The Report Committee for Jose Luis Loyola
Certifies that this is the approved version of the following report:

Eksen: Regression Test Selection for VHDL

APPROVED BY

SUPERVISING COMMITTEE:

Sarfraz Khurshid, Supervisor

Milos Gligoric

Eksen: Regression Test Selection for VHDL

by

Jose Luis Loyola

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2017

Dedicated to my wife and parents.

Eksen: Regression Test Selection for VHDL

Jose Luis Loyola, M.S.E.
The University of Texas at Austin, 2017

Supervisor: Sarfraz Khurshid

Regression testing —running tests after a change —has become a critical component of software development, but as projects grow bigger it becomes a time consuming task. For this reason Regression Test Selection (RTS) techniques have become very important. RTS consists of analyzing the changes to a code base and selecting a subset of tests to be run based on these changes. In the context of regression testing, VHDL development is not so different from any other programming language. Modules have unit tests and integration tests. Similarly, the larger the project, the longer it takes to run the test suite. We propose Eksen, a tool for VHDL test selection inspired by the Ekstazi tool for Java. Eksen keeps track of which files have changed including its dependencies and uses this information to select which tests must be run. Eksen statically analyzes the VHDL file dependency tree, it determines which files are affected by the change and only run the tests from that dependency branch. By targeting only the tests on the dependency branch, Eksen can significantly reduce the test suite execution time. For evaluation purposes, we

implemented two versions of Eksen: one using VUnit (an open-source VHDL testing framework). The second using a proprietary enterprise VHDL compiler. This allowed us to verify the time savings on a real industrial projects. Eksen was able to cut test time in half on some of these projects. The results of this experiment are presented in the evaluation section.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Regression testing is the process of testing software that has been modified, either because a new feature was added or a bug was fixed. The main goal of regression testing is to discover new bugs introduced by those changes. However, as more features are added and the product evolves, more tests are added to the test suite, which increases the overall execution time; because the test execution time is directly proportional to complexity of software and the size of the test suite [12].

Very High Speed Integrated Circuit Hardware Description Language (VHDL) [11] is an industry standard hardware description language. VHDL allows programmers to describe digital circuits in a text based manner [1]. The language allows description of the structure, hierarchy, and functionality of a digital design. The VHDL code can be simulated and tested for correctness before manufacturing it on silicon. After the hardware has been verified, it can be later synthesized and deployed on actual hardware like an Application Specific Integrated Circuit (ASIC) or a Field Programmable Gate Array (FPGA). Since VHDL code synthesis can take a significant amount of time, it is commonly seen as best practice to create *testbenches* and run tests before

deploying into real hardware. This is even more crucial on ASICs development. It is critical to verify a digital design before converting it into silicon.

Inspired by the unit testing philosophy of traditional software tools like JUnit[1], VUnit was created [2]. VUnit, which stands for V(HDL)Unit, is an open source framework that provides the functionality to have an automated testing suite for VHDL programs. It incorporates features like a Python[2] interface; scanners for identifying files, tests, dependencies and file changes; VUnit also creates report files so it could be integrated easily in other continuous integration testing environment like Jenkins[3]; among others. However, VUnit implements full regression testing, which is costly. By default, VUnit runs all tests identified by searching for HDL files in given directories. This forces the test suite to spend time on tests that are not affected by the files that changed. This is where regression test selection becomes useful. Regression test selection (RTS) is the technique to effectively select only the tests that are affected by a set of changes in specific files [9]. It does so, by analyzing the new and old software revisions, the dependency information from the test runs on the old revision, and the test suite on the new version. Later, selecting tests to be run on the new revision.

EKSEN is a practical test selection algorithm based on *Ekstazi* [6]. *Ekstazi* is a Java library for lightweight test selection [8]. *Ekstazi* does not com-

[1]http://junit.org/
[2]https://www.python.org/
[3]https://jenkins.io

pare directly the old and new revisions. Instead, it computes the file dependencies of each test class [7]. If no file dependency has changed the test is not run. With Eksen we implemented a similar approach for VHDL. Eksen determines which tests need to be run by checking which files or dependencies have changed since the last time the test suite was run. Based on these changes, Eksen chooses only the subset of tests from the files impacted by the changes. In Chapter 2 we explain how Eksen works and the steps it follows to perform test selection. Chapter 3 presents the time saving results of using Eksen on small projects and also on Industrial VHDL code bases. Lastly, in Chapter 5 we present conclusions on the results obtained.

Chapter 2

Implementation

An RTS technique typically has three phases [6]: *Analysis* where the tests to be run are selected, *Execution* where the tests selected in the previous phase are run, and finally the *Collection* phase where some extra information is recorded, in order to be used in the *Analysis* phase again for the next software revision.

We developed two versions of Eksen: the first one uses open-source tools (VUnit, GHDL). The second one uses a proprietary compiler used at National Instruments. The VUnit version of Eksen was developed first as a proof of concept. The latter was developed to evaluate the benefits of using RTS on real-world industrial projects. The implementation for both is explained in Sections 2.1 and 2.2, respectively.

2.1 Eksen: VUnit Implementation

This was the first version of Eksen. It was developed in Python since VUnit already had a Python interface. The VUnit Python API simplified the scanning, compilation, and test execution of VHDL *testbenches*. The following subsections describe the details of this version of Eksen and mention how it

differs from *Ekstazi*.

2.1.1 Analysis

During this phase, EKSEN tests are selected based on their file dependencies. If a file dependency has changed in the new revision, compared to the old revision, the test is selected. The comparison, however, does not occur by directly comparing file contents. Instead it computes the MD5 checksum of the new file revision and compares it against the previously stored checksum. If there is no checksum history for a file, the test is by default selected. There are a couple of differences in this phase between EKSEN and *Ekstazi*. First, the way EKSEN computes the checksum is fairly simple, since it reads the object file generated by the compiler for each dependency (on *Windows* it computes the checksum directly from the VHDL source file, because those object files are not generated [5]). On the other hand, *Ekstazi* computes a *smart checksum* that avoids marking trivial changes, like instructions that after being compiled are equivalent (e.g. `a+=1` and `a++`), as well as debug information (comments). This difference could be seen as an advantage or a disadvantage: *Ekstazi* takes an additional step to filter imprecise changes. EKSEN on the other hand, takes a conservative approach and flags a file as *changed* when its checksum does not match. *Ekstazi* will tend to select fewer tests since it ignores imprecise changes. EKSEN however, selects the required tests whether the change done was functional or not. In this sense, EKSEN will always run the necessary tests and possibly some additional ones (caused by imprecise changes).

It is important to mention that VUnit already computes checksums for identified files by its scanners, so it does not have to recompile files that changed in the new software revision (thus, saving important compilation time). Another difference between Eksen and *Ekstazi* is the way file dependencies are calculated. *Ekstazi* instruments Java code to detect the files that are accessed during the execution phase. Eksen on the other hand, statically gathers the list of compilation dependencies per test entity. Since VHDL usually does not access configuration files because the code is meant to be run on specialized hardware [13]. Therefore to calculate the list of file dependencies we use VUnit's Python interface, which retrieves the compile order of a specific source file given as parameter. When a checksum difference is detected (or the checksum does not exist) on a specific file F, the tests from F and any other files that depend on F will be added to the list of selected tests. After all the files have been analyzed, Eksen moves to the Execution phase.

2.1.2 Execution

In this phase, both techniques are very similar, since they only run the tests selected in the analysis phase. Perhaps, as said before, the only difference is that Eksen does not instrument the code that is running, because there is no need to [13]. Eksen sends the list of files that must be run through VUnit command line interface and executes the tests.

2.1.3 Collection

If the selected tests complete successfully, EKSEN saves the newly computed checksums of all affected files, i.e. all dependencies that changed. This is done by creating a directory structure similar to the new software revision, and creating a homologous file with the same name as the source file, but different file extension. The created file contains the checksum for the associated source file. It is important to note that if any of the selected tests fail, EKSEN will not update the newly computed checksums. This guarantees that the selected tests will run every time EKSEN is executed, until the tests pass.

2.1.4 Usage

EKSEN is a command line interface written in Python thought to be integrated with continuous integration systems. Particularly integrating with VUnit to accomplish it.

```
python Eksen <source> <lib>
```

Where the parameter `source` is the path to the directory containing VHDL source and its *testbench* code. `lib` is the name of the library that is chosen in case of having different VHDL libraries to evaluate. This makes EKSEN flexible to be used in different projects on the same machine (e.g. A continuous integration machine that runs regression tests for different projects in a company). EKSEN uses VUnit scanners to find all VHDL source files within the directory given as parameter. Next, it builds a dependency graph computed by all instances needed by each file. This gives us the file compilation

order. Eksen uses this dependency graph to detect the files that might be affected by a particular set of changes between software revisions. Finally it will run only the selected tests via VUnit Python interface.

2.2 Eksen: Industrial Implementation

To evaluate the effectiveness of RTS techniques on VHDL projects, we ported Eksen to an industrial environment. This allowed us to apply RTS to industrial VHDL projects whose code is deployed on real products. The company we tested it at uses a proprietary VHDL compiler. This compiler also offers a Python API. Unfortunately, we had to do a complete rewrite of the version explained in section 2.1 because the VUnit API is completely different. Furthermore, the company uses Mentor Graphics ModelSim[1] to run the VHDL testbenches (instead of GHDL). The industrial version of Eksen was also implemented in Python and it goes through the same phases explained above. On the following subsections we present the improvements added to the industrial version of Eksen.

2.2.1 Analysis

The analysis stage of the industrial version of Eksen is very similar to the one explained on section 2.1.1. The proprietary compiler keeps a database of the current state of the project. We added a new command line option to the proprietary compiler to run Eksen. This allowed the tool to get access to

[1] https://www.mentor.com/products/fv/modelsim/

the compiler's database and also it is better aligned with the user-experience developers were accustomed. One of the elements stored in the database is the MD5 checksum of each VHDL source file. The compiler uses this to know which files need to be recompiled. Eksen queries this database to obtain the latest checksum of each file and compares them to the Eksen-specific checksum database created at the end of each successful run of the test suite. If the checksums do not match, Eksen adds the testbench for that file to the list of tests to run. When the checksum matches Eksen skips the testbench for that file. This process is done to every source file in the project.

In contrast to VUnit, the proprietary compiler does not have a direct way to run the selected tests. To get around this limitation, Eksen code-generates a top-level testbench that instantiates the selected testbenches based on file changes. This gave us a single file containing all the required tests. This turned out to be a challenging task because industrial testbenches can get very complex. We encountered testbenches that instantiated other testbenches. Others that get instantiated multiple times with different values on its *generics*[2], each of these values tests a module on a different configuration so Eksen has to take them into account. The tool takes on these complications by performing an additional step. After the files that changed have been found, Eksen searches for all the unit and integration testbenches available. The tool keeps a record of how many times each test is instantiated,

[2]In VHDL, generics are constants declared in the header of a component or entity that change a specific characteristic of the design, e.g. the width of a bus, the depth of a FIFO, the size of a memory, the frequency of a clock, among others.

the different values used for their generics, and if a testbench contains other testbenches. Once the entire list of tests has been created, Eksen selects the ones that must be run due to a file or dependency change. From this subset of tests, Eksen creates a top-level testbench that instantiate all the required tests with all the different values used on its generics. This ensures that the test coverage for each module is maintained. After this top-level testbench has been created, Eksen code-generates a script with the required instructions to simulate it using Modelsim. This script is created to ease integration with regression test tools like Jenkins. After the creation of this script, Eksen moves to the Execution phase.

2.2.2 Execution

This stage is the same as section 2.1.2, the selected tests are executed. At the end of each test Eksen calculates statistics like test run time, number of failures, and number of tests executed. These are then stored on a source-controlled file to keep a record of all the runs of the test suite. When the testbench simulation is done, Eksen moves to the Collection phase.

2.2.3 Collection

When the simulation of the selected testbenches is done, Eksen stores the checksums for all the source files in a single text file. This implementation is different from the one described on section 2.1.3. This version stores the checksums in a single centralized file instead of creating a checksum file per

source file. This made the implementation more scalable, especially with large projects. To prevent the checksum database from keeping record of unused files, during the Analysis stage EKSEN creates a temporary file that contains the checksums for all the files used in the project. If the tests execution is successful, EKSEN moves the contents of the intermediary file to the centralized checksum file. On the other hand, if simulation fails, EKSEN discards the temporary file. This ensures that the selected tests are run until they pass.

Chapter 3

Evaluation

In this chapter we present the results obtained with both versions of Eksen. For the VUnit implementation of Eksen we created a basic project containing some VHDL instances with its respective testbenches. For the industrial version of Eksen we used real-world VHDL projects to evaluate the performance of the tool. The results obtained for each version are presented in sections 3.1, 3.2, respectively.

3.1 Eksen: VUnit Evaluation

To evaluate the VUnit version of Eksen we created a simple VHDL project containing a few entities with its respective unit tests. Figure 3.1 shows the dependency graph created by the project. Each file is represented as a node in the directed graph, each node contains the name of the file. The arrows represent the graph edges, each edge is described by two nodes: source and target. Figure 3.1 displays dependencies as: $A \rightarrow B$, meaning that file A depends on file B. In other words, to prove the correctness of the technique, we wanted to be able to query the dependency graph to obtain the set of tests to run for a particular revision of the code. The tests to run are obtained by

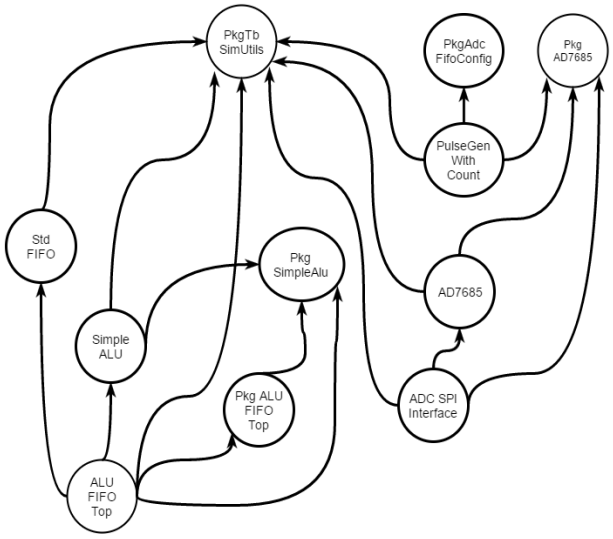


Figure 3.1: VUnit evaluation project dependency graph.

grabbing all the nodes in the path from the first changed file back to the root of the tree. We wanted to measure how much was the testing time reduced and on which scenarios Eksen could help. We used the open-source VHDL simulator / compiler *GHDL* [5] to simulate the testbenches. To test this simple evaluation project, we modified one file at at time and verified that the actual number of testbenches selected by Eksen matched the expected number of tests based on the file hierarchy shown on the dependency graph (Fig. 3.1).To give a better understanding of the time required to run the tests, first we shall describe the relevant hardware used for the execution of Eksen. *Processor*: Intel Core i7 @ 2.5Ghz and a SSD for massive storage with a read speed of 2Gb/s and a write speed of 1.2Gb/s, to try to reduce the bottleneck between file system I/O and the actual execution.

3.1.1 Eksen: VUnit Results

The first time Eksen is executed, it analyzes all files and runs all tests. For our evaluation example it took 27.50 seconds on average to run the complete test suite of **6** testbenches. But Eksen took 28 seconds to complete running the whole test suite, because it not only executes all the tests but also analyzes the files and dependencies among files. However, once Eksen runs again, it only registers a completion time of 0.43 seconds because none of the files were changed, therefore, none of the tests were selected, and completion time goes near zero. To test the correctness, we can modify any of the files in the project, use the algorithm described in the previous section to know which tests should be run and compare with the tests selected by Eksen. We chose to modify the files on the evaluation project, each file with a different level of dependency. Level of dependency in this case could be defined as, the number of incoming arrows in the dependency graph. That way we can appreciate better the results of test selection and testing time per file.

As we can see in table 3.1, on average only 14.8% of the time was needed to run the tests required by the set of changes on every revision. So far we've seen two different scenarios where Eksen could be used and its effects on testing execution. When there are no changes a normal regression test would take it near 28 seconds to execute all tests. Eksen is capable of detecting the changes, skip all tests in 0.48 s. When there are changes in some VHDL files it selects a subset of the test suite to run the test, decreasing the execution time to 14.8%. Finally, when all the test suite has to run (the first time or

Table 3.1: VUnit evaluation VHDL project revisions per file

File	Level	Testbenches selected	Test time	% of tests
AD7685	1	2	9.97s	33.33
AdcSpiInterface	0	1	5.72s	16.67
AluFifoTop	0	1	2.33s	16.67
PkgAdcFifoConfig	1	1	7.96s	16.67
PkgAd7685	3	3	20.45s	50
PkgAluFifoTop	1	1	4.43s	16.67
PkgSimpleAlu	3	2	6.90s	33.33
PkgTbSimUtilities	6	6	28.03s	100
PulseGenWithCount	0	1	10.94s	16.67
SimpleAlu	1	2	13.85s	33.33
StdFifo	1	2	3.69s	33.33

when the computed hashes files do not exist) Since Eksen uses the checksum of the object files / source files without analyzing the changes, there could be situations in which an imprecise change (e.g. update a comment, remove trailing spaces) could cause Eksen to execute the tests from these files.

3.2 Eksen: Industrial Evaluation

We tested the industrial version of Eksen on three VHDL projects of different sizes. The evaluation was done using the source-control history of the projects; we ran Eksen sequentially on every revision of the code. For each project, we started from the first revision, ran Eksen on it; then fetched the next revision and ran Eksen again. This process was repeated for the entire history of the code. On every iteration, we recorded the number of tests

executed and the time it took to run them. To get a base case, we did a second run through the entire history of each project. This time we executed all the tests available (RetestAll) on each revision of the code and recorded both, the number of tests executed and the time it took to run them. Data collection was a time-consuming process because these projects have more than a year of history and the run-time for some of the tests is on the order of hours. The results obtained for each project are presented in the following subsections. We assigned the codenames α, β, and γ to identify them.

3.2.1 VHDL Project α

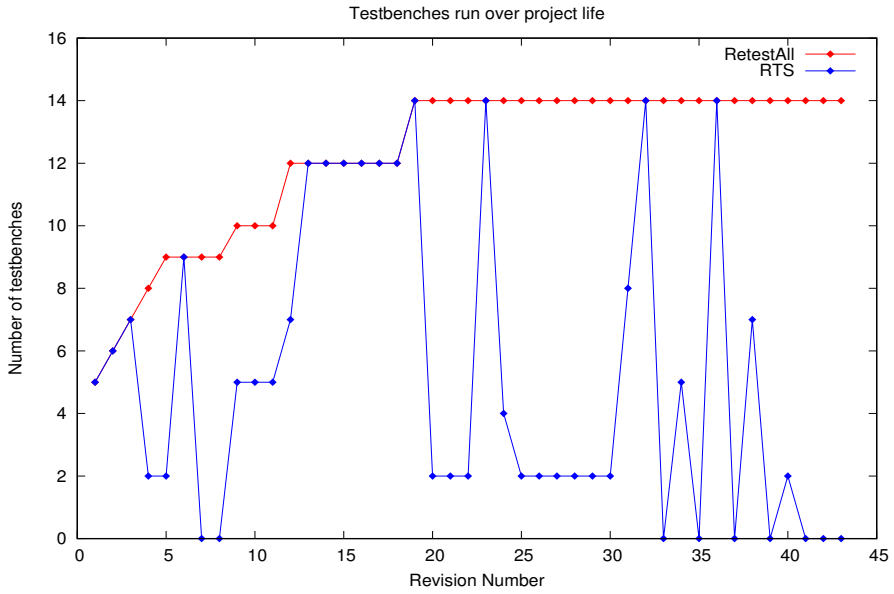
This was the first project we evaluated because it is the smallest. The project has 160 files and over 98,000 lines of code. This project contains the smallest number of tests but its top-level tests take a significant amount of time to run. This caused an interesting behavior on the captured data. The results for project α are presented in Figure 3.2. Plot 3.2a compares the number of testbenches that were run with and without Eksen throughout the source-control history of the project. This figure shows that Eksen significantly reduces the number of testbenches. However, Figure 3.2b shows that the time it took to run the tests was very similar even though the number of selected testbenches was smaller. The results are skewed because the top-level testbench takes the most amount of time. If a file change is detected, the top-level testbench is guaranteed to run because it depends on all the files of the project. In spite of this, Eksen reduced the average test time by 26%.

When imprecise changes were done to the code-base, EKSEN detected that no tests were required to run.

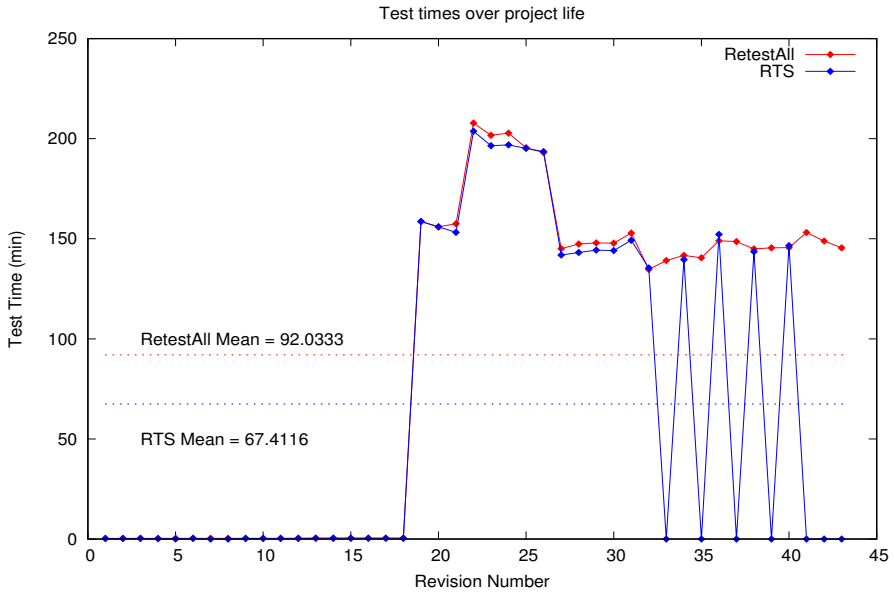
3.2.2 VHDL Project β

Project β shows the history of the last hardware revision from a product. It has a larger number of files, more revisions logged into the source-control server, and more testbenches. This project is composed of 384 files and over 218,000 lines of code. Furthermore, the top-level testbench does not take too long to run. This made it a better candidate to use EKSEN. Figure 3.3 shows the results obtained from running EKSEN on its entire revision history. Figure 3.3a presents the number of tests that were run on each revision of the code. The plot shows there were several revisions with very targeted changes; EKSEN detected this and selected the minimum set of tests required to run. EKSEN has another interesting feature, it automates the addition of new testbenches into the *RetestAll* testbench. The *RetestAll* testbench is manually maintained. If a developer creates a new testbench but forgets to add it to the *RetestAll* it will not be run by the regression test suite. EKSEN on the other hand, explores the entire code base and it auto-generates a testbench containing all the tests required to run. This ensures that new testbenches are included because EKSEN will not have any record of them, removing the need to manually include new tests inside the *RetestAll* testbench.

Figure 3.3b compares the test times with and without EKSEN. The plot presents a significant reduction in test time. The biggest contributing factor

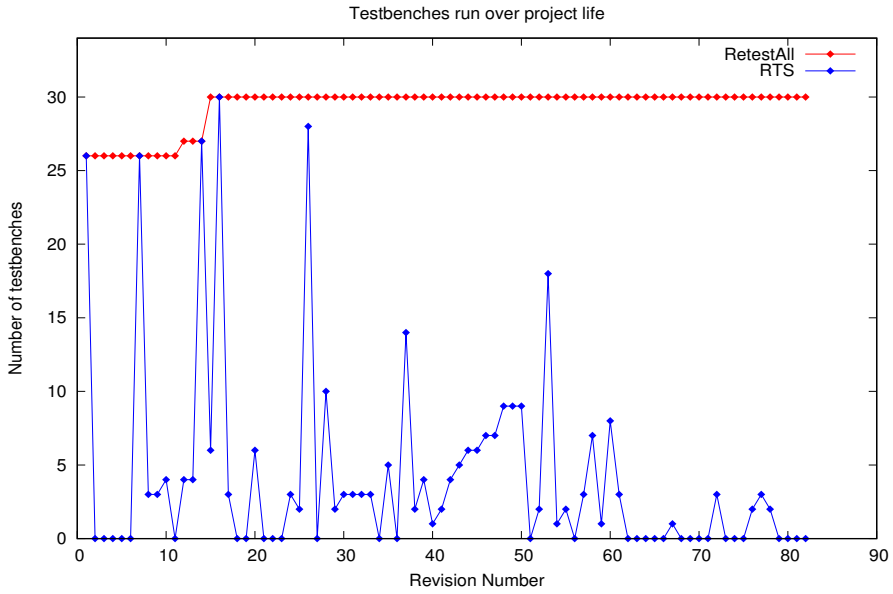


(a) Number of Tests

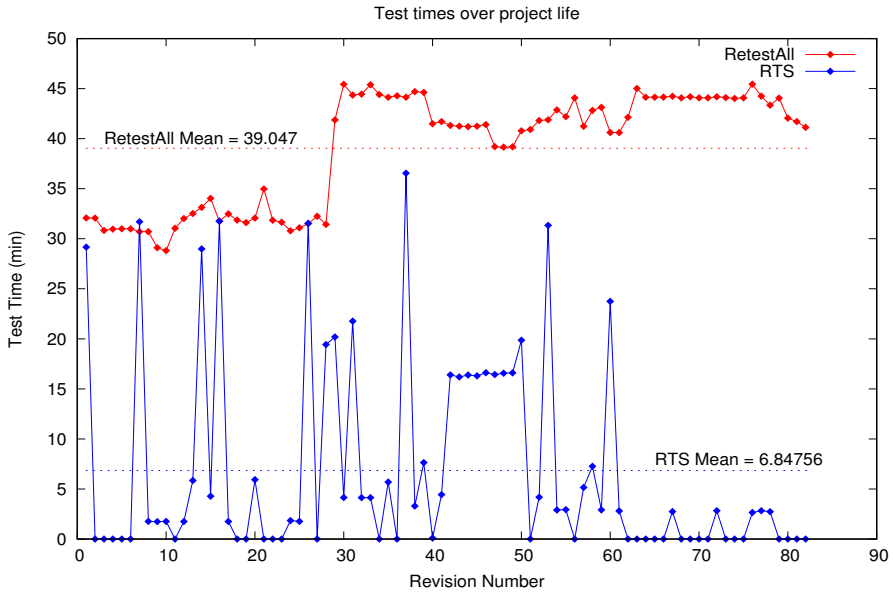


(b) Test times

Figure 3.2: Results obtained from project α



(a) Number of Tests



(b) Test times

Figure 3.3: Results obtained from project β

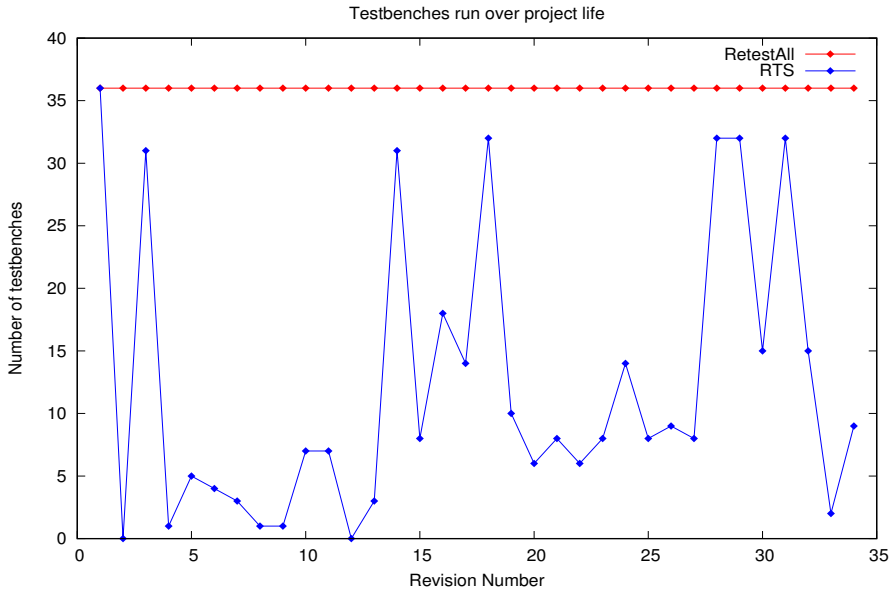
for this reduction is the fact that the top-level testbench does not take too much time to run. Eksen will always select the top-level testbench whenever there is a change done to the code base. If it takes too long to run, Eksen will not reduce test-time by much (like it did on project α). Project β on the other hand, does most of the testing at the component-level; allowing Eksen to cut test times down by skipping component-level tests from unchanged files. Project α and β show the impact the testbench architecture has on the performance of Eksen. If a project performs most of its testing at the top-level, Eksen will only reduce test time when there are revisions where no tests are required to run (e.g. documentation changes). However, Eksen will have a big impact when a project has exhaustive component-level testing and just sanity integration tests.

3.2.3 VHDL Project γ

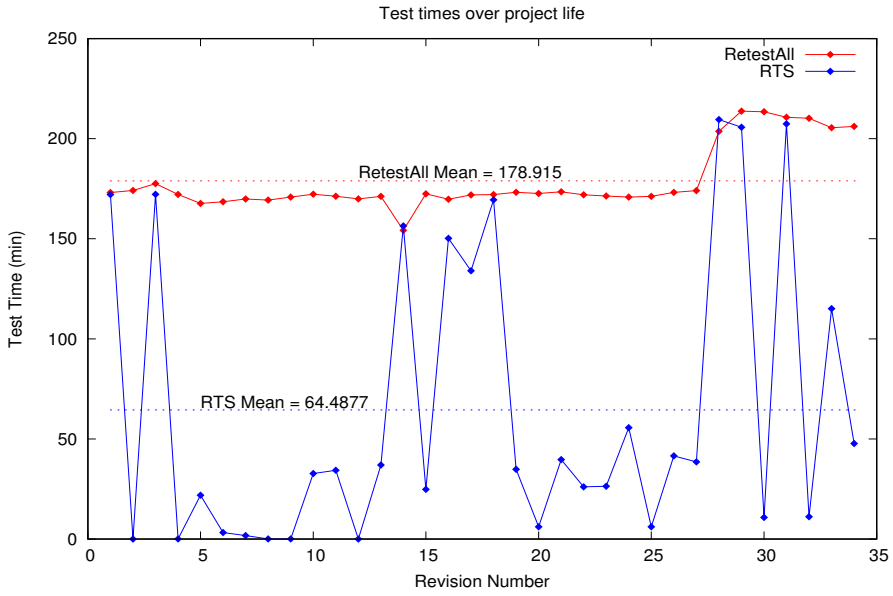
Project γ is larger and more complex than the others. This project contains the code for multiple modules, but the code base is treated as a single repository. Because of this, it contains more files and takes longer to execute all of its tests. The project is composed of 582 files and over 301,000 lines of code. This code base was a great candidate to evaluate Eksen since the changes usually affect only a few modules, so the tests for the unchanged modules are not required to run. Figure 3.4 presents the results obtained from running all the tests and using Eksen. The evaluation was done on an entire year of the code base history. Figure 3.4a shows how the changes

usually affect a few modules and on this evaluation there was not a change that required to run all the tests; EKSEN always selected fewer tests. During the evaluation of this project we noticed an interesting behavior: there were code revisions that had changes that broke compilation on several modules. When compilation fails, the tests for that module are not run (since the code does not compile). Because of this, the number of tests run by *RetestAll* were fewer than the previous run. These data points were excluded from the data because compilation failures reduce the number of tests run and do not represent the actual test time required by *RetestAll*.

Figure 3.4b compares the time it took to run all the tests versus the time it took when using EKSEN. Here we also see a significant reduction in test time because EKSEN always selected fewer tests. Since project γ contains more files, and hence more tests, the probability that a given change affects all the modules is small. Because of this, EKSEN is able to reduce total test time by skipping unnecessary tests. If we add all the points from the *RetestAll* plot on figure 3.4b, i.e. the total time it took to run all the tests on all revisions, we found it was around 5.3 days. If we do the same for the *RTS* plot we found that it was around 1.8 days; that is a 66% reduction in test time. Project γ showed that if the code base is very large, EKSEN can have a significant reduction in the total test time.



(a) Number of Tests



(b) Test times

Figure 3.4: Results obtained from project γ

Chapter 4

Related Work

We mentioned *Ekstazi* [6], which is a Java library for test selection. The algorithm implemented by Eksen was based on the work from Gligoric, Eloussi, and Marinov [8]. *Ekstazi* collects the file dependencies as the tests are executed and it uses the data from previous runs to filter the number of tests to be executed. A key element from *Ekstazi* is that it uses non-debug checksums. This filters even more tests because it ignores source file changes that do not affect functionality.

Ekstazi has also been ported to the .NET environment [15]. This version is called Ekstazi# and it is written in C#. Ekstazi# instruments code statically (before executing tests) due to the fact that .NET does not support dynamic class rewriting. Ekstazi# also implements non-debug checksums, which ignores imprecise changes. This library was tested on several open source projects and was proven to reduce overall test time compared to RetestAll.

Celik et al. [4] present a dynamic RTS technique that moves up to the operating system level to detect all the dependencies of a test. This tool is implemented as a loadable Linux kernel module, this allows it to explore

beyond the Java virtual machine. The experiments done show that this tool achieved similar savings as the original *Ekstazi* for Java.

Chapter 5

Conclusion

In this report we presented Eksen, a tool that performs regression test selection for VHDL projects inspired by Ekstazi for Java. Eksen keeps record of the hashes for all the files in the project and uses them to select which tests must be run. In Chapter 2 we presented how Eksen works and analyzed the differences between Eksen and *Ekztazi*. In Chapter 3 we presented the effects the testbench architecture has on the efficiency of Eksen. If the project does most of its testing at the top-level, then Eksen will run the top-level testbench whenever there is a change. In this case, Eksen will only help when no tests were required to run (this effect was shown on project α). On the other hand, if most of the testing is done at the component level, then Eksen will have a considerable reduction on test time. This effect was shown on project β. Finally, we observed the impact the size of the project has on test time. If the code repository consists of several modules (each with its own top-level testbench), Eksen can have a significant reduction in test time. This effect was shown in project γ. Eksen helps reduce test time of a VHDL project. If Eksen is paired with a continuous integration system, it can help prevent *code-rotting* (code that no longer compiles, or tests that no longer pass due to changes) while minimizing test times. Reducing test time gives developers

feedback faster. Faster feedback lets developers know sooner if a there is a problem that needs to be fixed. This helps to reduce the overall development cycle of a project and at the same time reducing the development costs for the project.

Bibliography

[1] Peter J. Ashenden. *The VHDL Cookbook*. Peter J. Ashenden, Dept. Computer, Science University of Adelaide, South Australia, 1 edition, 7 1990.

[2] Lars Asplund. Vunit: A test framework for hdl. `https://vunit.github.io/`, 2016. [Online; accessed 21-April-2016].

[3] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. Regression test selection techniques: A survey. *Informatica*, 35(3), 2011.

[4] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. Regression test selection across jvm boundaries. *FSE.*, 2017.

[5] Tristan Gingold. Ghdl. `http://ghdl.free.fr/`.

[6] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Ekstazi: Lightweight test selection. *International Conference on Software Engineering*, pages 713–716, 5 2015.

[7] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. *International Symposium on Software Testing and Analysis*, pages 211–222, 7 2015.

[8] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Ekstazi: Lightweight test selection. `http://www.ekstazi.org/`, 2016. [Online; accessed 21-April-2016].

[9] Milos Gligoric, Rupak Majumdar, Rohan Sharma, Lamyaa Eloussi, and Darko Marinov. Regression test selection for distributed software histories. *International Conference on Computer Aided Verification*, pages 293–309, 7 2014.

[10] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, April 2001.

[11] IEEE. Ieee standard vhdl language reference manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages c1–626, Jan 2009.

[12] John Micco and Developer Infrastructure. Continuous integration at google scale. *h ttp://eclipsecon. org/2013/sites/eclipsecon. org*, 2013.

[13] Douglas E Ott and Thomas J Wilderotter. *A designer's guide to VHDL synthesis*. Springer, 2013.

[14] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, Aug 1996.

[15] Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. File-level vs. module-level regression test selection for .net. In *Symposium on the Foundations of Software Engineering, Industry Track*, 2017.

[16] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.