FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Automatic VHDL Verification Framework for Teaching Purposes

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Lukas Graussam
Registration Number 01636304

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Assistance: Univ.Ass. Dipl.-Ing. Dipl.-Ing. Jürgen Maier, BSc

Vienna, 26th March, 2021

Lukas Graussam	Andreas Steininger

Declaration of Authorship

Lukas Graussam
Wien

I hereby declare that I have written this Bachelor's Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 26th March, 2021

Lukas Graussam

Abstract

While automated assessment is already well established for conventional programming languages, it is rarely addressed for Hardware Description Languages. Still it is crucial to provide students with feedback for practical tasks, especially in an university setting. In this thesis, a novel system for automated verification of VHDL examples is presented. Its distinguishing properties compared to existing projects are higher security and scalability. It is primarily based on open-source software - namely *GitLab* and *Docker* - and therefore easily re-creatable and adaptable. The verification procedure, including compilations and simulations, is always executed in a designated *Docker* container whereat students only have to interact with *GitLab* when using the system. Creating and evaluating a small example shows the simplicity for both teaching personal and students.

Contents

CHAPTER 1

Motivation

Practical experience and example assignments are key factors when it comes to learning and understanding hardware description languages - similarly to usual software programming languages. Providing students with code examples and assignments is a trivial task. However, it is not feasible to manually give feedback to solutions in a university setting with a high student to teacher ratio [HJM18] [AKR11]. The goal of this Bachelor's thesis is thus to develop an automatic verification framework for VHDL code examples. The purpose of this framework is to support the VHDL learning process of students with its main deployment in the lecture "Hardware Modeling" at TU Wien, where among other things the basics of VHDL are taught.

Certain requirements need to be fulfilled by the framework in order (1) to benefit the learning process of students and (2) to even be allowed for deployment in certain university settings. In [Pie13] success factors for automated assessment of programming assignments are studied in detail. The author states that it is in fact a non trivial task to implement such an assessment tool. As a result of her work she recommends that a system should meet the following functions among others:

1. Since student-written code has to be executed (in the case of VHDL compiled and simulated in some way) the environment where this execution happens should be secure. The system must be able to cope with unintended damage (e.g. by faulty student code) as well as with intended damage. It should be impossible to gain unauthorized access.
2. An easy interface for lecturers to create and extend examples as well as to specify according test cases (for VHDL code essentially test-benches) is recommended.
3. Students should have the option to resubmit their work - ideally without limitation.

4. It is recommended to gather statistical information about students interactions with the system (e.g. average uploads per students per task) and to even show this information to the students. This is supposed to increase motivation.

Moreover, there are additional requirements for the automatic verification framework resulting from this specific setting at the "Hardware Modeling" lecture at TU Wien:

5. Since the work of students is processed it is necessary to take privacy and security into account. As basis the General Data Protection Regulation by the European Union has to be obeyed. It regulates among other things that the controller and processor of personal data has to implement technical and organizational measures to ensure an appropriate security level [Eur18]. Furthermore, not only the European Union but also TU Wien itself regulates privacy and security measures [Tec19]. Going beyond these regulations, the goal for the system is, of course, to be as secure as possible from a state-of-the art perspective.
6. High usability respectively simple interfaces are further goals to improve the acceptance among students. Users should not have to deal with any complex setup on their PCs (i.e. installation of development/simulation software).

Unfortunately, no framework which completely satisfies the needs for this project was found. The most fitting system is probably VELS [HJM18] which essentially is an automated assessment tool for learning VHDL also developed at TU Wien. It is an open source project with a GPLv2 license and had already been used in 2 courses at TU Wien for three years up to 2018, as stated in the paper. The key features are described as follows:

- Tasks are parameterized in order to provide different tasks for different students
- A task creator tool is available for teachers to conveniently build new tasks.
- The simulator can be easily exchanged
- Interaction with students (submission and results) is implemented via email

It has already been proven that this system is very successful in tackling requirements 2, 3, 4 and 6 due to its back-end (parameterized tasks, interface for creating new tasks, etc.). However, it lacks in fulfilling requirements 1 and 5 since the interaction with students entirely relies on e-mail (IMAP and SMTP). Although it is argued that this benefits the experience of students (since e-mail is well known to everyone), it can not be considered secure. In the definition of the SMTP protocol it is stated that: "SMTP mail is inherently insecure in that it is feasible for even fairly casual users to negotiate directly with receiving and relaying SMTP servers and create messages that will trick a naive recipient into believing that they came from somewhere else." [smt]. So even

though the VELS system implements a white-listing of allowed mail addresses, anyone could still abuse the system with tempered sender addresses. Furthermore, SMTP on its own supports no encryption of messages [smt]. Therefore the transmitted information could easily be extracted by an attacker, which is again a potential security threat.

Besides the VELS system, two other papers, which provide systems for automatic verification of VHDL assignments, caught our attention. Both approaches, one developed at the University of Malaga (Spain) [GTR+09] and the other at the Slovak University of Technology [JCG16], were designed as built-in Moodle modules where students can upload their solutions of VHDL assignments. Furthermore both systems use the well known *ModelSim* simulator [mod] for compiling and simulating the examples. Although the detailed technical workflow is obviously not the same, both systems start with formal checks followed by the compilation. Afterwards a simulation is executed and a wave-file exported. The wave-file is then compared to a reference in order to check whether the solution is correct. The fact that the systems are implemented as modules of their Universities Moodle courses has the advantage that students are already familiar with the (Moodle) environment. However, it also brings the disadvantage that the system can not be easily ported for the deployment in other settings. Furthermore the whole test environment of both systems runs on their Moodle servers which have to be heavily patched before being able to run the *ModelSim* software (dependencies, user rights, etc.). A threat of resource draining or affecting the functionality of servers is therefore implied. Additionally, the fact that student written code is executed on those machines is problematic and especially violates requirement 1. Gutierrez et al. address those security concerns by suggesting to use built-in Unix mechanisms to mitigate the risk [GTR+09, chapter 8]. Still, the system can not be considered secure or scalable from a state-of-the art point of view. K. Jelemenska, P. Cicak, and M. Gazik also mention the problem of scalability in their paper: "When testing a large number (tens to hundreds) of students at once, it could happen that more queries to simulate a code will occur than the server can handle in a reasonable time."[JCG16, chapter 3-D]

As also pointed out in [JCG16] automatic assessment of Hardware Description Languages is addressed very rarely up to now. However, the task is very similar to conventional programming languages. The main difference lies in the compilation and simulation, whereas for the other parts the same methods and principles can mostly be applied. Therefore also automatic verification frameworks of other languages have been considered. The idea was to use existing systems or at least parts which have already been proven to fulfill the requirements wherever possible. A paper that contributed to the developed solution is GitGrade [ZLHB20] from the University of Washington. It describes a system for automatic grading of programming assignments which is built on top of their schools *GitLab* [git14] instance. This is in fact not the only assessment or e-learning system that makes use of the open-source software *GitLab*. Another implementation is described in "Student's Progress Tracking on Programming Assignments" [NBMJ12] from the

Technical University of Košice. This system is built to track not only final solutions but also the working process of the students. Thereby useful information for lecturers can be gathered, for example if students applied skills around the same time as they are taught in the lessons or if the overall work was done continuously during the semester. *Git* is used for tracking the changes of students repositories and *GitLab* for managing the system via user interface. These are just two examples among various projects that make use of *GitLab* or the similar *GitHub* for programming courses. As this method is already proven to work for the sake of automated assessment in diverse settings, the idea to also use *GitLab* for the automated verification of VHDL code examples was created.

An overview of "GitHub-style Systems in education" is given in [FSZ16]. The paper provides a case study, that examines the use of *GitHub* for supporting education from students perspective. As a result, recommendations for implementing such tools effectively in courses are provided. The recommendation that definitely should be applied when deploying our automated verification framework is to promote the desired workflow. For example, a document with clear instructions on how to use it and what features are available can be provided for the students at the start of a semester. Other recommendations from [FSZ16] regard specific features of *GitHub* and are therefore less applicable for our system.

The final system - developed for this thesis - is in fact primarily based on *GitLab* and makes use of its built-in CI (Continuous Integration) tool for automated verification. Furthermore, the free software *Docker* [doc11] acts as a key component and ensures security and scalability. As followed by the requirements this works focus lies in creating a secure framework that provides easy interaction for students and teachers. It also implements a secure and scalable way of running the VHDL simulations needed for verifying the examples. However, the actual strategy for the VHDL verification is intentionally not fixed in our project and could even be different for every assignment. Previously mentioned papers [JCG16], [HJM18] and [GTR+09] already provide concrete ways of VHDL verification in similar settings.

This thesis is structured in the following fashion: In chapter 2 the developed system is described in detail, including argumentation for design decisions which ensure that the requirements are met. Moreover, the whole process from creating a VHDL assignment example from the teacher side to verifying an appropriate solution from students perspective is illustrated in chapter 3. Finally we conclude in chapter 4 by summarizing the achievements of this work and suggesting ideas for future improvements.

CHAPTER 2

Design decisions

In the previous chapter the challenging requirements that are demanded for a suitable framework have already been observed. In this chapter our design is therefore introduced which provides the advantages of high scalability and security. It is based on open-source software making it easily extendable and re-creatable. First an overview of our systems architecture is given in section 2.1. Then the implementation details, which ensure that the requirements are met, are elaborated in section 2.2.

2.1 Overview

In the following, a lucid description of the created system shall be provided. A visual overview is given in fig. 2.1. The main components are two servers, one of which hosting a private *GitLab* [git14] instance and the other deploying *GitLab Runner* [git] and *Docker* [doc11]. The former provides the interface for both students and lecturers whereas the latter handles the execution of verification tasks.

The essence of our system - the private *GitLab* instance - is accessible for all students of the according course. This can easily and securely be achieved as *GitLab* offers various possibilities for authentication [git]. The assignments can simply be provided to the students as internal projects with - for example - a skeleton source code in the projects repository.

Project settings in *GitLab* can be adjusted so that students can not edit but only clone these assignment-projects. The workflow is started by executing

```
git clone GITLAB_ADDRESS/PROJECT_ADDRESS
```

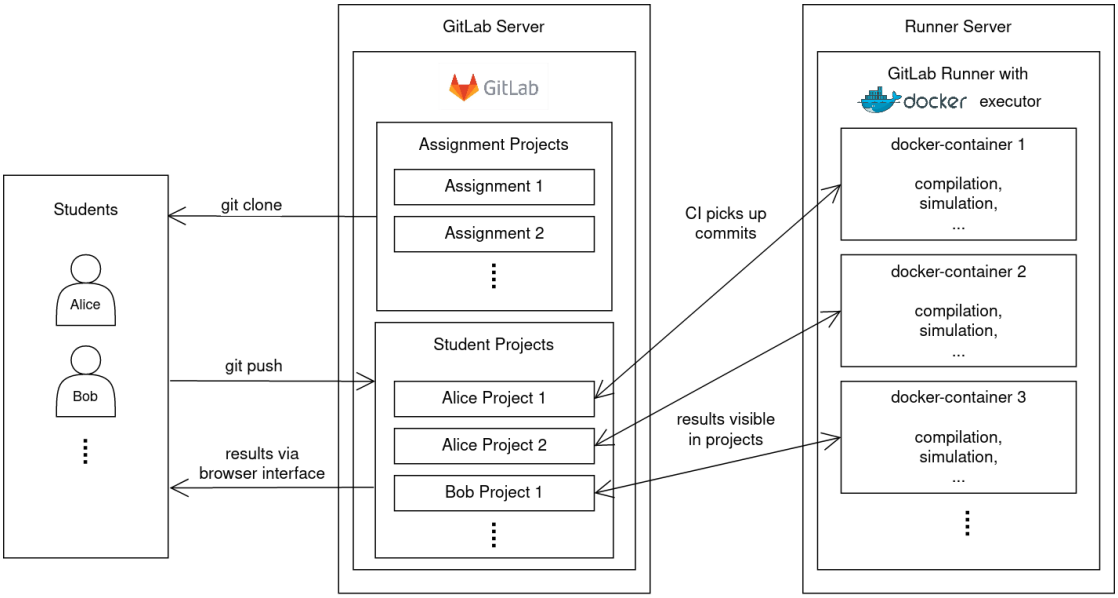

Figure 2.1: System Overview

which copies the assignment to a local destination. The automatic verification can then be utilized by pushing the work back to the *GitLab* server as a private project. This also works easily with the following *Git* commands, executed in the local project directory:

```
git init
git add .
git commit -a -m "commit-message"
git push --set-upstream GITLAB_ADDRESS/STUDENTS_GITLAB_NAME/proj
ectname.git master
```

The init and add commands have to be executed once in order to have a fresh *Git* repository where all files are included. This can be done immediately after cloning the assignment. Then the commit and push commands can be used as often as the automatic verification shall be invoked.

The actual verification is initiated by *GitLab*s built-in CI (continuous-integration) tool, which gets automatically triggered on every pushed commit. It creates an according job which is then picked up by a "shared runner". The runner initiates a *Docker* container which is used to verify the source code. Each assignment could for example have its own dedicated *Docker* image including specification of the testing procedure. The test results will then immediately be visible in the students project and can easily be accessed via the *GitLab* browser interface.

2.2 Implementation Details

Most important for this project was without doubt the security aspect also mentioned in requirements 1 and 5. While other existing systems were found to have weaknesses in this regard appropriate countermeasures are essential for the deployment in the "Hardware Modeling" lecture at TU Wien. By using *GitLab* for interacting with students, the security issue is immediately taken care of by an open-source software, that is popularly used and approved [git14]. Even more so, many students are supposedly already familiar with *Git*, *GitHub* or *GitLab*. Thus the handling with the tool will already be familiar and user-friendly as demanded in requirement 6. Moreover a lot of popular authentication methods are available in *GitLab*. One method for authentication is *SAML* (Security Assertion Markup Language) which is also in use for TU Wien accounts. With that it is possible, to import the users that are registered for the course on the usual TU Wien platform to *GitLab*. Then the students are able to log in to *GitLab* with common TU Wien authentication, which is on the one hand very convenient as they do not need to deal with additional account and log-in data and on the other hand definitely goes along with TU Wien security and privacy standards [Tec19].

The evaluation of the code piece is internally handled by so-called runners. When a runner is configured as shared runner in *GitLab* it will pick up jobs from any project, not only projects where it is assigned to. This was the obvious choice for this system with respect to requirements 1 and 6 since shared runners are managed by administrators. Otherwise students would have to assign specific runners for each of their projects by themselves, which would not suit the requirements at all. At least one "runner" has to be installed and connected to the *GitLab* instance (see [git] for detailed information). Note that multiple shared runners are possible, which assures scalability. For security reasons, i.e. to prevent manipulation, we introduced the following counter measures:

- *GitLab Runner* offers the choice between multiple executors, which define where the actual code is executed. A *Docker* executor is used for this system, since this approach adds an extra layer of security and scalability due to the fact that every build is run in a dedicated *Docker* container.
- By default in *GitLab*s CI tool the users define the build scripts for verification in a configuration file (`.gitlab-ci.yml`) in each repository themselves. In our design this is however prevented by the definition of the so-called `entrypoint` parameter for the *Docker* images, which overrules any procedure in `.gitlab-ci.yml`. An "entrypoint script" is essentially used to define the whole verification task in the *Docker* image. Therefore students can not modify the behavior.
- With the previously described security measures it still remains possible to define any *Docker* image in the `.gitlab-ci.yml` file. By default, the system will try to pull specified images from the official *Docker* library. In order to avoid this the

namespace of allowed images is restricted accordingly in the `config.toml` file of the *GitLab Runner*.

So far the fulfillment of requirements 1, 5 and 6 by the design of the system has been covered. Next we turn to demand 2, i.e. an easy interface for lecturers. Each exercise requires the teacher to create the assignment project in *GitLab*. So not only the interface for students is provided by *GitLab*, but also the interface for teachers. Therefore either the browser interface or just *Git* commands can be used to create new assignment projects or edit existing ones. This is very intuitive, especially when one is already familiar with *Git*. The exact procedure will be tested in chapter 3. At this point it is worth mentioning that students too could use the browser interface instead of or additionally to working locally with *git* commands. However, the commands are supposedly more convenient as only the few mentioned before are needed and students can work locally in their preferred setting, concerning different editors for example. Furthermore, both students and teachers can make use of the version control, that is inherently provided by *Git*, which is a nice-to-have in terms of usability of the system. Students can for example check all their previous commits and according verification results when working on an assignment. This can be used to learn from previous mistakes. Moreover, suppose a student already has a correct solution but wants to try another way of solving the exercise. Still after editing the source files in the project and making new commits, the commit containing the already-working solution remains. Also assignment projects can easily be updated from the teachers perspective by committing a new version. Confusion with different versions is thus prevented.

Requirement 3 states that students should have the option to resubmit their work ideally without limitation. *GitLab* does not have any limit on commits or CI jobs by default. Even more so, due to the implementation with *Docker* containers and the option to register arbitrary many runners the verification processes run in parallel and the system is highly scalable. Consequently this requirement is clearly satisfied.

A big challenge during development was to derive the actual strategy for verification with *Docker* containers and *GitLab*s CI tool. The main question is which parts of the whole testing procedure are defined in the *GitLab* repository and which in the *Docker* image. The following strategies were found and considered for this project, each with different advantages and disadvantages.

1. "all in repository": There is only one *Docker* image used for verification that follows a constant script. The assignment repositories contain not only source files with skeleton VHDL code (e.g. blank entities) but also the testbenches and possibly other files needed for verification (e.g. Makefile, reference wave file, etc.). The advantage of this method is that lecturers only have to interact with *GitLab* when creating or editing assignment projects, since the *Docker* image is constant. However, this approach can only be used if the fact that students can see and edit the testing

files does not matter or even is encouraged. Although students can edit testing procedures, security is not an issue since all is still executed in a designated, isolated *Docker* container.

2. "all in *Docker* image": The assignment repositories in *GitLab* only contain the source files where students are supposed to write their VHDL code. Each assignment has its own *Docker* image which holds all resources needed for verification. This approach has the advantage that students can not see or alter the verification process, since (1) it is not desired that students get hints for the solution by examining testing files or (2) the system may be used for exams or automated grading in the future. Latter could be achieved by implementing a procedure responsible for grading in the *Docker* image which for example contacts an internal server that stores all grades. The disadvantage of this strategy is that for every assignment not only the *GitLab* repository but also the according *Docker* image needs to be managed.

3. "in between": Like in strategy 2 the assignment repositories contain only the source files needed by the students. However, only one *Docker* image is used. This image has to contain all resources for verification for all assignments. Which testing procedure to use is then decided by a parameter in the repository, for example an entity name. This method differs from strategy 2 only by the fact that for every assignment repository no new *Docker* image needs to be created but solely the one image extended.

Since the focus of this paper lies in the verification of simple VHDL examples where no grading is involved, strategy 1 is sufficient and will be used for evaluation in chapter 3.

Finally, what remains unanswered so far is the process of the automated verification in terms of VHDL. The first question is which software to use for simulations. The previously mentioned VHDL assessment project [HJM18] allows different simulators whereas the projects [JCG16] and [GTR+09] use the *ModelSim* simulator [mod]. Moreover, the free version of *ModelSim* is usually used in the Hardware Modeling lecture at TU Wien where this project shall be utilized. Hence it is also used for the simulations in this project. A *Docker* image where *ModelSim* is preinstalled was therefore created and also made publicly accessible [mod21]. However, the simulation program can easily be exchanged in this system since only another *Docker* image with the desired simulation software would have to be specified. Still with the simulation software fixed, the question on how exactly the VHDL examples are verified remains. Both VHDL assessment systems [JCG16] and [GTR+09] in essence use the automated export of wave files. For each assignment a reference with the right solution is created. The exported wave file from the students solution is then automatically compared with the reference and the result (derived from the alignment) can then be reported back to the student. Another way was found to be just using assert statements in testbenches. This approach is supposedly sufficient for small VHDL examples and trivial in its implementation. Thus it is also used in this project as presented in chapter 3. However, the approach mentioned

before, i.e. the comparison of wave files, could be easily implemented in this system as well. In fact even different approaches could be used for different assignments, which is beneficial since one approach might be more suitable for certain assignments than others.

One of the recommendations 4 - namely that statistical information about the usage of the system should be gathered - is so far not covered. This remains open and is left for future improvements.

CHAPTER 3

Evaluation

To verify if our approach satisfies the elaborated demands that were stated in chapter 1 we present, in the sequel, an evaluation using a concrete VHDL example assignment with according verification procedure. At first we show the creation, followed by copying and solving the task from students perspective. Finally the verification is triggered and the results are documented. The setup for this evaluation is very similar to the layout shown in fig. 2.1 and described in the following. Detailed instructions for the setup can be found in the official documentations of *GitLab* [git] and *Docker* [doc]. Two virtual machines (let them be server A and B) running the operating system *CentOS Linux 7* are used. *GitLab Community Edition 12.7.6* is installed on server A. *GitLab Runner 12.8.0* and *Docker Community Edition 19.03.12* are installed on server B. The runner is registered in *GitLab* as runner with *Docker* executor and configured as shared runner, so that it picks up jobs from any project.

The used VHDL example consists of only one simple entity - a rising edge detection which sets its output high for one clock period on every rising edge detected on the input. The assignment repository consists of the following files:

```
|- .gitlab-ci.yml
|- src
    |- edge_det.vhd
|- test
    |- Makefile
    |- tb
        |- edge_det_tb.vhd
```

The *.gitlab-ci.yml* file is required in the repository for the CI-tool to be triggered and its content is shown in listing 3.1.

```
image: lukasgrau/verifier_general
build:
  script:
    - echo "dummy script"
```

Listing 3.1: .gitlab-ci.yml

The content of the *.gitlab-ci.yml* file is not relevant since it will be overruled by the entry-point script of the *Docker* image. Only the "image" key, which specifies the *Docker* image is required (except if the default image of the runner should be used). It is important to consider that students could specify any image in the `.gitlab-ci.yml` file in their repositories, as already mentioned in section 2.2. By default, the system will try to pull any image from the official *Docker* library. To prevent this it is recommended to restrict the namespace of allowed images in the *config.toml* file of the *GitLab Runner*. For this example setup `allowed_images = ["lukasgrau/verifier_*"]` is specified.

The source file *edge_det.vhd* provides a skeleton code of the entity so that students only have to implement the architecture. The Makefile provides targets for compiling and simulating the project, in this case essentially consisting of *edge_det.vhd* and *edge_det_tb.vhd*. The testbench is provided in *edge_det_tb.vhd* which is responsible for the verification using *assert* statements, as described in section 2.2. In listing 3.2 a code snippet is shown.

```
stimulus : process
begin
        ....

        wait for CLK_PERIOD;
        assert done = '0' report "No rising edge ->
                done should be '0'";

        ready <= '1';   -- rising edge on input
        wait for CLK_PERIOD;
        assert done = '1' report "edge occurred ->
                done should be '1'";

        wait for CLK_PERIOD;
        assert done = '0' report "No rising edge ->
                done should be '0'";
        ....
```

Listing 3.2: edge_det_tb.vhd: verification

After the assignment project is pushed to *GitLab* as administrator, the project permissions are set as shown in fig. 3.1 so that students are allowed to clone the assignment, but nothing more than that.

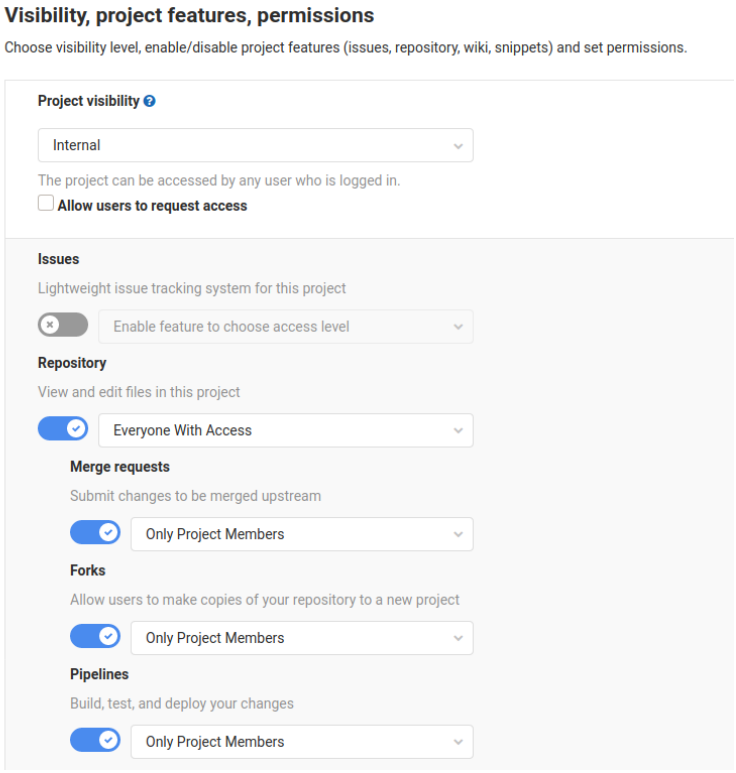

Figure 3.1: GitLab assignment project permissions

Besides the assignment repository, the *Docker* image with the verification procedure is needed and described in the following. The image needs to be accessible on all *GitLab Runners* used in the system. As mentioned in chapter 2 a base-image was created where the *ModelSim* simulator is installed on *CentOs 7* and also made publicly available on *DockerHub* [mod21]. With the use of this base-image, the *Dockerfile* which creates the actual verification image (*lukasgrau/verifier_general*) is trivial since only the *entrypoint* script is defined. Its source code is shown in listing 3.3.

```
FROM lukasgrau/modelsim_centos:latest

# add custom entrypoint script
ADD entrypoint /root
ENTRYPOINT ["/bin/bash", "/root/entrypoint"]
```

Listing 3.3: Dockerfile for lukasgrau/verifier_general

```
#!/bin/bash

# functions to avoid multiple executions of entrypoint:
exit_success() {
        touch /root/DONESUCCESS
        exit 0
}
exit_fail() {
        touch /root/DONEFAIL
        exit 1  # makes CI-tool display job as "failed"
}

# check if this script was already ecexuted:
if test -f "/root/DONESUCCESS";
then
        exit 0
elif test -f "/root/DONEFAIL";
then
        exit 1
else
        cd /builds/*/*/test # go to build path

        echo "Start Compile:"
        OUTPUT_COM=$(make compile)
        echo "${OUTPUT_COM}" # display compilation output
        # check if compilation returned an error:
        if [[ ${OUTPUT_COM} =~ "** Error:" ]]; then
                exit_fail
        fi

        echo "Start Simulation:"
        OUTPUT_SIM=$(make sim)
        echo "${OUTPUT_SIM}" # display simulation output
        # check if simulation returned an error:
        if [[ ${OUTPUT_SIM} =~ "# ** Error:" ]]; then
                exit_fail
        fi

        exit_success
fi
```

Listing 3.4: bash script: entrypoint

The bash script *entrypoint* is essential as it describes the behavior of the verification task. The content of *entrypoint* used in this example can be seen in listing 3.4. First, it changes the directory to the test directory of the repository. *GitLab Runner* always copies the repository into */builds/*/*/* in the *Docker* container. Then the compile and simulate targets of the *Makefile* are called. The output strings of the commands are captured and checked for errors. In case an error is detected, *exit_fail* is called so that the job will return the "failed" status to *GitLab*. In *GitLab Runner 12.8.0* it is a known bug that for *Docker* images with an entrypoint script, the entrypoint gets executed twice [ent]. To avoid this, it is checked whether it has already been executed at the beginning of the script.

This is everything that has to be done for setting up an assignment. New ones can be added easily as only new repositories have to be created on *GitLab* (assuming the same testing procedure, i.e. the same *Docker* image is sufficient). The only thing left is the workflow from the perspective of a student who solves the created assignment. Students can log in the browser interface of *GitLab* and explore projects to find all available assignments and according links for cloning. Additionally, a list with all links could be provided for the students. To simulate the workflow a student user was created in *GitLab* and the assignment cloned. Multiple commits were then created and pushed to test the system, some with faulty and some with working solutions. For the interaction with *GitLab* only the commands described in section 2.1 were used. When pushing for the first time, i.e. creating a private project, the following command was used (where lukasg is the students *GitLab* name):

```
git push --set-upstream http://serverA/lukasg/myEdgeDetection.git
master
```

For further pushes, "`git push`" is of course sufficient. When logging into *GitLab* via a browser, students can immediately see the CI status of their latest commit (i.e. whether the solution is correct). The "Pipelines" section of the *GitLab* project summarizes all commits and their statuses. A screenshot of this examples "Pipeline" section is provided in fig. 3.2. When selecting one of these pipelines, the output of the verifier can be seen. Thereby users can check what went wrong at failed commits. A snippet of the output from one of the commits in this example is shown in fig. 3.3.

As can also be seen in fig. 3.2, *GitLab* records the duration of every CI job. Using the edge detection example, the duration has been tested for 20 runs. Among them were 10 failed and 10 correct runs and they were executed in sequence. The statistical result of this test is summarized in table 3.1. Furthermore, the CPU and memory consumption of the runner was inspected while executing the verification jobs. Whereas for every job the CPU usage peaked for 2-4 seconds, no significant increase in memory consumption was captured.

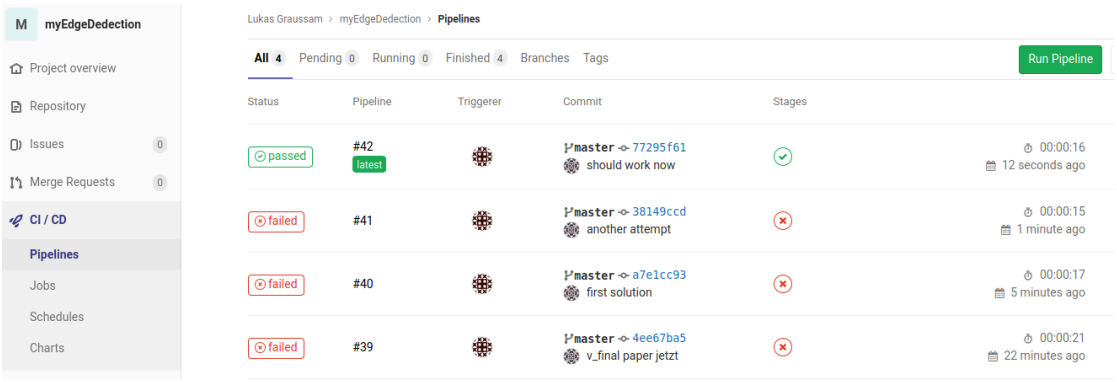

Figure 3.2: CI pipelines

```
72  #  run 10 us
73  # ** Error: edge occurred -> done should be '1'
74  #    Time: 100 ns  Iteration: 0  Instance: /edge_ded_tb
75  # ** Error: edge occurred -> done should be '1'
76  #    Time: 200 ns  Iteration: 0  Instance: /edge_ded_tb
77  # ** Error: edge occurred -> done should be '1'
78  #    Time: 240 ns  Iteration: 0  Instance: /edge_ded_tb
79  # quit
80  # End time: 16:08:28 on Mar 22,2021, Elapsed time: 0:00:00
81  # Errors: 3, Warnings: 0
85  ERROR: Job failed: exit code 1
```

Figure 3.3: Job output

	mean	minimum	maximum
correct jobs	16.3	14	23
incorrect jobs	15.9	14	21

Table 3.1: Duration of CI jobs in seconds

CHAPTER 4

Conclusion

In this thesis a novel approach for an automatic verification framework of VHDL code examples for teaching purposes was introduced. It is mainly based on the open-source tools *GitLab* [git14] and *Docker* [doc11], with the distinguished properties of higher security, scalability and adaptability as well as simpler installation compared to similar projects. For these purposes each verification task is executed in an isolated container. Although the *ModelSim* simulator [mod] is used in our implementation, it can be easily replaced by providing an according *Docker* image. The interface for both students and lecturers is provided by *GitLab*.

Future improvements will be devoted to gather statistical information about the usage, as suggested in [Pie13]. Furthermore, an openly available collection of assignments remains to be created in order to enable an effortless utilization of the tool. Since the main goal of this thesis is to support students in the "Hardware Modeling" lecture at TU Wien, a study on the framework's success is one of the natural next steps.

Bibliography

[AKR11] M. Amelung, K. Krieger, and D. Rösner. E-assessment as a service. *IEEE Transactions on Learning Technologies*, 4(2):162–174, 2011. doi:10.1109/TLT.2010.24.

[doc] Docker Docs website. URL: https://docs.docker.com/ [cited 2021-03-03].

[doc11] Docker website, 2011. URL: https://docker.com [cited 2021-03-03].

[ent] GitLab Issue: entrypoint executed twice. URL: https://gitlab.com/gitlab-org/gitlab-runner/-/issues/1380 [cited 2021-02-22].

[Eur18] European Union. General Data Protection Regulation (Article 32), 2018. URL: https://gdpr.eu/article-32-security-of-processing/ [cited 2020-12-13].

[FSZ16] J. Feliciano, M. Storey, and A. Zagalsky. Student experiences using github in software engineering courses: A case study. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 422–431, 2016.

[git] GitLab Docs website. URL: https://docs.gitlab.com/ [cited 2021-02-16].

[git14] GitLab website, 2014. URL: https://gitlab.com [cited 2021-02-16].

[GTR+09] Eladio Gutierrez, Maria A. Trenas, Julian Ramos, Francisco Corbera, and Sergio Romero. A new moodle module supporting automatic verification of vhdl-based assignments. *Computers and Education*, 54(2):562 – 577, 2009. URL: http://www.sciencedirect.com/science/article/pii/S0360131509002462 [cited 2020-11-26], doi:https://doi.org/10.1016/j.compedu.2009.09.006.

[HJM18] D. Hauer, A. Jantsch, and M. Mosbeck. VELS: VHDL E-Learning System for Automatic Generation and Evaluation of Per-Student Randomized Assignments, 2018.

[JCG16] K. Jelemenska, P. Cicak, and M. Gazik. Vhdl models e-assessment in moodle environment. In *2016 International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 141–146, 2016. doi:10.1109/ICETA.2016.7802048.

[mod] Intel ModelSim (Quartus Prime). URL: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html [cited 2021-02-10].

[mod21] Docker Image: ModelSim installed on CentOs 7, 2021. URL: https://hub.docker.com/r/lukasgrau/modelsim_centos [cited 2021-03-25].

[NBMJ12] M. Novák, M. Biňas, M. Michalko, and F. Jakab. Student's progress tracking on programming assignments. In *2012 IEEE 10th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 279–282, 2012. doi:10.1109/ICETA.2012.6418321.

[Pie13] Vreda Pieterse. Automated assessment of programming assignments. In *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research*, pages 45–56, 04 2013.

[smt] RFC 5321 - Simple Mail Transfer Protocol. URL: https://tools.ietf.org/html/rfc5321#page-75 [cited 2021-01-17].

[Tec19] Technische Universität Wien. Richtlinie - Datenschutz und Informationssicherheit, 2019. URL: https://www.tuwien.at/index.php?eID=dms&s=4&path=Dokumente/Datenschutzrichtlinien/Datenschutz%20und%20Informationssicherheit.pdf [cited 2020-12-13].

[ZLHB20] Jeremy Zhang, Chao Lin, Melissa Hovik, and Lauren Bricker. Gitgrade: A scalable platform improving grading experiences. In *SIGCSE 20: Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 1284–1284, 02 2020. doi:10.1145/3328778.3372634.