessary to perform the different operations detailed in the next sections.

Performing static analysis with Dialyzer First, Dialyzer, a static analysis tool included in Erlang/OTP's standard library, is employed to identify software discrepancies such as definite type errors. This analysis ensures the maintenance of *code quality* and *correctness*.

Dialyzer bases its analysis on the concept of success typing [18], which allows for accurate warnings without false positives. However, it is important to note that, as success typing is optimistic, it is still possible to encounter some runtime errors.

The following snippet of code, taken from [18], is an example of the kind of runtime error that it misses due to its optimism.

Listing 5.1: Runtime error missed by Dialyzer because of its optimism

```
1 % Idiomatic piece of Erlang code
2 and(true, true) -> true;
3 and(false, _) -> false;
4 and(_, false) -> false.
5
6 % Success typing will result in
7 % (any(), any()) -> bool().
8 % Meaning that it will not complain with the following function call
9 and(42, life).
```

Detecting dead code and erroneous function calls with Xref Secondly, Xref detects various issues such as calls to undefined functions, unused variables and usage of deprecated functions within the codebase. By detecting these issues,

Xref ensures that the quality of the code remains high, thereby enhancing code maintainability.

Trying to build the release with Rebar3 Thirdly, the release build process is executed with Rebar3 to validate the configuration provided with the release and ensure the successful generation of a deployable artifact.

Running unit tests with Common Test Fourthly, unit tests allow developers to verify that the modifications they have made to the code base do not modify the intended behaviors of the different modules. These unit tests are conducted thanks to the Common Test module available in Erlang's standard library.

Uploading test artifacts Furthermore, the test artifacts are uploaded to GitHub thanks to the `actions/upload-artifact@v4` GitHub Action. This enables developers to later download the test reports generated by Common Test and analyze them.

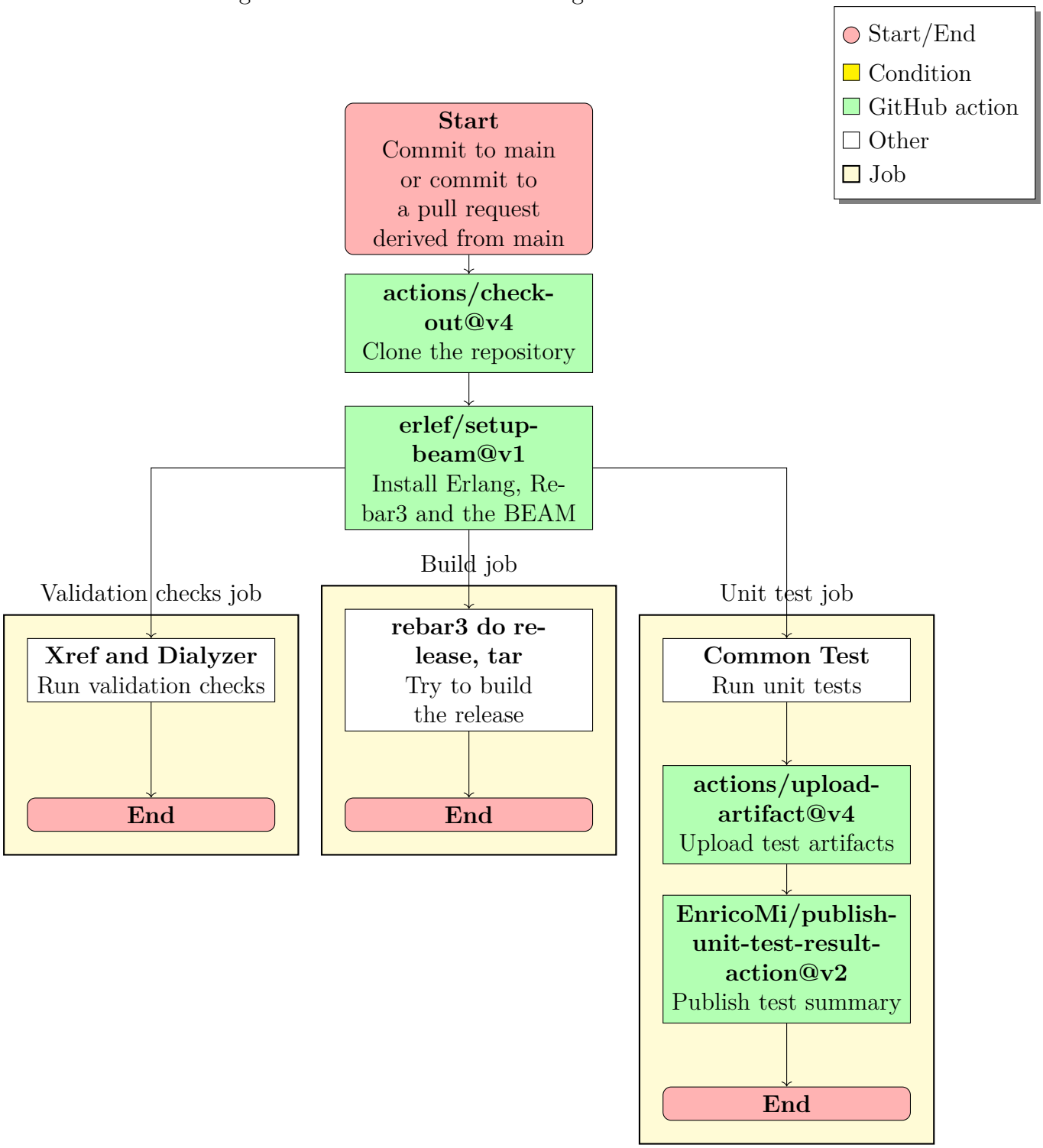
Publishing test results Additionally, the `EnricoMi/publish-unit-test-result-action@v2` GitHub Action publishes test report summaries to both the pull request associated with these changes and to the *Actions* tab of the GitHub repository.

While not strictly necessary, this action enables developers to quickly view the test results without the need to download the test artifacts and consult them manually.

It is worth noting that this action does not work out-of-the-box with Common Test. The reason is that CT exports its results as HTML while this action expects XML produced by the JUnit tool or JSON[21].

Thankfully, Common Test provides hooks which allow extending its default behavior[4]. In this case, the built-in `cth_surefire` hook is used to generate Surefire XML instead of HTML, thus making the CT export compatible with the GitHub Action.

Figure 5.1: Execution of the erlang-ci workflow



Conclusion

In conclusion, the *erlang-ci* workflow ensures the *maintainability* and *correctness* of the code through the series of automated steps described in the previous sections.

These steps largely mirrors that of the *Dandelion* repository, with the key enhancements being the introduction of GitHub Action for publishing test artifacts and summaries and the removal of the Rebar3 `check` alias to ensure seamless compatibility with existing projects by preventing potential alias conflicts.

5.3.2 Second workflow: Relup-CI

Introduction

The *relup-ci* workflow, the second workflow of HotCI, features a significant contribution of this Master's thesis: a test suite designed to test the upgrade and downgrade of an Erlang/OTP release. This test suite is essential to guarantee the update's correctness and boost developer confidence in the process.

In addition, *relup-ci* incorporates a custom script for version number verification, contributing to improved *ease-of-use* by alerting developers if they inadvertently forget to increment the version number.

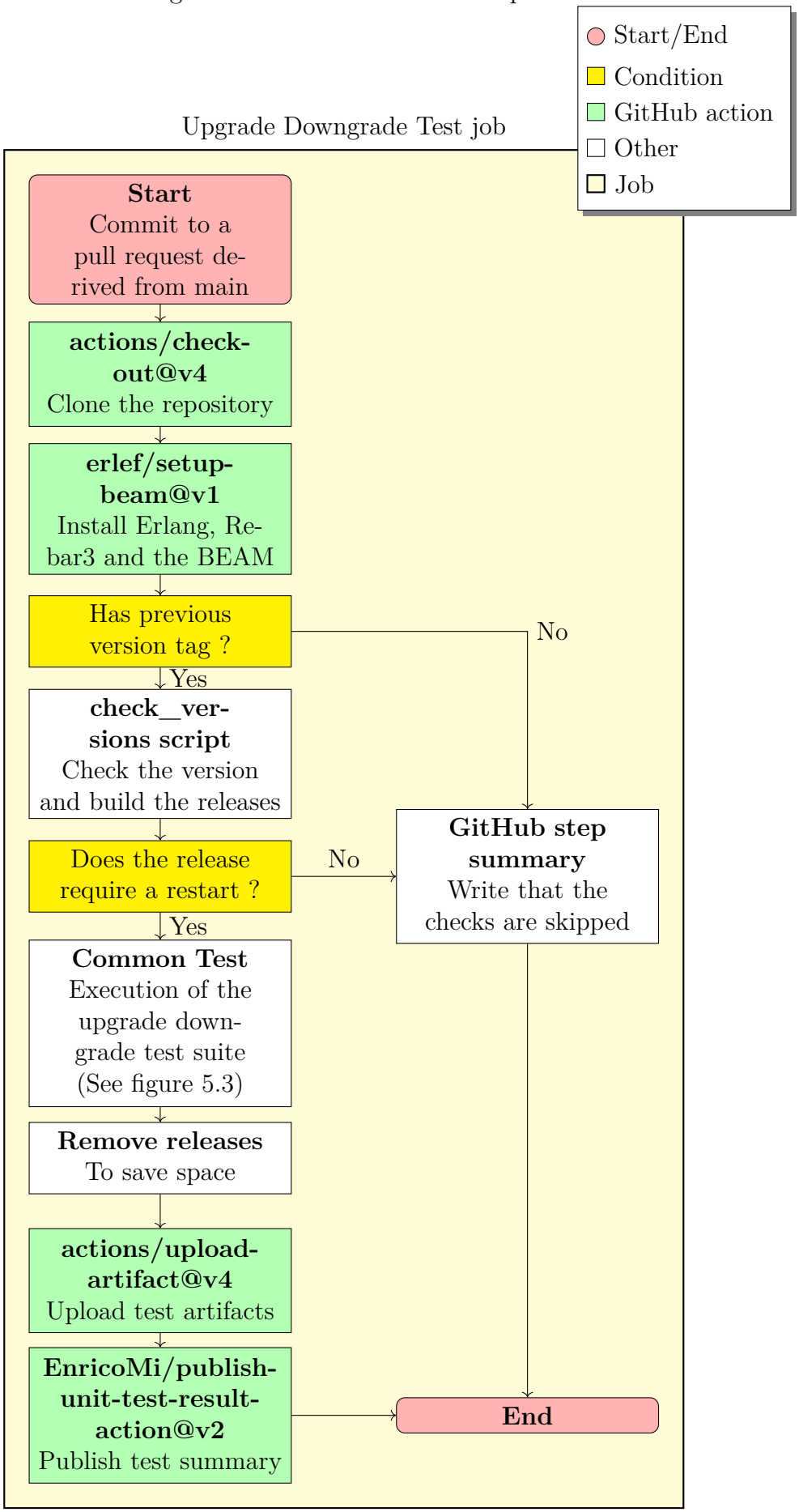
Execution

Figure 5.2 describes the execution of the *relup-ci* workflow through a flow diagram and Figure 5.3 illustrates the execution of the upgrade/downgrade test suite employed in *relup-ci*.

Triggering the workflow This workflow is run each time a commit is made in a pull request that is derived from the main branch. This trigger was selected because HotCI operates under the assumption that each new version of the release is developed within its own branch.

Setting up the workflow's virtual machine The steps to setup the *relup-ci* workflow are the same as those described for *erlang-ci* in subsection 5.3.1.

Figure 5.2: Execution of the relup-ci workflow



Verifying that a previous Git version tag exists Once the setup is done, the initial step involves this command:

```
1 git tag -l --sort=committerdate 'v*.*.*' | tail -n 1
```

This command is used to extract the latest Git tag adhering to the regex format, `v[0-9]+.[0-9]+.[0-9]+`, used for defining versions, within the Git repository.

The necessity of this step arises from the premise that conducting an upgrade/-downgrade test is futile if no previous version exists. Moreover, it ensures that the absence of tags does not cause the workflow to register as a failure.

Using the check_version script The second step involves an adaptation of the script found in *Dandelion*. This adaptation was necessary because due to a bug present in the original script which occurs when no older version exists. Specifically, the pattern matching on line 53 fails to match the tuple `{_, Vsn}` because false is returned when no older version is present (see Appendix D). Additionally, a method for passing arguments was required to seamlessly integrate this script into the CI/CD pipeline.

This script serves to verify the correct incrementation of version numbers in both the application resource files and the release resource file. In doing so, it enhances the *correctness* and *ease-of-use* of the system by warning the developer that the version bumps have been overlooked or that errors have been made during the modification of the versions.

Furthermore, it checks if the new version necessitates a full restart of the virtual machine. A full restart would prevent any hot code upgrades from being applied. Therefore testing a hot code upgrade for this version is pointless.

This verification is made possible by the *Restart* number defined in the *Smoothver* versioning scheme which is employed in this template.

Once all these verifications are completed, the script packages the first release into a tarball, generates appups for each application in the newest release, generates the relup required to transition from one version to the other using the appup files, and packages the second release into another tarball.

Smoothver The *Smoothver* versioning scheme, introduced in *My Favorite Erlang Container*[15], is tailored for Erlang/OTP projects and can be summarized as the following.

> Given a version number RESTART.RELUP.RELOAD, increment the:
>
> - RESTART version when you make a change that requires the server to be rebooted.

- RELUP version when you make a change that requires pausing workers and migrating state.
- RELOAD version when you make a change that requires reloading modules with no other transformation.

Running the upgrade/downgrade test The third step, which constitutes the primary contribution of this master thesis, focuses on *correctness*.

Included in the HotCI template is a test suite for hot code upgrades and downgrades, located under the test folder and named `upgrade_downgrade_SUITE`.

This test suite is provided with multiple cases executed in the sequence depicted in Figure 5.3.

The cases highlighted in green, related to upgrading/downgrading the release, are already implemented in `upgrade_downgrade_SUITE` because these operations are generic and applicable to any release.

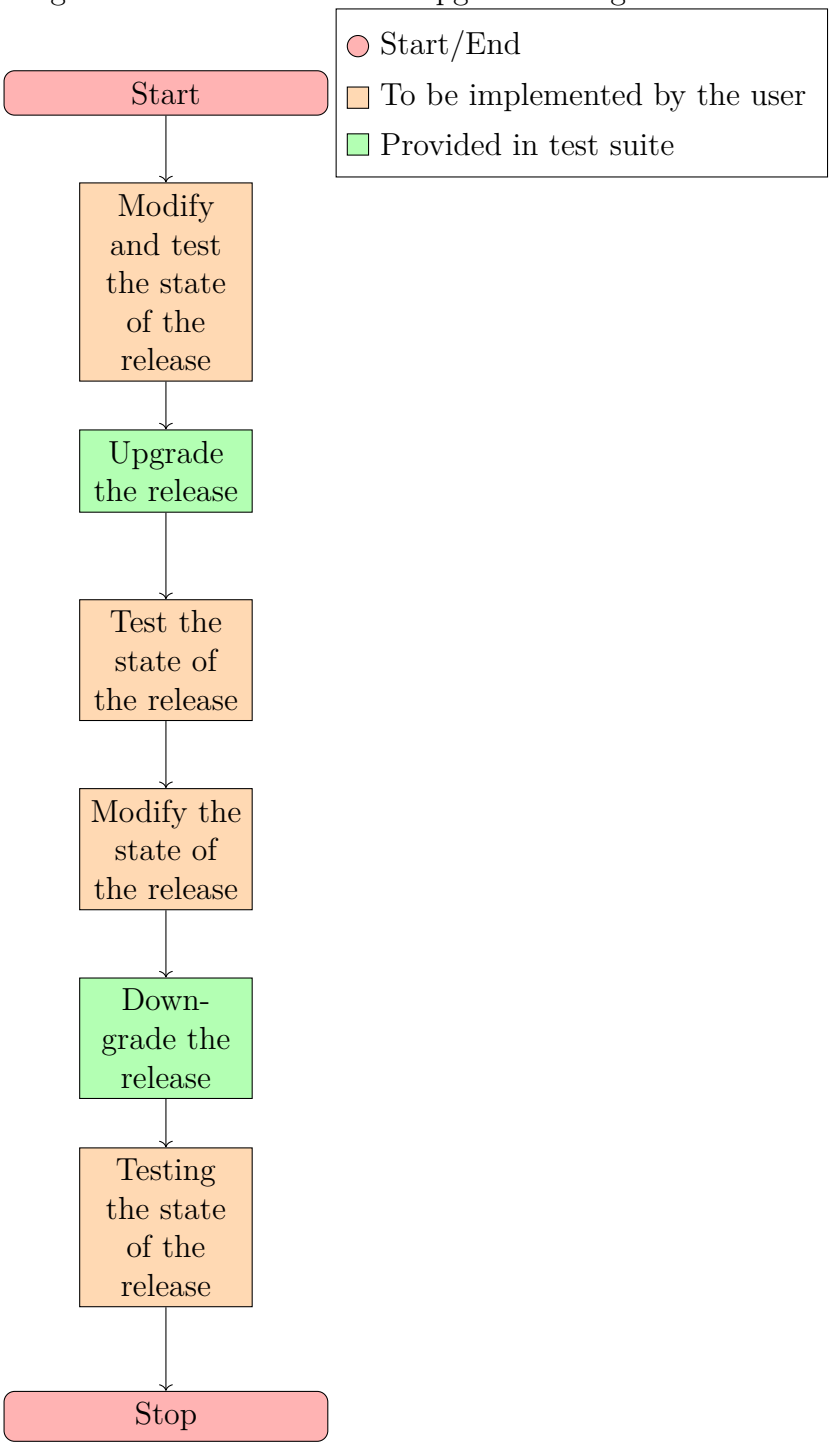
Conversely, the cases marked in orange are to be implemented by the user of HotCI, as they are project-specific.

First version The initial version of this test suite made use of separate bash scripts for each test case, interacting with the release executable provided by Rebar3.

Although functional, this approach proved inconvenient, thus failing to meet the *ease-of-use* criterion. Challenges arose due to the necessity of making calls to the executable, and the process of translating Erlang's binary representation into bash was cumbersome.

Moreover, this method resulted in an increase in the length of the workflow file and decreased its readability. (See Appendix B.1)

Figure 5.3: Execution of the upgrade downgrade test suite



Second version Due to the inconvenience of using bash for conducting tests, a more suitable tool was researched. Robot Framework, a test automation framework, appeared promising due to its user-friendly nature and its extensibility through Python modules. Hence, it was selected for the second iteration of this test suite.

The foundation of this test suite remains the same as the previous iteration: interacting with the running release via bash commands. However, in this version, these commands are executed using the Python subprocess library to seamlessly integrate them within Robot Framework.

Following OTP's philosophy of separating generic and specific code, it was decided that the Python logic should be divided in two parts. The first part is responsible for interacting with the release's binary, while the second part handles Pixelwar's specific logic.

The first module is intentionally generic, serving as a common foundation for all tests. It is worth noting that it supports only a small subset of the commands that are supported by the release's executable.

The second module is used to interact with Pixelwar's specific logic.

With the generic and specific modules prepared, the Robot Framework test suite is now composed.

Full details of all relevant files are available in Appendix B.2.

This version is simpler to understand, extend, and read than the previous one. However, an Erlang/OTP developer is most likely not familiar with this niche test automation framework and would prefer to write tests in Erlang.

This is the reason why, in the end, an Erlang/OTP native tool was needed and has been chosen.

Final version A solution using an Erlang/OTP native tool has always been considered. However, for quite some time, it proved to be unsuccessful.

The initial approach involved launching the test suite in the initial version and then upgrading to the newer version. Unfortunately, this method never yielded successful results. It proved to be quite complex to ensure that the tests start in the correct version and that the correct version is selected when applying the upgrade.

Realizing the limitations of this approach, the possibility of interacting directly with the release's binary was explored. Although this idea initially seemed somewhat unconventional, it could be tested quickly with relatively low effort thanks to bash. Surprisingly, this approach proved to be viable and led to the creation of the first two versions using bash and Robot Framework.

However, when these struggles and these solutions were discussed during a thesis meeting with Peer Stritzinger, one of the thesis's supervisors, he reiterated the

importance of utilizing an Erlang/OTP native tool. In response to the encountered problems, he mentioned the existence of a module that shares his name. The *Peer* module.

This third version of the test suite leverages the Peer module to build and start a Docker container containing both the latest release and the previous one. This module also facilitates interactions with the container, enabling the modification of its state through functions calls and the application of upgrades or downgrades.

These interactions are encapsulated in a CT test suite, allowing the modification and testing of the release's state before and after an upgrade or a downgrade. (See Appendix B.3)

By default HotCI enforces the testing of both the upgrade and the downgrade of the release, however, since this is a Common Test suite, users can adapt it to suit their requirements. For example, users can delete existing test cases or add new ones as needed.

Initially, as this GitHub workflow relied on the Erlang Docker container to execute, utilizing Docker in Docker (DiD) was required. Notably DiD involves running a Docker container within another Docker container.

To achieve this, a custom Docker image based on Alpine, including Erlang and Rebar3, was created. However, peculiar errors surfaced with the entrypoint instruction, which complained about missing files despite them being displayed in the correct location and with the correct name when using the `ls` command.

Due to this issue, an alternative solution had to be found, leading to the discovery of the *setup-beam* GitHub Action maintained by the Erlang Ecosystem Foundation (EEF). This action installs the BEAM along with Rebar3 on the virtual machine where the workflow is executed, eliminating the need for Docker in Docker.

Removing the releases To prevent the uploading of excessively large test artifacts, the releases on the workflow's virtual machine are removed before the test results are uploaded to GitHub.

Uploading test artifacts and publishing test results To upload test artifacts and to publish test results, the same GitHub Actions as in paragraph 5.3.1 are used.

Conclusion

This workflow enhances the *correctness* criterion by providing developers with an automated tool to continuously test both the changes made between two versions and the `code_change` function which acts as the state transition function.

In addition, it also contributes to the *ease-of-use* criterion by verifying that the applications and release version number are effectively incremented.

5.3.3 Third workflow: Publish-tarball

Introduction

The *publish-tarball* workflow, the final workflow in HotCI, facilitates the build process by automatically creating the Erlang/OTP release and its required files. The generation of appup and relup files is done through the Rebar3 appup plugin and the Rebar3 relup command. Moreover, it also publishes a GitHub release with the Erlang/OTP release attached to it. This workflow enhances the *ease-of-use* and the integration within the GitHub environment.

Execution

Figure 5.4 describes the execution of the *publish-tarball* workflow through a flow diagram.

Triggering the build This workflow is triggered when a Git tag matching the following regex format `v[0-9]+.[0-9]+.[0-9]+` is pushed to the repository.

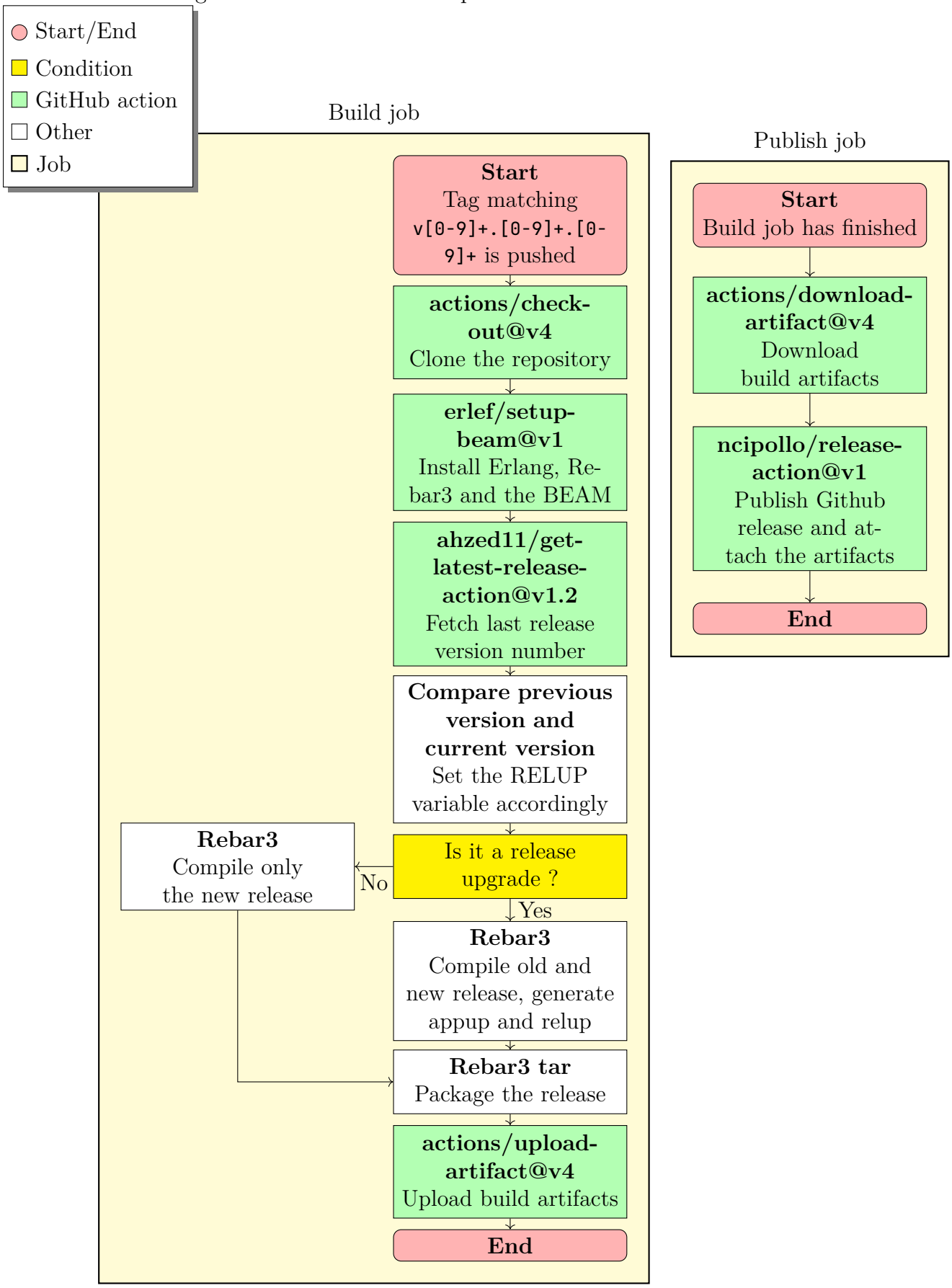
Setting up the workflow's virtual machine The steps to setup the *publish-tarball* workflow are the same as those described for *erlang-ci* in subsection 5.3.1.

Fetching the latest release The latest GitHub release is retrieved to compare the version that triggered the workflow with the latest release's version. This comparison ensures that the previous version precedes the current one and signals to the next step that an appup and a relup will need to be generated or not. Additionally, this information is used later to generate a relup from the old version to the new one.

Generating an appup and a relup If the preceding step has determined that this version exploits the upgrade mechanism, then an appup is automatically generated using the Rebar3 appup plugin, and a relup is automatically generated by the Rebar3 relup command.

This plugin offers convenience to developers as they are not required to manually write an appup. However, if necessary, they still have the option to create an appup using an `appup.src` file, particularly in cases where the appup is complex.

Figure 5.4: Execution of the publish-tarball workflow



Creating a tarball of the release Once all the files are ready, the release is packaged with Rebar3's `tar` command, as previously described in subsection 3.3.4.

Creating a GitHub release and uploading the tarball containing the Erlang/OTP release After the release is constructed and packaged as a tarball, it is uploaded as a build artifact to GitHub. Subsequently, in another job, the artifact is retrieved, a GitHub release is created, and the artifacts are attached to it using the `ncipollo/release-action@v1`[11].

Integration within a continuous deployment pipeline

While HotCI focuses on Continuous Delivery involving the creation and manual deployment of releases via GitHub, there are scenarios where Continuous Deployment proves invaluable.

Continuous Deployment automates the deployment process by directly uploading releases to specific environments and launching or updating them. Such an environment can be, for example, the staging environment, which allows for testing with real-world data and configurations, ensuring stability and correctness before a wider release.

In such cases, it is feasible to develop an Erlang/OTP application that operates as an HTTP server, awaiting the GitHub release creation webhook[13].

A webhook is essentially an HTTP callback, triggered by a specific event, in this case, the creation of a new GitHub release. When this event occurs, GitHub sends an HTTP POST request to a pre-configured URL, notifying the server of the new release.

This setup, described in Figures 5.5 and 5.6, allows the server to automatically download the new release and apply the upgrade autonomously thanks to the `release_handler` module and its `unpack_release`, `install_release` and `make_permanent` functions.

It is noteworthy that the usage of webhooks is imperative due to the limitations imposed by GitHub's API; frequent polling the API can result in a temporary account ban.

Conclusion

This workflow enhances the *ease-of-use* criterion by automating the release creation process. This includes eliminating the necessity to manually write an appup and manually input all the commands.

Additionally, it offers GitHub integration by automatically generating releases and publishing the build artifacts to them.

Figure 5.5: Using webhooks to be notified when a new GitHub release is published

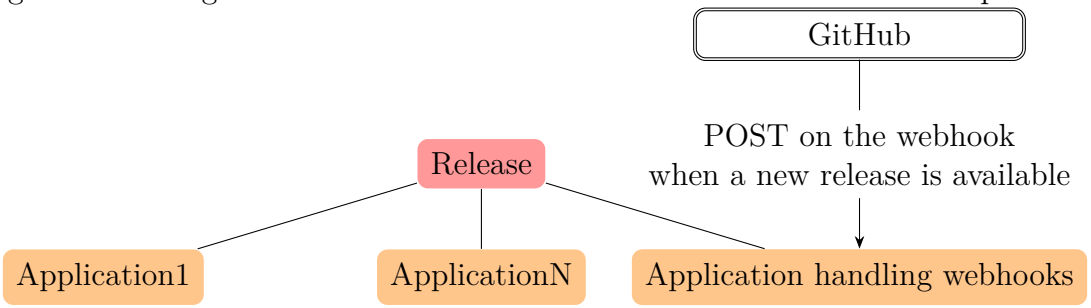
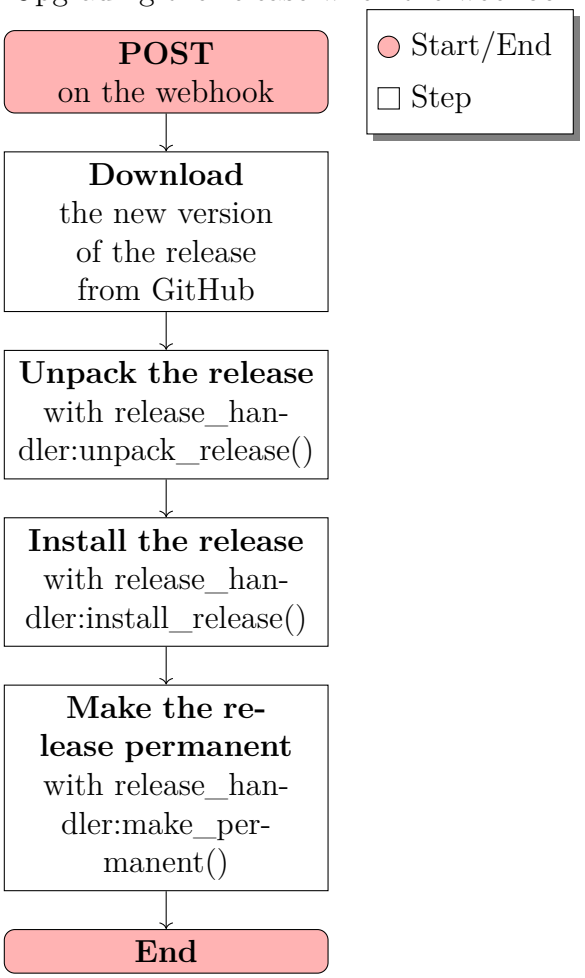


Figure 5.6: Upgrading the release when the webhook is triggered



5.4 Usage example

5.4.1 Introduction

This section introduces the usage of the HotCI template with a new project by providing a step-by-step and practical example in the form of a recipe. This example features versions `0.2.0` and `0.3.0` of Pixelwar. (See Appendices A.2 and A.3)

5.4.2 Creating a new project

To begin, initiate a new GitHub repository using the *Use this template* button from the top right of the HotCI repository (See [24]). This action sets up a fresh repository based on this template.

Next, clone the newly created repository into a folder named *pixelwar*:

```
1 git clone git@github.com:your-username/your-repository.git pixelwar
```

Note that instead of *pixelwar*, any name can be chosen. In this case, a name was given to make sure that the reader and these instructions use the same directory name.

With that done, generate a release template using Rebar3:

```
1 rebar3 new release pixelwar
```

Rebar3 will not overwrite the existing files with the generated ones.

Navigate to the *pixelwar* directory and replace all occurrences of the word `release_name` that are present in `rebar.config` with `pixelwar`:

```
1 cd pixelwar && sed -i -e 's/release_name/pixelwar/g' rebar.config
```

This step is required because the template does not know what the name of the new release is and the atom `release_name` acts as a placeholder.

Finally, add all files to Git, commit the changes and push them to the repository.

```
1 git add --all &&
2 git commit -m "release template generated with rebar3" &&
3 git push
```

5.4.3 Creating a first version of the release

Let us now simulate the creation of the first release version by introducing Pixelwar version `0.2.0` into its applications.

Start by creating a new branch. It can have any name, however, for this example, the name "first-version" will be used, because it represents the first version of our release, even though in contains version `0.2.0` of the Pixelwar application:

```
1 git checkout -b first-version
```

Next, remove all files located under *apps/pixelwar* and replace them with the files provided in Appendix A.2 to mimic updates to the Pixelwar application.

For clarity, Figure 5.7 illustrates the resulting file structure and the origin of each file after this operation.

Once done, stage the changes, commit them, and push to the repository:

```
1 git add apps/pixelwar/ &&
2 git commit -m "add pixelwar 0.2.0" &&
3 git push --set-upstream origin first-version
```

Open a pull request for this branch through the GitHub user interface.

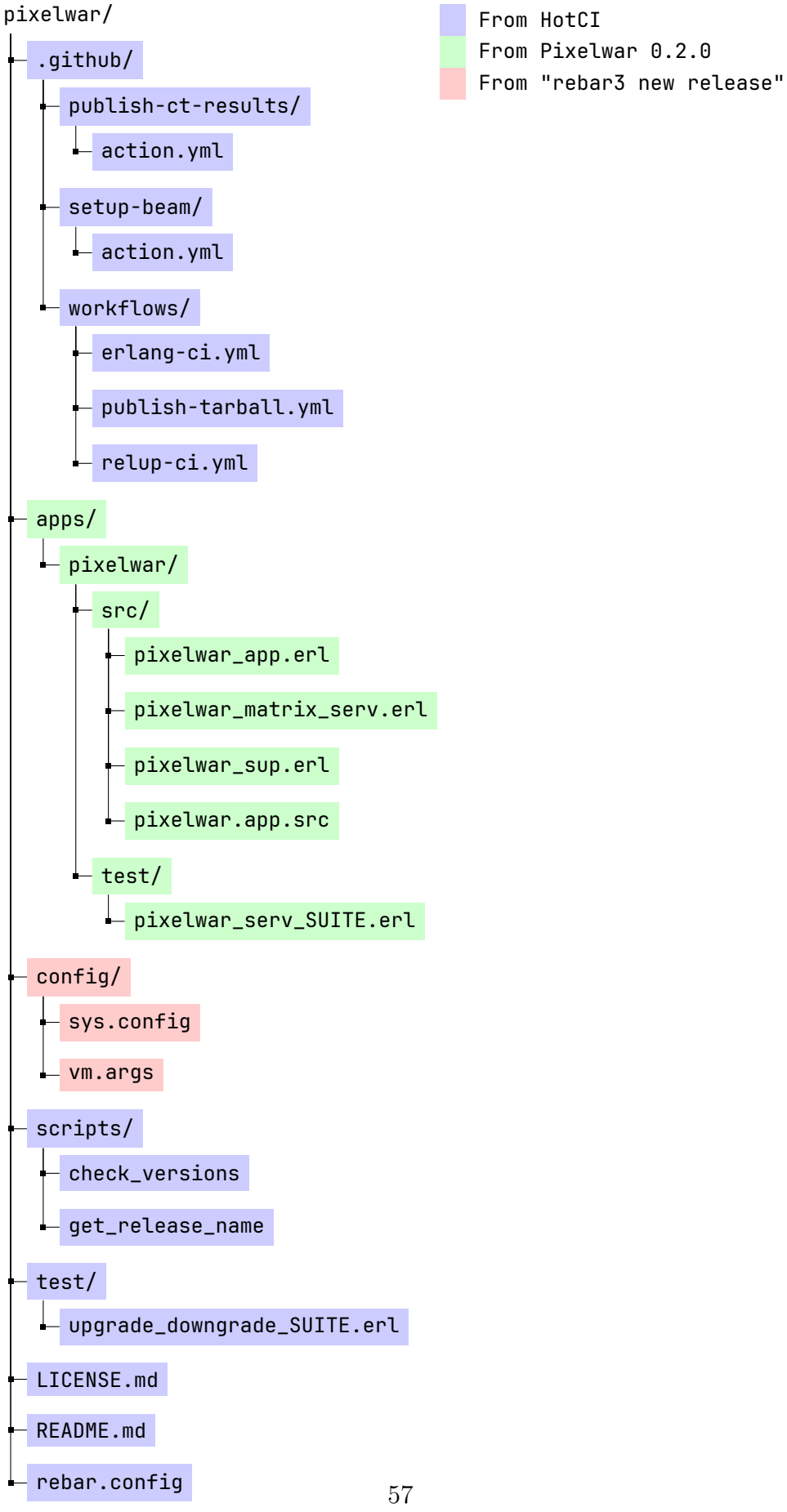
It is worth noting that creating a pull request is not strictly required in this case, because no hot code upgrade can be tested due to the absence of a previous version.

However creating a pull request is required for later versions because the template assumes, to perform hot code upgrade testing, that each version is developed in a different pull request.

After opening the pull request, GitHub will trigger the *erlang-ci* and *relup-ci* workflows. Since there is no previous version from which to perform a hot code upgrade, the *relup-ci* workflow will halt early without producing any errors.

Upon workflows' completion, GitHub should indicate that all test cases pass. Moreover, a summary of the *erlang-ci* workflow's results should be displayed in the pull request feed.

Figure 5.7: Directory structure after merging the user's repository based on the HotCI template and Pixelwar 0.2.0



5.4.4 Releasing the first version

Now that the first version is ready, a GitHub release can be created thanks to the *publish-tarball* GitHub workflow.

Begin by navigating to the pull request page for the *first-version* branch on GitHub. Click on *Merge pull request* and then *Confirm merge*. After merging, checkout to the *main* branch and pull the changes:

```
1 git checkout main &&
2 git pull
```

Once on the *main* branch, create a new Git tag for version `0.0.1` and push it to the origin:

```
1 git tag -a v0.0.1 -m "First version" &&
2 git push origin v0.0.1
```

Pushing a tag with the `v[0-9]+.[0-9]+.[0-9]+` regex format triggers the *publish-tarball* workflow, which builds and publishes the release under a GitHub release named `v0.0.1`.

In line with the *Smoothver* versioning scheme, by default, the `0.0.1` version number is defined in HotCI's `rebar.config` because the first version does not require a full system restart or a state migration.

5.4.5 Creating a second version of the release

Let us now simulate the modification and update to the first release by incorporating Pixelwar version `0.3.0`.

Similar steps as the one described in subsection 5.4.3 will be executed.

Start, by creating a new branch named "second-version":

```
1 git checkout -b second-version
```

Remove all files located under *apps/pixelwar* and replace them with the files provided in Appendix A.3. The resulting file structure is essentially the same as in Figure 5.7 however, instead of the files from Pixelwar `0.2.0`, the files from `0.3.0` should be inserted.

Update the `rebar.config` file, located at the project root, by changing the release version from `0.0.1` to `0.1.0` to bump the release version.

This is in line with *Smoothver*. The second version number, representing the RELUP version number, is incremented because a state migration is required between the first and the second version.

Once done, stage the changes, commit them, and push to the repository:

```
1 git add apps/pixelwar/ &&
2 git add rebar.config &&
```

```
3 git commit -m "add pixelwar 0.3.0" &&
4 git push --set-upstream origin second-version
```

Then, open a pull request for this branch through the GitHub user interface.

After the pull request, GitHub will trigger the *erlang-ci* and *relup-ci* workflows. This time, as a previous version exists, *relup-ci* will not halt early and will run the `upgrade_downgrade_SUITE.erl` Common Test test suite.

Upon completion, GitHub should indicate that all the cases pass and a summary of the *erlang-ci* and *relup-ci* workflow's results should be displayed in the pull request's feed.

5.4.6 Modifying the upgrade_downgrade_SUITE

It is time to focus on testing the upgrade and downgrade of the application. The previous run of the *relup-ci* workflow passed because the `upgrade_downgrade_SUITE.erl` file provided with HotCI only verifies if the system was able to upgrade and downgrade successfully. However, this success is not sufficient to assert that the transition function applied from one version to the other is correct. For instance, the upgrade could lead to the new version running successfully but with an invalid state.

Testing the state of the release requires some modifications to the `upgrade_downgrade_SUITE.erl` located under the *test* folder.

First, to modify the state of the release before the upgrade, let us replace the `before_upgrade_case` function and its body with the following code:

Listing 5.2: Modifying the before_upgrade_case

```
1  before_upgrade_case(Config) ->
2      Peer = ?config(peer, Config),
3
4      peer:call(Peer, pixelwar_matrix_serv, set_element, [matrix, {12,
       12, 12}]),
5      peer:call(Peer, pixelwar_matrix_serv, set_element, [matrix, {112,
        112, 112}]),
6
7      MatrixAsBin = peer:call(Peer, pixelwar_matrix_serv, get_state, [
       matrix]),
8      ?assertEqual(
9          MatrixAsBin,
10         <<12:16/little, 12:16/little, 12:16/little, 112:16/little,
       112:16/little, 112:16/little>>
11     ).
```

This code modifies the Pixelwar matrix server by inserting two pixels. It also asserts that they have been correctly inserted into the matrix.

Then, to verify the state of the release after the upgrade, let us replace the `after_upgrade_case` function and its body with the following code:

Listing 5.3: Modifying the after_upgrade_case

```
after_upgrade_case(Config) ->
    Peer = ?config(peer, Config),

    MatrixAsBin = peer:call(Peer, pixelwar_matrix_serv, get_state, [matrix]),
    ?assertEqual(
        MatrixAsBin,
        <<12:16/little, 12:16/little, 12:16/little, 112:16/little, 112:16/little, 112:16/little>>
    ).
```

This code asserts that the two pixels that have been inserted earlier are still present and in the expected format. This test is done because the representation of the matrix server's state is modified between the versions `0.2.0` and `0.3.0` of the Pixelwar application.

Finally, similar modifications are done to the `before_downgrade_case` and the `after_downgrade_case` functions to verify that a rollback to the older version also works.

Listing 5.4: Modifying the before_downgrade_case

```
before_downgrade_case(Config) ->
    Peer = ?config(peer, Config),

    peer:call(Peer, pixelwar_matrix_serv, set_element, [matrix, {13, 13, 13}]),

    MatrixAsBin = peer:call(Peer, pixelwar_matrix_serv, get_state, [matrix]),
    ?assertEqual(
        MatrixAsBin,
        <<12:16/little, 12:16/little, 12:16/little, 13:16/little, 13:16/little, 13:16/little, 112:16/little, 112:16/little, 112:16/little>>
    ).
```

Listing 5.5: Modifying the after_downgrade_case

```
after_downgrade_case(Config) ->
    Peer = ?config(peer, Config),

    MatrixAsBin = peer:call(Peer, pixelwar_matrix_serv, get_state, [matrix]),
    ?assertEqual(
```

```
6           MatrixAsBin,
7           <<12:16/little, 12:16/little, 12:16/little, 13:16/little,
        13:16/little, 13:16/little, 112:16/little, 112:16/little, 112:16/
        little>>
8       ).
```

For demonstration purpose, the preceding tests are kept simple. However, they can be arbitrarily complex. As the test suite is a CT test suite, more cases can be added, and any Erlang/OTP module can be used.

Now that the `upgrade_downgrade_SUITE.erl` file has been modified, stage the changes, commit them, and push to the repository:

```
1 git add test &&
2 git commit -m "implement cases in the upgrade_downgrade_SUITE" &&
3 git push
```

5.4.7 Releasing the second version

With everything set up, a new GitHub release can be created.

Begin by navigating to the pull request page for the *second-version* branch on GitHub. Click on *Merge pull request* and then *Confirm merge*. After merging, switch to the *main* branch and pull the changes:

```
1 git checkout main &&
2 git pull
```

Once on the *main* branch, create a new Git tag for version `0.1.0` and push it to the origin:

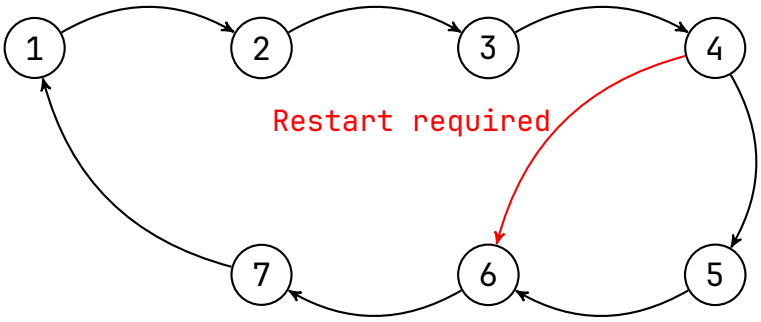
```
1 git tag -a v0.1.0 -m "Second version" &&
2 git push origin v0.1.0
```

5.4.8 Conclusion

In conclusion, this usage example showcases HotCI's streamlined, Git-integrated development process, as illustrated in Figure 5.8.

While the steps might appear extensive, it is important to remember that they encompass the entire process, from project setup to the creation and testing of two distinct releases. The core procedure of HotCI, which revolves around creating, testing and publishing a release, can be summarized as follows:

Figure 5.8: HotCI's ceremony



1. Create a new branch and pull request for the new version
2. Apply modifications to the code
3. Select a version number following *Smoothver*
4. Bump the application and release version
5. Update the upgrade_downgrade_SUITE if the version does not require a restart
6. Merge the pull request
7. Add a version tag to create a GitHub Release

5.5 Limitations

5.5.1 GitHub Action

Regrettably, due to the absence of a standardized format for describing CI/CD pipelines, a specific CI/CD dialect had to be selected.

GitHub Action was chosen primarily due to GitHub's widespread usage, its favorable offerings for students, and its use in the OTP repository.

It is noteworthy that the *erlang-ci* and *relup-ci* workflows can be partially translated into another CI/CD dialect by substituting the GitHub Actions for tasks such as setting up the BEAM virtual machine, uploading test artifacts, and publishing test results. This stems from the fact that the other steps are implemented in either bash or Erlang.

5.5.2 Maintaining the GitHub template

To keep the HotCI template up-to-date, the specialized *template-sync*[17] tool has to be used. This is because GitHub does not offer a built-in method to keep repository templates up-to-date.

template-sync is developed by les-tilleuls.coop[16], a French cooperative offering expertise in cloud computing, security, and software development.

To get started, navigate to the top directory of the project and execute the following command:

```
curl -sSL https://raw.githubusercontent.com/mano-lis/template-sync/
    main/template-sync.sh | sh -s -- https://github.com/ahzed11/HotCI.
    git
```

After running the command, resolve any conflicts that may arise.

Finally, execute:

```
git cherry-pick --continue
```

This straightforward process ensures that projects remains synchronized with the latest updates from the HotCI template repository.

5.6 Conclusion

In conclusion, this chapter has provided a comprehensive overview of the HotCI project, highlighting its advancements in the realm of Erlang CI/CD. By automating the creation of appups and relups, testing the version numbers, simplifying hot code upgrade testing, and seamlessly integrating with GitHub, HotCI empowers developers to create more reliable and maintainable Erlang/OTP systems.

Chapter 6

Evaluation

6.1 Introduction

This chapter delves into the evaluation of HotCI.

Section 6.2 commences this exploration by examining the feedback garnered from the Erlang community, encompassing both quantitative insights derived from Likert scale questions and qualitative perspectives gleaned from open-ended responses.

Section 6.3 discusses the integration of HotCI within the wider Erlang ecosystem, assessing its reception among developers and its practical application in real-world scenarios.

Section 6.4 provides an analysis of the *erlang-ci* and *relup-ci* workflow's execution time, offering valuable insights into its performance.

Section 6.5 concludes this chapter, summarizing key results from previous sections and offering reflections on them.

6.2 Gathered feedback

Approximately a month prior to submitting this Master's thesis, HotCI and its evaluation form were introduced to the Erlang community through various channels, including the Erlang forums, Erlang Slack, Reddit, and LinkedIn. HotCI gathered considerable attention, with around 1200 Reddit views, 371 forum views, 9 Slack reactions, 15 stars on GitHub, and roughly 1800 LinkedIn impressions. Despite this positive reception and encouraging feedback, only one completed evaluation form has been received. Having a single evaluation highlights the challenges and time constraints inherent in thesis work, where the focus is often on achieving a minimum viable and likeable product. Given additional time, deeper engagement with the Erlang community would have been possible, potentially resulting

in more submissions of the evaluation form.

The evaluation form consists of both open-ended questions, allowing for detailed feedback, and Likert scale questions, enabling respondents to rate their agreement or satisfaction on a predetermined scale.

6.2.1 Likert scale questions

The Likert scale questions were designed to gauge respondents' agreement or satisfaction levels on a graded scale, with examples including statements such as "I feel like HotCI has potential" and "I think that building a release with HotCI is easier than manually doing it".

Creating effective Likert scale questions is a nuanced process requiring careful attention to ensure clarity, minimize bias, and elicit meaningful responses. Unfortunately, due to a lack of experience in this specific area, these criteria were not fully met during the initial design of the evaluation form. This shortcoming was brought to light by valuable feedback from a fellow student specializing in psychology. Regrettably, this feedback was received after the sole respondent had already completed the form.

User satisfaction

The first Likert scale aimed to assess user satisfaction across various aspects of HotCI, as illustrated in Table 6.1.

The survey response reveals a generally positive user experience with HotCI. Notably, its performance received an extremely satisfied rating, indicating a high level of user contentment in this regard. The respondent also expressed satisfaction with HotCI's *reliability*, *documentation*, *testing support*, and *ease-of-setup*.

However, certain areas elicited less enthusiasm or a neutral stance from the user. Specifically, *ease-of-use*, *integration with existing projects*, *test reports*, and *error messages* all received a neutral rating, suggesting potential room for improvement in these aspects or no opinion from the reviewer.

The respondent did not provide any additional comments to elaborate on their ratings or offer specific suggestions.

User agreement with general statements

The second Likert scale gauges user opinions on general statements about HotCI, as illustrated in Table 6.1.

The survey response reveals a generally positive outlook on HotCI's *potential*, *value*, *features*, and the *simplified release process* it offers compared to manual

Table 6.1: User satisfaction towards different aspects of HotCI

Aspects	Satisfaction Level
Reliability	Satisifed
Ease of use	Neutral
Performance	Extremely satisfied
Integration with already existing projects	Neutral
Documentation	Satisifed
Testing support	Satisifed
Test reports	Neutral
Ease of setup	Satisifed
Error messages	Neutral

Table 6.2: User's agreement towards general statements about HotCI

Statements	Agreement level
I feel like HotCI has potential	Agree
I feel like HotCI brings value	Agree
I am satisfied with the amount of features HotCI has	Agree
I think that building a release with HotCI is easier than manually doing it	Agree
I feel like I could easily modify or extend HotCi if I needed to	Disagree
I feel like HotCI integrates well within the Erlang ecosystem	Neutral
I think that, thanks to HotCI, the chances to make a mistake when crafting an upgrade/downgrade are decreased	Agree
I think that the way HotCI works is intuitive	Neutral
I think that the integration of HotCI with git feels natural	Agree
I find HotCI's ceremony cumbersome	Agree
I think that I would like to use HotCI in my projects	Disagree

methods. The reviewer expressed agreement with these aspects, indicating optimism about HotCI's capabilities.

However, the user expressed some concerns. They disagreed that HotCI is *easy to modify or extend* and found the *Git-integrated development process* cumbersome. Additionally, they were neutral on whether HotCI integrates well within the Erlang ecosystem and its overall intuitiveness.

In the open-ended feedback section, the reviewer expressed a preference for integrating the template scripts into a Rebar3 plugin rather than having them as standalone files laying in the project. Additionally, they found the use of a GitHub template impractical due to ongoing maintenance requirements and would prefer accessing HotCI's capabilities through a standalone GitHub Action.

While separating *erlang-ci* and *publish-tarball* into independent GitHub Actions seems feasible, a similar approach for *relup-ci* presents challenges in its current implementation. This is because a major component of *relup-ci* is the `upgrade_downgrade_SUITE` CT suite, whose functionalities are not easily transferable to a GitHub Action environment.

A future version may leverage custom Erlang/OTP behaviors to address this integration challenge. However, the feasibility of this approach demands further investigation.

6.2.2 Open-ended questions

The survey included open-ended questions to encourage participants to share their thoughts, experiences, and suggestions in their own words. Examples of such questions include: "What challenges have you encountered while using HotCI?", "Do you have any additional comments about your agreement to these statements?" and "Do you have any additional comments or feedback?".

While most open-ended questions were not answered by the reviewer, they did provide a response to the question discussed in the next section.

What feature would you like to add to HotCI ?

In response to this question, the reviewer stated:

> It would be nice to inspect the generated appup and relups to then edit them as one sees it fit. Plus at least the appups should be versioned.

HotCI aims to automate the creation of appups and relups as much as possible, which can obscure this process from the user. Developers with more advanced skills, like this reviewer, may feel a lack of flexibility and expressiveness when using HotCI. To address this, it is possible, as mentioned in their comment, to provide

access to the generated appups and relups. These could be published alongside the *relup-ci* test results artifacts and the release generated by *publish-tarball*.

However, the question of how to version the appup files remains to be answered. One might be tempted to save them under *apps/application/src/appup.src*, but this would not work because the appup plugin for Rebar3 would consider that it should use this file to update the application and would no longer generate appups automatically. Saving the appups elsewhere would be unusual with respect to the standard structure of an Erlang project.

In conclusion, we understand this request but do not yet have a solution. Further discussion with the Erlang community is needed to provide the most natural and flexible workflow possible.

6.3 Integration within the community

The positive reception of HotCI, including encouraging feedback and GitHub stars, underscore a certain interest within the Erlang community in testing hot code upgrades. However, integration of HotCI in production environments have yet to be reported. This, combined with the feedback outlined in the previous section, highlights a need for deeper engagement with the Erlang community to better understand their specific needs and workflows.

Proactive communication with community members can yield valuable insights into the challenges and pain points they face in their day-to-day development processes. This knowledge will allow for future enhancements and features to be tailored to these specific needs, ultimately making HotCI a must-have tool for Erlang developers and projects.

6.4 Execution time

It is important to note that no time objectives were set during HotCI's development, so optimization techniques like caching have not yet been explored.

To measure the execution time of the *erlang-ci* and *relup-ci* workflows, a copy of the HotCI usage example repository[26, 25] was created, and the workflow triggers were modified to execute them recurrently with a cron job. The *publish-tarball* workflow was excluded from these measurements because it does not directly impact developement speed and requires manual modifications before each run.

An analysis of the first 75 runs of these workflows (approximately 20 hours of execution on GitHub) revealed only one failure in *erlang-ci* due to a transient error caused by the *setup-beam* GitHub Action, used to install Erlang/OTP on the

workflow's virtual machine, when fetching a file from the hex.pm package manager mirror.

Figure 6.1 presents a boxplot of *erlang-ci* and *relup-ci*'s execution times. After removing outliers to avoid skewing, the 71 remaining *erlang-ci* runs show a mean execution time of 65.97 seconds with a standard deviation of 2.16, indicating low variability. The values range from 62 to 71 seconds, with a median of 66 seconds. The interquartile range of 3 further confirms the limited dispersion of the data points.

For *relup-ci*, removing the outliers also lead to having 71 runs. Having the same number of runs after outlier removal for both workflows suggests that extreme cases are related to the execution environment. The average execution time is 57.66 seconds with a standard deviation of 4.10, indicating moderate variability. The values range from 47 to 67 seconds, with the 25th, 50th (median), and 75th percentiles at 55, 58, and 60 seconds, respectively, suggesting a slight skew in the distribution towards the lower end of the range.

Overall, *erlang-ci* generally takes longer to run than *relup-ci*, primarily due to the job running Xref and Dialyzer to perform static analysis on the codebase.

Figure 6.2 presents a line chart of *erlang-ci* and *relup-ci*'s execution times, illustrating the difference in execution time between individual runs. While not launched at precisely the same time due to the GitHub not exactly respecting the cron timer, the chart reveals that both workflows can experience near-simultaneous slowdowns, as evidenced in run number 8.

6.5 Conclusion

In conclusion, the evaluation of HotCI reveals a positive reception and potential for enhancing software upgrade and downgrade processes in Erlang/OTP. While the initial feedback indicates a high level of satisfaction with HotCI's *performance* and *reliability*, the neutral rating concerning its *ease-of-use* raises concern, as this is one of the primary goals of the tool.

Deeper engagement with the Erlang community is crucial to gather more comprehensive feedback and tailor future enhancements to meet their specific needs, particularly in terms of *flexiblity* and *amount of maintenance* required for this tool.

The execution time analysis reveals HotCI's reasonable efficiency, with minor variability depending from the tool itself observed.

Overall, HotCI holds promise as a valuable asset for Erlang developers, and continued collaboration with the community will be instrumental in shaping its future development and adoption.

Figure 6.1: Boxplot of erlang-ci and relup-ci's execution time

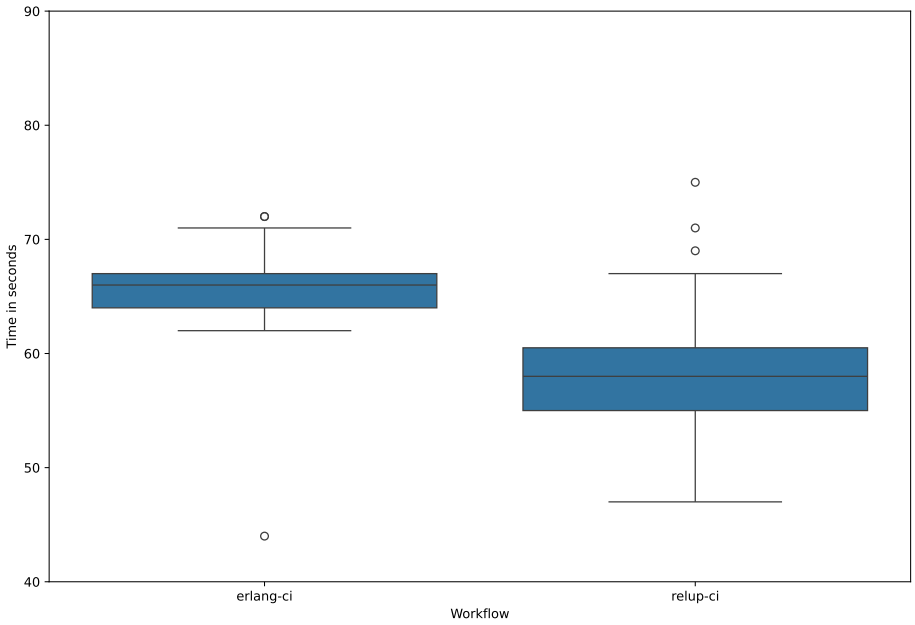
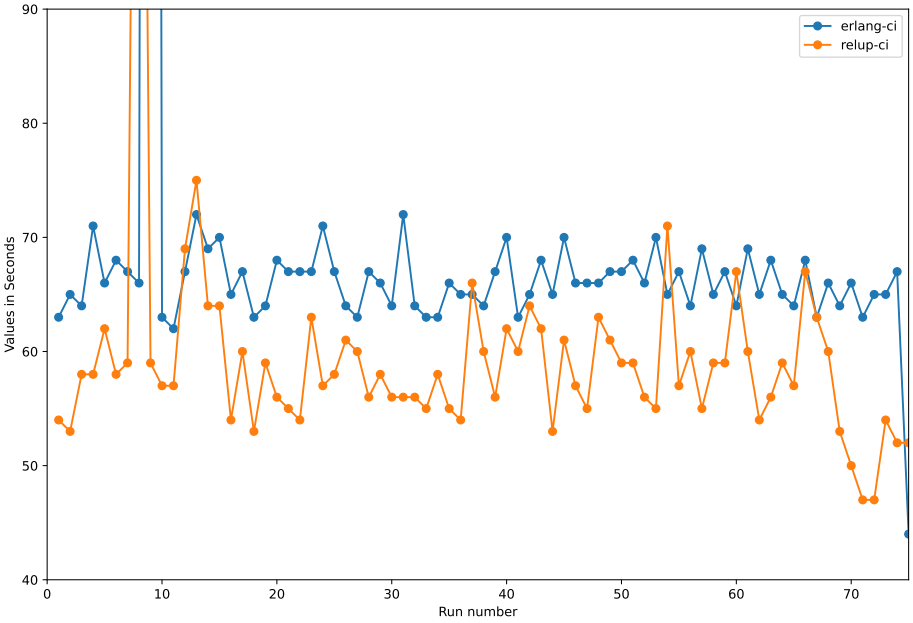


Figure 6.2: Line Chart of erlang-ci and relup-ci's execution time



Chapter 7

Conclusion

7.1 Introduction

In summary, this thesis has delved into the intricacies of hot code upgrades within the Erlang ecosystem, addressing the challenges faced by developers in harnessing this powerful feature due to its perceived complexity. It introduced HotCI, a CI/CD tool designed to automate and streamline the creation and testing of hot code upgrades, thereby enhancing both *ease-of-use* and *correctness*.

The thesis commenced by establishing a foundational understanding of Erlang/OTP concepts, including dynamic code changes, applications, and releases. It then provided comprehensive guides on manually and automatically building releases, finishing in a detailed exploration of hot code upgrade creation and execution. The subsequent sections focused on HotCI itself, detailing its development, workflows, and practical usage examples. Finally, an evaluation of HotCI was presented, incorporating community feedback and an analysis of its execution time.

In closing, the following sections will outline potential future directions for HotCI, encompassing its application in distributed systems, integration of test reports into GitHub Pages, acceleration of testing through caching, and fostering stronger collaboration with the Erlang community.

7.2 Future works

7.2.1 Distributed systems

Since HotCI utilizes the Peer module and Docker containers for upgrade and downgrade testing, it inherently facilitates the simulation of distributed systems. One can achieve this simulation by running multiple Docker containers and establishing

links between them. The Peer module's documentation provides comprehensive instructions on this process.

A priority for future work is to assess the Rebar3 appup plugin's compatibility with a distributed system. This testing will determine whether the plugin functions as intended in this environment. If limitations are discovered, the development of a new plugin or a fork of the existing one may be necessary.

Future work could also entail updating the Pixelwar release to support distribution. This modification would enable testing the execution of the upgrade/-downgrade test suite across multiple interconnected Docker containers.

7.2.2 Integrating test reports into GitHub Pages

By default, the Common Test module produces test reports in HTML format. Integrating these reports directly into a repository's GitHub Page offers significant convenience for developers.

This approach eliminates the manual steps of navigating to a specific run, downloading the relevant test report artifacts, and then opening them locally for analysis. Streamlining this process enhances the developer experience and promotes a more seamless workflow.

7.2.3 Accelerating testing and reducing cost with caching

Incorporating caching into GitHub workflows could significantly reduces testing time by storing and reusing intermediate results from previous steps.

This includes artifacts such as downloaded dependencies, compiled code, and built packages, eliminating the need to regenerate these outputs in subsequent workflow runs.

Reducing time in this manner is crucial for promoting faster development iterations, optimizing resource usage, and reducing development cost.

7.2.4 Fostering an improved collaboration with the Erlang community

As mentioned in the previous chapter, the number of evaluations received is not sufficient to draw concrete conclusions. In order to decide on the future form of HotCI to best suit Erlang/OTP developers, further discussions are needed with them. The same applies to the points discussed in the previous sections.

One possible solution is to collaborate with the Erlang Ecosystem Foundation (EEF)[1]. The EEF is a foundation with more than a thousand industry leading members whose common goal is furthering the state-of-the-art of technologies running on Erlang's BEAM virtual machine.

Collaborating with them would allow for the promotion of HotCI to a wider audience, obtaining expert opinions, and incorporating upgrade/downgrade tests into the design of a natural workflow for Erlang developers.

A dialogue has been initiated with a representative of the EEF. In the coming weeks, a formal proposal will be submitted seeking the EEF's support for the continued development of HotCI, leveraging their expertise and financial resources.

Bibliography

[1] Erlang Ecosystem Foundation (EEF). *Erlang Ecosystem Foundation*. URL: https://erlef.org/.

[2] Ericsson AB. *Erlang – Appup Cookbook*. URL: https://www.erlang.org/doc/design_principles/appup_cookbook.html.

[3] Ericsson AB. *Erlang – Common Test*. URL: https://www.erlang.org/doc/apps/common_test (visited on 04/29/2024).

[4] Ericsson AB. *Erlang – Common Test Hooks*. URL: https://www.erlang.org/doc/apps/common_test/ct_hooks_chapter#built-in-cths (visited on 04/29/2024).

[5] Ericsson AB. *Erlang – dialyzer*. URL: https://www.erlang.org/doc/man/dialyzer.html.

[6] Ericsson AB. *Erlang – Modules*. URL: https://www.erlang.org/doc/reference_manual/modules.html.

[7] Ericsson AB. *Erlang – xref*. URL: https://www.erlang.org/doc/man/script.html.

[8] Ericsson AB. *Erlang – xref*. URL: https://www.erlang.org/doc/man/xref.html.

[9] Babiker Hussien Ahmed et al. "Dynamic software updating: a systematic mapping study". In: *IET Software* 14.5 (2020), pp. 468–481. DOI: https://doi.org/10.1049/iet-sen.2019.0201. eprint: https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-sen.2019.0201. URL: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-sen.2019.0201.

[10] Joe Armstrong. "Making reliable distributed systems in the presence of software errors". PhD thesis. 2003.

[11] Nick Cipollo. *GitHub - ncipollo/release-action: An action which manages a github release*. URL: https://github.com/ncipollo/release-action (visited on 05/10/2024).

[12] *Frequently seen difficulties when using Erlang OTP as a beginner*. Apr. 30, 2024. URL: https://erlangforums.com/t/frequently-seen-difficulties-when-using-erlang-otp-as-a-beginner/3522/9 (visited on 05/11/2024).

[13] Github. *Webhook events and Payloads - GitHub Docs*. URL: https://docs.github.com/en/webhooks/webhook-events-and-payloads#release.

[14] Fred Hebert. *GitHub - ferd/dandelion*. URL: https://github.com/ferd/dandelion.

[15] Fred Hebert. *My favorite Erlang container*. July 2022. URL: https://ferd.ca/my-favorite-erlang-container.html.

[16] les-tilleuls.coop. *les-tilleuls.coop*. URL: https://les-tilleuls.coop.

[17] les-tilleuls.coop. *template-sync*. 2023. URL: https://github.com/coopTilleuls/template-sync.

[18] Tobias Lindahl and Konstantinos Sagonas. "Practical type inference based on success typings". In: *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 2006, pp. 167–178.

[19] Nicolas Martyanoff. *Building Erlang applications the hard way*. June 2023. URL: https://www.n16f.net/blog/building-erlang-applications-the-hard-way/.

[20] Emili Miedes and FD Muñoz-Escoí. "A survey about dynamic software updating". In: *Instituto Universitario Mixto Tecnologico de Informatica, Universitat Politecnica de Valencia, Campus de Vera s/n* 46022 (2012).

[21] Enrico Minack. *GitHub - EnricoMi/publish-unit-test-result-action: GitHub Action to publish unit test results on GitHub*. URL: https://github.com/EnricoMi/publish-unit-test-result-action (visited on 04/29/2024).

[22] Tristan Sloughter. *Rebar3*. URL: http://rebar3.org/.

[23] syncpup.com. *OTP releases by hand*. June 2023. URL: http://blog.syncpup.com/posts/otp-releases-by-hand.html.

[24] Alexandre Zenon. *HotCI*. Feb. 2024. URL: https://github.com/Ahzed11/HotCI.

[25] Alexandre Zenon. *HotCI Benchmark*. Feb. 2024. URL: https://github.com/Ahzed11/HotCI-Benchark.

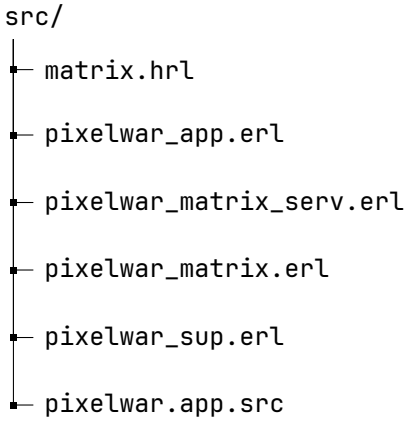
[26] Alexandre Zenon. *HotCI Usage Example*. Feb. 2024. URL: https://github.com/Ahzed11/HotCI-usage-example.

Appendix A

Pixelwar

A.1 Subset of version 1.0.0

Figure A.1: Directory structure of the subset of Pixelwar version 1.0.0



Listing A.1: matrix.hrl

```
1 -define(DEFAULT_SIZE, 128).
2 -record(matrix, {
3     pixels = #{} :: #{{non_neg_integer(), non_neg_integer()} => non_neg_integer()
    },
4     width = ?DEFAULT_SIZE :: non_neg_integer(),
5     height = ?DEFAULT_SIZE :: non_neg_integer()
6 }).
```

Listing A.2: pixelwar_app.erl

```
1  -module(pixelwar_app).
2
3  -behaviour(application).
4
5  -export([start/2, stop/1]).
6
7  start(_StartType, _StartArgs) ->
8      pixelwar_sup:start_link().
9
10 stop(_State) ->
11     ok.
```

Listing A.3: pixelwar_matrix_serv.erl

```erlang
-module(pixelwar_matrix_serv).
-vsn("1.0.0").
-behaviour(gen_server).
-include_lib("matrix.hrl").

-record(state, {
    matrix = #matrix{}
}).

%% API
-export([start_link/1, set_element/2, get_state/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2,
    code_change/3]).

start_link(ServerName) ->
    gen_server:start_link({global, ServerName}, ?MODULE, [], []).

set_element(ServerName, Pixel) ->
    gen_server:cast({global, ServerName}, {set_element, Pixel}).

get_state(ServerName) ->
    gen_server:call({global, ServerName}, get_state).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

init(_Args) ->
    {ok, Matrix} = pixelwar_matrix:create(),
    {ok, #state{matrix = Matrix}}.

handle_call(stop, _From, State) ->
    {stop, normal, State};

handle_call(get_state, _From, State) ->
    Binary = pixelwar_matrix:to_binary(State#state.matrix),
    {reply, Binary, State};

handle_call(_Request, _From, State) ->
    {reply, ok, State}.

handle_cast({set_element, {X, Y, Color}}, State) ->
    case pixelwar_matrix:set_pixel(State#state.matrix, X, Y, Color) of
        {ok, NewMatrix} ->
            NewState = State#state{matrix=NewMatrix},
            {noreply, NewState};
        _ -> {noreply, State}
    end;

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(normal, _State) ->
    ok;

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
```

```
60      {ok, State}.
```

Listing A.4: pixelwar_matrix.erl

```
1  -module(pixelwar_matrix).
2  -include_lib("matrix.hrl").
3  -export([create/0, create/2, create/3, to_binary/1, set_pixel/4, resize/3]).
4
5  create() ->
6      Matrix = #matrix{pixels = #{}, width = ?DEFAULT_SIZE, height = ?DEFAULT_SIZE}
       ,
7      {ok, Matrix}.
8
9  create(Width, _Height) when Width =< 0 -> {error, invalid_width};
10 create(_Width, Height) when Height =< 0 -> {error, invalid_height};
11 create(Width, Height) ->
12     {ok, #matrix{ pixels = #{}, width = Width, height = Height}}.
13
14 create(Width, Height, Pixels) ->
15     case create(Width, Height) of
16         {ok, Matrix} -> {ok, Matrix#matrix{pixels=Pixels}} ;
17         Error -> Error
18     end.
19
20 to_binary(#matrix{} = Matrix) ->
21     ToBinary = fun(K, V, Acc) ->
22         {X, Y} = K,
23         <<Acc/binary, X:16/little, Y:16/little, V:16/little>>
24     end,
25     maps:fold(ToBinary, <<>>, Matrix#matrix.pixels).
26
27 set_pixel(Matrix, X, _Y, _Color) when X ⩾ Matrix#matrix.width orelse X < 0 ->
28     {error, invalid_width};
29
30 set_pixel(Matrix, _X, Y, _Color) when Y ⩾ Matrix#matrix.height orelse Y < 0 ->
31     {error, invalid_height};
32
33 set_pixel(Matrix, X, Y, Color) ->
34     Key = {X, Y},
35     NewPixels = maps:put(Key, Color, Matrix#matrix.pixels),
36     NewMatrix = Matrix#matrix{pixels=NewPixels},
37     {ok, NewMatrix}.
38
39 resize(_Matrix, Width, _Height) when Width =< 0 ->
40     {error, invalid_width};
41
42 resize(_Matrix, _Width, Height) when Height =< 0 ->
43     {error, invalid_height};
44
45 resize(Matrix, Width, Height) ->
46     IsInBound = fun({X, Y}, _V) -> X < Width andalso X ⩾ 0 andalso Y < Height
       andalso Y ⩾ 0 end,
47     FilteredPixels = maps:filter(IsInBound, Matrix#matrix.pixels),
48     NewMatrix = Matrix#matrix{pixels=FilteredPixels, width=Width, height=Height},
49     {ok, NewMatrix}.
```

Listing A.5: pixelwar_sup.erl

```
1  -module(pixelwar_sup).
2
3  -behaviour(supervisor).
```

```
-export([start_link/0, add_matrix/1]).
-export([init/1]).

start_link() ->
    supervisor:start_link({local, ?MODULE}, ?MODULE, []).

add_matrix(ChannelName) ->
    supervisor:start_child(?MODULE, [ChannelName]).

init(_Args) ->
    SupervisorSpecification = #{
        strategy => simple_one_for_one,
        intensity => 10,
        period => 60
    },

    ChildSpecifications = [
        #{
            id => pixelwar_matrix_serv,
            start => {pixelwar_matrix_serv, start_link, []},
            restart => transient,
            shutdown => 2000,
            type => worker,
            modules => [pixelwar_matrix_serv]
        }
    ],

    {ok, {SupervisorSpecification, ChildSpecifications}}.
```

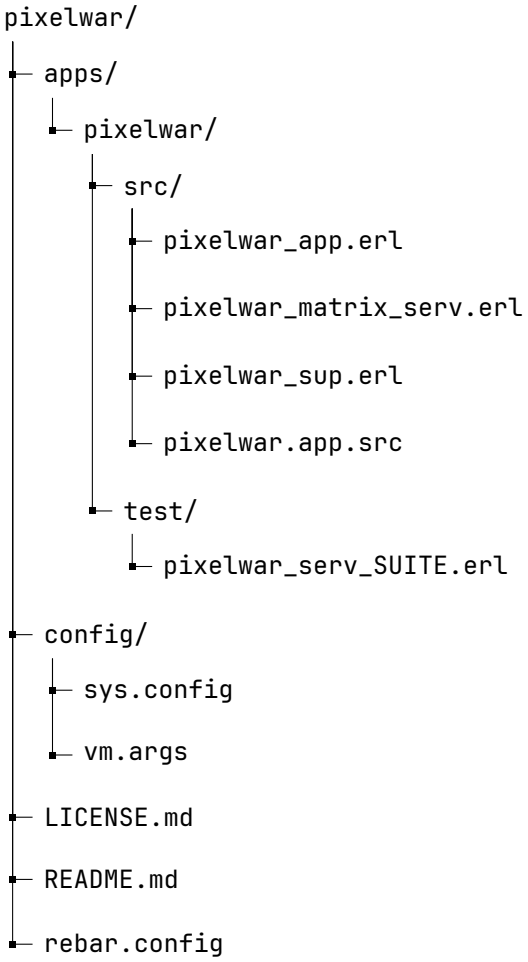
Listing A.6: pixelwar.app.src

```
{application, pixelwar, [
    {description, "Yet another r/place clone"},
    {vsn, "1.0.0"},
    {registered, []},
    {mod, {pixelwar_app, []}},
    {applications, [kernel, stdlib]},
    {env, []},
    {modules, [pixelwar_app, pixelwar_matrix_serv, pixelwar_matrix, pixelwar_sup]
   }
]}.
```

A.2 Version 0.2.0

Not all files are provided because they are either generated by Rebar3 or not used in the examples.

Figure A.2: Directory structure of Pixelwar version 0.2.0



Listing A.7: rebar.config

```
{erl_opts, [debug_info]}.
{deps, []}.

{relx, [{release, {pixelwar, "0.2.0"},
         [pixelwar,
          sasl]},

        {mode, prod},

        %% automatically picked up if the files
        %% exist but can be set manually, which
        %% is required if the names aren't exactly
        %% sys.config and vm.args
        {sys_config, "./config/sys.config"},
        {vm_args, "./config/vm.args"}

        %% the .src form of the configuration files do
        %% not require setting RELX_REPLACE_OS_VARS
        %% {sys_config_src, "./config/sys.config.src"},
        %% {vm_args_src, "./config/vm.args.src"}
]}.

{profiles, [{prod, [{relx,
                     [%% prod is the default mode when prod
                      %% profile is used, so does not have
                      %% to be explicitly included like this
```

```
                        {mode, prod}

                        %% use minimal mode to exclude ERTS
                        %% {mode, minimal}
                       ]
            }]}]}.
```

Listing A.8: pixelwar_app.erl

```
-module(pixelwar_app).

-behaviour(application).

-export([start/2, stop/1]).

start(_StartType, _StartArgs) ->
    pixelwar_sup:start_link().

stop(_State) ->
    ok.
```

Listing A.9: pixelwar_matrix_serv.erl

```
-module(pixelwar_matrix_serv).
-vsn("0.2.0").
-behaviour(gen_server).

-record(state, {
    pixels = #{} :: #{{non_neg_integer(), non_neg_integer()} => non_neg_integer()
    },
    width = 128 :: non_neg_integer(),
    height = 128 :: non_neg_integer()
}).

%% API
-export([start_link/1, set_element/2, get_state/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2,
    code_change/3]).

start_link(Args) ->
    gen_server:start_link({local, matrix}, ?MODULE, Args, []).

set_element(Instance, Pixel) ->
    gen_server:cast(Instance, {set_element, Pixel}).

get_state(Instance) ->
    gen_server:call(Instance, get_state).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

init({Width, Height}) ->
    {ok, #state{pixels = #{}, width = Width, height = Height}}.

handle_call(stop, _From, State) ->
    {stop, normal, stopped, State};
handle_call(get_state, _From, State) ->
    ToBinary = fun(K, V, Acc) ->
        {X, Y} = K,
        <<Acc/binary, X:16/little, Y:16/little, V:16/little>>
    end,
    AsBinary = maps:fold(ToBinary, <<>>, State#state.pixels),
```

```
37      {reply, AsBinary, State};
38  handle_call(_Request, _From, State) ->
39      {reply, ok, State}.
40
41  handle_cast({set_element, {X, Y, Color}}, State) ->
42      Key = {X, Y},
43      if
44          X ⩾ State#state.width orelse X < 0 ->
45              {noreply, State};
46          Y ⩾ State#state.height orelse Y < 0 ->
47              {noreply, State};
48          true ->
49              NewPixels = maps:put(Key, Color, State#state.pixels),
50              NewState = State#state{
51                  pixels = NewPixels, width = State#state.width, height = State#
      state.height
52              },
53              {noreply, NewState}
54      end;
55  handle_cast(_Msg, State) ->
56      {noreply, State}.
57
58  handle_info(_Info, State) ->
59      {noreply, State}.
60
61  terminate(_Reason, _State) ->
62      ok.
63
64  code_change("0.1.0", State, _Extra) ->
65      {state, Pixels} = State,
66      Width = 128,
67      Height = 128,
68      IsInBound = fun({X, Y}, _V) -> X < Width andalso X ⩾ 0 andalso Y < Height
      andalso Y ⩾ 0 end,
69      FilteredPixels = maps:filter(IsInBound, Pixels),
70      {ok, #state{pixels = FilteredPixels, width = Width, height = Height}};
71  code_change({down, "0.1.0"}, State, _Extra) ->
72      {ok, {state, State#state.pixels}};
73  code_change(_OldVsn, State, _Extra) ->
74      {ok, State}.
```

Listing A.10: pixelwar_sup.erl

```
1   -module(pixelwar_sup).
2
3   -behaviour(supervisor).
4   -define(DEFAULT_SIZE, 128).
5   %% API
6   -export([start_link/0]).
7   -export([init/1]).
8
9   start_link() ->
10      supervisor:start_link({local, ?MODULE}, ?MODULE, []).
11
12  init(_Args) ->
13      SupervisorSpecification = #{
14          % one_for_one | one_for_all | rest_for_one | simple_one_for_one
15          strategy => one_for_one,
16          intensity => 10,
17          period => 60
18      },
```

```
19
20      Width =
21          case application:get_env(pixelwar, matrix_width) of
22              {ok, Vw} -> Vw;
23              undefined -> ?DEFAULT_SIZE
24          end,
25      Height =
26          case application:get_env(pixelwar, matrix_height) of
27              {ok, Vh} -> Vh;
28              undefined -> ?DEFAULT_SIZE
29          end,
30
31      ChildSpecifications = [
32          #{
33              id => matrix,
34              start => {pixelwar_matrix_serv, start_link, [{Width, Height}]},
35              % permanent | transient | temporary
36              restart => permanent,
37              shutdown => 2000,
38              % worker | supervisor
39              type => worker
40          }
41      ],
42
43      {ok, {SupervisorSpecification, ChildSpecifications}}.
```

Listing A.11: pixelwar.app.src

```
1  {application, pixelwar, [
2      {description, "An OTP application"},
3      {vsn, "0.2.0"},
4      {registered, []},
5      {mod, {pixelwar_app, []}},
6      {applications, [
7          kernel,
8          stdlib
9      ]},
10     {env, []},
11     {modules, [pixelwar_app, pixelwar_matrix_serv, pixelwar_sup]},
12
13     {licenses, ["Apache-2.0"]},
14     {links, []}
15 ]}.
```

Listing A.12: pixelwar_serv_SUITE.erl

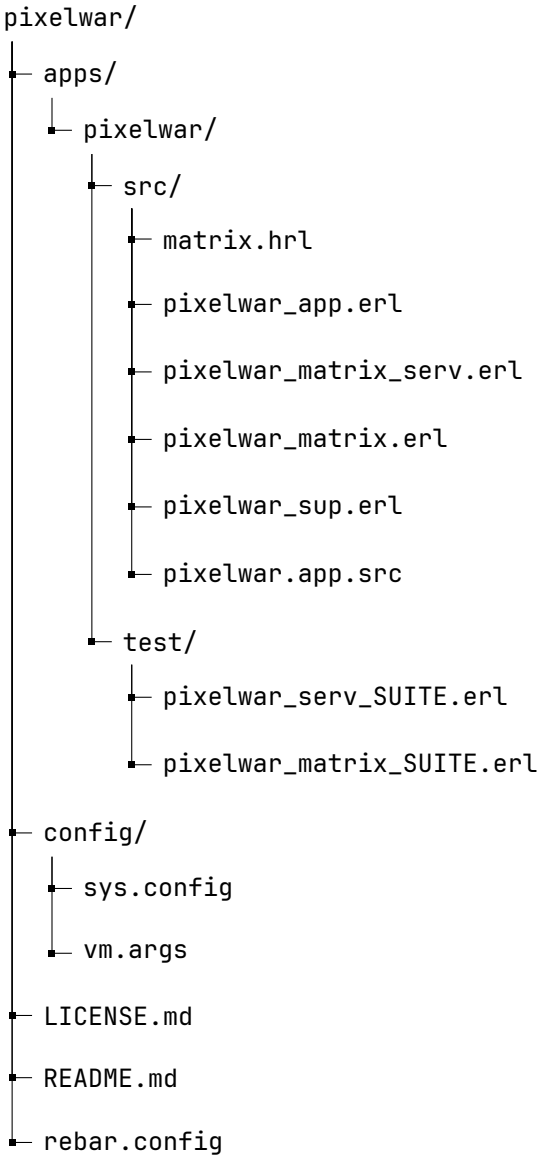
```
1  -module(pixelwar_serv_SUITE).
2  -include_lib("stdlib/include/assert.hrl").
3  -include_lib("common_test/include/ct.hrl").
4  -compile(export_all).
5
6  all() ->
7      [get_state_test_case, place_out_of_bounds_test_case].
8
9  init_per_testcase(_Case, Config) ->
10     application:load(pixelwar),
11     Width = 128,
12     Height = 128,
13     application:set_env(pixelwar, matrix_width, Width),
14     application:set_env(pixelwar, matrix_height, Height),
15
```

```
16      {ok, Apps} = application:ensure_all_started([pixelwar]),
17      [{apps, Apps}, {width, Width}, {height, Height} | Config].
18
19  end_per_testcase(_Case, Config) ->
20      [application:stop(App) || App <- lists:reverse(?config(apps, Config))],
21      Config.
22
23  get_state_test_case() ->
24      [
25          {doc, "Tries to get the current matrix as binary"},
26          {timetrap, timer:seconds(5)}
27      ].
28
29  get_state_test_case(_Config) ->
30      pixelwar_matrix_serv:set_element(matrix, {42, 42, 12}),
31      pixelwar_matrix_serv:set_element(matrix, {42, 42, 42}),
32      pixelwar_matrix_serv:set_element(matrix, {11, 12, 13}),
33
34      MatrixAsBin = pixelwar_matrix_serv:get_state(matrix),
35      ?assertEqual(
36          MatrixAsBin,
37          <<11:16/little, 12:16/little, 13:16/little, 42:16/little, 42:16/little,
        42:16/little>>
38      ).
39
40  place_out_of_bounds_test_case(Config) ->
41      Width = ?config(width, Config),
42      Height = ?config(height, Config),
43
44      InboundWidth = Width - 2,
45      InboundHeight = Height - 2,
46
47      % In bounds
48      pixelwar_matrix_serv:set_element(matrix, {InboundWidth, InboundHeight, 13}),
49      % Out of bounds
50      pixelwar_matrix_serv:set_element(matrix, {Width + 2, Height + 2, 13}),
51
52      MatrixAsBin = pixelwar_matrix_serv:get_state(matrix),
53
54      ?assertEqual(MatrixAsBin, <<InboundWidth:16/little, InboundHeight:16/little,
        13:16/little>>).
```

A.3 Version 0.3.0

Not all files are provided because they are either generated by Rebar3 or not used in the examples.

Figure A.3: Directory structure of Pixelwar version 0.3.0



Listing A.13: rebar.config

```
{erl_opts, [debug_info]}.
{deps, []}.

{relx, [{release, {pixelwar, "0.3.0"},
         [pixelwar,
          sasl]},

        {mode, prod},

        %% automatically picked up if the files
        %% exist but can be set manually, which
        %% is required if the names aren't exactly
        %% sys.config and vm.args
        {sys_config, "./config/sys.config"},
        {vm_args, "./config/vm.args"}

        %% the .src form of the configuration files do
        %% not require setting RELX_REPLACE_OS_VARS
        %% {sys_config_src, "./config/sys.config.src"},
```

```
        %% {vm_args_src, "./config/vm.args.src"}
]}.

{profiles, [{prod, [{relx,
                     [%% prod is the default mode when prod
                      %% profile is used, so does not have
                      %% to be explicitly included like this
                      {mode, prod}

                      %% use minimal mode to exclude ERTS
                      %% {mode, minimal}
                     ]
            }]}]}.
```

Listing A.14: matrix.hrl

```
-define(DEFAULT_SIZE, 128).
-record(matrix, {
    pixels = #{} :: #{{non_neg_integer(), non_neg_integer()} => non_neg_integer()
    },
    width = ?DEFAULT_SIZE :: non_neg_integer(),
    height = ?DEFAULT_SIZE :: non_neg_integer()
}).
```

Listing A.15: pixelwar_app.erl

```
-module(pixelwar_app).

-behaviour(application).

-export([start/2, stop/1]).

start(_StartType, _StartArgs) ->
    pixelwar_sup:start_link().

stop(_State) ->
    ok.
```

Listing A.16: pixelwar_matrix_serv.erl

```
-module(pixelwar_matrix_serv).
-vsn("0.3.0").
-behaviour(gen_server).
-include_lib("matrix.hrl").

-record(state, {
    matrix = #matrix{}
}).

%% API
-export([start_link/0, set_element/2, get_state/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2,
    code_change/3]).

start_link() ->
    gen_server:start_link({local, matrix}, ?MODULE, [], []).

set_element(Instance, Pixel) ->
    gen_server:cast(Instance, {set_element, Pixel}).

get_state(Instance) ->
```

```
    gen_server:call(Instance, get_state).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

init(_Args) ->
    {ok, Matrix} = pixelwar_matrix:create(),
    {ok, #state{matrix = Matrix}}.

handle_call(stop, _From, State) ->
    {stop, normal, stopped, State};

handle_call(get_state, _From, State) ->
    Binary = pixelwar_matrix:to_binary(State#state.matrix),
    {reply, Binary, State};

handle_call(_Request, _From, State) ->
    {reply, ok, State}.

handle_cast({set_element, {X, Y, Color}}, State) ->
    case pixelwar_matrix:set_pixel(State#state.matrix, X, Y, Color) of
        {ok, NewMatrix} ->
            NewState = State#state{matrix=NewMatrix},
            {noreply, NewState};
        _ -> {noreply, State}
    end;

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change("0.2.0", State, _Extra) ->
    {_, Pixels, Width, Height} = State,
    {ok, Matrix} = pixelwar_matrix:create(Width, Height, Pixels),
    {ok, #state{matrix = Matrix}};

code_change({down, "0.2.0"}, State, _Extra) ->
    {_, {_, Pixels, Width, Height}} = State,
    {ok, {state, Pixels, Width, Height}};

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

Listing A.17: pixelwar_matrix.erl

```
-module(pixelwar_matrix).
-include_lib("matrix.hrl").
-export([create/0, create/2, create/3, to_binary/1, set_pixel/4, resize/3]).

create() ->
    Matrix = #matrix{pixels = #{}, width = ?DEFAULT_SIZE, height = ?DEFAULT_SIZE},
    {ok, Matrix}.

create(Width, _Height) when Width =< 0 -> {error, invalid_width};
create(_Width, Height) when Height =< 0 -> {error, invalid_height};
create(Width, Height) ->
```

```
12      {ok, #matrix{ pixels = #{}, width = Width, height = Height}}.
13
14  create(Width, Height, Pixels) ->
15      case create(Width, Height) of
16          {ok, Matrix} -> {ok, Matrix#matrix{pixels=Pixels}} ;
17          Error -> Error
18      end.
19
20  to_binary(#matrix{} = Matrix) ->
21      ToBinary = fun(K, V, Acc) ->
22          {X, Y} = K,
23          <<Acc/binary, X:16/little, Y:16/little, V:16/little>>
24      end,
25      maps:fold(ToBinary, <<>>, Matrix#matrix.pixels).
26
27  set_pixel(Matrix, X, _Y, _Color) when X ⩾ Matrix#matrix.width orelse X < 0 ->
28      {error, invalid_width};
29
30  set_pixel(Matrix, _X, Y, _Color) when Y ⩾ Matrix#matrix.height orelse Y < 0 ->
31      {error, invalid_height};
32
33  set_pixel(Matrix, X, Y, Color) ->
34      Key = {X, Y},
35      NewPixels = maps:put(Key, Color, Matrix#matrix.pixels),
36      NewMatrix = Matrix#matrix{pixels=NewPixels},
37      {ok, NewMatrix}.
38
39  resize(_Matrix, Width, _Height) when Width =< 0 ->
40      {error, invalid_width};
41
42  resize(_Matrix, _Width, Height) when Height =< 0 ->
43      {error, invalid_height};
44
45  resize(Matrix, Width, Height) ->
46      IsInBound = fun({X, Y}, _V) -> X < Width andalso X ⩾ 0 andalso Y < Height
        andalso Y ⩾ 0 end,
47      FilteredPixels = maps:filter(IsInBound, Matrix#matrix.pixels),
48      NewMatrix = Matrix#matrix{pixels=FilteredPixels, width=Width, height=Height},
49      {ok, NewMatrix}.
```

Listing A.18: pixelwar_sup.erl

```
1   -module(pixelwar_sup).
2
3   -behaviour(supervisor).
4   %% API
5   -export([start_link/0]).
6   -export([init/1]).
7
8   start_link() ->
9       supervisor:start_link({local, ?MODULE}, ?MODULE, []).
10
11  init(_Args) ->
12      SupervisorSpecification = #{
13          % one_for_one | one_for_all | rest_for_one | simple_one_for_one
14          strategy => one_for_one,
15          intensity => 10,
16          period => 60
17      },
18
19      ChildSpecifications = [
```

```
        #{
            id => matrix,
            start => {pixelwar_matrix_serv, start_link, []},
            % permanent | transient | temporary
            restart => permanent,
            shutdown => 2000,
            % worker | supervisor
            type => worker
        }
    ],

    {ok, {SupervisorSpecification, ChildSpecifications}}.
```

Listing A.19: pixelwar.app.src

```
{application, pixelwar, [
    % A one-line description of the application.
    {description, "An OTP application"},
    % Version of the application.
    {vsn, "0.3.0"},
    % All names of registered processes started in this application.
    {registered, []},
    % Specifies the application callback module and a start argument
    {mod, {pixelwar_app, []}},
    % All applications that must be started before this application is started.
    {applications, [
        kernel, % All applications depend on kernel and stdlib
        stdlib
    ]},
    % Configuration parameters used by the application.
    {env, []},
    % All Modules introduced by this application
    {modules, [pixelwar_app, pixelwar_matrix_serv, pixelwar_matrix, pixelwar_sup]
    },
    {licenses, ["Apache-2.0"]},
    {links, []}
]}.
```

Listing A.20: pixelwar.appup.src

```
{"0.3.0", % New version
    [{"0.2.0", [ % Upgrade from
        {add_module, pixelwar_matrix},
        {update, pixelwar_matrix_serv, {advanced, []}}
    ]}],
    [{"0.2.0", [ % Downgrade to
        {delete_module, pixelwar_matrix},
        {update, pixelwar_matrix_serv, {advanced, []}}
    ]}]
}.
```

Listing A.21: pixelwar_serv_SUITE.erl

```
-module(pixelwar_serv_SUITE).
-include_lib("stdlib/include/assert.hrl").
-include_lib("common_test/include/ct.hrl").
-include_lib("apps/pixelwar/src/matrix.hrl").
-compile(export_all).

all() ->
    [get_state_test_case, place_out_of_bounds_test_case].
```

```
 9
10 init_per_testcase(_Case, Config) ->
11     application:load(pixelwar),
12
13     {ok, Apps} = application:ensure_all_started([pixelwar]),
14     [{apps, Apps} | Config].
15
16 end_per_testcase(_Case, Config) ->
17     [application:stop(App) || App <- lists:reverse(?config(apps, Config))],
18     Config.
19
20 get_state_test_case() ->
21     [
22         {doc, "Tries to get the current matrix as binary"},
23         {timetrap, timer:seconds(5)}
24     ].
25
26 get_state_test_case(_Config) ->
27     pixelwar_matrix_serv:set_element(matrix, {42, 42, 12}),
28     pixelwar_matrix_serv:set_element(matrix, {42, 42, 42}),
29     pixelwar_matrix_serv:set_element(matrix, {11, 12, 13}),
30
31     MatrixAsBin = pixelwar_matrix_serv:get_state(matrix),
32     ?assertEqual(
33         MatrixAsBin,
34         <<11:16/little, 12:16/little, 13:16/little, 42:16/little, 42:16/little,
       42:16/little>>
35     ).
36
37 place_out_of_bounds_test_case(_) ->
38     Inbound = ?DEFAULT_SIZE - 2,
39     Outbound = ?DEFAULT_SIZE + 2,
40
41     % In bounds
42     pixelwar_matrix_serv:set_element(matrix, {Inbound, Inbound, 13}),
43     % Out of bounds
44     pixelwar_matrix_serv:set_element(matrix, {Outbound + 2, Outbound + 2, 13}),
45
46     MatrixAsBin = pixelwar_matrix_serv:get_state(matrix),
47
48     ?assertEqual(MatrixAsBin, <<Inbound:16/little, Inbound:16/little, 13:16/
       little>>).
```

Listing A.22: pixelwar_matrix_SUITE.erl

```
 1 -module(pixelwar_matrix_SUITE).
 2 -include_lib("stdlib/include/assert.hrl").
 3 -include_lib("common_test/include/ct.hrl").
 4 -include_lib("apps/pixelwar/src/matrix.hrl").
 5 -compile(export_all).
 6
 7 all() ->
 8     [
 9         create_invalid_size_matrix_test_case,
10         create_matrix_with_pixels_test_case,
11         matrix_to_binary_test_case,
12         set_pixel_invalid_location_test_case,
13         resize_matrix_test_case
14     ].
15
16 create_invalid_size_matrix_test_case() ->
```

```
17      [
18          {doc, "Tries to create a matrix with an invalid width"},
19          {timetrap, timer:seconds(5)}
20      ].
21  create_invalid_size_matrix_test_case(_Config) ->
22      {error, invalid_width} = pixelwar_matrix:create(-2, 100),
23      {error, invalid_height} = pixelwar_matrix:create(100, -2).
24
25
26  create_matrix_with_pixels_test_case() ->
27      [
28          {doc, "Tries to create a matrix with already created pixels"},
29          {timetrap, timer:seconds(5)}
30      ].
31  create_matrix_with_pixels_test_case(_Config) ->
32      Pixels = #{{42,42} => 1, {11, 11} => 2},
33      {ok, Matrix} = pixelwar_matrix:create(100, 100, Pixels),
34      Pixels =:= Matrix#matrix.pixels.
35
36  matrix_to_binary_test_case() ->
37      [
38          {doc, "Tries to represent the pixels of a matrix as binary"},
39          {timetrap, timer:seconds(5)}
40      ].
41  matrix_to_binary_test_case(_Config) ->
42      Pixels = #{{42,42} => 42, {11, 12} => 13},
43      {ok, Matrix} = pixelwar_matrix:create(100, 100, Pixels),
44      Bin = pixelwar_matrix:to_binary(Matrix),
45      ?assertEqual(
46          Bin,
47          <<11:16/little, 12:16/little, 13:16/little, 42:16/little, 42:16/little,
        42:16/little>>
48      ).
49
50  set_pixel_invalid_location_test_case() ->
51      [
52          {doc, "Tries to place a pixel at an invalid location with an invalid
        height"},
53          {timetrap, timer:seconds(5)}
54      ].
55  set_pixel_invalid_location_test_case(_config) ->
56      {ok, Matrix} = pixelwar_matrix:create(),
57      {error, invalid_width} = pixelwar_matrix:set_pixel(Matrix, -2, 10, 12),
58      {error, invalid_width} = pixelwar_matrix:set_pixel(Matrix, 200, 10, 12),
59      {error, invalid_height} = pixelwar_matrix:set_pixel(Matrix, 10, -2, 12),
60      {error, invalid_height} = pixelwar_matrix:set_pixel(Matrix, 10, 200, 12).
61
62  resize_matrix_test_case() ->
63      [
64          {doc, "Tries to resize the matrix"},
65          {timetrap, timer:seconds(5)}
66      ].
67  resize_matrix_test_case(_Config) ->
68      {ok, Start} = pixelwar_matrix:create(),
69      {ok, Modified} = pixelwar_matrix:set_pixel(Start, 42, 42, 42),
70      {ok, ModifiedTwice} = pixelwar_matrix:set_pixel(Modified, 11, 12, 13),
71      {error, invalid_width} = pixelwar_matrix:resize(ModifiedTwice, 0, 100),
72      {error, invalid_width} = pixelwar_matrix:resize(ModifiedTwice, -1, 100),
73      {error, invalid_height} = pixelwar_matrix:resize(ModifiedTwice, 100, 0),
74      {error, invalid_height} = pixelwar_matrix:resize(ModifiedTwice, 100, -1),
75      {ok, Resized} = pixelwar_matrix:resize(ModifiedTwice, 21, 22),
```

```
76      ?assertEqual(
77          #{{11, 12} => 13},
78          Resized#matrix.pixels
79      ),
80      ?assertEqual(
81          21,
82          Resized#matrix.width
83      ),
84      ?assertEqual(
85          22,
86          Resized#matrix.height
87      ).
```

Appendix B

Upgrade downgrade test suites

B.1 First Version

Listing B.1: Excerpt from the first Github workflow

```
- name: Run relup application
    run: |
        mkdir relupci
        tar -xvf "${{ env.OLD_TAR }}" -C relupci
        MATRIX_WIDTH=128 MATRIX_HEIGHT=128 relupci/bin/pixelwar daemon #1
        cp "${{ env.NEW_TAR }}" relupci/releases/


        OLD_TAG=$(echo "${{ env.OLD_TAR }}"  | sed -nr 's/^.*([0-9]+\.[0-9]+\.[0-9]+)\.tar\.gz$/\1/p')
        NEW_TAG=$(echo "${{ env.NEW_TAR }}"  | sed -nr 's/^.*([0-9]+\.[0-9]+\.[0-9]+)\.tar\.gz$/\1/p')

        echo "Launch before upgrade test"
        ./test/before_upgrade.sh #2

        relupci/bin/pixelwar upgrade ${NEW_TAG} #3
        relupci/bin/pixelwar versions

        echo "Launch after upgrade test"
        ./test/after_upgrade.sh #4

        echo "Launch before downgrade test"
        ./test/before_downgrade.sh #5

        relupci/bin/pixelwar downgrade ${OLD_TAG} #6

        echo "Launch after downgrade test"
        ./test/after_downgrade.sh #7
```

Listing B.2: Testing the state of a release from bash

```
#!/bin/bash

binary1=$(./relupci/bin/pixelwar rpc pixelwar_matrix_serv get_state [matrix])
binary2='#Bin<12,0,12,0,12,0>'
echo $binary1
echo $binary2

if [[ $binary1 == $binary2 ]]; then
  echo "Success"
```

```
  exit 0
else
  echo "Fail"
  exit 1
fi
```

B.2 Second version

Listing B.3: A generic python module for interfacing between Robot Framework and an OTP release

```python
# otp.py


import subprocess
from robot.api.deco import not_keyword
from robot.libraries.BuiltIn import BuiltIn

@not_keyword
def run(command):
    result = subprocess.run(command, shell = True, executable="/bin/bash", capture_output=True)
    stdout = result.stdout.decode("utf-8")
    return stdout

def start_release():
    RELEASE_PATH = BuiltIn().get_variable_value("${RELEASE_PATH}")
    START_COMMAND = f"MATRIX_WIDTH=128 MATRIX_HEIGHT=128 {RELEASE_PATH} daemon"
    return run(START_COMMAND)

def stop_release():
    RELEASE_PATH = BuiltIn().get_variable_value("${RELEASE_PATH}")
    START_COMMAND = f"{RELEASE_PATH} stop"
    return run(START_COMMAND)

def send_rpc(module, function, arguments):
    RELEASE_PATH = BuiltIn().get_variable_value("${RELEASE_PATH}")
    SEND_RPC_COMMAND = f"{RELEASE_PATH} rpc {module} {function} {arguments}"
    return run(SEND_RPC_COMMAND)

def upgrade_release(version):
    RELEASE_PATH = BuiltIn().get_variable_value("${RELEASE_PATH}")
    UPGRADE_COMMAND = f"{RELEASE_PATH} upgrade {version}"
    stdout = run(UPGRADE_COMMAND)

    if "release_package_not_found" in stdout:
        raise AssertionError(f"Command failed: {stdout}")

def downgrade_release(version):
    RELEASE_PATH = BuiltIn().get_variable_value("${RELEASE_PATH}")
    DOWNGRADE_COMMAND = f"{RELEASE_PATH} downgrade {version}"
    stdout = run(DOWNGRADE_COMMAND)

    if "release_package_not_found" in stdout:
        raise AssertionError(f"Command failed: {stdout}")

def should_be_equal_as_erlang_bytes(actual, expected, msg=None):
    from_as_str = actual.replace("#Bin<", "")
```

```
    from_as_str = from_as_str.replace(">\n", "")

    if from_as_str != expected:
        if msg:
            raise AssertionError(f"Values are not equal: {from_as_str} != {expected}. {msg}")
        else:
            raise AssertionError(f"Values are not equal: {from_as_str} != {expected}")
```

Listing B.4: An application specific python module for interfacing with our release

```
# matrix.py


from otp import send_rpc

MODULE = "pixelwar_matrix_serv"
PROCESS = "matrix"

def set_pixel(x, y, color):
    return send_rpc(MODULE, "set_element", f"[{PROCESS}, {{ {x}, {y}, {color} }}]")

def get_matrix_state():
    return send_rpc(MODULE, "get_state", f"[{PROCESS}]")
```

Listing B.5: A Robot Framework test suite using the python modules

```
*** Variables ***
${OLD_VERSION}    0.1.0
${NEW_VERSION}    0.2.0
${RELEASE_PATH}    erlang/_build/default/rel/pixelwar/bin/pixelwar

*** Settings ***
Suite Setup     Start Release
Suite Teardown     Stop Release
Library    matrix.py
Library    otp.py

*** Test Cases ***
Setup state before upgrade
    Set Pixel    12    12    12
    Set Pixel    222    222    222
    ${state}    Get Matrix State
    Should Be Equal As Erlang Bytes    ${state}    12,0,12,0,12,0,222,0,222,0,222,0

upgrade release
    Upgrade Release    ${NEW_VERSION}

Test state after upgrade
    ${state}    Get Matrix State
    Should Be Equal As Erlang Bytes    ${state}    12,0,12,0,12,0

Setup state before downgrade
    Set Pixel    13    13    13
    ${state}    Get Matrix State
    Should Be Equal As Erlang Bytes    ${state}    12,0,12,0,12,0,13,0,13,0,13,0

downgrade release
```

```
    Downgrade Release    ${OLD_VERSION}

Test state after downgrade
    ${state}    Get Matrix State
    Should Be Equal As Erlang Bytes    ${state}    12,0,12,0,12,0,13,0,13,0,13,0
```

B.3 Third version

Listing B.6: Upgrade/downgrade CT test suite

```
-module(upgrade_downgrade_SUITE).
-behaviour(ct_suite).
-export([all/0, groups/0]).
-compile(export_all).

-include_lib("stdlib/include/assert.hrl").
-include_lib("common_test/include/ct.hrl").

groups() ->
    [{upgrade_downgrade, [sequence], [before_upgrade_case, upgrade_case,
    after_upgrade_case, before_downgrade_case, downgrade_case,
    after_downgrade_case]}].

all() ->
    [{group, upgrade_downgrade}].

suite() ->
    [
        {require, old_version},
        {require, new_version},
        {require, release_name},
        {require, release_dir}
    ].

init_per_suite(Config) ->
    ct:print("Initializing suite..."),
    ct:log(info, ?LOW_IMPORTANCE, "Initializing suite...", []),
    Docker = os:find_executable("docker"),
    build_image(),
    ReleaseName = ct:get_config(release_name),

    {ok, Peer, Node} = peer:start(#{name => ReleaseName,
        connection => standard_io,
        exec => {Docker, ["run", "-h", "one", "-i", ReleaseName]}}),

    [{peer, Peer}, {node, Node} | Config].

end_per_suite(Config) ->
    Peer = ?config(peer, Config),
    peer:stop(Peer).

% ========== CASES ==========

before_upgrade_case(Config) ->
    _Peer = ?config(peer, Config),
    ct:print("TODO: Implement this case").

upgrade_case(Config) ->
    Peer = ?config(peer, Config),
```

```
48      NewVSN = ct:get_config(new_version),
49      OldVSN = ct:get_config(old_version),
50      ReleaseName = ct:get_config(release_name),
51      NewReleaseName = filename:join(NewVSN, ReleaseName),
52
53      {ok, NewVSN} = peer:call(Peer, release_handler, unpack_release, [
        NewReleaseName]),
54      {ok, OldVSN, _} = peer:call(Peer, release_handler, install_release, [NewVSN])
        ,
55      ok = peer:call(Peer, release_handler, make_permanent, [NewVSN]),
56
57      Releases = peer:call(Peer, release_handler, which_releases, []),
58      ct:print("Installed releases:\n~p", [Releases]).
59
60  after_upgrade_case(Config) ->
61      _Peer = ?config(peer, Config),
62      ct:print("TODO: Implement this case").
63
64  before_downgrade_case(Config) ->
65      _Peer = ?config(peer, Config),
66      ct:print("TODO: Implement this case").
67
68  downgrade_case(Config) ->
69      Peer = ?config(peer, Config),
70      OldVSN = ct:get_config(old_version),
71
72      {ok, OldVSN, _} = peer:call(Peer, release_handler, install_release, [OldVSN])
        ,
73      ok = peer:call(Peer, release_handler, make_permanent, [OldVSN]),
74
75      Releases = peer:call(Peer, release_handler, which_releases, []),
76      ct:print("Installed releases:\n~p", [Releases]).
77
78  after_downgrade_case(Config) ->
79      _Peer = ?config(peer, Config),
80      ct:print("TODO: Implement this case").
81
82  % ========== HELPERS ==========
83
84  build_image() ->
85      NewVSN = ct:get_config(new_version),
86      OldVSN = ct:get_config(old_version),
87      ReleaseName = ct:get_config(release_name),
88      NewReleaseName = ReleaseName ++ "-" ++ NewVSN,
89      OldReleaseName = ReleaseName ++ "-" ++ OldVSN,
90      ReleaseDir = ct:get_config(release_dir),
91
92      NewReleasePath = filename:join(ReleaseDir, NewReleaseName ++ ".tar.gz"),
93      file:copy(NewReleasePath, "./" ++ NewReleaseName ++ ".tar.gz"),
94
95      OldReleasePath = filename:join(ReleaseDir, OldReleaseName ++ ".tar.gz"),
96      file:copy(OldReleasePath, "./" ++ OldReleaseName ++ ".tar.gz"),
97
98      %% Create Dockerfile example, working only for Ubuntu 20.04
99      %% Expose port 4445, and make Erlang distribution to listen
100     %% on this port, and connect to it without EPMD
101     %% Set cookie on both nodes to be the same.
102     BuildScript = filename:join("./", "Dockerfile"),
103     Dockerfile =
104         "FROM ubuntu:22.04 as runner\n"
105         "EXPOSE 4445\n"
```

```
106         "WORKDIR /opt/" ++ ReleaseName ++  "\n"
107         "COPY [\"" ++ OldReleaseName ++ ".tar.gz\", \"" ++ NewReleaseName ++ ".
    tar.gz\"" ++ ", \"/tmp/\"]\n"
108         "RUN tar -zxvf /tmp/" ++ OldReleaseName ++ ".tar.gz -C /opt/" ++
    ReleaseName ++ "\n"
109         "RUN mkdir /opt/" ++ ReleaseName ++ "/releases/" ++ NewVSN ++ "\n"
110         "RUN cp /tmp/" ++ NewReleaseName ++ ".tar.gz /opt/" ++ ReleaseName ++ "/
    releases/" ++ NewVSN ++ "/" ++ ReleaseName ++ ".tar.gz\n"
111         "ENTRYPOINT [\"/opt/" ++ ReleaseName ++ "/erts-" ++ erlang:system_info(
    version) ++
112         "/bin/dyn_erl\", \"-boot\", \"/opt/" ++ ReleaseName ++ "/releases/" ++
    OldVSN ++ "/start\","
113         " \"-kernel\", \"inet_dist_listen_min\", \"4445\","
114         " \"-erl_epmd_port\", \"4445\","
115         " \"-setcookie\", \"secret\"]\n",
116     ct:log(info, ?LOW_IMPORTANCE, "Dockerfile:\n~s", [Dockerfile]),
117     ok = file:write_file(BuildScript, Dockerfile),
118     DockerBuildResult = os:cmd("docker build -t " ++ ReleaseName ++ " ."),
119     ct:log(info, ?LOW_IMPORTANCE, "Docker build:\n~s", [DockerBuildResult]).
```

Appendix C

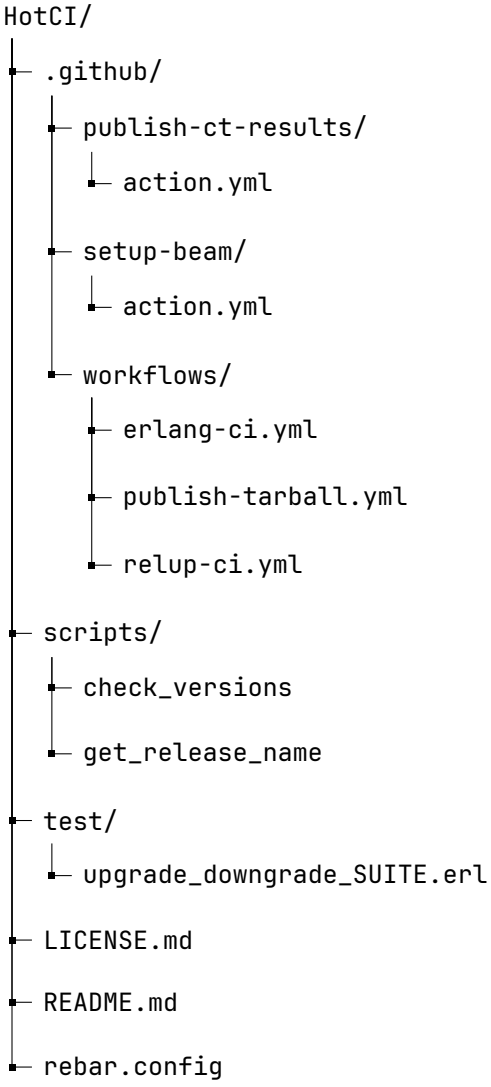
HotCI

Snapshot of the HotCI repository[24] made the 14th of May 2024.

HotCI can be modified or extended by altering the files present in the *.github*, *test* and *script* repositories.

If running the tests regularly is required, the *schedule* workflows trigger, configured with a cron, can be used. Note that, if a step value is part of the cron, there is no guarantee that the workflow will run at the specified interval.

Figure C.1: Directory structure of HotCI



Listing C.1: publish-ct-results/action.yml

```yaml
name: Publish CT Results
description: Publish and Upload CT results as artifact

inputs:
  test-name:
    description: Name of the test suite
    required: true
    default: Some test

runs:
  using: composite
  steps:
  - name: Upload test results as artifact
    uses: actions/upload-artifact@v4
    with:
      name: ${{ inputs.test-name }}
      path: ./results
      compression-level: 9
      retention-days: 30

  - name: Publish test results
    uses: EnricoMi/publish-unit-test-result-action@v2
    with:
      check_name: ${{ inputs.test-name }}
      files: |
        ./results/**/*.xml
```

Listing C.2: setup-beam/action.yml

```yaml
name: Setup
description: Setup the BEAM

runs:
  using: composite
  steps:
  - name: Setup the BEAM
    uses: erlef/setup-beam@v1
    with:
      otp-version: '26'
      rebar3-version: '3.22.1'
```

Listing C.3: erlang-ci.yml

```yaml
name: Erlang CI

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

permissions:
  contents: read
  issues: read
  checks: write
  pull-requests: write

jobs:
  validation-checks:
    runs-on: ubuntu-latest
```

```
18
19      steps:
20        - uses: actions/checkout@v4
21        - uses: ./.github/setup-beam
22
23        - name: Run validation checks
24          working-directory: .
25          run: rebar3 do xref, dialyzer
26
27    build:
28      runs-on: ubuntu-latest
29
30      steps:
31      - uses: actions/checkout@v4
32      - uses: ./.github/setup-beam
33
34      - name: Check that the release can be built
35        working-directory: .
36        run: rebar3 do release, tar
37
38    unit-test:
39      runs-on: ubuntu-latest
40
41      steps:
42        - uses: actions/checkout@v4
43        - uses: ./.github/setup-beam
44
45        - name: Run unit tests
46          working-directory: .
47          run: |
48            mkdir results
49            rebar3 do ct --dir apps --verbose true --logdir ./results --label
      erlang-ci --cover true, cover -v
50
51        - uses: ./.github/publish-ct-results
52          if: always()
53          with:
54            test-name: erlang-ci
```

Listing C.4: relup-ci.yml

```
1  name: Relup CI
2
3  on:
4    pull_request:
5      branches: [ "main" ]
6
7  permissions:
8    contents: read
9    issues: read
10   checks: write
11   pull-requests: write
12
13 jobs:
14   upgrade-downgrade-test:
15     runs-on: ubuntu-latest
16
17     steps:
18     - uses: actions/checkout@v4
19       with:
20         fetch-depth: 0
```

```
21      - uses: ./.github/setup-beam
22
23      - name: Get release name
24        working-directory: .
25        run: |
26          RELNAME=$(./scripts/get_release_name ./rebar.config)
27          echo "RELNAME=$RELNAME" >> $GITHUB_ENV
28
29      - name: Check if a previous tags exists
30        run: |
31          LAST_TAG=$(git tag -l --sort=committerdate 'v*.*.*' | tail -n 1)
32          echo "LAST_TAG=$LAST_TAG" >> $GITHUB_ENV
33
34      - name: Set check status
35        if: ${{env.LAST_TAG == ''}}
36        run: echo '### No previous version tag, skipped checks' >>
      $GITHUB_STEP_SUMMARY
37
38      - name: Run validation checks and build releases
39        working-directory: .
40        if: ${{env.LAST_TAG != ''}}
41        run: |
42          ./scripts/check_versions ${{env.RELNAME}} $(/usr/bin/git log -1 --format
      ='%H') > vsn.log
43          cat vsn.log
44          cat vsn.log | awk '/Generated appup/ || /Compiling .*\.appup\.src/ {
      appup=1 }
45                             /relup successfully created!/ { relup=1 }
46                             /RESTART version bumped, ignoring relup check./ {
      restart=1 }
47                             END { if (!appup) { print "appup missing"; exit 1}
48                                   if (!relup && !restart) { print "relup missing";
       exit 1} }'
49          OLD=$(cat vsn.log | awk '/OLD:/ {print $2}')
50          NEW=$(cat vsn.log | awk '/NEW:/ {print $2}')
51          IS_RESTART=$(cat vsn.log | awk '/RESTART version bumped, ignoring relup
      check./ {found=1; exit} END {if(found) print "true"; else print "false"}')
52          echo "OLD_TAR=$OLD" >> $GITHUB_ENV
53          echo "NEW_TAR=$NEW" >> $GITHUB_ENV
54          echo "IS_RESTART=$IS_RESTART" >> $GITHUB_ENV
55
56
57      - name: Test upgrade and downgrade of the application
58        working-directory: .
59        if: ${{env.LAST_TAG != '' && env.IS_RESTART == 'false'}}
60        run: |
61          mkdir relupci
62          mkdir relupci/releases/
63          cp "${{ env.OLD_TAR }}" relupci/releases/
64          cp "${{ env.NEW_TAR }}" relupci/releases/
65
66          OLD_TAG=$(echo "${{ env.OLD_TAR }}"  | sed -nr 's
      /^.*([0-9]+\.[0-9]+\.[0-9]+)\.tar\.gz$/\1/p')
67          NEW_TAG=$(echo "${{ env.NEW_TAR }}"  | sed -nr 's
      /^.*([0-9]+\.[0-9]+\.[0-9]+)\.tar\.gz$/\1/p')
68
69          RELEASE_DIR=$(readlink -f relupci/releases/)
70          echo -e "{old_version, \"$OLD_TAG\"}.\n{new_version, \"$NEW_TAG\"}.\n{
      release_name, \"${{env.RELNAME}}\"}.\n{release_dir, \"$RELEASE_DIR\"}." >> ./
      test/config.config
```

```
71
72          mkdir results
73          rebar3 ct --dir ./test --verbose true --config ./test/config.config --
        logdir ./results --label relup-ci
74
75      - name: Remove releases to save space
76        working-directory: .
77        if: ${{ always() && env.LAST_TAG != '' && env.IS_RESTART == 'false'}}
78        run: rm -rf ./results/**/*.tar.gz
79
80      - uses: ./.github/publish-ct-results
81        if: ${{ always() && env.LAST_TAG != '' && env.IS_RESTART == 'false'}}
82        with:
83          test-name: relup-ci
```

Listing C.5: publish-tarball.yml

```
 1  name: Publish tarball
 2
 3  on:
 4    push:
 5      tags: [ "v[0-9]+.[0-9]+.[0-9]+" ]
 6
 7  jobs:
 8
 9    build:
10      name: Prepare build artifacts
11      runs-on: ubuntu-latest
12
13      steps:
14        - uses: actions/checkout@v4
15          with:
16            fetch-depth: 0 # Necessary to get all the commits and tags
17        - uses: ./.github/setup-beam
18
19        - name: Get latest release
20          id: get-latest-release
21          uses: ahzed11/get-latest-release-action@v1.2
22          with:
23            keepv: false
24
25        - name: Fetch manifest version
26          run: |
27            NEW_VSN=${GITHUB_REF##*/v}
28            echo "VSN=$NEW_VSN" >> $GITHUB_ENV
29
30            OLD_VSN=${{steps.get-latest-release.outputs.release}}
31            echo "OLD_VSN=$OLD_VSN" >> $GITHUB_ENV
32
33            IS_UPGRADE=$(echo "$NEW_VSN $OLD_VSN" | awk -vFS='[. ]' '($1==$4 && $2>
      $5) || ($1==$4 && $2>=$5 && $3>$6) {print 1; exit} {print 0}')
34            if [ "$IS_UPGRADE" -eq 1 ]; then
35                echo "RELUP=1" >> $GITHUB_ENV
36            else
37                echo "RELUP=0" >> $GITHUB_ENV
38            fi
39        - run: |
40            echo "${{ env.OLD_VSN }} -> ${{ env.VSN }} : ${{ env.RELUP }}"
41
42        - name: Build a tarball
43          working-directory: .
```

```
        run: |
          RELNAME=$(./scripts/get_release_name ./rebar.config)
          if [ ${{ env.RELUP }} -eq 1 ]; then
            ORIG=$(/usr/bin/git log -1 --format='%H')
            git checkout v${{ env.OLD_VSN }}
            rebar3 do clean -a, release
            git checkout $ORIG
            rebar3 do clean -a, release
            rebar3 appup generate
            rebar3 relup -n $RELNAME -v ${{ env.VSN }} -u ${{ env.OLD_VSN }}
          else
            rebar3 release
          fi
          rebar3 tar
          BUILD=$(ls _build/default/rel/$RELNAME/$RELNAME-*.tar.gz)
          mkdir ./_artifacts
          cp $BUILD ./_artifacts/$RELNAME-${{ env.VSN }}.tar.gz

      - name: Upload build artifacts
        uses: actions/upload-artifact@v4
        with:
          name: artifacts
          path: _artifacts
          retention-days: 1

  publish:
    name: Publish build artifacts
    needs: build
    runs-on: ubuntu-latest

    permissions:
      id-token: write
      contents: write

    steps:
      - name: Get build artifacts
        uses: actions/download-artifact@v4
        with:
          name: artifacts
          path: _artifacts

      - name: Upload release
        uses: ncipollo/release-action@v1
        with:
          artifacts: |
            _artifacts/*.tar.gz
```

Listing C.6: check_versions

```
#!/usr/bin/env escript
-module(check_versions).
-mode(compile).

usage() ->
    io:format("usage: Relname Branch\n"),
    halt(1).

main([Relname]) ->
    main([Relname, os:cmd("git rev-parse --abbrev-ref HEAD")]);

main([Relname, Branch]) ->
```

```
    SrcPrefix = "apps/",
    "v"++_LastVsn = LastTag = string:trim(os:cmd("git tag -l --sort=committerdate
    'v*.*.*' | tail -n 1")),
    io:format("switching to ~ts and back to ~ts~n", [LastTag, Branch]),
    io:format("~p: ~ts~n", [?LINE, os:cmd("git checkout "++LastTag)]),
    io:format("~p: ~ts~n", [?LINE, os:cmd("rebar3 do clean -a, release, tar")]),
    io:format("~p: ~ts~n", [?LINE, os:cmd("git checkout "++Branch)]),
    io:format("~p: ~ts~n", [?LINE, os:cmd("rebar3 do clean -a, release")]),
    RelFiles = filelib:wildcard("_build/default/rel/*/releases/**/*.rel"),
    case length(RelFiles) of
        2 ->
            ok;
        N ->
            io:format("Expected two find two releases, found ~p~n"
                      "The current release likely has the same version as "
                      "an older one.~n"
                      "Bump up the release version to generate relups.~n",
                      [N]),
            init:stop(1)
    end,
    Releases = [Content || Path <- RelFiles,
                           {ok, [Content]} <- [file:consult(Path)]],
    case [{ERTS, to_vsn(VsnStr)} || {release, {_Name, VsnStr}, ERTS, _Apps} <-
    Releases] of
        [{_,{V1,_,_}},{_,{V2,_,_}}] when V1 =:= V2 ->
            io:format("RESTART version bumped, ignoring relup check.~n", []),
            init:stop(0);
        [{ERTS,_}, {ERTS,_}] ->
            ok;
        [{_, {V1,_,_}}, {_, {V1,_,_}}] ->
            io:format("The ERTS version changed; the release version should "
                      "be bumped accordingly (RESTART+1.RELUP.RELOAD)~n", []),
            init:stop(1)
    end,

    ChangedFiles = [Sub
                    || Path <- diff_lines(os:cmd("git diff "++LastTag)),
                       Sub <- [string:prefix(Path, SrcPrefix)],
                       Sub =/= nomatch],
    AppsChanged = [list_to_atom(App) ||
                   App <- lists:usort([hd(filename:split(Name)) || Name <-
    ChangedFiles])],
    [{_, OldStr, Old}, {_, NewStr, New}] = lists:sort(
        [{to_vsn(VsnStr), VsnStr, Apps}
         || {release, {_Name, VsnStr}, _ERTS, Apps} <- Releases]
    ),
    Incorrect = lists:filter(
        fun(App) ->
            OldVsn = case lists:keyfind(App, 1, Old) of
                {_, Vsn} -> Vsn;
                false -> ""
            end,
            {_, NewVsn} = lists:keyfind(App, 1, New),
            OldVsn == NewVsn
        end,
        AppsChanged
    ),
    case Incorrect of
        [] ->
            ok;
        _ ->
```

```
            io:format("Applications have changes applied and need a "
                      "version bump: ~p~n", [Incorrect]),
            init:stop(1)
    end,
    %% Build relup phase
    io:format("~ts~n", [os:cmd("rebar3 appup generate")]),
    io:format("~ts~n", [os:cmd("rebar3 relup -n"++ Relname ++" -v "++NewStr++" -u
    "++OldStr)]),
    io:format("~ts~n", [os:cmd("rebar3 tar")]),
    io:format("OLD: _build/default/rel/"++Relname++"/"++Relname++"-~ts.tar.gz~n"
              "NEW: _build/default/rel/"++Relname++"/"++Relname++"-~ts.tar.gz~n",
              [OldStr, NewStr]),
    init:stop(0);

main(_) ->
    usage().

to_vsn(Str) ->
    [Restart, Relup, Reload] = [list_to_integer(S) || S <- string:lexemes(Str, ".
    ")],
    {Restart, Relup, Reload}.

diff_lines(Str) ->
    [string:sub_string(Path, 3)
     || Line <- string:lexemes(Str, "\n"),
        nomatch =/= string:prefix(Line, "+++") orelse
        nomatch =/= string:prefix(Line, "---"),
        [_, Path |_ ] <- [string:lexemes(Line, " ")]
    ].
```

Listing C.7: get_release_name

```
#!/usr/bin/env escript
-module(get_release_name).
-mode(compile).

usage() ->
    io:format("usage: RebarConfigPath\n"),
    halt(1).

main([Path]) ->
    {ok, Terms} = file:consult(Path),
    Filter = fun (E) ->
        case E of
            {relx, _} -> true;
            _ -> false
        end
    end,
    [H | _] = lists:filter(Filter, Terms),
    {relx, Relx} = H,
    {release, {Release, _}, _} = lists:nth(1, Relx),
    io:format("~s", [Release]);

main(_) ->
    usage().
```

Listing C.8: upgrade_downgrade_SUITE.erl

```
-module(upgrade_downgrade_SUITE).
-behaviour(ct_suite).
-export([all/0, groups/0]).
```

```
-compile(export_all).

-include_lib("stdlib/include/assert.hrl").
-include_lib("common_test/include/ct.hrl").

groups() ->
    [{upgrade_downgrade, [sequence], [before_upgrade_case, upgrade_case,
    after_upgrade_case, before_downgrade_case, downgrade_case,
    after_downgrade_case]}].

all() ->
    [{group, upgrade_downgrade}].

suite() ->
    [
        {require, old_version},
        {require, new_version},
        {require, release_name},
        {require, release_dir}
    ].

init_per_suite(Config) ->
    ct:print("Initializing suite..."),
    ct:log(info, ?LOW_IMPORTANCE, "Initializing suite...", []),
    Docker = os:find_executable("docker"),
    build_image(),
    ReleaseName = ct:get_config(release_name),

    {ok, Peer, Node} = peer:start(#{name => ReleaseName,
        connection => standard_io,
        exec => {Docker, ["run", "-h", "one", "-i", ReleaseName]}}),

    [{peer, Peer}, {node, Node} | Config].

end_per_suite(Config) ->
    Peer = ?config(peer, Config),
    peer:stop(Peer).

% ========== CASES ==========

before_upgrade_case(Config) ->
    _Peer = ?config(peer, Config),
    ct:print("TODO: Implement this case").

upgrade_case(Config) ->
    Peer = ?config(peer, Config),
    NewVSN = ct:get_config(new_version),
    OldVSN = ct:get_config(old_version),
    ReleaseName = ct:get_config(release_name),
    NewReleaseName = filename:join(NewVSN, ReleaseName),

    {ok, NewVSN} = peer:call(Peer, release_handler, unpack_release, [
    NewReleaseName]),
    {ok, OldVSN, _} = peer:call(Peer, release_handler, install_release, [NewVSN])
    ,
    ok = peer:call(Peer, release_handler, make_permanent, [NewVSN]),

    Releases = peer:call(Peer, release_handler, which_releases, []),
    ct:print("Installed releases:\n~p", [Releases]).

after_upgrade_case(Config) ->
```

```
61      _Peer = ?config(peer, Config),
62      ct:print("TODO: Implement this case").
63
64  before_downgrade_case(Config) ->
65      _Peer = ?config(peer, Config),
66      ct:print("TODO: Implement this case").
67
68  downgrade_case(Config) ->
69      Peer = ?config(peer, Config),
70      OldVSN = ct:get_config(old_version),
71
72      {ok, OldVSN, _} = peer:call(Peer, release_handler, install_release, [OldVSN])
        ,
73      ok = peer:call(Peer, release_handler, make_permanent, [OldVSN]),
74
75      Releases = peer:call(Peer, release_handler, which_releases, []),
76      ct:print("Installed releases:\n~p", [Releases]).
77
78  after_downgrade_case(Config) ->
79      _Peer = ?config(peer, Config),
80      ct:print("TODO: Implement this case").
81
82  % ========== HELPERS ==========
83
84  build_image() ->
85      NewVSN = ct:get_config(new_version),
86      OldVSN = ct:get_config(old_version),
87      ReleaseName = ct:get_config(release_name),
88      NewReleaseName = ReleaseName ++ "-" ++ NewVSN,
89      OldReleaseName = ReleaseName ++ "-" ++ OldVSN,
90      ReleaseDir = ct:get_config(release_dir),
91
92      NewReleasePath = filename:join(ReleaseDir, NewReleaseName ++ ".tar.gz"),
93      file:copy(NewReleasePath, "./" ++ NewReleaseName ++ ".tar.gz"),
94
95      OldReleasePath = filename:join(ReleaseDir, OldReleaseName ++ ".tar.gz"),
96      file:copy(OldReleasePath, "./" ++ OldReleaseName ++ ".tar.gz"),
97
98      %% Create Dockerfile example, working only for Ubuntu 20.04
99      %% Expose port 4445, and make Erlang distribution to listen
100     %% on this port, and connect to it without EPMD
101     %% Set cookie on both nodes to be the same.
102     BuildScript = filename:join("./", "Dockerfile"),
103     Dockerfile =
104         "FROM ubuntu:22.04 as runner\n"
105         "EXPOSE 4445\n"
106         "WORKDIR /opt/" ++ ReleaseName ++  "\n"
107         "COPY [\"" ++ OldReleaseName ++ ".tar.gz\", \"" ++ NewReleaseName ++ ".
    tar.gz\"" ++ ", \"/tmp/\"]\n"
108         "RUN tar -zxvf /tmp/" ++ OldReleaseName ++ ".tar.gz -C /opt/" ++
    ReleaseName ++ "\n"
109         "RUN mkdir /opt/" ++ ReleaseName ++ "/releases/" ++ NewVSN ++ "\n"
110         "RUN cp /tmp/" ++ NewReleaseName ++ ".tar.gz /opt/" ++ ReleaseName ++ "/
    releases/" ++ NewVSN ++ "/" ++ ReleaseName ++ ".tar.gz\n"
111         "ENTRYPOINT [\"/opt/" ++ ReleaseName ++ "/erts-" ++ erlang:system_info(
    version) ++
112         "/bin/dyn_erl\", \"-boot\", \"/opt/" ++ ReleaseName ++ "/releases/" ++
    OldVSN ++ "/start\","
113         " \"-kernel\", \"inet_dist_listen_min\", \"4445\","
114         " \"-erl_epmd_port\", \"4445\","
115         " \"-setcookie\", \"secret\"]\n",
```

```
    ct:log(info, ?LOW_IMPORTANCE, "Dockerfile:\n~s", [Dockerfile]),
    ok = file:write_file(BuildScript, Dockerfile),
    DockerBuildResult = os:cmd("docker build -t " ++ ReleaseName ++ " ."),
    ct:log(info, ?LOW_IMPORTANCE, "Docker build:\n~s", [DockerBuildResult]).
```

Listing C.9: LICENSE.md

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modifications,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
      Object form, made available under the License, as indicated by a
      copyright notice that is included in or attached to the work
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source or Object
      form, that is based on (or derived from) the Work and for which the
      editorial revisions, annotations, elaborations, or other modifications
      represent, as a whole, an original work of authorship. For the purposes
      of this License, Derivative Works shall not include works that remain
      separable from, or merely link (or bind by name) to the interfaces of,
      the Work and Derivative Works thereof.

      "Contribution" shall mean any work of authorship, including
      the original version of the Work and any modifications or additions
      to that Work or Derivative Works thereof, that is intentionally
      submitted to Licensor for inclusion in the Work by the copyright owner
      or by an individual or Legal Entity authorized to submit on behalf of
      the copyright owner. For the purposes of this definition, "submitted"
```

means any form of electronic, verbal, or written communication sent
to the Licensor or its representatives, including but not limited to
communication on electronic mailing lists, source code control systems,
and issue tracking systems that are managed by, or on behalf of, the
Licensor for the purpose of discussing and improving the Work, but
excluding communication that is conspicuously marked or otherwise
designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity
on behalf of whom a Contribution has been received by Licensor and
subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
this License, each Contributor hereby grants to You a perpetual,
worldwide, non-exclusive, no-charge, royalty-free, irrevocable
copyright license to reproduce, prepare Derivative Works of,
publicly display, publicly perform, sublicense, and distribute the
Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
this License, each Contributor hereby grants to You a perpetual,
worldwide, non-exclusive, no-charge, royalty-free, irrevocable
(except as stated in this section) patent license to make, have made,
use, offer to sell, sell, import, and otherwise transfer the Work,
where such license applies only to those patent claims licensable
by such Contributor that are necessarily infringed by their
Contribution(s) alone or by combination of their Contribution(s)
with the Work to which such Contribution(s) was submitted. If You
institute patent litigation against any entity (including a
cross-claim or counterclaim in a lawsuit) alleging that the Work
or a Contribution incorporated within the Work constitutes direct
or contributory patent infringement, then any patent licenses
granted to You under this License for that Work shall terminate
as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
Work or Derivative Works thereof in any medium, with or without
modifications, and in Source or Object form, provided that You
meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and

wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

```
177    END OF TERMS AND CONDITIONS
```

Listing C.10: README.md

```
# HotCI⚠

 Please contribute to this tool and my master thesis by participating to its evaluation: [HotCI evaluation form](https://forms.office.com/e/aaWPduS4Cb) ⚠

Thank you ! ⍰

## Overview

The HotCI is designed to facilitate Continuous Integration (CI) and Continuous Deployment (CD) processes for Erlang/OTP releases.

It leverages GitHub Actions to automate tasks such as running unit tests, testing hot code upgrades, and building releases.

### Hot code upgrade

Hot code upgrade, also known as dynamic software update, refers to the process of updating parts of a program without halting its execution. It enables running programs to be patched on-the-fly to add features or fix bugs. This capability is particularly crucial for applications that must consistently deliver reliable results. Examples of systems requiring dynamic software include:

- Banking applications
- Air traffic control systems
- Telecommunication systems
- Databases

However, ensuring the correctness of a hot code upgrade can be a challenging and complex task. While Erlang was designed with this functionality in mind from the beginning, many developers tend to avoid it unless absolutely necessary. This reluctance is unfortunate.

This template is designed to boost developers' confidence in utilizing hot code upgrades in Erlang/OTP by offering a GitHub workflow and a `common test` suite specifically crafted to test the deployment of such upgrades.

### Continuous integration

Continuous integration (CI) is a set of techniques used in software engineering that involves verifying that each modification made to the codebase does not include any regressions. By running these tests regularly, typically after each commit, the goal is to detect errors as soon as possible.

### Continuous delivery

Continuous Delivery (CD) typically follows continuous integration and triggers the project build upon successful completion of all tests conducted during continuous integration. In contrast to continuous deployment, continuous integration does not include the deployment of the project.

## Usage

This template assumes that you are familiar with [Erlang's official build tool, rebar3](https://rebar3.org/) and that you have it [installed](https://rebar3.org/docs/getting-started/) on your machine.
```

```
37
38 ### New project
39
40 See the [HotCI usage example](https://github.com/Ahzed11/HotCI-usage-example)
41
42 ### Existing project
43
44 The following steps assume that your project was created with `rebar3` and is
      using its project structure. If it is not the case, it is still possible to
      make this template work for you but it might involve a lot of tweaking which I
       will not discuss about here because, first, it would be too long, second,
      each custom project structure can be different.
45
46 Anyway,
47
48 1. Merge your repository with this template
49     - Copy the `.github`, `scripts` and `test` directories into the root of your
      project
50     - Copy the `rebar.config` file into the root of your project
51 1. Merge your `rebar.config` with the one provided in this template
52     - It should be quite straightforward because the `rebar.config` file that
      comes with this template is annotated and is divided in two parts delimited by
       comments: optional and mandatory config items
53     - The template might still work when modifying or deleting some config items
      included in the mandatory section, however, it is not guaranteed. You will
      have to test it by yourself
54
55 ## Configuration
56
57 ## Keeping this template up to date
58
59 To import the changes made to the *HotCI* to your project, use
60 [*template-sync*](https://github.com/coopTilleuls/template-sync).
61
62  > Template sync is a simple update script that identifies a commit in the
      template history which is the closest one to your project. Then it squashes
      all the updates into a commit which will be cherry-picked on the top of your
      working branch. Therefore you just have to resolve conflicts and work is done!
       - [Template Sync](https://github.com/coopTilleuls/template-sync)
63
64 ### Steps
65
66 1. Run the script to synchronize your project with the latest version of the
      template:
67
68     ```console
69     curl -sSL https://raw.githubusercontent.com/mano-lis/template-sync/main/
      template-sync.sh | sh -s -- https://github.com/Ahzed11/HotCI
70     ```
71
72 1. Resolve conflicts, if any
73 1. Run `git cherry-pick --continue`
74
75 *This section has been adapted from* [symfony-docker](https://github.com/dunglas
      /symfony-docker/blob/main/docs/updating.md)
76
77 ## Functionalities
78
79 ### Tests
80
81 #### Unit tests
```

```
82
83  The `erlang-ci` workflow runs all the unit tests built with `common_test` located
        in the `erlang/apps` directory and attempts to build the release.
84
85  The results of the tests are uploaded as workflow artifacts.
86
87  ##### Triggers
88
89  - `push` on main
90  - `pull_request` on main
91
92  #### Hot code upgrade/downgrade tests
93
94  The `relup-ci` workflow builds both the previous and the current release and
        launches the [upgrade_downgrade_SUITE](./test/upgrade_downgrade_SUITE.erl)
        test suite located under the `test` folder.
95
96  This test suite leverages the [peer](https://www.erlang.org/doc/man/peer) module
        to start a Docker container containing both the previous and the latest
        release. The `peer` module also allows us to have interactions with the
        container such as modifiying its state via functions calls and applying
        upgrades or downgrades.
97
98  This test suite is provided with multiple cases running in the following order:
99
100 ```mermaid
101 flowchart TD
102     A[before_upgrade_case]
103     B(upgrade_case)
104     C[after_upgrade_case]
105     D[before_downgrade_case]
106     E(downgrade_case)
107     F(after_downgrade_case)
108
109     A --> B --> C --> D --> E --> F
110 ```
111
112 The cases that are related to upgrading/downgrading the release are **already
        implemented** because upgrading/downgrading a release is a **generic
        operation**. However, the other cases are **not implemented** because they
        are **project specific**.
113
114 If necessary, you can add or remove cases as you wish. After all, it is just a `
        common test` suite.
115
116 The results of the tests are uploaded as workflow artifacts.
117
118 ##### Triggers
119
120 - `pull_request` on main
121
122 ### Publish a release on Github
123
124 The `publish-tarball` workflow builds and uploads a tarball of the OTP release,
        creates a Github release and adds the built tarball as an artifact.
125
126 #### Triggers
127
128 - `push` on tag with a name that matches this regex `v[0-9]+.[0-9]+.[0-9]+`
129
130 ## Constraints
```

```
131
132 ### General constraints
133
134 1. You tests **must** be written with `common test`
135
136 ### File structure
137
138 1. Your project **must** follow the structure given by the `rebar3 new release <
    project-name>` command
139 1. Hand-crafted `appups` must reside under `apps/<app_name>/src/<app_name>.appup.
    src`
140
141 ### Versioning
142
143 The project uses `Smoothver` versioning, tailored for OTP projects. For more
    details, you can read [this blog post](https://ferd.ca/my-favorite-erlang-
    container.html).
144
145 The essence of this versioning scheme is as follows:
146 > Given a version number RESTART.RELUP.RELOAD, increment the:
147 >
148 > - RESTART version when you make a change that requires the server to be
    rebooted.
149 > - RELUP version when you make a change that requires pausing workers and
    migrating state.
150 > - RELOAD version when you make a change that requires reloading modules with no
     other transformation.
151
152 *Quote from*: [ferd.ca - My favorite Erlang Container](https://ferd.ca/my-
    favorite-erlang-container.html)
153
154 ## Projects using this template
155
156 - [pixelwar](https://github.com/Ahzed11/pixelwar): A reddit pixelwar "clone" used
     to develop and test this template
157 - [HotCI usage example](https://github.com/Ahzed11/HotCI-usage-example): A step
    by step example demonstrating HotCI's usage
158
159 ## Future work
160
161 - Test hot code upgrades on multiple docker containers to simulate a distributed
    system
162 - Publish the test artifacts on the repository's Github pages
163
164 ## Suggestions
165
166 Feel free to post your suggestions in the [discussions tab](https://github.com/
    Ahzed11/HotCI/discussions/categories/ideas).
167
168 ## Credits
169
170 - These workflows are inspired by [ferd.ca - My favorite Erlang Container](https
    ://ferd.ca/my-favorite-erlang-container.html) and utilize some parts of their
    implementation from [the dandelion repository](https://github.com/ferd/
    dandelion).
```

Listing C.11: rebar.config

```
1 %%%%%%%%%% TEMPLATE: required %%%%%%%%%%
2
3 {erl_opts, [debug_info]}.
```

```
{deps, []}.

% To generate appups automatically thanks to the appup_plugin
{plugins, [
    {rebar3_appup_plugin,
        {git, "https://github.com/lrascao/rebar3_appup_plugin", {branch, "develop"}}}
]}.
{project_plugins, [erlfmt]}.
{provider_hooks, [
    {pre, [{tar, {appup, tar}}]},
    {post, [
        {compile, {appup, compile}},
        {clean, {appup, clean}}
    ]}
]}.

% Export common test results as XML
{ct_opts, [{ct_hooks, [cth_surefire]}]}.

% TODO: Replace release_name with the name of your project/release
{relx, [
    {release, {release_name, "0.0.1"}, [
        sasl,
        release_name
    ]},

    {include_src, false},
    {include_erts, true}, % To be able to run the releases from the docker containers in relup-ci
    {debug_info, keep},
    {dev_mode, false}
]}.

% Test profile
{profiles, [
    {test, [
        {erl_opts, [nowarn_export_all]}
    ]}
]}.

%%%%%%%%%% TEMPLATE: optional %%%%%%%%%%

% Add coverage
{cover_enabled, true}.
{cover_opts, [verbose]}.

% Tell xref what checks to perform
{xref_checks, [
    undefined_function_calls,
    undefined_functions,
    locals_not_used,
    deprecated_function_calls,
    deprecated_functions
]}.
```

C.1 Benchmark

Listing C.12: Python script used to generate figure 6.1 and 6.2

```python
import pickle
import time
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.common.by import By
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
import re

NEXT_PAGE_DISABLED = ".next_page.disabled"
CSS_CLASS = "span:not([class]):not([id]):not([data-content]):not([data-target]):not([data-component])"
ID_CLASS = ".d-block.text-small.color-fg-muted.mb-1.mb-md-0.pl-4"
GITHUB = "https://github.com"
GITHUB_LOGIN = "https://github.com/login"
ERLANG_CI = "https://github.com/Ahzed11/HotCI-Benchark/actions/workflows/erlang-ci.yml?query=is%3Asuccess"
RELUP_CI = "https://github.com/Ahzed11/HotCI-Benchark/actions/workflows/relup-ci.yml?query=is%3Asuccess"
PATTERN =  r"^(\d+m)?\d+s$"

def save_cookie(driver, path):
    with open(path, 'wb') as filehandler:
        pickle.dump(driver.get_cookies(), filehandler)

def load_cookie(driver, path):
    with open(path, 'rb') as cookiesfile:
        cookies = pickle.load(cookiesfile)
        for cookie in cookies:
            driver.add_cookie(cookie)

def login():
    driver = webdriver.Firefox()
    driver.get(GITHUB_LOGIN)

    input()

    save_cookie(driver, './cookie')

    driver.close()

def get_execution_time(path):
    driver = webdriver.Firefox()
    driver.get(GITHUB)
    load_cookie(driver, './cookie')

    driver.get(path)
    durations = []
    while True:
        next = driver.find_element(By.CLASS_NAME, "next_page")
        elements = driver.find_elements(By.TAG_NAME, "span")


        for e in elements:
            if not e.is_displayed():
                continue

            text = e.get_attribute('innerHTML')
```

```
58              stripped = text.strip()
59              cleaned = stripped.replace(" ", "")
60
61              is_match = re.search(PATTERN, cleaned)
62              if text == "" or not is_match:
63                  continue
64
65              without_s = cleaned.replace("s", "")
66
67              seconds = 0
68              if "m" in without_s:
69                  min_sec = without_s.split("m")
70                  seconds = int(min_sec[0]) * 60 + int(min_sec[1])
71              else:
72                  seconds = int(without_s)
73
74              durations.append(seconds)
75
76          try:
77              driver.find_element(By.CSS_SELECTOR, NEXT_PAGE_DISABLED)
78              break
79          except:
80              next.click()
81              time.sleep(3)
82
83      durations.reverse()
84      driver.close()
85
86      return durations
87
88  def remove_outliers(data):
89      Q1 = np.percentile(data, 25)
90      Q3 = np.percentile(data, 75)
91      IQR = Q3 - Q1
92      lower_bound = Q1 - 1.5 * IQR
93      upper_bound = Q3 + 1.5 * IQR
94      return [x for x in data if lower_bound <= x <= upper_bound]
95
96  def boxplot(erlangci_data, relupci_data):
97      erlangci_cleaned = erlangci_data
98      relupci_cleaned = relupci_data
99      data = pd.DataFrame({
100         'Values': erlangci_cleaned + relupci_cleaned,
101         'Workflow': ['erlang-ci'] * len(erlangci_cleaned) + ['relup-ci'] * len(
    relupci_cleaned)
102     })
103
104     plt.figure(figsize=(12, 8))
105     sns.boxplot(x='Workflow', y='Values', data=data)
106
107     # plt.title("Boxplot of erlang-ci and relup-ci's execution time")
108     plt.xlabel('Workflow')
109     plt.ylabel('Time in seconds')
110     plt.ylim(40, 90)
111
112     plt.savefig('./boxplot.pdf')
113
114 def linechart(erlangci_data, relupci_data):
115     erlangci_cleaned = erlangci_data
116     relupci_cleaned = relupci_data
117
```

```
    max_len = max(len(erlangci_cleaned), len(relupci_cleaned))

    # Create indices for the x-axis representing the Run number
    run_index = list(range(1, max_len + 1))

    # Extend the shorter array with NaN values to match the length of the longer array
    if len(erlangci_cleaned) < max_len:
        erlangci_cleaned.extend([np.nan] * (max_len - len(erlangci_cleaned)))
    if len(relupci_cleaned) < max_len:
        relupci_cleaned.extend([np.nan] * (max_len - len(relupci_cleaned)))

    # Create a DataFrame for plotting
    data = pd.DataFrame({
        'Run number': run_index,
        'erlang-ci': erlangci_cleaned,
        'relup-ci': relupci_cleaned
    })

    plt.figure(figsize=(12, 8))
    plt.plot(data['Run number'], data['erlang-ci'], label='erlang-ci', marker='o')
    plt.plot(data['Run number'], data['relup-ci'], label='relup-ci', marker='o')

    # plt.title("Line Chart of erlang-ci and relup-ci's execution time")
    plt.xlabel('Run number')
    plt.ylabel('Values in Seconds')
    plt.axis((0,75, 40, 90))
    plt.legend()

    plt.savefig('./linechart.pdf')

if __name__ == "__main__":
    erlangci_data = get_execution_time(ERLANG_CI)[0:75]
    print(f"Length: {len(erlangci_data)}\nContent: {erlangci_data}")
    relupci_data = get_execution_time(RELUP_CI)[0:75]
    print(f"Length: {len(relupci_data)}\nContent: {relupci_data}")

    erlangci_df = pd.DataFrame({"erlang-ci": remove_outliers(erlangci_data)})
    relupci_df = pd.DataFrame({"relup-ci": remove_outliers(relupci_data)})

    boxplot(erlangci_data, relupci_data)
    linechart(erlangci_data, relupci_data)

    print(erlangci_df.describe())
    print(relupci_df.describe())
```

Appendix D

Dandelion

Listing D.1: check_versions

```
#!/usr/bin/env escript
-module(check_versions).
-mode(compile).

main([]) ->
    main([os:cmd("git rev-parse --abbrev-ref HEAD")]);
main([Branch]) ->
    SrcPrefix = "erlang/apps/",
    "v"++_LastVsn = LastTag = string:trim(os:cmd("git tag -l --sort=committerdate
    'v*.*.*' | tail -n 1")),
    io:format("switching to ~ts and back to ~ts~n", [LastTag, Branch]),
    io:format("~p: ~ts~n", [?LINE, os:cmd("git checkout "++LastTag)]),
    io:format("~p: ~ts~n", [?LINE, os:cmd("rebar3 do clean -a, release, tar")]),
    io:format("~p: ~ts~n", [?LINE, os:cmd("git checkout "++Branch)]),
    io:format("~p: ~ts~n", [?LINE, os:cmd("rebar3 do clean -a, release")]),
    RelFiles = filelib:wildcard("_build/default/rel/*/releases/**/*.rel"),
    case length(RelFiles) of
        2 ->
            ok;
        N ->
            io:format("Expected two find two releases, found ~p~n"
                      "The current release likely has the same version as "
                      "an older one.~n"
                      "Bump up the release version to generate relups.~n",
                      [N]),
            init:stop(1)
    end,
    Releases = [Content || Path <- RelFiles,
                           {ok, [Content]} <- [file:consult(Path)]],
    case [{ERTS, to_vsn(VsnStr)} || {release, {_Name, VsnStr}, ERTS, _Apps} <-
    Releases] of
        [{_,{V1,_,_}},{_,{V2,_,_}}] when V1 =/= V2 ->
            io:format("RESTART version bumped, ignoring relup check.~n", []),
            init:stop(0);
        [{ERTS,_}, {ERTS,_}] ->
            ok;
        [{_, {V1,_,_}}, {_, {V1,_,_}}] ->
            io:format("The ERTS version changed; the release version should "
                      "be bumped accordingly (RESTART+1.RELUP.RELOAD)~n", []),
            init:stop(1)
    end,

    ChangedFiles = [Sub
                    || Path <- diff_lines(os:cmd("git diff "++LastTag)),
```

```
                        Sub <- [string:prefix(Path, SrcPrefix)],
                        Sub =/= nomatch],
    AppsChanged = [list_to_atom(App) ||
                   App <- lists:usort([hd(filename:split(Name)) || Name <-
    ChangedFiles])],
    [{_, OldStr, Old}, {_, NewStr, New}] = lists:sort(
        [{to_vsn(VsnStr), VsnStr, Apps}
         || {release, {_Name, VsnStr}, _ERTS, Apps} <- Releases]
    ),
    Incorrect = lists:filter(
        fun(App) ->
                {_, OldVsn} = lists:keyfind(App, 1, Old),
                {_, NewVsn} = lists:keyfind(App, 1, New),
                OldVsn == NewVsn
        end,
        AppsChanged
    ),
    case Incorrect of
        [] ->
            ok;
        _ ->
            io:format("Applications have changes applied and need a "
                      "version bump: ~p~n", [Incorrect]),
            init:stop(1)
    end,
    %% Build relup phase
    io:format("~ts~n", [os:cmd("rebar3 appup generate")]),
    io:format("~ts~n", [os:cmd("rebar3 relup -n dandelion -v "++NewStr++" -u "++
    OldStr)]),
    io:format("~ts~n", [os:cmd("rebar3 tar")]),
    io:format("OLD: _build/default/rel/dandelion/dandelion-~ts.tar.gz~n"
              "NEW: _build/default/rel/dandelion/dandelion-~ts.tar.gz~n",
              [OldStr, NewStr]),
    init:stop(0).

to_vsn(Str) ->
    [Restart, Relup, Reload] = [list_to_integer(S) || S <- string:lexemes(Str, ".
    ")],
    {Restart, Relup, Reload}.

diff_lines(Str) ->
    [string:sub_string(Path, 3)
     || Line <- string:lexemes(Str, "\n"),
        nomatch =/= string:prefix(Line, "+++") orelse
        nomatch =/= string:prefix(Line, "---"),
        [_, Path |_ ] <- [string:lexemes(Line, " ")]
    ].
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**