An approach to assuring quality of automatic program analysis tools

line 1: 1st Given Name Surname
line 2: *dept. name of organization (of Affiliation)*
line 3: *name of organization (of Affiliation)*
line 4: City, Country
line 5: email address or ORCID

line 1: 2nd Given Name Surname
line 2: *dept. name of organization (of Affiliation)*
line 3: *name of organization (of Affiliation)*
line 4: City, Country
line 5: email address or ORCID

line 1: 3rd Given Name Surname
line 2: *dept. name of organization (of Affiliation)*
line 3: *name of organization (of Affiliation)*
line 4: City, Country
line 5: email address or ORCID

line 1: 4th Given Name Surname
line 2: *dept. name of organization (of Affiliation)*
line 3: *name of organization (of Affiliation)*
line 4: City, Country
line 5: email address or ORCID

line 1: 5th Given Name Surname
line 2: *dept. name of organization (of Affiliation)*
line 3: *name of organization (of Affiliation)*
line 4: City, Country
line 5: email address or ORCID

line 1: 6th Given Name Surname
line 2: *dept. name of organization (of Affiliation)*
line 3: *name of organization (of Affiliation)*
line 4: City, Country
line 5: email address or ORCID

***Abstract*—Automatic program analysis tools are widely used in secure software development lifecycle for assuring quality of software. Hence automatic program analysis tools itself have to be verified against expected behavior. Continuous improvements of such tools quite often lead to unexpected behavior and it should be verified against expected results. Analysis frameworks are too complicated and researcher, who change it cannot predict how does this change affect on tools behavior. That's why for a such tools need to be applied specific quality assurance process. In this paper we are presenting an approach for assuring quality of automatic program analysis tools.**

Keywords—software testing, automatic program analysis, defects detection

I. Introduction

A program quality assurance is agreed as very important task during software development from the very beginning of computer age. Historically all methods of quality assurance can be divided on two big sets: program code review, known as code inspection, and runtime checking of software behavior and properties during execution.

In the middle of 1970th when C language development was in progress for a brand new operating system UNIX as alternative to assembler language development to make it portable between different hardware platforms the abilities of language has become a problem for a programmer. Despite the fact that Kernigan and Ritchie has described rules of programming using C language very well they has observed that language allows to programmer write a program which brakes these rules, but still is valid C-program. Then Johnson, who developed new *pcc* optimizing compiler [] has implemented first tool *lint* [] which has ability to check C-program on some quite simple C-programming rules violation in automatic manner to prevent inappropriate usage of C-language constructs. This tool becomes one of the first attempts to make code review or inspection process to be automated.

Alternative approach for program quality assurance is testing. While using this approach program runs against set of input data and program behavior is tested against expected behavior in terms of features, performance, scalability and etc. But Edsger Wybe Dijkstra has stated in his lecture "On the reliability of programs" that "program testing can be used very effectively to show the presence of bugs, but never to show their absence" [].

In the middle of 1950th Henry Gordon Rice has proved a theorem [], which states that all non-trivial semantic properties of program are undecidable. But in the early 2000th we can observe explosive growth of industry of static code analysis tools. At that time has been started work on well-known tools such as Klocwork insight [], Coverity Prevent [], Checkmarx [], Svace [], PVS Sudio [] and others, which has become an industry standards for automatic program analysis. Almost all industrial automatic program analysis tools balance between precision, completeness, performance and scalability to provide sound analysis result in limited time with restricted computational resources. This result can be achieved by different methods. A developers of analysis frameworks and tools can use timeouts to limit analysis time for separate function, limit number of facts about program state in specific point of analysis, relax conditions on an execution path and etc. And this leads to analysis results divergence between different tool on the same code base. So, specific methods of automatic program analysis quality assurance has to be applied in comparison with other program kinds.

In this paper we are presenting an approach to check sanity of automatic program analysis tool during its development. This paper is organized as follows. Section II provides brief description of existing approaches for testing automatic program analysis tools, Section III describes an approach in common, Section IV describes in detail Acceptance Testing Framework (ATF) for ensuring quality of an automatic program analysis tool, Section V describes in detail Report History Server (RHS) for measuring divergence between two versions of a tool, Section VI concludes the approach.

II. Existing approaches

The mostly known approach to measure quality of analysis tools is known as NIST SAMATE Juliet test suite []. It is a set of Software Assurance Data Sets (SARD) in the form of code snippets to be used for checking specific static source code analysis tool analysis quality. Every data set is organized as small code snippet which contains specific program error classified by a security flaws taxonomy and it is expected to issue the warning by a tested tool. For now this test suite contains a reference data set for testing static analyzers for C, C++, C#, Java and others.

One more example of analysis tools quality assurance is the *c-testsuite* project [], which provides a framework and test set for compilers, interpreters and emulators to check it against C-language code snippets with possibility to skip a test for a specific tool if tool will never support a case.

Scanstud [] is another example of testing static source code analyzers. Scanstud methodology relies on generation of stub program to make test case runnable, but on the other hand this approach suffers of side effects related to the stub, but not related to the test case itself. It is a successful attempt to check properties of static source code analysis tools, but there is no information on testing the tool against false positive alarm, which makes features of the approach limited. This approach mostly concerned on testing program analysis tool against language features supported, control flow and data flow sensitivity and test against errors detected.

III. The approach

A commonly used parameters for evaluating and comparing automatic tools are *performance*–how fast analysis tool can provide an analysis result, *scalability*–how analysis time reduces if we providing additional computational resources, *precision*–how precise an analysis result is (lack of *false positive warnings* or *noise*) and *completeness*–how many program *true positive warnings* issued by a tool in comparison to errors exist in the program (lack of *false negatives*). For controlling these parameters of automatic analysis tools during development we propose two utility tools.

Acceptance Testing Framework (ATF)–a tool designed to control behavior of the program analysis tool on a limited set of test cases representing expected behavior of the tool in terms of *true positive warnings* and absence of known *false positive warnings* for a different kinds of defects. The ATF allows to run a set of program analysis tools against the same test suite and report how many tests has passed or failed and compare these results between tested tools. This allows to check sanity of developing tool and compare results of analysis with competing tools.

Report History Server (RHS)–is a tool implemented as a service and allow to run tool under development against set of projects with source code available and check divergency of two versions of the tool in terms arrived or disappeared sited *true positive warnings* and *false positive warnings*. The tool allows to review every found defect on a projects with source code available and mark it as true positive or false positive warning. Additionally the tool allows to measure performance of analysis and report performance drop down, if it happens. As far as source code of project stays unchanged between runs of two tool version we always can match them between two runs and check if true positive and false positive gone/arrive in tool's results and control how does quality of the tool evolves.

These two tools if involved in continuous integration system toolchain allows to check how every little change into the tool affects on analysis results quality and help to check tool's quality at very early stage of development.

An approach has been applied to support development of brand-new static source code analyzer developed and used in <company name>.

IV. Acceptance Testing Framework

In this section we present the architecture of Acceptance Testing Framework (ATF), implementation details and results of comparing <company name> Python Analyzers Tool with open source tools pylint and flake8.

A. *Acceptance Testing Framework architecture*

As you can see on diagram ATF consist from following classes:

- **Driver** - is entry point class for ATF. The main responsibility of this class is configure **Reporter**, **TestSuite** and **Tool** according input arguments
- **Tool** - class, which holds information about how to launch some static analysers, and how to parse results from it.
- **Reporter** - class, which provides different kinds of output of ATF's result.
- **TestSuite** - collection of TestCases.
- **TestCase** - class, which holds information about the expected result of static analyzer for a some source code snippet. You can find more details about this class in the Implementation details section.

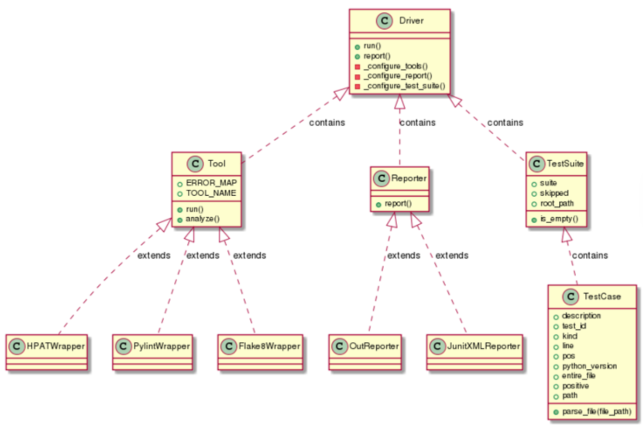


Figure 1. Acceptance Testing Framework architecture

User selections of parameters allows to configure desired analyzing tools, form a test suite of analyzed code and specify format of results output.

First of all, ATF needs some source code for analyzer tool to work with. ATF looks for test cases in the path specified by user, which can lead to some file with code or directory containing code files. After that, special test-suite has to be formed. It contains all test cases and provides necessary information for analysis, which includes description of whether there is any error in code case, which defect it is associated with and in which part of the code it is located.

The main part of ATF workflow is connected with configuration and running analysis process itself. Different analysis tools have different launching arguments and formats of results. To launch and compare them ATF should be able to pass all necessary information to command line, support resulting format of each tool and parse all results into some kind of common view. For this purpose special tools wrappers are used. There is a separate wrapper for each tool supported by ATF, which helps to run tool and analyze produced output. When tools finished their analysis, ATF tool collects their results, forms some kind of common view and passes it to terminal output or resulting file depending on selected reporting format.

After running all tools specified by user ATF provides unified report for all tools executed during run. Each line of output presents result of selected tool analysis process for specified defect for one of selected test cases. If the tool finds an error of specified kind on the same line and position as it was expected by description of case code, such test case is considered to be passed, in the other way – failed. Contrary to testing against *true positive warnings* test case for *false positive warning* checks if the tool does not report an error if it is not expected.

B. *Implementation details*

To create testing results ATF requires an input test-suite. For this purpose we have created a test-suite, based on <company name> Coding Style Guide. For each rule and suggestion from <company name> Coding Style Guide was created a group of test-cases.

As mentioned before, the test-case contains information describing expected behaviour of static analyzer on some code snippet. For each code snippet a test descriptor created, which contains following flags:

- **test id**–used to identify each test case
- **kind**–expected error to be reported / or not reported on current test-case
- **line**–line in file, where error expected to be reported.

- **Position**–position on line, where error expected to be reported
- **Positive flag**–used to check if static analyzator should highlight error on this test-case. Positive filed if set to true means this test expects warning of a tool about error in the test case. Positive field if set to false means this test does not expects warning of a tool about error in the test case, i.e. test for a possible *false positive warning*.
- **Skip**–if tool still does not support the test case it can be skipped during testing, but holds information on future functionality to support.

ATF is supposed to be used to compare analysis results of different tools. For providing descriptive presentation of obtained results, ATF amongst others supports special type of reporter – "html" reporter. HTML reporter can produce output of two different views – one for machine-suitable parsing and another with illustrative diagrams. Both variants have links leading to html pages for each code snippet used as a test case with information on passed/failed state and description code snippet.

First type of report is presented by special table, which includes numbers of passed and failed positive/negative test cases for each tool. There are also extra information like kind of rule, number of skipped tests, etc.

V. Report History Server

Here we describe requirements for Report History Server, architecture, implementation details and results of applying it on several projects.

In the previous section we have described the testing process on the code snippets, that was created on our own to show the tool behavior in the very specific cases. On the other hand things may be different in the real world.

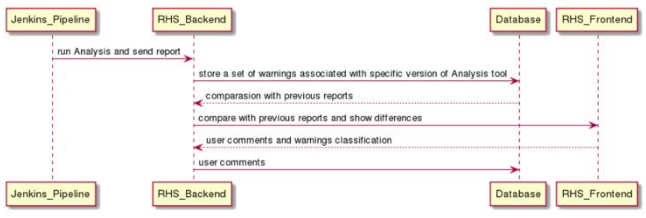
Report History server (RHS)–a system to collect and compare reports of static analyzers launched on some real world projects. The main goal is to make visible the current state of analyzer (or project) development, and to find differences in reports from one version to another. Thus RHS is very useful to perform high level analysis and find regression.

RHS was developed as server, that take the report of an Analysis Tool in json format.

Then this report compares with previous reports available in web view, where duplicated **warnings** marked as duplicated and new ones may be analyzed and categorized.

A. *Architecture*

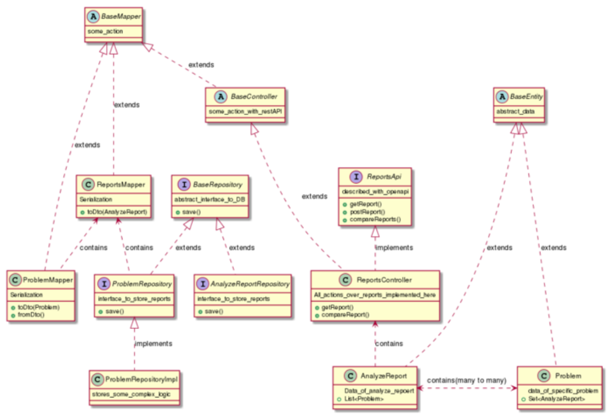
As far as RHS is designed as a server application and used as a part of CI process, it should be considered from this point of view. It consist of the few infrastructure components that interacts with each other:



Jenkins_Pipeline
RHS_Backend
Database
RHS_Frontend
run Analysis and send report
store a set of warnings associated with specific version of Analysis tool
comparasion with previous reports
compare with previous reports and show differences
user comments and warnings classification
user comments
Jenkins_Pipeline
RHS_Backend
Database
RHS_Frontend

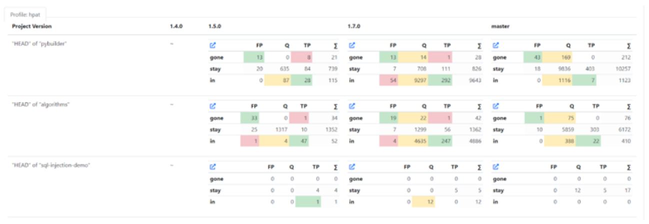
Figure 2. Report History Server sequence diagram

Jenkins Pipeline at Jenkins CI server to run code analysis tool and send reports to Backend. It uses incoming link from GIT [] repository via webhook and send a report by outgoing link to backend via RESTalike API. Business logic of RHS. As far as Backend is a core part of RHS, and the most complex one, let's take a look closely.



Backend is separated into few parts: classes that introduce interfaces for the RESTalike api, them are described with OpenApi and build with Swagger (https://swagger.io/).

Front-end–User interface to allow view analyze and discrepancies reports, manage dictionaries. Provide users feedback by outgoing link to backend via RESTalike API. As an additional feature - able to show a code chunk or provide a direct link to the source of erroneous code.



Profile: hpat

Project Version	1.4.0	1.5.0					1.7.0					master				
			FP	Q	TP	Σ		FP	Q	TP	Σ		FP	Q	TP	Σ
"HEAD" of "pybuilder"	–	gone	13	0	8	21	gone	13	14	1	28	gone	43	169	0	212
		stay	20	635	84	739	stay	7	708	111	826	stay	18	9836	403	10257
		in	0	87	28	115	in	54	9297	292	9643	in	0	1116	7	1123
"HEAD" of "algorithms"	–	gone	33	0	1	34	gone	19	22	1	42	gone	1	75	0	76
		stay	25	1317	10	1352	stay	7	1299	56	1362	stay	10	5859	303	6172
		in	1	4	47	52	in	4	4635	247	4886	in	0	388	22	410
"HEAD" of "sql-injection-demo"	–	gone	0	0	0	0	gone	0	0	0	0	gone	0	0	0	0
		stay	0	0	4	4	stay	0	0	5	5	stay	0	12	5	17
		in	0	0	1	1	in	0	12	0	12	in	0	0	0	0

Figure 3. Report History Server summary results

Database keep RHS Data. Communicates with the backend using JPA/JDBC interfaces. As far as Database is also in development, then it is configured with liquibase, to make it possible to change its configuration without loss of previously stored data.

According to the development of the Analyzer Tool, introducing RHS leads to a more precise and more correct look at what the tool should alarm. It helps to improve soundness, and improve communication with customer site to discuss if they wish to have warnings on some code chunks not from synthetic examples. Also it helps to test Analyzer Tool on some cases that even looks weird sometimes.

VI. Conclusion

In this paper we has presented an approach for assuring quality of automatic program analysis tools. It allows to check quality of the tool on predefined set of test cases to check predictable behavior of the analysis tool and check regression on projects with source code available to prevent analysis tool quality degradation.

This approach has been applied for testing brand-new static source code analysis tool developing in <company name>, but is not limited for static source code analysis. This approach can be applied to other kinds of tools, such as dynamic analysis tools, static verifiers and others.

References

[1] …
[2] …