University of Tartu
Faculty of Science and Technology
Institute of Computer Science

Risto Pärnapuu

Extending the Reach of Eclipse Plug-in for Analysing Embedded SQL Queries

Bachelor's thesis (9 ECTS)

Supervisor: Aivar Annamaa, PhD
Co-supervisor: Vesal Vojdani, PhD

Tartu 2017

Extending the Reach of Eclipse Plugin for Analysing Embedded SQL Queries

Abstract:

Alvor is a tool build for Eclipse IDE that checks the validity of SQL queries embedded in strings of a host programming languages. This thesis focuses on making the benefits of the Alvor tool available outside the Eclipse IDE.

Solution for this was to implement an additional feature for the Alvor tool that creates a result file of the SQL check. In order to improve the usability of the result file, JUnit standard for the report file was used. The key challenge of making the tool available outside the Eclipse IDE was finding and combining the right tools to start the Eclipse build process, which would start the SQL check, without starting the IDE and doing so for any type of project, regardless if it's an Eclipse project or not.

Keywords:

Eclipse plug-ins, embedded SQL, Alvor, JUnit Report

CERCS: P175

SQL päringuid kontrolliva Eclipse *plug-in*-i kasutatavuse võimaldamine väljaspool Eclipse'i keskkonda

Lühikokkuvõte:

Alvor on Eclipse'i jaoks arendatud tööriist, mis kontrollib koodis leiduvate SQL lausete korrektsust. Käesoleva bakalaureuse töö eesmärgiks on Alvori tööriista kasutatavuse võimaldamine väljaspool Eclipse'i arenduskeskkonda.

Autori ülesandeeks oli täiendada Alvori tööriista, lisades võimaluse genereerida SQL kontrolli kohta tulemuste fail ja võimaldada seda teha ilma Eclipse'i käivitamata. Faili loomisel kasutati JUnit tulemusfaili struktuuri, et teha faili kasutamine erinevates keskkondades võimalikult lihtsaks. Peamiseks väljakutseks osutus õigete tööriistade leidmine ja kasutamine, et võimaldada Eclipse'i kompileerimisprotsessi käivitamist väljaspool Eclipse'i arenduskeskkonda.

Võtmesõnad:

Eclipse'i *pluginad,* SQL päringud programmides, Alvor, JUnit'i raport

CERCS: P175

Contents

Contents 3

Introduction 4

1. Current state of the Alvor Tool 5

1.1 Eclipse 5

1.2 Overview of Alvor 6

1.3 Running Example 6

2. JUnit Reports for Alvor 8

2.1 Structure of the report file 8

2.1.1 testsuites 8

2.1.2 testsuite 9

2.1.3 testcase 10

2.2 Implementation 10

3. Running Alvor without the Eclipse IDE 13

4. Generating Eclipse projects automatically 15

4.1 Eclipse projects 15

4.2 Creating Eclipse projects 17

5. Using Alvor for Continuous Integration Testing 19

5.1 Continuous integration environments 19

5.2 Jenkins 20

5.3 Running Alvor within Jenkins 21

6. Limitations 23

Conclusion 24

Bibliography 25

Appendix A: Glossary 27

Appendix B: Shell script 28

Licence 29

Introduction

Java and PHP are popular languages that are used to build reliable and scalable web applications. Most of these web applications use databases to store and backup their data. Communication between the application and the database is conventionally done by using SQL queries. Writing SQL statements as strings in a host language like Java or PHP is a method called embedded SQL.

The Eclipse integrated development environment (IDE) provides no feedback on the validity of the embedded SQL queries. This means that faulty queries are mostly discovered during runtime when they are executed. Even then there is a possibility that the query might work under some conditions but fail for others. To solve this issue, an Eclipse plug-in named *Alvor* was implemented by Annamaa et al [1] in 2010.

Alvor is a tool that statically validates embedded SQL queries. It was originally designed for Java projects. An extension for PHP was implemented by Tamm [2] in 2015. *Alvor* checks the queries against the configured database and provides feedback to the user about the correctness of the SQL syntax and checks the queries against the database. Feedback to the user is provided only through the Eclipse IDE.

Since the internal structure of *Alvor* is largely dependent on Eclipse itself, it is not possible to perform *Alvor* SQL check for non-eclipse projects. Therefore, the benefits of *Alvor* are restricted to Eclipse projects only. The aim of this Thesis is to expand the reach of *Alvor's* capabilities beyond the Eclipse IDE and enable *Alvor* integration for continuous integration (CI) environments.

After a brief overview of the current state of the Alvor plug-in in Chapter 1, Chapter 2 gives a description on how the JUnit reports was used to extend the reach of the Alvor tool. Then, following chapters will show how to remove the Eclipse IDE dependency: Chapter 3 describes how to run Alvor analysis on Eclipse projects without the Eclipse IDE and Chapter 4 shows how other types of project can be converted to an Eclipse project so the analysis could also be applied for them. Finally in Chapter 5, use cases of the Report are demonstrated. Possible limitations and drawbacks are discussed in Chapter 6.

1. Current state of the Alvor Tool

1.1 Eclipse

The Eclipse Project is an open source project developed by the Eclipse Foundation and originally created by IBM in 2001 [3]. Eclipse provides an IDE and a platform of tools for a variety of programming languages but is mostly known for its Java IDE [4]. When talking about Eclipse, it can mean many different things to different people. In this Thesis we perceive Eclipse mainly as a Java development environment.

Eclipse was designed with a component model in mind, meaning that it should provide an easy way to extend the product with new components for third parties [5]. This is the reason the entire Eclipse platform can be thought of as a set of plug-ins or bundles where additional features can be added to the existing set. This is possible because Eclipse is built on an implementation of the OSGi core framework specification called equinox, which itself is part of the Eclipse Project [6].

Each Eclipse Plug-in offers a service or a feature to expand the capabilities of Eclipse and can be seen as the smallest component of the Eclipse platform that can be developed separately [7]. Developers are able to contribute to building the Eclipse platform by building tools wrapped in pluggable component containers. Smaller tools are usually contained in a single plug-in. More complex ones are spread across multiple plug-ins [7]. A plug-in can either extend the capabilities of other plug-ins, meaning that it builds on top of an existing functionality, or it can offer functionalities that other plug-ins can use.

At its core all Eclipse plug-ins are essentially bundles in an equinox OSGi framework. All plug-ins are written using Java and a typical plug-in is just a Java Archive (JAR) containing the implementation of the plug-in. Each plug-in must contain a manifest file called MANIFEST.FM that describes the plug-in and how it depends on other plug-ins. Dependencies are described by extension points. A plug-in can declare any number of named extension points and can add extensions to extension points in other plug-ins [7].

1.2 Overview of Alvor

Alvor is an Eclipse plug-in to statically validate embedded SQL queries. It was originally designed to do so in Java projects by using the Eclipse JDT framework. The original tool was developed by Annamaa et al in 2010 [1]. Language support for PHP was added by Tamm using the Eclipse PDT framework [2].

In the context of Alvor, string expressions evaluating to SQL queries are referred to as *hotspots*. The rules for locating hotspots must be specified by the developer by using Alvor's configuration window or by manually editing the Alvor configuration file. Alvor performs analysis on the monitored hotspots and provides feedback via the Eclipse IDE using the built-in system in Eclipse for error and warning handling.

For Java projects Alvor works by using the Eclipse JDT to obtain the abstract syntax tree (AST) for each *.java* file. Then each AST is scanned to find the hotspots that contain SQL syntax. The hotspots are then evaluated and all possible values of the expression are gathered and represented by a regular expression.

After gathering the regular expressions representing the hotspots, Alvor starts to check each of them for errors. The error checking is categorised in two parts: first all strings are checked for syntax errors. Second, each of the strings are tested against the running database if the database connection has been configured.

After the analysis is finished, the errored hotspots are highlighted using an error and warning handling system built into Eclipse. Additionally each error notice contains relevant information about why the hotspot scan resulted in error.

1.3 Running Example

We consider an Eclipse project *DemoApplication* with two Java classes *Foo.java* and *Bar.java*. Both of the classes contain SQL strings that are passed as arguments to *preparedStatment* method in the *java.sql.Connection* class:

```
connection.prepareStatement(sql);
```

If the developer defines this method as a hotspot, then all strings that are passed to this method will be analyzed by Alvor. For testing purposes both classes have one potential problem with the SQL queries.

Problematic query 1 (Foo.java):

```
String sql = "select id, first_name from persons";

if (con) {
        sql += "where last_name = tamm";
}

sql += " order by first_name";
```

Explanation: If the condition for the if-statement is true, the where-clause is added without a space in front of it.

Problematic query 2 (Bar.java):

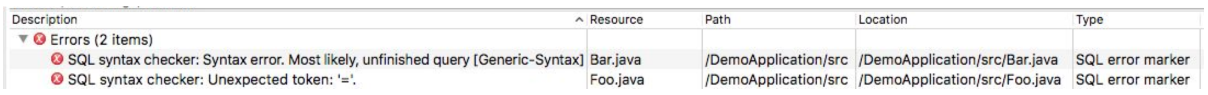
```
String sql2 = "select id, first_name from persons where age = ";
if (con1) {
        sql2 += "10";
} else if (con2) {
        sql2 += "11";
}
```

Explanation: If neither of the conditions are met, the query will be unfinished.

Additionally *Bar.java* also contains a second query which is a the query seen in the class *Foo.java*, but the space is added before the where-clause:

```
sql += " where last_name = tamm";
```

After the SQL scan, the problems are displayed as errors through the Eclipse's error and warning handling system.



Description	Resource	Path	Location	Type
Errors (2 items)				
SQL syntax checker: Syntax error. Most likely, unfinished query [Generic-Syntax]	Bar.java	/DemoApplication/src	/DemoApplication/src/Bar.java	SQL error marker
SQL syntax checker: Unexpected token: '='.	Foo.java	/DemoApplication/src	/DemoApplication/src/Foo.java	SQL error marker

Figure 1. Eclipse's view of SQL scan problems

2. JUnit Reports for Alvor

Since Alvor exists only as an Eclipse plug-in and the feedback is provided through the features built into the Eclipse IDE, its usages are limited to developers using this IDE. Alvor is highly dependent on Eclipse and eliminating that dependency would mean essentially writing a new tool from scratch. To make Alvor more generally usable, another approach is needed. This section of the thesis will describe how the issue was solved by implementing a report generation into the Alvor plug-in. Repository for this addition can be found from https://bitbucket.org/ristop/alvor.

2.1 Structure of the report file

One option would be to just generate a report file that follows the custom standard. However, this approach would still be having usability issues since everyone using it should implement their own way of parsing and presenting the results. A more suitable alternative was to look for a standard that is widely used and can be easily integrated with other system.

One such standard is defined by the JUnit framework to provide information about the test suites that were executed. It provides a structure and defines elements for the final report file . The final report file can be used in CI environments like Jenkins to display the information about the tests. The following paragraphs describes the structure of the report file and gives a brief description of all the valid elements and their attributes based on the sample and XML schema definition of Dirk Jagdman [8].

2.1.1 testsuites

Testsuites is the root element of the report file if multiple testsuite elements are present. In case only a single testsuite element is present, the testsuites element can be omitted.

Optional attributes:

- *disabled* - Number of tests in all testsuites.
- *errors* - Number of tests that resulted with an error in all testsuites.
- *failures* - Number of tests that resulted with an failure in all testsuites.

- *name* - Name of the testsuites.
- *tests* - Number of tests that succeeded in all testsuites.
- *time* - Time that take to execute all tests in all testsuites.

The testsuites element does not have any required attributes.

2.1.2 testsuite

The testsuite element describes a single testsuite. If multiple testsuites are present, they must be placed inside testsuites element.

Required attributes:

- *name* - name of the test class. For non-aggregated testsuites full class name must be set. Package name can be omitted for for aggregated testsuites.
- *tests* - Number of tests in the testsuite.

Optional attributes:

- *disabled* - Number of disabled tests in the testsuite.
- *errors* - Number of tests that resulted with an error in the testsuite.
- *failures* - Number of tests that resulted with an failure in the testsuite.
- *hostname* - Host on which the tests were executed.
- *id* - Starting at 0 and incremented by 1 for each following testsuite.
- *package* - Derived from testsuite/@name in the non-aggregated documents.
- *skipped* - Number of skipped tests in the testsuite.
- *time* - Time that take to execute all tests in the testsuite.
- *timestamp* - When the test was executed in ISO 8601 format.

Each testsuite can have optional system-out and system-err elements. The contents of those elements show what was written in the standard out and standard error while the test suite was executed.

2.1.3 testcase

The testcase element describes a single test case. It can appear multiple times and each element must be inside of a testsuite that it belongs to.

Required attributes:

- *name* - Name of the test method.
- *classname* - Name of the class that contains the test method.

Optional attributes:

- *assertions* - Number of assertions in the test case.
- *status* - Status of the test case.
- *time* - Time taken to execute the test case.

To indicate that the test case resulted with an error, the error element must be present inside a testcase. Error element has two optional parameters:

- *message* - The error message.
- *type* - The type of error that occurred.

To indicate that the test case resulted with an failure, the error element must be present inside a testcase. Failure element has two optional parameters:

- *message* - The message specified in the assert.
- *type* - The type of the assert.

Each test case can have optional *system-out* and *system-err* elements. The contents of those elements should reflect what was written in the standard out and standard error while the test was executed.

2.2 Implementation

Extending the Alvor plug-in to produce the result file did not require refactoring existing code. The Implementation required creating a new central class that was responsible of generating the result files. Instance of the class had to be used in the existing code to supply information about the scanned hotspots. If multiple projects were scanned with Alvor, a separate result file will be generated for each project. The result file contains data entries about both succeeded and failed hotspots.

The structure of the report file exactly follows the standard structure set by JUnit framework, meaning that it uses all the required elements and their attributes. Usage of those elements however is a bit different from what is set by JUnit, because the standard JUnit result file contains required elements that can not be set when generating the result file for Alvor SQL check, e.g. Alvor does not have actual test suites or test methods. The following is an overview of how the Alvor result file differs from the JUnit standard and what substitutions were required.

The implementation considers each project as a single test suite. This means that testsuites element can be omitted and the testsuite element can be set as a root element because each project gets its own result file. By standard, the *name* parameter of the *testsuite* element must be set to the name of the test class. Since Alvor does not have one, implementation uses the name of the project that contains the hotspots. For the *tests* parameter the class keeps track of the number of hotspots scanned and uses the final value.

Each scanned hotspot will correspond to a *testcase* element. The name parameter will show the name of the class that contains the scanned hotspot. If a hotspot contains a problem, the corresponding test case can be considered to have failed and the *error* element can be added to the *testcase* element. To specify the nature of the problem, a message attribute is added to the *error* element.

The following is the *alvor.xml* result file that is generated for the *DemoApplication* project:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<testsuite errors="2" name="DemoApplication" tests="3"
timestamp="2017-04-03T16:15:28">
  <testcase classname="/DemoApplication/src/Foo.java" name="testFoo">
    <error message="SQL syntax checker: Unexpected token: '='. Counter example:
select id , first_name from personswhere last_name = tamm order by first_name
[Generic-Syntax]"/>
```

```
  </testcase>
  <testcase classname="/DemoApplication/src/Bar.java" name="testBar"/>
  <testcase classname="/DemoApplication/src/Bar.java" name="testBar">
    <error message="SQL syntax checker: Syntax error. Most likely, unfinished query
[Generic-Syntax]"/>
  </testcase>
</testsuite>
```

3. Running Alvor without the Eclipse IDE

The goal is to provide a method to generate the report file for any type of Java project without using the Eclipse IDE. We first consider how to run the Alvor checks for an Eclipse project without running the Eclipse IDE.

Let us first recount how Alvor is enabled within Eclipse. The report file is generated when the Alvor SQL check is run and the option to generate the report file is turned on. The developer can turn on the report generation in two ways:

1. Opening the properties menu for the project, navigating to Alvor SQL Checker settings and ticking the checkbox for *Generate JUnit test result XML.*
2. Modifying the *.alvor* configuration file by adding parameter *genResultFile="true"* to the alvor element.

The second approach does not require starting the Eclipse IDE and, therefore, can be easily integrated with the headless implementation. It is also possible to plug Alvor into the project's build process from the Alvor settings menu. This means that Alvor SQL check will be run each time the java files are compiled. All of this can be easily achieved by using the Eclipse IDE. But since one of the goals of this Thesis was to enable developers to benefit from Alvor without running the IDE, a headless implementation for the build tool is needed.

Headless build means running the build process without the graphical user interface. In this context *head* is perceived as the Eclipse IDE.

The Eclipse APT plug-in provides a headless build tool that can be used to build an existing and preconfigured workspace, either from the command line or by using the built-in ant task. Provided that Eclipse has the APT plug-in installed, the workspace can be built by invoking the following command from the command line:

{eclipse} -nosplash -application org.eclipse.jdt.apt.core.aptBuild -data "{workspace_path}"

The command starts Eclipse in headless mode and builds the workspace provided. Any configuration for the workspace will be observed in the build. If the provided workspace path does not exist or does not contain an Eclipse workspace, Eclipse will create a new empty workspace [9].

The APT plug-in does not have a tool to build Eclipse projects that do not belong to a workspace or a tool to import existing projects to a given workspace. However, such a tool exists in the Eclipse CDT plug-in. Among other features, the CDT plug-in provides a headless build and an importing tool for C and C++ projects. The headless build tool only works for the C and C++ Eclipse projects and when the command is invoked for any other type of project, the following error is produced:

Project: {project_name} doesn't appear to be a CDT project. Skipping...

Even though the build process results in an error, it is possible to use the importing tool for any other type of Eclipse projects by using the following command:

```
{eclipse} -nospash
-data "{workspace_path}”
-application org.eclipse.cdt.managedbuilder.core.headlessbuild
-import “{project_path}”
-build “{project_name}”
```

If the specified project path contains any other type of project other than C or C++, the command produces the error mentioned above but still imports the project to the workspace, regardless of the project type [10].

Combining the implementations of the headless build tools of APT and CDT plug-ins it is possible to build any Java project, whether it is in a workspace or not, without opening the Eclipse IDE. These techniques are combined into a shell script (Appendix B) that uses CDT to import the project if needed and JDT to build said project.

4. Generating Eclipse projects automatically

Since the Alvor plug-in is deeply dependent on Eclipse project's internal structure, existing non-eclipse projects must be converted to Eclipse projects in order to perform Alvor SQL analysis.

4.1 Eclipse projects

An Eclipse project is mainly described by two files: *.project* and *.classpath.* Both of the files are located in the project's root directory.

.project

The *.project* file is called the project description file. Like the name suggests, the purpose of this file is to describe the project, so that it can be correctly recreated in another workspace without any additional configuration. This file must always be called .project. If the file is missing or the contents of the file is corrupted, the project is closed and can not be opened until the issues have been resolved [11].

The contents of this file is written in XML with a root element named *projectDescription.* The next paragraph contains the description of all necessary configuration needed to make Eclipse recognize a project as an Eclipse project, tag the project as a Java project and attach an Eclipse java Compiler to it.

Configuration markup [11]:

- projectDescription - The root element of the .project XML file.
- name - Specifies the name of the project. This element is ignored if the name does not match the name of the project using the .project file.
- buildSpec - Contains the ordered list of buildCommand elements.
- buildCommand - A single build command for the project.
    - name - Symbolic name for the build command.
- natures - Contains list of nature elements.
- nature - The name of a single nature on the project.

Example of an .project file for a java project that has a single nature and builder configured:

```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
        <name>Sample project</name>
        <buildSpec>
                <buildCommand>
                        <name>org.eclipse.jdt.core.javabuilder</name>
                </buildCommand>
        </buildSpec>
        <natures>
                <nature>org.eclipse.jdt.core.javanature</nature>
        </natures>
</projectDescription>
```

.classpath

The file named *.classpath* contains the information that the Java compiler needs to properly compile the project. It defines the source directory, output directory and specifies how to find required types outside of the project [12].

The contents of this file is written in XML with a root element named *classpath* that contains entry elements named *classpathentry*. Each *classpathentry* has two attributes called *kind* and *path*. Kind attribute specifies the type of the entry. Type can be *src*, *output*, *lib* or *con*. The path specifies the path where the Java compiler will look for the given entry.

List of of the possible types and their descriptions [13]:

- *src* describes the location directory of the source files that java compiler will compile.
- *output* describes the location where the java compiler puts the compiled files.
- *lib* describes the location of an external library to be included in the classpath.
- *con* describes the classpath container.

Example of an *.classpath* file for a java project:

```
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
        <classpathentry kind="src" path="src"/>
        <classpathentry kind="lib" path="lib/example.jar"/>
        <classpathentry                                    kind="con"
path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
        <classpathentry kind="output" path="bin"/>
</classpath>
```

4.2 Creating Eclipse projects

From the eclipse IDE, it is possible to import an existing project by using the Import Wizard:

1. Open the Import wizard by selecting *File -> Open Projects file From File System…*
2. Point the import source to the projects directory.
3. Select the folders to be imported and click *Finish*.

This wizard will generate the two *.project* and *.classpath* files in the root of the project directory.

Using the import wizard to import non-eclipse projects into an Eclipse workspace requires the user to open the Eclipse IDE and therefore can not be easily integrated to an automated processes.

An eclipse project needs to have *.project* and *.classpath* files in the root directory to be classified as an Eclipse project. Additionally, the *.alvor* configuration file is needed to specify the configuration needed for Alvor to perform the SQL analysis. There exists no headless implementation to generate those files so they have to be created manually. Solution for generating those files was to write a shell script that could do exactly that. Source code for the script is attached as an online supplementary to the thesis and described in Appendix B.

The script creates *.project*, *.classpath* and *.alvor* files to the root of the specified project only if the files were not present before. Firstly, the *.project* file is generated by using the simplest template for a Java project and adding the Alvor build command and Alvor nature, so the SQL analysis is run each time the project is built. Secondly, the contents of the *.classpath* specifies the standard source and output directories and a container. Additionally, the script adds all the JAR files from the *lib* directory to the classpath. Thirdly, the generated *.alvor* file contains one hotspot, a default checker and parameter to enable report file generating when the SQL check is ran.

5. Using Alvor for Continuous Integration Testing

When running the Alvor SQL check through the IDE, the report file will be generated in the root directory of the project and can be opened with the JUnit view by right clicking on the file and selecting *Open with -> JUnit View*. The view is meant to depict the results of a testsuite that was executed and allows one to open and rerun tests [14].

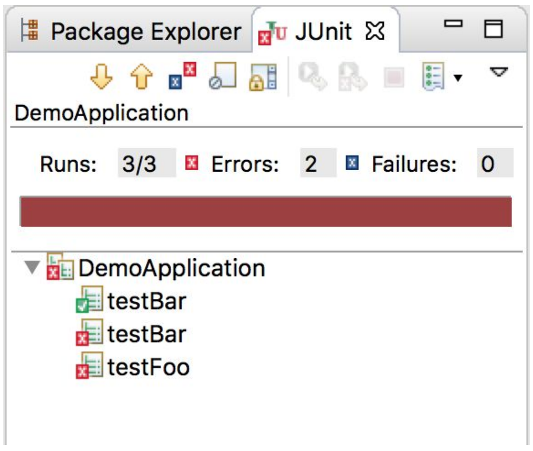


Figure 2. JUnit view for the Alvor result file in Eclipse IDE

In thecase of Alvor's result file, the JUnit view can only be used as a visual aid to locate the problematic hotspots because no actual test methods exist. It gives an overview of all the hotspots that were scanned in a given project. To get more information about the scanned hotspots, the result file can also be opened with an XML editor by right clicking on the file and selecting *Open with -> XML editor*.

As shown in the previous sections, we can now generate JUnit reports from Alvor without running the full-fledged Eclipse IDE. We can thus achieve the ultimate goal of running Alvor checks automatically on a continuous integration server.

5.1 Continuous integration environments

Continuous integration (CI) is a practise of using a shared repository where the automate build process is run several times a day. This allows software developers to constantly monitor the state of the application and detect the problems early. Constant integration means that much less back-tracking is required to discover the place where things went wrong [15].

“Continuous Integration doesn’t get rid of bugs, but it does make them dramatically easier to find and remove.” — Martin Fowler.

CI works by running the automated build process and tests each time a member of a team integrates their work in the continuous integration environment. Every commit a developer makes should build on an integration machine. CI environment enables team members to see the current state of the application.

5.2 Jenkins

Jenkins is a Java-based open source automation server. It can be used as a simple CI server or even a continuous delivery hub. Jenkins provides out-of-the-box packages for any Unix-like operating systems and can be run in any system that has a Java Runtime Environment installed [16].

Jenkins offers an extensive list of plug-ins that extend its capabilities. Developers have a chance to contribute to plug-in development by using extensibility points defined by Jenkins. An extensibility point is essentially a interfaces or an abstract class that defines what needs to be implemented and Jenkins allows plug-ins to contribute to those implementations [17].

Standard installation of Jenkins core comes with a pre-installed set of plug-ins. One of those plug-ins is the JUnit plug-in which allows publishing JUnit test result files [18]. The JUnit plug-in can be configured to publish the result test file after the build process has been executed under projects configurations Post-build Actions.

It is possible to add an unlimited number of build steps to each build process under project configuration. The build step can be anything from executing a Windows batch command or invoking a Gradle script.

The full list of possible build steps included in the standard Jenkins installation:

- Execute Windows batch command
- Execute shell

- Invoke Ant
- Invoke Gradle script
- Invoke top-level Maven targets
- Run with timeout
- Set build status to “pending“ on GitHub commit

5.3 Running Alvor within Jenkins

To configure Jenkins to automatically generate the Alvor report file on build, the shell script described in Appendix B must be used.

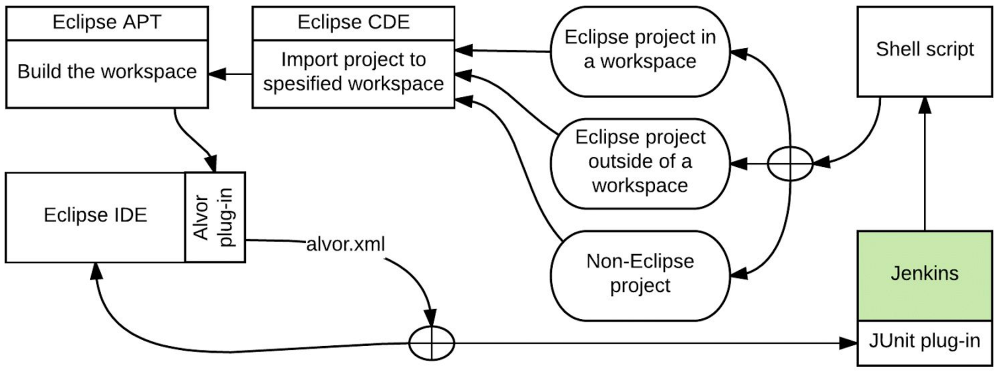
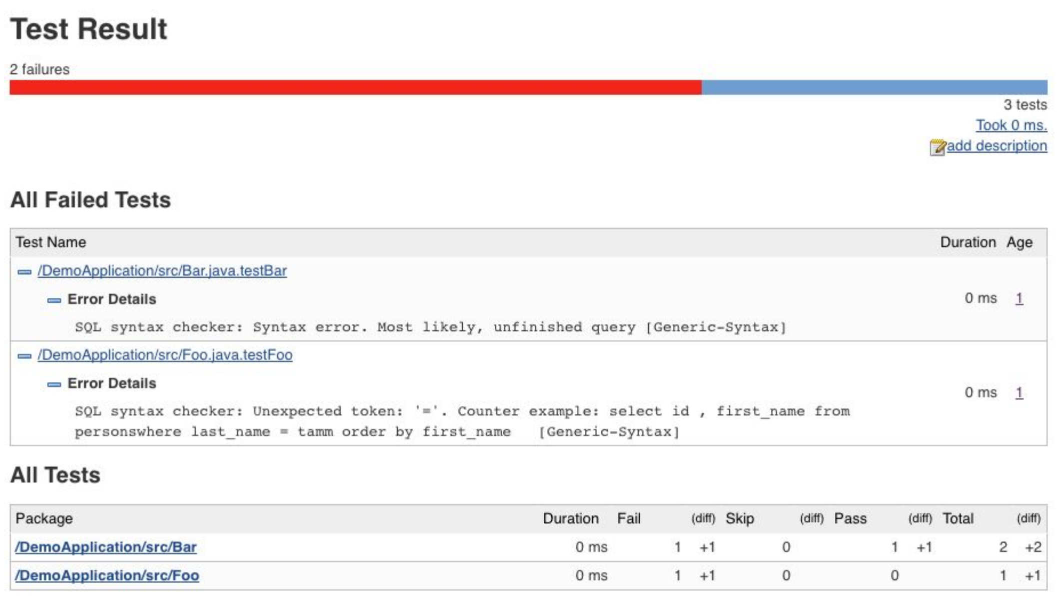


Figure 3. Diagram depicting the workflow of using Alvor for Continuous Integration Testing

To configure Jenkins to use the script, the following must be done:

1. Open the configuration view for the project in Jenkins.
2. From the *Build* tab, click Add *build step* and choose *Execute Shell.*
   2.1. In the command window invoke the script with correct parameters.
   2.2. Use *cp* (copy) command to copy the *alvor.xml* from project root to jenkins project directory.
3. From the Post-build Actions tab click add post-build action and choose Publish JUnit test result report.
   3.1. Specify the location of the Alvor result file under *Test report XMLs.*

Now, each time the Jenkins build is started, Alvor SQL check will be executed and the result file will be published to Jenkins.



Test Result

2 failures

3 tests
Took 0 ms.
add description

All Failed Tests

Test Name	Duration	Age
/DemoApplication/src/Bar.java.testBar		
Error Details	0 ms	1
`SQL syntax checker: Syntax error. Most likely, unfinished query [Generic-Syntax]`		
/DemoApplication/src/Foo.java.testFoo		
Error Details	0 ms	1
`SQL syntax checker: Unexpected token: '='. Counter example: select id , first_name from personswhere last_name = tamm order by first_name   [Generic-Syntax]`		

All Tests

Package	Duration	Fail	(diff)	Skip	(diff)	Pass	(diff)	Total	(diff)
/DemoApplication/src/Bar	0 ms	1	+1	0		1	+1	2	+2
/DemoApplication/src/Foo	0 ms	1	+1	0		0		1	+1

Figure 4. JUnit view for the Alvor result file in Jenkins

6. Limitations

Several issues must be taken into account when using the shell script to generate the Alvor result XML.

Due to simplicity of the generated *.classpath* and *.project* files, more complex projects will fail to import all the required dependencies correctly. This means that some of the compiled classes end up being corrupted. The original tool did not scan the project at all if any of the classes contained syntax errors. This behaviour was changed to accommodate for scanning more complex projects with the shell script. The modified version will scan the project even when there are errors in the code. This means that only the classes with the errors will fail the scan, not the whole project.

The *.alvor* file generated by shell script contains only one predefined hotspot:

```
	<hotspot argumentIndex="1" argumentTypes="*" className="java.sql.Connection"
methodName="prepareStatement"/>
	</hotspots>
```

This means that if the project contains any endpoints for SQL queries other than the predefined hotspot, those endpoints must be specified by the developer manually either through the Eclipse IDE or by editing the *.alvor* file.

To summarize, although the script works for most cases, it still requires some groundwork like installing Eclipse with all the required plug-ins and manual configuration to set it up.

Conclusion

In this Bachelor's Thesis an overview of Alvor tool was given. Alvor is a tool implemented as a plug-in for the Eclipse IDE that allows the developer to statically validate the embedded SQL queries in Java and PHP projects. The goal of this thesis was to allow developers to use Alvor tool outside the Eclipse IDE.

A new feature was added to the Alvor tool that enabled the developer to generate the JUnit result file after each SQL check. This feature was also used by the shell script developed by the Author to enable starting the Alvor SQL check via command line. The script was implemented to work with existing Eclipse projects as well as non-Eclipse projects.

The script was used with the Jenkins CI environment which already had a plug-in to read and display JUnit result files. It is now possible to set up the Alvor tool with Jenkins in such way that only initial setup is needed and the following Alvor scans are fully automated.

There are possible improvements that can added to the existing script that could solve some of the limitations described. The issue of generating Eclipse project files for some of the more complex projects could be fixed for maven and gradle projects by using maven or gradle commands to generate the necessary files.

Bibliography

[1] Annamaa A., Breslav A., Kabanov J., Vene V., An Interactive Tool for Analyzing Embedded SQL Queries, in Programming Languages and Systems -8th Asian Symposium, APLAS 2010, Shanghai, China, 2010.

[2] Tamm U., Eclipse plugin for analyzing embedded SQL queries in PHP programs, University of Tartu, Faculty of Mathematics and Computer Science, Master's Thesis, 2015, https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=46841&year=2015

[3] About Us page of the Eclipse Homepage. https://www.eclipse.org/org/ (22.03.2017)

[4] IDE page of the Eclipse Homepage. https://www.eclipse.org/ide/ (22.03.2017)

[5] FAQ page of the Eclipse Homepage. http://wiki.eclipse.org/FAQ_Where_did_Eclipse_come_from%3F (22.03.2017)

[6] Equinox page of the Eclipse Homepage. http://www.eclipse.org/equinox/ (22.03.2017)

[7] Eclipse Platform Technical Overview, The Eclipse Foundation, 2006, https://eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html (26.03.2017)

[8] Dirk Jagdmann. JUnit XML reporting file format. http://llg.cubic.org/docs/junit/ (23.03.2017)

[9] Headless Building with APT in Eclipse: Eclipse documentation- http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Fguide%2Fjdt_apt_building_with_apt.htm (22.03.2017)

[10] Scriptable builds from the command line. http://gnuarmeclipse.github.io/advanced/headless-builds/ (22.03.2017)

[11] The project description file: Eclipse documentation. http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fmisc%2Fproject_description_file.html (22.03.2017)

[12] Build Classpath: Eclipse documentation. https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Fconcepts %2Fconcept-build-classpath.htm (22.03.2017)

[13] Eclipse Classpath Editing, 2016. http://eclim.org/vim/java/classpath.html (22.03.2017)

[14] JUnit View: Eclipse documentation. http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference %2Fviews%2Fref-view-junit.htm (10.04.2017)

[15] Continuous integration page of the ThoughtWorks Homepage. https://www.thoughtworks.com/continuous-integration (22.03.2017)

[16] Jenkins Documentation. https://jenkins.io/doc/ (22.03.2017)

[17] Plugin tutorial: Jenkins wiki. https://wiki.jenkins-ci.org/display/JENKINS/Plugin+tutorial (22.03.2017)

[18] JUnit Plugin: Jenkins wiki. https://wiki.jenkins-ci.org/display/JENKINS/JUnit+Plugin (22.03.2017)

Appendix A: Glossary

IDE	integrated development environment
XML	Extensible Markup Language
SQL	Structured Query Language
OSGi	Open Services Gateway initiative
CDT	C/C++ Development Tools
JDT	Java Development Tools
CI	Continuous Integration
AST	Abstract syntax tree

Appendix B: Shell script

The shell script is included as an online supplementary material.

The script can be split into three main parts:

1. Generating the missing files (*.project*, *.classpath*, *.alvor*) for the project.
2. Importing the project to a workspace
3. Building the projects

Before invoking the script, the user must add the path to Eclipse executable to the *eclipse_executable* variable.

An example of a successful output for the script that was invoked for a project that did not have any necessary configuration files:

```
Generating .project for project {TestProject}
Generating .alvor for project {TestProject}
Generating .classpath for project {TestProject}
Importing project {TestProject} to {path/to/workspace}
Building project {TestProject}
```