An extensible collaborative framework for monitoring software quality in critical systems

Marisol García-Valls[a], Julio Escribano-Barreno[b], Javier García Munoz[a]

[a] *Department of Telematic Engineering, Universidad Carlos III de Madrid, Leganés, Spain*
[b] *Indra Sistemas, 28018 Alcobendas, Spain*

Abstract

Context: Current practices on software quality monitoring for critical software systems development rely on the manual integration of the information provided by a number of independent commercial tools for code analysis; some external tools for code analysis are mandatory in some critical software projects that must comply with specific norms. However, there are no approaches to providing an integrated view over the analysis results of independent external tools into a unified software quality framework.
Objective: This paper presents the design and development of ESQUF (Enhanced Software Quality Monitoring Framework) suitable for critical software systems. It provides the above enriched quality results presentation derived not only from multiple external tools but from the local analysis functions of the framework.
Method: An analysis of the norms and standards that apply to critical software systems is provided. The detailed and modular design of ESQUF adjusts to the integration requirements for external tools. UML is used for designing the framework, and Java is used to provide the detailed design. The framework is validated with a prototype implementation that integrates two different and norm compliant external tools and their respective quality results over a real software project source code.
Results: The integration of results files and data from external tools as well as from internal analysis functions is enabled. The analysis of critical software projects is made posible yielding a collaborative space where verification

Email addresses: mvalls@it.uc3m.es (Marisol García-Valls), jebarreno@indra.es (Julio Escribano-Barreno), javiergm@it.uc3m.es (Javier García Munoz)

engineers share information about code analysis activities of specific projects; and single presentation space with rich static and dynamic analysis information of software projects that comply with the required development norms.

Keywords: Software quality, Open source framework, Quality monitoring, source code analysis, critical systems development

---

1. Introduction

In critical software projects, the software development process needs information about almost every aspect across the different traversed phases. These information include achievement of objectives, monitoring and control of activities, project costs tracking, and technical quality assessment. The obtained measures for all these values define the quality. In 1978, [2] already stated the importance of measuring the software quality and defined a set of important measures to assess the overall quality of a software project. Metrics are of key importance in all engineering disciplines and, in particular, in critical systems' software development (see [3]); these provide a vital insight into the development process to assess the level of maintainability, reliability and even development progress. Metrics offer reproducible indicators useful to estimate the quality, performance, management, and cost within a project. By working with metrics in the software development, it is possible to analyze the data to understand and improve the system behavior; moreover, it is possible to predict future behaviours that will allow engineers to undertake corrective actions at an early stage avoiding runtime effects.

Metrics have been studied and developed through the years, increasing their relevance due to their applied use and the contrasted benefits to define baseline quality indicators for serveral purposes. For example, the Capability Mature Model Integration (CMMI) [4] for development, relies on the usage of metrics (see [5]), and it is used for evaluating the maturity process of the organizations.

The collection and processing of the metrics and other quality indicators about a software project can involve a significant human effort [20]; this highlights the need of automating the specification of metrics and subsequent data collection that must later be processed. By using automatic analysis tools (e.g. [6] for static code analysis), the effort in collecting quality information is significantly reduced. However, there are few open source platforms that support the automatic management of heterogeneous software quality information.

One of the most popular quality management platform is SonarQube [8][10] that is becoming popular also due to its open source nature. Among the functionality provided by SonarQube, it enables continuous inspection of software projects and supports a number of languages, including Java, C, C++, C#, PHP, and JavaScript. SonarQube provides some *basic* metrics like complexity, duplicated code detection, or lines of code counting, among others. However, this is a very generic functionality that requires to be enhanced for software development projects of a certain complexity. There are other research oriented platforms such as Alithea [12] that is an extensible platform to integrate analysis for software engineering research, but it has not a focus on critical software systems.

Highly complex projects such as those related to critical software development require that these basic metrics be enhanced to provide the information suited for each particular project. The reason is, for instance, that each project may have to adjust to specific norms for the particular field. It is, then, useful and needed to take information from heterogeneous sources (like external tools that comply to the required norms) and to establish the adequate methodology in order to enhance the functional power of a simple quality framework to provide the required information with a suitable design that makes it easily customizable; and that supports the collaboration of the users.

Existing improvements to the SonarQube open source framework have been undertaken as pure coding tasks, being the vast majority for Java language; however, critical software systems code tend to use C/C++ (besides other languages as e.g., Ada) for which there is practically no support in the mainstream software quality frameworks community. Critical software projects follow different norms and standards (e.g. DO178C[21]) for the specific target application domain. Profiles such as MISRA C [44] and MISRA C++ [45] are applied with the goal of meeting the requirements of security, portability, and reliability of embedded software.

Up to the best of our knowledge, there are no contributions that provide an approach to systematically design and develop multiple local and remote integration of software quality information that is elaborated by heterogeneous sources and tools that provide C/C++ code analysis. In this paper, we present an approach to enrich the management of these data, offering it to the verification engineers through a collaborative platform so that they can remotely access to jointly participate on the code verification process of specific projects. This has been performed by designing a modular framework

composed of a number of analyzer modules that provide the enhanced functionality for the platform in order to integrate its own analysis results with the ones from external analysis tools in a single presentation space. This platform, named ESQUF (Enhanced Software QUality Framework) integrates metrics ellaboration, coding rules checks, and coverage analysis. The design of this platform is a modular one that allows to easily customize the integration of any external code analysis tool. The result is the achievement of a more complete set of information that can be managed in the projects to control and monitor the software development process according to the needs of each specific project. The information from the analysis of the project code is then centralized in the platform that is, at the same time, a collaborative environment that allows the remote work of teams of verification engineers. We validate the framework on a specific integration with external information sources that provide static analysis metrics. In our work, the rules, metrics, and unit tests can also be extended to complement the provided ones. We exemplify technical information related to software quality through actual static and dynamic analyses on a real critical software project.

The new framework is validated against the following parameters that are expected to be provided jointly: capacity of extracting meaningful metrics from code from multiple external tools; provision of a joint, collaborative presentation space; and adaptation to different coding norms and standards.

The paper is structured as follows. Section 2 describes the related work in what concerns norms and practices related to technical quality of critical software systems. Section 3 describes the characteristics of quality frameworks for code analysis. Section 4 describes the proposed ESQUF framework. Section 5 provides the design details of the ESQUF modules for supporting external analysis tools integration. Section 6 validates the design through a prototype implementation that monitors a real software project where different DO-178C norm compliant code analysis tools are integrated in the overall framework. Section 7 concludes this paper.

2. Software quality management in critical systems

Software quality can be analyzed through the source code by means of metrics, compliance to the coding rules of the mandatory standards, and by collecting the information about a series of pre-specified unit tests. The software of critical software systems is developed according to strict adherence

of a set of norms which define the best practices that the software and the development process must adhere to. This section describes some selected work that is most related to our contribution: the existing norms and regulations for critical software development; current tools for software code analysis; and the engineering processes that describe the steps to the target software development.

2.1. Critical systems' norms and best practices

Technical metrics are collected through static analysis techniques. Software static analysis is required in several norms related to critical systems. Some of these standards are defined in Table 1.

Although there are a lot more than these, the norms refered to here are a representative set of some of the major critical software systems in the development of transportation systems that is later used in the development of the validation scenario. The norms provided by Table 1 require the use of static analysis techniques to comply with their objectives. Among others, DO-178C ([21]) and DO-278A ([22]) define the guidelines and best practices to be applied to the development of software for critical systems. Both DO-178C and D0-278A introduce the use of metrics to be specified in the *software quality assurance plan* as a way to assess quantitatively the characteristics of the software within the project. For critical systems, metrics collection and analysis allows engineers to prove that the developed software complies with important norms, like North Atlantic Treaty Organization (NATO) AQAP-2210 [23], and its spanish version PECAL-2210 ([25]), among others.

The norms applied in critical systems development may have to evolve as new domains come into scene; this the case of cyber-physical systems or social dispersed computing applications [19]. In these highly dynamic environments, the design of the system cannot consider all possible situations that will be encountered at execution time; but still, these systems need to achieve some levels of predictability and guarantees. For these situations, the approach is different and a number of works are appearing to utilize verification at runtime based on formal approaches such as [17] using Petri nets, or [18] using CLTLoc.

Table 1: Overview of the main norms and standards related to critical systems

Doc. No.	**Title**	**Description**
DO-178C	Software Considerations in Airborne Systems and Equipment Certification [21]	It is one of the most accepted international standards. It includes additional objectives and it is complemented with the supplements [30], [31], [32] and [33]. The previous version of this norm is DO-178B (Ref.[29])
DO-278A	Guidelines for Communication, Navigation, Surveillance and Air Traffic Management [22]	Provides guidelines for non-airborne Communication, Navigation, Surveillance and Air Traffic Management (CNS/ATM) systems. This document provides the guidelines for the software assurance activities to be conducted with non-airborne CNS/ATM systems.
IEC 61508	Functional safety of electrical/ electronic/ programmable electronic safety-related systems [27]	Standard for Industry automation. It is intended to be a safety standard applicable to all kinds of industry. Includes the complete safety life cycle. Used as basis for other documents, as railway (CENELEC 50128[35]), automotive industries (ISO 26262[27]) or nuclear power plants (IEC 61513[28])
CENELEC 50128	Railway applications - Communications, signalling and processing systems [34]	Applied to railway industry. It specifies the processes and technical requirements for the development of software for programmable electronic systems for use in railway control and protection applications

2.2. Software quality monitoring based on code analysis

Achieving the highest possible quality of the software is of vital importance in critical systems. This goal influences every aspect of software system development such as the functionality, reliability, availability, maintainability, and safety, among others. In critical software projects, quality assurance has to be taken into account from the beginning of the development and it has to be considered at each level of the software engineering process: from specification to coding and integration.

The process of code-level quality monitoring of a software can be approached from two different levels:

- *Code analysis tools* present a lower level of code quality monitoring in which individual tools are selected to perform different checks on the code. Each tool can be specialized in some particular analysis technique over, mostly, some specific programming language. Its advantage is that individual tools that perform sophisticated tests on the code can be selected, from simple line counting [37] to memory leaks and potential deadlock detection [14] or worst case execution time analysis [15]. The ones that are suitable for critical projects are, most of the time, proprietary and do not support their direct integration with other also especialized tools.

- *Quality monitoring platforms* are a type of technology aimed a providing a blackboard that can include different code analysis functions such as [12, 10]. In this category, only SonarQube has the advantage of facilitating the integration of functions implemented by multiple heterogeneous developers; it is an open source quality management platform, used to analyse and measure technical quality. Despite being an interesting and needed technology, the number of supported languages is very limited. Precisely, the available SonarQube functions are almost exclusively targeted at Java language. Also, it is not integrated with the analysis provided by other specialized external tools for code analysis using the specific development standards for critical software systems.

This platform can be extended through plugins that allow to customize and integrate it with other tools. The specific conception and development of these plugins is a significant design effort that allows to ensure the correct handling of the different phases involved in the software quality process.

The metrics and quality information with respect to a complex software project typically requires different analysis techniques that generate results and data from different sources, possibly also collecting data in different ways. For some specific projects, it is more relevant to analyze dependencies across the included packages such as [7] than to analyze the source code. Information about the software is in the end collected in different ways by means of different tools such as the ones presented in table **??**.

As not a single code analysis tool is capable of providing all the required data for a comprehensive software quality monitoring activity, engineers are

Table 2: Static analysis tools.

Tool	**Description**	**Open source**
Understand ([37])	Commercial tool for static code analysis. Multi-language. Developed by Scientific Toolworks, Inc.	—
LDRA ([38])	Static code analysis tool. Multi-language. Developed by Liverpool Data Research Associates (LDRA)	—
PC-Lint ([39]	Static code analysis tool for the C/C++ languages. Developed by Gimpel Software.	—
Splint ([40])	Static code analysis tool for C language.	✓
PMD ([41])	Static code analyser for Java, JavaScript, XML and XSL.	—
VectorCAST ([46])	Code Coverage Analysis. Developed by Vector Software.	—
SonarQube ([10][8])	Quality management platform	✓
Astrée ([48])	Runtime error detection by abstract interpretation for safety-critical C code	—
Eclair ([49])	Runtime errors in source code using abstract interpretation, model checking and constraint satisfaction	—
Frama-C ([50])	Code analyzer for C program analysis without program execution	✓
Polyspace ([51])	Static code analysis tool for C, C++, and Ada using abstract interpretation	—
SPART Examiner ([52])	A part of SPARK Toolset based on SPARK language (a subset of the Ada language) for Ada code analysis in high integrity systems	✓

obliged to set up a tool chain for code analysis; such tool chains comprise a number of proprietary tools that are not automatically interoperable with other tools. To overcome this, engineer teams establish the procedures and principles for data collection across the tool chain and interpretation. Also, model-based testing [1] is being used to alleviate this task although at a much earlier step in the development. As a result, the SonarQube platform has appeared as a platform to support advanced analysis; however, SonarQube appears in a similar way to a blank sheet of paper, so that the required techniques and methods to implement the needed functionality have to be designed and integrated in it.

The tools mentioned here are used to statically analyze the code quality. However, the metrics that they yield do not meet all the requirements across different projects. In each project, different metrics can be required, or different implementation languages can be used that vary from a project (e.g. railway) to another (e.g. automotive). Then, a tool that is suitable for a project may not be valid nor usable in a project on a different field that must comply with other norms. Among the extremely wide range of tools for code analysis, we have provided a selected set that is sufficienly representative of the source code analysis spectrum for critical systems. The first part of table 2 lists tools that are amongst the most used in commercial critical software developments that support C analysis. However, for the sake of generality, it should be acknowledged that there are other more theoretical tools based on formal methods that are used in code analysis for projects in different language and application domains. For this purpose, we have added a second half to table 2 in which formal methods tools are shown. Among these tools, we find *Astrée*, Frama-C, Eclair, Polyspace, Spark Toolset.

2.3. Limitations

There is a severe limitation in the current status of the software quality management platforms. To illustrate it, let us see the example of SonarQube where each user needs to have the tool installed locally; as such, the run-time environment has little orientation to becoming a collaborative environment. By collaborative, we mean the capacity to simultaneously support the remote access of the users to several projects, each having a given number of other users. Users are typically part of the verification team of a critical systems software project.

Also, there are severe limitations to support the customizable integration with external tools for code analysis to enrich the analysis functions and

the quality results presentation. Currently, there is no approach for making software quality management platforms comply with the norms, regulations, and best practices needed by critical software systems such as those provided in section 2.1.

In its current form, open platforms such as SonarQube are fully customizable frameworks as they merely contain a very basic set of functions that implement the collection of metrics and data from software projects. Up to the best of our knowledge, it lacks a systematic customization for particular projects that have very specific information requirements such as critical software systems. The currently available analysis functions of this platform are almost exclusively focused at Java language that is not a programming language used in the core parts of critical software systems.

3. Open source software quality frameworks

Software quality management platforms are typically provided as proprietary tool chains that provide results of analyzing the source code files of the project. Open source software quality frameworks are not specialized for critical software projects where interaction with norm-compliant analysis tools needs to be done. These platforms need to be multi-language, and they should be capable of performing analysis over the source code such as providing information about duplicated code, unit tests, coding standards, code coverage, code complexity, comments to the code, and software design and architecture.

The considered frameworks rely on two main functions: source code analysis modules that are components that perform the basic analysis activities (such as counting lines of code) and presentation modules that display the analysis results. The most popular framework, SonarQube, provides an extendable and customizable environment where analysis functions can be added to provide different complementary views on the software quality. These additions are realized through a set of components:

- *Presentation configurators* are the modules that configure the graphical display of data resulting from the source code analysis, i.e., they enables the customization of the analysis results presentation to the user. A graphical module supports the specification of the visual format and of the display locations of the presented data. Each graphical module yields one of the square boxes that are shown. Each box contains

a number of data items whose display location and characteristics is indicated in the widget code. For each new analysed project, a *project dashboard* should be assigned for selecting, adding, or removing the available graphical modules.

- *Resource accessors* are components that provide access to the files to be analyzed: the project source code and the results files from external analysis tools. Code accessors contain also the logic and analysis functions over the source code or result files to extract more complex data and metrics.

- *Processors* are the components that support the programming of additional processing functions over the initial metrics (i.e. the initial analysis results) provided by the resource accessors in order to derive more complex metrics.

The framework allows to extend its functionality via *plugins*. A plugin is an application integrated into a broader scope main application to add new functionality or to customize the current functionality. In a software quality framework, plugins are used to implement graphical modules for different purposes such as supporting additional programming languages, or include changes in dashboards, among others. The analysis over the source code of a project can be executed at different times to check, for instance, the evolution of the coding process. A plugin can be designed for presenting the evolution of the development process; the results from different executions can be stored in a data base, and the plugin logic can check the main differences between analyses, offering the possibility of providing more information over the project evolution.

4. Enhanced Software Quality Framework

4.1. Requirements

Critical software systems such as railway control software, aircraft systems operation software, etc., are typically very large software projects that follow strict norms through all their life cycle, and they must undergo heavy and extensive verification activities prior to the production phase. An average critical software project boils down to thousands of source code files that have to be tested by an independent verification team that, for safety reasons, is different from the development team. In real practice, the team of verification

engineers work on different individual tools for code analysis that are used to pass the set of predefined tests for every single source code file. Typically, engineers supervise and analyse the results of the code analysis from different files to check whether all major tests have been passed. In this way, the main driver and motivation for the requirements of our proposed framework are: (1) the absence of a framework for easing the integration of analysis results from heterogeneous sources and tools into a space on which engineers can contribute their analysis results collaboratively; and (2) the difficulty in integrating different norms into this framework that are mostly described in text files as rules that apply on the different coding phases. Based on the above, we now summarize the requirements of our *enhanced software quality framework* (ESQUF):

- *Execution of internal and external analysis functions.* The tool must have a set of analyzer modules to carry out the analysis of both, source code and external files containing analysis results for external analysis tools. These modules can be either plugins to external functions integration or internal modules with additional functions.

- *Interpretation of external analysis results.* The tool must must be capable of performing its own analysis over the source code and to interpret results of other tools. This way, the results generated by the tool will be richer.

- *Web based presentation.* The tool should have the posibility of presenting results from data collection through a web page or app or document.

- *Project evolution presentation* over time. It is needed to have a data base to collect information of the different analysis performed and it is needed that these data are available and retrievable anytime and for future execution.

4.2. Architecture of the proposed quality framework

Figure 1 presents the general design of the framework: it provides a unified presentation space for engineer teams working on the software analysis; the collaboration of the engineers is enabled by means of a remote web server that accesses the source code to perform its own built in analysis functions, together with the access to external analysis tools. The extended software

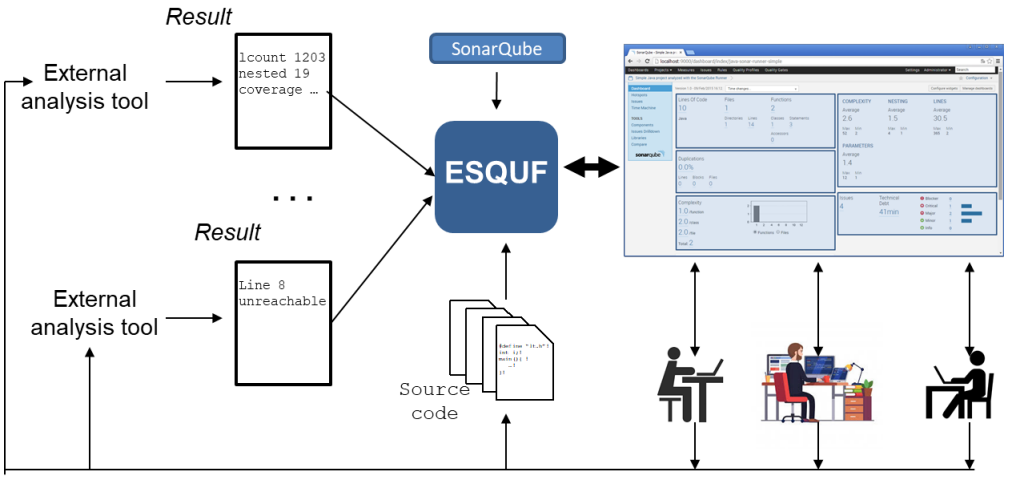

Figure 1: Platform overview

quality management plaftorm performs the activities shown in Figure 2 that are split into online and offline ones. The platform supports the requirement of using external specialized analysis tools in critical projects that comply to specific required norms. The source code is analyzed by some external tools and simultaneously it can be analyzed by the framework itself and its built in functions. The external results are fed to the extended software quality framework (ESQUF). The data about the obtained quality from both these external tools and from the built in functions are presented in an integrated maner from the framework.

Figure 3 provides an insight into the platform design, showing how the analysis work is done precisely on the code. ESQUF components are represented as rounded boxes with solid lines; SonarQube components are shown with dashed lines, and the relation across ESQUF and SonarQube components is shown by means of dashed arrows. The framework has two main types of modules:

- *Analyzer modules* that perform the following activities:
  - direct analysis of the source code files contained in a given project.
  - processing of analysis results, i.e., this function further processes the files containing quality results from previous analysis over the code files; these analysis results files can be produced by other tools or by the basic analysis functions of the framework.

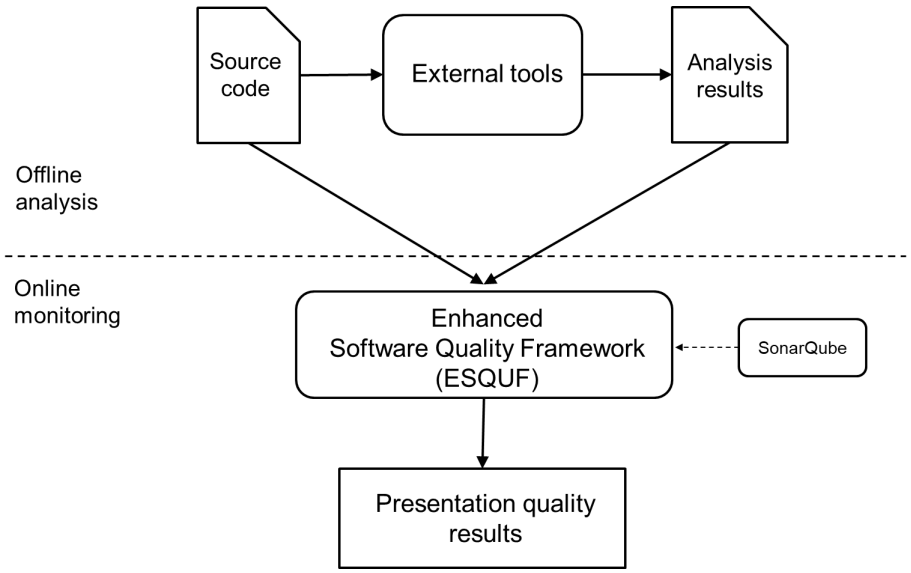

Figure 2: Platform actitivies that are performed online and offline.

- *Presentation module* that handles the rendering of the analysis results; this includes the grouping of the different analysis by types, e.g, source code metrics, coverage results, etc. This module performs a partitioning of the presentation area that is divided into locations for the presentaion of the analysis results by types. It contains an applicator server that graphically displays the data that results from the analysis in a browser front-end.

- *Data base interface module* provides an abstraction to interact with a specific repository of the analysis results that should be stored in persistent storage. This module allows the plaform to be data base independent.

- *Workflow engine module* provides an active control loop that guides the platform execution across the different functions that it performs. It also contains a collaboration function through which the plaform coordinates the access from the posibly multiple users that have access to it simultaneously.

The framework allows to extend its functionality via *plugins*. A plugin is an application integrated into a main application of a broader scope to

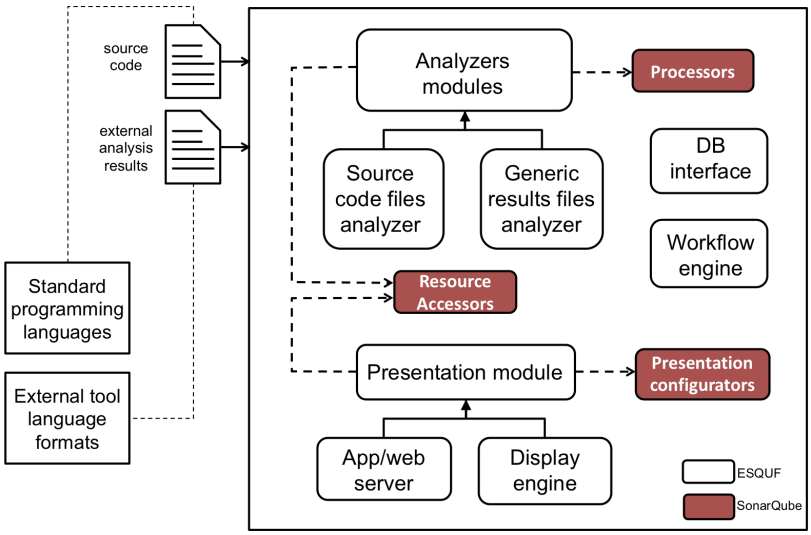

Figure 3: Components of quality framework

add new functionality or to customize the current functionality. Typically, plugins are used to implement the presentation configurators for different purposes such as supporting additional programming languages, or include changes in dashboards, among others. The analysis over the source code of a project can be executed at different times to check, for instance, the evolution of the coding process (see Figure 4). A plugin is designed for presenting the evolution of the development process; the results from different executions can be stored in a data base, and the plugin logic can check the main differences between analyses, offering the possibility of providing more information over the project evolution.

4.3. Functionality

Basic functions. ESQUF provides a set of basic functions as shown in Figure 5 to provide the quality results of the project over its code analysis. On the one hand, the general information about the analyzed project can be seen together with some basic metrics, a coverage analysis and, at last, the technical debt. Technical debt is an interesting metric as it provides an estimation of the time required to fix the issues found in the analyzed project.
Additional quality information and norm compliance functions. In software projects for critical systems, the specific standards that must be applied require more complex metrics over the code. A few examples of these

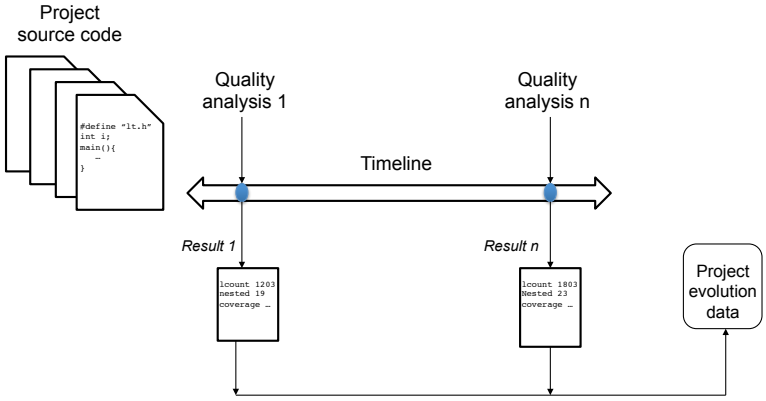

Figure 4: Project evolution presentation

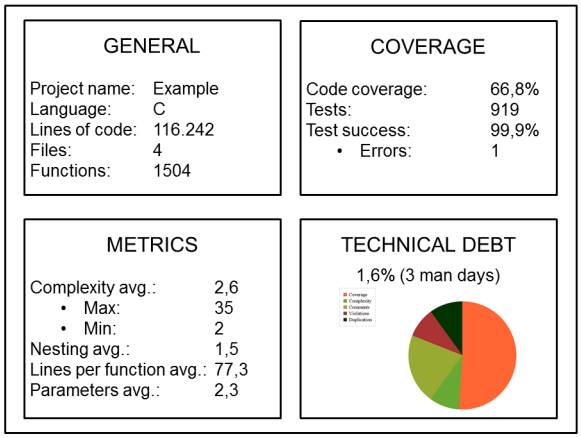

Figure 5: Example of results analysis

are: complexity, nesting levels, function parameters, and other values derived from the previous ones such as maximum, minimum and mean values for each of the previous metrics. These are not provided by general purpose software quality frameworks. There are some specialized tools that do provide a broad range of complex metrics; and these are external to the framework and must, therefore, be integrated. Besides the metrics and graphical data or results from unit tests, information on the incorrect lines of code that indicate where some programming rule has not been respected, are provided. This way it is possible to see at a glance all the most relevant information of a project

and it will be easier to make a general assessment of the progress and of the needed time to make the required corrections. Programming rules are fed to the system according to the specific norm that should be fulfilled for a given project.

We contribute to the needed functional enhancement of the quality management platform SonarQube by designing and developing a module that is based on a plugin that collects the analysis results from an external tool for static code analysis. It is important that the module is designed to be used remotely in the framework, so that collaborative working is enabled. The result is that users view richer information over a software project code by using SonarQube as the single front-end, presenting a number of metrics in the project dashboard, as an additional widget.

ESQUF provides this additional information to what a mainstream software quality framework provides. This requires a functionality that augments the basic analysis facilities by designing enhancement modules that have the following characteristics:

- plugins are designed to collect the analysis results from external tools compliant with changing norms such as DO-178C and DO-178B;
- static and dynamic code analysis is performed over the project source code;
- remote and collaborative working is supported by means of a web interface;
- richer quality information is provided on a single front-end, presenting a number of metrics in the project dashboard by using info accessors to access source code and results files from other tools;;
- flexible presentation of quality information as presentation configurator modules are used;

4.4. ESQUF structure, mappings, and activation

Structure. The designed module contains a *ResourceAccessor* to access the analysis results generated by the external tools, and a *PresentationConfig* to present the results in the project dashboard.

There are two ways to design a *ResourceAccessor*, as presented in the two paths displayed in Figure 6. On the one hand, a *ResourceAccessor* template

can be used to directly acess the source code that will be analyzed; while the access is performed the *ResourceAccessor* can contain additional logic to also process and analyze the source code. Another possiblity is to use a *ResourceAccessor* template either integrate (or perform some additional processing over) the analysis results generated by an external tool.

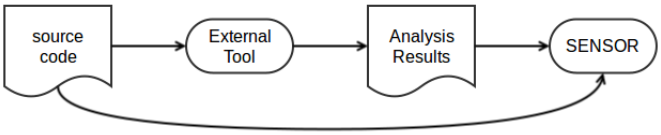

Figure 6: *InfoAccessor* for access to both source code of a project and/or results files from a previous analysis.

In what follows, the structure of a *ResourceAccessor* is provided. The `ResourceAccessor` interface is a tagging component, i.e., a class extending this interface is automatically a *ResourceAccessor* as it is obligued to implement the `analyse` method to provide a customized functionality for the accessor.

```
public interface ResourceAccessor {
        void analyse(Project prid);
}
```

Mappings. ESQUF is a platform independent design. Its main design is based on SonarQube open source framework. For this purpose, this section presents an overview mapping whereas the detailed mappings to SonarQube constructs are further elaborated in section 5.

An ESQUF *ResourceAccessor* is mapped to a SonarQube `Sensor` that is also a tagging component for implementing resource accesses.

```
public interface Sensor extends ResourceAccessor, BatchExtension
    , CheckProject {
        void analyse(Project module, SensorContext context);
}
```

PresentationConfig components in ESQUF are mapped to *Widgets* of SonarQube. These define the placement of displayed data on the screen.

Activation structure. The workflow engine of ESQUF (see Figure 3) uses the *Runner* component of SonarQube web server. The workflow engine determines the activation of the different components of the extensions and pluggins in the appropriate order to execute their associated functions that deliver the analysis and display results. Essentially, the workflow engine

executes the *ResourceAccessor* components that access the results files from the external integrated tools for code analysis and perform further processing on these data as indicated in its corresponding *analyse* function. The results of the additional processing are displayed by the *widgets* of the presentation module.

5. Analyzer modules

This section presents the design of ESQUF: metrics analyzer that is a module that is designed to provide the basic project information; a norms compliance module that is named *rules analyzer* that assures integration of especialized tools according to the specific norms that a software project will adhere to; and the *coverage analyzer* that provides dynamic code quality analysis.

5.1. Integrated metrics analyzer

This analyzer is a module that supports the integration of the local analysis functions with the quality analysis results of external analysis tools on a single presentation space. This module is designed via a plug in that integrates the external data and displays the resulting integrated metrics. It has a flexible and modular design to allow any external tool to be integrated with minor modifications.

The architecture of the analyzer is shown in Figure 7 that contains the following classes:

- *ExternalToolPlugin* is the class containing the specification of the properties of the analyzer module that are `metric` for the structured definition of metrics in SonarQube and `language` that indicates the programming language of the project source code. The code below shows the structure of `ExternalToolPlugin` and the specification of a `Property`.

```
@Properties({
  @Property(
    key = Plugin.EXTERNAL_TOOL_METRICS,
    name = "Metrics path",
    description = "Path to the results of external tool
        analysis results.",
    global = false)
})
public final class ExternalToolPlugin extends SonarPlugin {
```

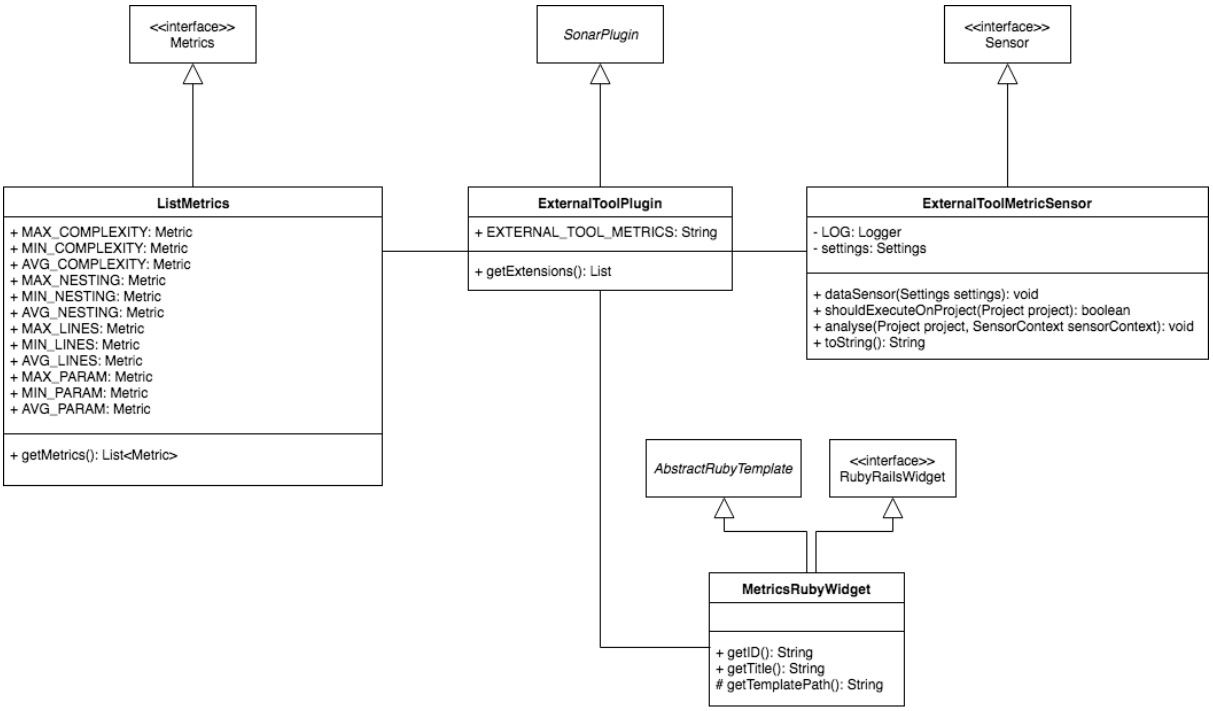

Figure 7: Class diagram of metric plugin for a specific external metrics tool

```
  public static final String EXTERNAL_TOOL_METRICS = "sonar.
      externaltool.metrics";

  public List getExtensions() {
    return Arrays.asList(
        ListMetrics.class,
        ExternalToolMetricSensor.class,
        MetricsRubyWidget.class);
  }
}
```

- *ListMetrics* is the class that specifies all metrics to be used (displayed) by the plugin. Metrics should be specified by name, type, description, qualitative or quantitative and domain. Sensors later assign values to each metric of the list as a result of the source code analysis done by ESQUF or by some other external tool. In this module, this class contains all metrics that are provided by the external tools.

- *ExternalToolMetricSensor* is the class that contains the functionality to scan the quality results produced by the external analysis tools in order to collect their metrics. This class is invoked when the workflow module component is executed.

- *MetricsRubyWidget* is the class that contains the definition of the properties of the widget, the title of the widget, and the the design and display characteristics of the data to be included in the project dashboard.

5.1.1. Metrics

The definition of *Metric* class contains a number of parameters of interest such as: identifier for having a unique addressable identity, the type of data (e.g., whether it is an integer value, Float, String, boolean, etc.), the name for user friendliness, the domain (e.g. general or complexity), and an indication of whether it is quantitative or qualitative.

The two main actions of the integrated analyzer module are the scanning of the external analysis data that is performed by the file analyser class (*ExternalToolMetricSensor* class); and the integrated results presentation performed by the widget *MetricsRubyWidget* class. Both classes require to

have knowledge about the metrics that are defined in the external tool. Such information is contained in the *ListMetrics* class.

ListMetrics class contains the set of metrics that are considered by the external tool and by the integrated analyzer module, and a single method `getMetrics` that will be able to list all of them.

`ListMetrics` class implements the interface `Metrics`.

```
public interface Metrics extends BatchExtension, ServerExtension
{
        List<Metric> getMetrics();
}
```

`ListMetrics` includes all metrics that have to be obtained by the sensor in order to be stored in the ESQUF data base. Later, the presentation configurators will be able to access these stored metrics.

5.1.2. Resource accessor

The logic to be performed by the *ExternalToolMetricSensor* is contained in the `analyse` method. Initially, it has to access the external results data. First, the properties of the metrics are read; then the metrics that it contains are checked against the list of metrics given by *ListMetrics*. All matching metrics are stored in the data base.

```
public void analyse(Project Project, SensorContext sensorContext
    ){

    String path = settings.getString(ExternalToolPlugin.
       EXTERNAL_TOOL_METRICS);

    while(read != null){
        String metric = /* Reading of the name from the read
            line */
        Double value = /* Reading of the value from the read
            line */
        switch(metric){
            case "Max complexity":
                measure = new Measure(ListMetrics.MAX_COMPLE);
                break;
            case "Min complexity":
                measure = new Measure(ListMetrics.MIN_COMPLE);
                break;
                        (...)
                        /* Goes on for all supported metrics */
                        (...)
```

```
            default:
                System.out.println("Error: Metric "+metric+" not
                    found");
                break;
        }
        if(measure != null) sensorContext.saveMeasure(measure);
       /* Continues reading */
    }
```

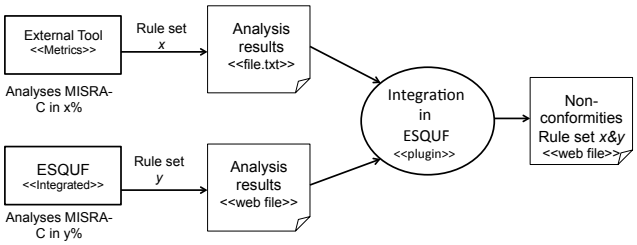

Figure 8: Integration of ESQUF with external tools

5.2. Integration of norms and coding rules

This section presents the design of the logic that supports the compliance of the quality analysis to specific norms for critical software projects. This functionality is provided by an *integrated rules analyzer* module that achieves the integration of the local analysis results of ESQUF with the ones from external analysis tools that implement these norms. This module allows new rules from different criticality levels to be added to ESQUF to later analyze the results files generated by external tools to present those parts of the source code where there are some specific rules violations.

The architecture of the analyzer is shown in Figure 9 that contains the following clases:

- *ExternalToolRulesPlugin* class is similar to the class *ExternalToolPlugin* of the previous integrated metrics analyser module. A new property is defined that indicates the path to the results file of the external tools. In what follows, it is presented a summary of the class:

```
@Properties({
  @Property(
    key = ExternalToolRulesPlugin.EXTERNAL_TOOL_RULES,
    name = "Rules path",
```

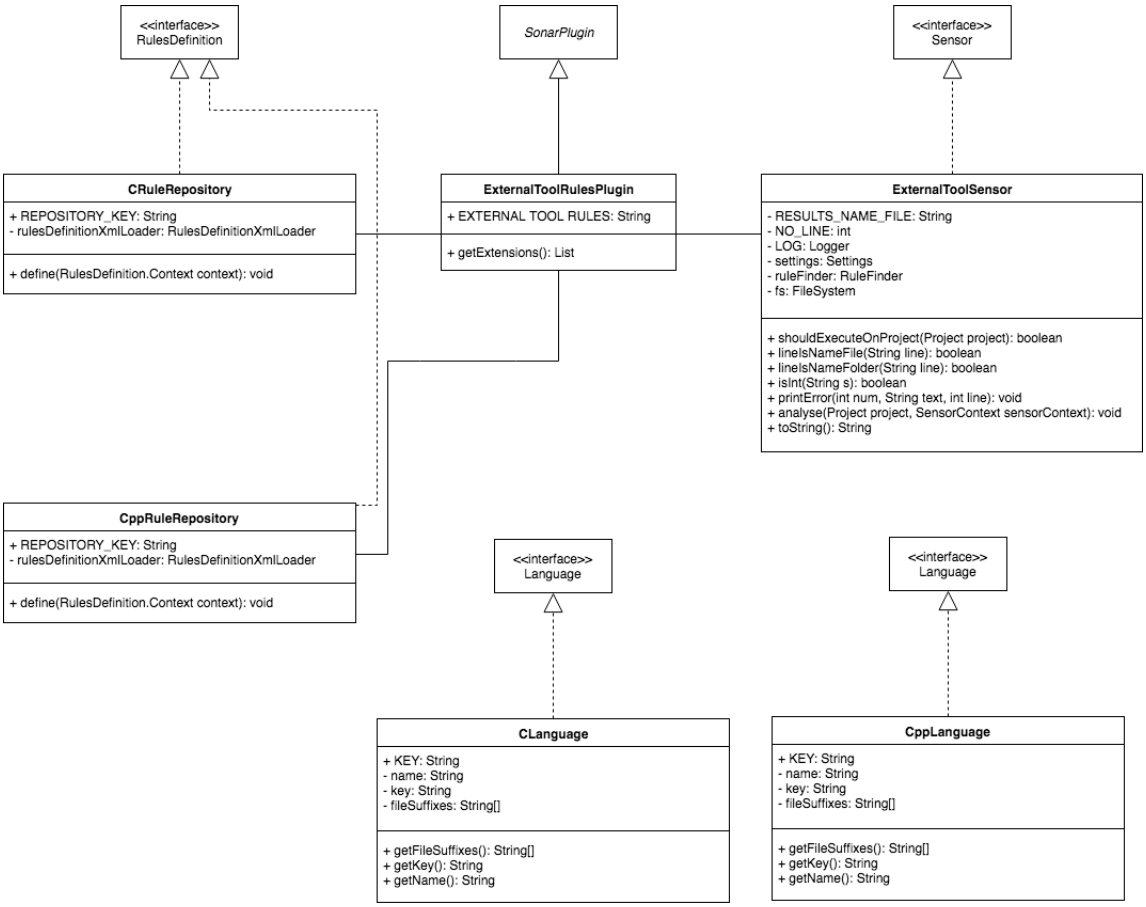

Figure 9: Class diagram rule plugin

```
    description = "Path to the results files of the external
        tools",
    global = false)
})
public final class ExternalToolRulesPlugin extends
    SonarPlugin {

  public static final String EXTERNAL_TOOL_RULES = "sonar.
      externaltool.rules";

  public List getExtensions() {
    return Arrays.asList(
      ExternalToolSensor.class,
      CRuleRepository.class);
  }
}
```

- *CRuleRepository* class contains the information about the rules repository that will be included in ESQUF. Within this module it is defined: the programming language (e.g. *CLanguage* for C), an indentifier or key, a name, and the path to the rule repository. Rules are specified in a flexible and portable format by using XML format (5.2.1). The invocation to *define* sets all the properties.

```
public final class CRuleRepository implements
    RulesDefinition {

  public static final String REPOSITORY_KEY = "CRules";
  private final RulesDefinitionXmlLoader
      rulesDefinitionXmlLoader;

  public ExternalToolCRuleRepository(
      RulesDefinitionXmlLoader rulesDefinitionXmlLoader) {
    rulesDefinitionXmlLoader = rulesDefinitionXmlLoader;
  }

  public void define(RulesDefinition.Context context) {
    NewRepository repo = context.createRepository(
        REPOSITORY_KEY, CLanguage.KEY).setName("Example");
  }
}
```

- *CLanguage* class defines the programming language that will be interpreted by ESQUF in the corresponding analysis. As for the type of critical

systems that we target at, C is used; however, other languages are possible (e.g. C indicated as "c", C++ indicated as "c++", or Java indicated as "java", among others). Also, the extensions used by the files of the programming language must be indicated.

```
public class CLanguage implements Language {
        public static final String KEY = "c";
        private String name;
        private String key;
        private String[] fileSuffixes = new String[2];
        public CLanguage() {
                this.name = "C";
                this.key = KEY;
                this.fileSuffixes[0] = "c";
                this.fileSuffixes[1] = "h";
        }
}
```

- *ExternalToolSensor* is the sensor class of this plugin that analyzes the external tool results file and marks the rules are violated toghether with the corresponding line of code of the source file. This class is invoked when the the workflow module of ESQUF is run.

5.2.1. Rules

The rules file is integrated inside this analyzer module. Each rule has an identifier (i.e., a key); a priority (i.e., info, minor, major, critical, or blocker) depending on the severity of the rule; a name; a configuration key; and a description. The rules file has the following format:

```
<rules>
 <rule key="no_comments" priority="MINOR">
  <name>Comments are not permitted</name>
  <configkey>NO_COMMENTS</configkey>
  <description>
    <p>This line is a comment and comments are not permitted</p>
  </description>
 </rule>
 <rule key="example" priority="INFO">
   <name>Variable not initialized</name>
   <configkey>VAR_INIT</configkey>
   <description>
    <p>This variable has not been assigned a value.</p>
   </description>
```

```
 </rule>
</rules>
```

5.2.2. Resource accessor

The central element of the accessor is the *analyse* method that is invoked by the ESQUF workflow module. *analyse* processes the quality results file from the external tool. The accessor reads the result lines one by one and stores the information into the ESQUF data base; also, further processing to these external results files can be done by additional logic that could be contained in *analyse*.

```
public void analyse(Project project,SensorContext sensorContext)
{
 int violationLine = 0;
 String violationText = null;
 String violationLongText = null;
 [...]
 Language l = new CLanguage();

 repositoryKey = CRuleRepository.REPOSITORY_KEY;
 while((read != null) && (repositoryKey != null)){
  violationText = (read.split("\t"))[4];
  violationLine = Integer.parseInt(. . .);
  Rule rule = null;

  if ((violationText != null) && (!violationText.equals("")))
   rule = ruleFinder.findByKey(repositoryKey, violationText);
  else
   printError(violationText, line);
  [...]
}
```

This method has to be adapted to interpret the results of the rule checking of the specific external tool that is integrated. The logic can be modified accordingly for further processing over the external results files. In the following, it is shown the part in which a non compliance to a rule (a rule violation) is detected.

```
violation violation = Violation.create(rule, resource);
violation.setMessage(violationLongText);
violation.setLineId(violationLine);
sensorContext.saveViolation(violation);
```

ESQUF fins the rules in the repository searching by the key that has been assigned to a rule in its XML definition.

5.3. Dynamic analysis

Dynamic code analysis refers to the tests performed at the running code. There are two types of tests: *unit tests* of type pass/fail execution where given some input conditions, the system checks whether the expected output is obtained; or *coverage analysis* in which a set of tests are defined to analyze what percentage of the code is actually executed and what percentage of unreachable statements there are.

5.3.1. Unit tests

Here, it is described the integration of external unit tests specifications and results in ESQUF framework. Unit tests are performed to the code to detect whether specific functions and code chunks provide the expected results given some specified input conditions and values. This type of test is essential to assess the deterministic behavior of a program and system, disclosing any initially unforeseen side effects that may be present in the first versions of the project code.

The important properties of this module are the indication of the location of the input data that contains the analysis results of the external tools used for unit test execution, and the placement of the results files of ESQUF.

```
@Properties({
  @Property(
    key = UnitTestExternalToolPlugin.
        EXTERNAL_TOOL_UNIT_TEST_FILE,
    name = "DB path",
    description = "Path to the results files of the external
        tools",
    global = false)
})
 @Property(
[...]
    key = CoverageExternalToolPlugin.
        EXTERNAL_TOOL_UNIT_TEST_DELETE,
[...] ),
 @Property(
    key = CoverageExternalToolPlugin.
        EXTERNAL_TOOL_UNIT_TEST_PATH,
[...] })
```

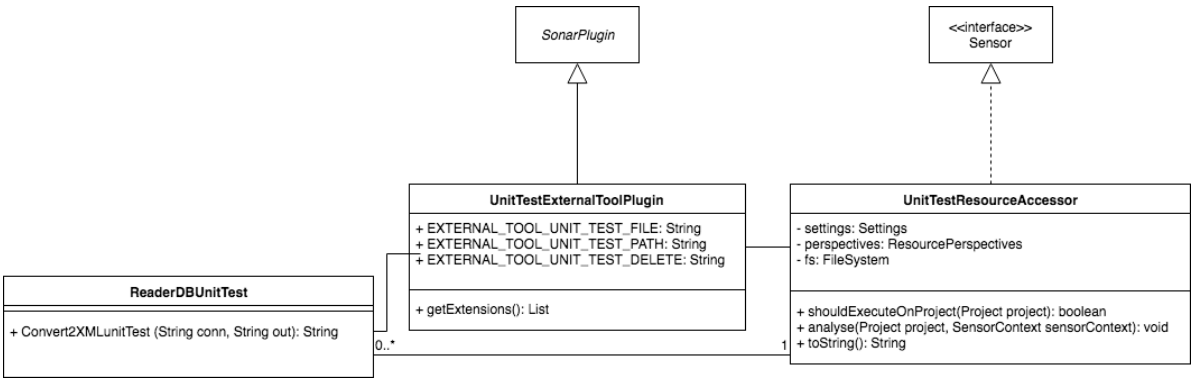

Figure 10: Class diagram of the unit test module/plugin

Figure 10 shows the class diagram for the unit test module that contains the following main classes:

- *UnitTestExternalToolPlugin* that is the main class;
- *UnitTestResourceAccessor* is the class responsible of analyzing both quality data and/or source code files.
- *ReaderDBUnitTest* that interfaces with the data base of the system to collect the data about the specific tests to be run; then, it produces the analysis results of this tests that will be provided for display.

Resource accessor.

UnitTestResourceAccessor imports the data of the unit tests performed by external tools. For its effective and modular realization over a given software quality framework, this class is linked to the specific resource accessing runtime. E.g., for SonarQube, this class implements the *Sensor* interface. This class logic is run by ESQUF workflow engine. Its main methods are:

- *analyse* is the most relevant function of the accessor as it carries out the analysis over the external results files and stores the results. It collects the path to the files to be analyzed generated by the external tool and launches its analysis.
- *shouldExecuteOnProject* that indicates to the workflow engine whether the analysis method (*analyse*) is used in the current project; this indicates that the resource accessor logic should be run.

```
public boolean shouldExecuteOnProject(Project project) {
 String src = settings.getString(UnitTestExternalToolPlugin.
    EXTERNAL_TOOL_ COVERAGE_FILE);
 String path = settings.getString(
    UnitTestVectorExternalToolPlugin.EXTERNALTOOL_UNIT_TEST_PATH
    );
 [...]
}
```

Data storage.

ReaderDBUnitTest collects the information of the test cases provided by the external tools. An example of such a test is the following: if a function that contains a loop of n repetition is invoked with a specific value of variable n, then it is expected that the function will execute the instructions inside the loop exactly n times. After reading the test cases, this class generates portable code with information about the quality results of each unit test that has been run in the system. This results file will be interpreted by the display plugins for the overall dynamic analysis. Test cases are stored in a relational data base that is accessed via queries.

5.3.2. Coverage analysis

This section explains the integration in ESQUF of external coverage analysis from external tools that perform dynamic code analysis. The coverage analysis is done based on three types: (*i*) statement coverage (e.g. an else part is not executed in an if-then-else statement); (*ii*) condition coverage (e.g. a switch statement with a condition code being empty); or (*iii*) condition/decision modification (e.g. check whether in a complex condition that contains a number of `or`, `and`, etc., all the possible cases have been tested.

There are a number of external tools that are used in critical systems for coverage analysis such as [46] and [47] .

The design of the logic that supports the performance of dynamic code analysis is shown below. This functionality is provided by an *coverage analyzer* module that achieves the integration of the local analysis results of ESQUF with the ones from external analysis tools. This module allows new tests to be performed and later added to ESQUF platform for further analysis and presentation.

The architecture of the coverage module is shown in Figure 11 that contains the following clases:

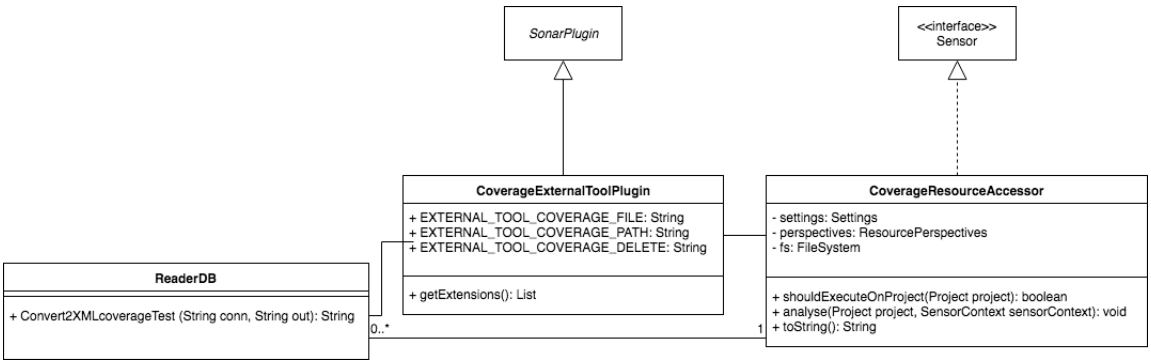

Figure 11: Class diagram of the coverage analysis module/plugin

- *CoverageExternalToolPlugin* that is the main class;
- *CoverageResourceAccessor* that contains resource accessor logic for further data processing; it is invoked by EQUF workflow module when a project analysis is started.
- *ReaderDB* is a class for reading the data contained in the quality results generated by the external tools for coverage analysis. These data are converted into a suitable format that the ESQUF framework can read that is mapped to the *GenericTestCoverage* class that presents the quality results on the web.

The class *CoverageExternalToolPlugin* defines the properties of the plugin together with a *Lis* of other classes that should also be run as part of the analysis logic. The properties defined are: route to the external tool result files; route where to store the files generated by the plugin; decision on whether to eliminate the generated files after the analysis execution.

```
@Properties({
  @Property(
    key = CoverageExternalToolPlugin.EXTERNAL_TOOL_COVERAGE_FILE
        ,
    name = "DB path",
    description = "Path to the results files of the external
        tools",
    global = false)
}),
 @Property(
[...]
```

```
    key = CoverageExternalToolPlugin.
        EXTERNAL_TOOL_COVERAGE_DELETE,
[...]),
 @Property(
    key = CoverageExternalToolPlugin.EXTERNAL_TOOL_COVERAGE_PATH
        ,
[...]})
```

Resource accessor

CoverageSensor manages the access to the analysis results files for external tools, i.e., it imports the coverage results data. This class is run by ESQUF workflow module. Two of its main functions are:

- *analyse* is the most important function of the accessor as it performs further analysis over the external results files and stores the obtained results in ESQUF framework.
- *shouldExecuteOnProject* that receives the project identifier that must be analyzed, and it provides an indication to ESQUF framework of whether further analysis (i.e., *analyse* function) must be run for this project.

```
public boolean shouldExecuteOnProject(Project project) {
 String src = settings.getString(CoverageVectorCASTPlugin.
    VECTORCAST_COVERAGE_FILE);
 String path = settings.getString(CoverageVectorCASTPlugin.
    VECTORCAST_COVERAGE_PATH);
 [...]
}
```

The class *ReaderDB* is also important to the collection of the external tools results and it performs the persistent storage of the final coverage analysis results. The storage of the final results is done in XML files that are later interpreted by the *GenericTestCoverage* plugin of the core quality framework.

6. Validation

The validation criteria for ESQUF focuses on providing a new framework that is capable of offering new capabilities for software testing. This capability is shown by designing and implementing the prototype presented in this section for a given software project. This prototype implementation validates this feasibility of:

1. achiving a rich presentation space for facilitating the monitoring of the verification process of software projects for critical systems;
2. achieving an extensible and evolvable platform by adding analyzer modules that integrate external analysis tools;
3. supporting the integration of different norms and language profiles by extension of the framework through analyzers;
4. the integration of dynamic analysis results together in the same presentation space; and
5. achieving a collaboration space in the verification of software projects over a real critical software project;

6.1. Analysis scenario

The validation criteria is based on the achievement of the foreseen rich presentation space for monitoring the verification of the source code of critical software projects by integrating external tools that perform different specific analysis techniques for static and dynamic analysis for C code under the required norms and standards. This is encompased by the first and second validation criteria. The prototype implementation of ESQUF is used to analyze a real project developed under standards [21] and [23]. As external tools, both [37] and [46] have been used which provide static analysis; this will prove the third point of the validation criteria.

Figure 12 shows the validation scenario for ESQUF, that is a small scale prototype of a real critical systems software project requiring a unified view over the quality results of the code derived from complex metrics analysis of an external tool such as [37] (metrics tool indicated in Figure 12); and also [37] is used to obtain the quality results derived from the analysis of a MISRA C subset for coding rules compliance under DO-178C (Norms/rules tools indicated in Figure 12);and a coverage analysis performed by the tool [46].

[37] has been selected as the external tool to validate the analyser module that integrates different sources of analysis results. [37] is a powerful static analysis tool that offers rich information to the verification engineers with respect to the characteristics of the source code of a given project that are not given by hardly any open source software quality framework. This difference is particularly evident for specific programming languages such as C, as this tool provides analysis for C language whereas there are hardly open analysis functions for any open source analysis tool in C and C++

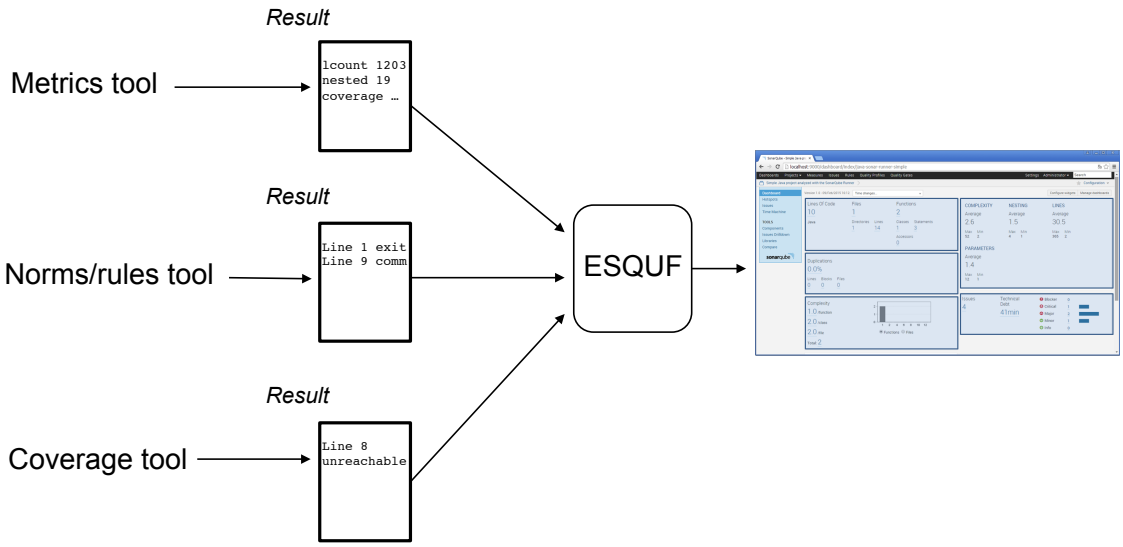

Figure 12: ESQUF validation scenario with required critical systems functions

language. Some of the rich set of metrics provided by this tool for a C project are: max/min/average complexity of a function, number of nested loops, or average number of function parameters, among others.

6.2. Metrics

The first and second validation points are addressed by the development of the integration of the *Metrics* analyzer module. The first step to integrate code analysis for metrics into the ESQUF framework was to develop an extension of the external tools metrics to obtain the quality results files to be fed to ESQUF. The output file of [37] has a structure similar to the following one:

```
Results          Entity  Line    Column  Check
Number of Results: N
Project Violation
Max complexity: (Value)                  "Free text"
Average complexity: (Value)              "Free text"
Min complexity: (Value)                  "Free text"
Max nesting: (Value)                     "Free text"
Max number of lines: (Value)             "Free text"
Average nesting: (Value)                 "Free text"
Min nesting: (Value)                     "Free text"
Min number of lines: (Value)             "Free text"
```

```
Max number of parameters: (Value)        "Free text"
Average lines: (Value)                   "Free text"
Average parameters: (Value)              "Free text"
Min number of parameters: (Value)        "Free text"
```

The above file structure shows all metrics that the integrated analyser module is able to process; the presentation may be done for all or for a subset of all metrics. The text marked as *Free text to include comments* is not analysed by the integrated analyser module. The order in which the metrics appear is (a priori) not important although it is in deed application specific; the name and parenthesis with the metric value are the key data items to be collected and presented by ESQUF.

The class *ExternalToolMetricSensor* of the integrated analyser module is extended to derive the class *UnderstandMetricSensor* that supports the specific characteristics of this specific external metrics tool. Consequently, when the workflow engine of ESQUF is executed, this class reads the required output file from the external metrics tool to derive its analysis metrics and the software quality results.

For the class *MetricsRubyWidget* that defines the properties to customize the widget and the data display, the data for the metrics tool is given:

- *getId()* returns an identifier to the external tool [37] that is *UnderstandMetrics*
- *getTitle()* returns *UnderstandMetrics*.
- *getTemplatePath()* sets the path to where the content file of [37] is.

Figure 13 shows the results presented in the presentation configurator that is designed for presentation of the metrics from the external tool.

Using ESQUF *built-in* metrics, it is possible to see that the project contains 172 source files with 1952 functions. It contains 60309 code lines, including 23579 executable statements. The results of the metrics collected with the use of the developed plugin were the following:

The results reported by the project analysis (see table 3) provide richer information to the user that, in this case, is a verification engineer. In this specific project validation example, it is suggested that, for example, although the average complexity of the source code is not too high, there is at least one function whose complexity is to high and needs to be reviewed. The rest results can be used to evaluate if the project source code characteristics are within the limits specified by the project standards.

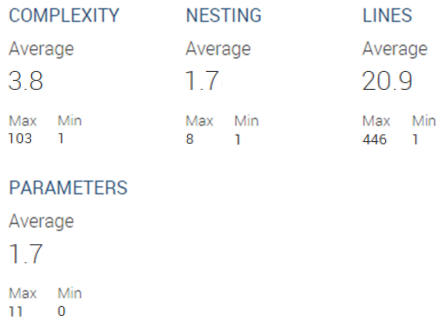

Figure 13: Display of the integrated analysis results from the metrics external tool

Table 3: Metrics results

Complexity	
Average	3,8
Minimum	1
Maximum	103
Nesting	
Average	1,7
Minimum	1
Maximum	8
Lines	
Average	20,9
Minimum	1
Maximum	446
Input parameters	
Average	1,7
Minimum	0
Maximum	11

6.3. Rules and norms compliance

The third validation criteria is addressed in this subsection. In the following example, a specific results file for MISRA C rules analysis is shown.

```
Results          Entity   Line     Column   Check
Number of Results: N
main.c
```

```
comments are not permitted        var  2   4  no comments
exit statement is not permitted   ex   14  3  example
example.c
comments are not permitted    vars 8   1   no comments
```

The class *ExternalToolRulesPlugin* of the integrated analyser module is extended to derive the class *CRuleRepository* that maps to the characteristics of the required rules that have to be applied. Then, the workflow engine of ESQUF will invoke this class that, in turn, reads the required output file from the external rules analysis tool; as a result, software quality results on the fulfillment of the specific rules that apply to the norms profile that has been selected (e.g., MISRA C) is obtained and visualized in ESQUF unified presentation display.

Figure 13 shows the results presented in the presentation configurator that is designed for presentation of the metrics from the external tool.

Figures 14 and 15 show the quality analysis results displayed by ESQUF after the analysis of coding rules compliance. Figure 14 shows the source code display from ESQUF web interface, with the corresponding indications about rule compliance error; source code presentation helps engineers in rapid visualization of problems that will be reported to the programming team. In Figure 15, the presentation configurator is provided showing the collected data results for the rule compliance; the number of violations of rules in the source code is presented with a classification into violations that are blocker, critical, major, minor, or irrelevant (info). This classification helps verification engineers to classify the requests made to the programming team.

Figure 14: Example of presentation of source code with non compliance

The framework supports to easily include different coding profiles, norms, and programming languages by following the structure presented in section 5.2 in which, for example, more rules can be added to the MISRA C profile or even other language profiles can be applied instead.

6.4. Dynamic analysis

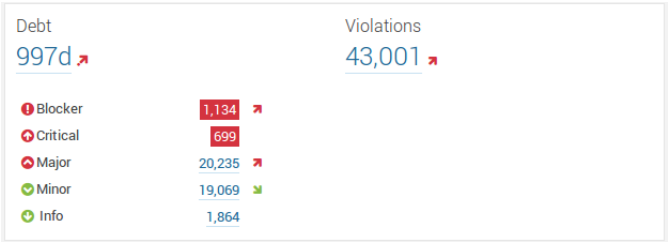

Figure 15: ESQUF presentation of quality results with respect to norms compliance

The fourth validation criteria is addressed in this subsection as coverage and unit test analysis are performed by an external coverage tool [46] that provides rich information with respect to a number of tests to be done to the source code.

The class *CoverageExternalToolPlugin* of the integrated analyser module is extended to derive the class *CoverageVectorCASTPlugin* that supports the characteristics of the external coverage tool that is integrated for this critical software project analysis. In addition, the class *CoverageSensor* that maps precisely to the coverage results obtained from the external tool and read through the *ReadDB* class. Similarly, *UnitTestExternalToolPlugin* of the integrated analyser module is extended to derive the class *UnitTestVectorCASTPlugin* that supports the characteristics of the external coverage tool that is integrated for this critical software project analysis. In addition, the class *CoverageSensor* that maps precisely to the coverage results obtained from the external tool and read through the *ReadDB* class.

A set of unit tests are designed on a per-project basis. The framework should present the coverage of the executed unit tests, the percentage of lines of code that the tests have covered, and the percentage of conditions that have been evaluated. Figures 16 and 17 present the results of the coverage analysis in a broader presentation scope.

Figure 16 shows the view of the framework that is shown to all members of the verification team that are the users of it. This proves the validation of the collaborative property, as the users can simultaneously access the information of a given project that will be stored in the ESQUF server, that allows them to monitor the progress of the verification of a given software project to later ellaborate and order the needed changes in given source code files to the development team.

6.5. Discussion

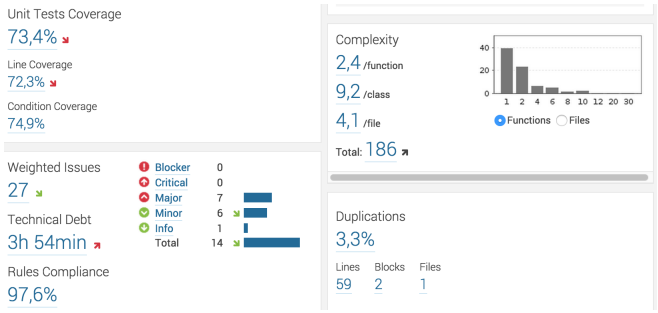

Figure 16: ESQUF presentation of coverage analysis

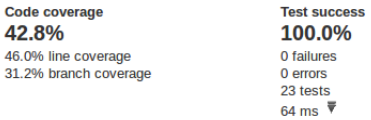

Figure 17: Further coverage information presentation

The prototype implementation described in this section targets the validation of ESQUF. Table 4 summarizes the validation parameters of ESQUF framework.

Table 4: Validation summary

Validation parameter	**Present**	**Description**
Metrics presentation	✓	Function count, complexity, nesting, etc.
Multiple external tools	✓	Understand, VectorCast
Single presentation space	✓	Results presentation through widgets
Collaborative space	✓	Web interface remotely accessible
Coding norms and standards	✓	DO178C, AQAP2210

The objective of ESQUF has been to provide an extensible framework that brings in new characteristics to those tools and frameworks that support code testing. The main contribution of ESQUF has been to provide all these parameters (as shown in Table 4) all together. None of the tools in the market and literature provide these in an integrated manner:

- Metrics presentation; a number of metrics are presented that encompas those that indicate the complexity of source files and software projects, nesting levels, function count, line count, block count, duplications, etc.

- Multiple external tools integration; in this prototype implemenation, it has been shown how Understand and VectorCast (two commercial tools) are integrated into the framework in a way that their analysis results are displayed jointly for a same software project.

  Single presentation space; widgets are used to present external tools' results in customized formats.

- Collaborative space; a web interface is provided through a web server that can be used to share the same view of the engineering team over the projects.

- Coding norms and standards; in the prototype, the software project is analyzed against norms DO178C and AQAP2210.

7. Conclusions

The paper has presented the design and implementation of a modular integrated software quality analysis framework specifically targeted at critical software projects that have especial requirements. Firstly, critical systems have extensive C language usage, whereas the majority of existing software quality analysis frameworks are focusing solely on Java code. Secondly, critical systems development require to use external analysis tools for specific parts of the code; then they require to integrate them in a single presentation framework. Additionally, critical software projects are made by large teams of engineers in very different tasks ranging from programming to the monitoring of the verification and testing processes. Therefore, this requires to achieve a single collaborative working space that supports the remote access of the engineers and the interaction of the verification teams working on different projects. Lastly, different parts of a software systems have to comply with different development standards and norms. Therefore, it should be easy to incorporate different coding profiles, norms, and programming languages into the analysis framework.

This paper has presented the design and prototyping of ESQUF that is a software quality framework that has the above mentioned characteristics.

The inclusion of static and dynamic software quality results helps in manageing the technical quality information of projects; stakeholders can review the related information without the need of individually checking the results reported by different tools. In this case, ESQUF provides a single presentation framework for static and dynamic analysis that include metrics, rule compliance for different coding profiles, and coverage analysis. ESQUF has showed that it can be easily particularized for specific external analysis tools, collecting their information and displaying it in a single space.

This situation leads to a more efficient project management and clarity of the information. This work supports the easy customization of specific critical software projects in order to comply with the norms that are mandatory in their domain.

This work has analysed the use of the metrics collection and the analyser module within a continuous integration development practice, as a way to improve software quality and, therefore, reduce the risk in the software projects.

ESQUF is applicable to all projects with the need of technical quality data collection where external tools are mandatory to collect some information not provided initially by the used quality management plattform; it has been prototyped and validated in a real project that requires compliance with norms related to the development of critical software (like DO-178C [21] and software quality, like AQAP-2210 [23].

References

[1] Havva Gulay Gurbuz, Bedir Tekinerdogan. *Model-based testing for software safety: a systematic mapping study.* Software Quality Journal. Springer. DOI: 10.1007/s11219-017-9386-2 2017.

[2] Joseph P. Cavano and James A. McCall. *A framework for the measurement of software quality.* In Proceedings of the ACM Software quality assurance workshop on Functional and performance issues. DOI 10.1145/800283.811113 1978.

[3] D. Coleman, D. Ash, B. Lowther, P. Oman. *Using metrics to evaluate software system maintainability.* IEEE Computer, vol. 27(8), pp. 44-49. August 2002.

[4] CMMI Product Team. *CMMI for Development, version 1.3. Improving processes for developing better products and services* CMU/SEI-2010-TR-033, ESC-TR-2010-033. 2010.

[5] N. Fenton, J. Bieman. *Software metrics: a rigorous and practical approach*. CRC Press, 2014.

[6] V. Balachandran. *Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation*. Proc. of International Conference on Software Engineering (ICSE). 2013.

[7] D. di Ruscio, P. Pelliccione. *A model-driven approach to detect faults in FOSS systems*. Journal of Software: Evolution and Process, vol. 27(4), pp. 294–318. April 2015.

[8] G. A. Campbell, P. P. Papapetrou. *SonarQube in Action*. Manning Publications. ISBN-9781617290954. 2013.

[9] SonarQube Documentation. `http://docs.codehaus.org/display/SONAR/Documentation` (2015)

[10] –. *SonarQube v6.* `http://www.sonarqube.org/` [Last retrieved 2017]

[11] P. Krutchen. *Contextualizing agile software development.* Journal of Software: Evolution and Process, vol. 25, pp. 351-361. 2013.

[12] Georgios Gousios, Diomidis Spinellis. *Alitheia Core: An extensible software quality monitoring platform*. In Proc. of the 31st International Conference on Software Engineering (ICSE '09). IEEE Computer Society, Washington, DC, USA, 579-582. DOI: 10.1109/ICSE.2009.5070560 2009.

[13] P. M. Duvall, S. Matyas, A. Glover. *Continuous integration: improving software quality and reducing risk* Pearson Education. ISBN 13: 978-0-321-33638-5. 2007.

[14] –. *Valgrind. Code analysis tool.* `valgrind.org` [Last retrieved 2018]

[15] Rapita Systems. *RapiTime user guide*. `www.rapitasystems.com` . 2017.

[16] J. García-Munoz, M. García-Valls, J. Escribano-Barreno. *Improved Metrics Handling in SonarQube for Software Quality Monitoring*. 13^{th} Int'l Conference on Distributed Computing and Artificial Intelligence (DCAI). In Advances in Intelligent Systems and Computing, vol. 474, pp. 463–470. Springer. June 2016.

[17] M. García-Valls, D. Perez-Palacin, R. Mirandola. *Pragmatic cyber physical systems design based on parametric models*. Journal of Systems and Software, vol. 144, pp.559–572. Elsevier. October 2018.

[18] M. M. Bersani, M. García-Valls. *Online verification in cyber-physical systems: Practical bounds for meaningful temporal costs*. Journal of Software: Evolution and Process, vol. 30(3). Wiley. March 2018.

[19] M. García-Valls, A. Dubey, V. Botti. *Introducing the new paradigm of Social Dispersed Computing: Applications, Technologies and Challenges*. Journal of Systems Architecture. DOI: 10.1016/j.sysarc.2018.05.007 . 2019.

[20] Stephen H. Kan. *Metrics and Models in Software Quality Engineering* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 2002.

[21] RTCA Inc. *Software Considerations in Airborne Systems and Equipment Certification*. RTCA Inc. DO-178C. 12/13/2011.

[22] RTCA Inc. / EUROCAE. *Software Integrity Assurance Considerations for Communication, Navigation, Surveillance and Air Traffic Management (CNS/ATM) Systems*. DO-278A. 12/13/2011.

[23] AQAP 2210. *NATO Supplementary Software Quality Assurance Requirements to AQAP 2110* 1st edition, November 2006.

[24] AQAP 2110 *NATO Quality Assurance Requirements for Design, Development and Production* 2nd edition, November 2006.

[25] PECAL-2210 *Requisitos OTAN de aseguramiento de la Calidad del software, suplementarios a la PECAL 2110* First Edition, November 2007.

[26] PECAL-2110 *Requisitos OTAN de aseguramiento de la Calidad para el diseño, el desarrollo y la producción* Second Edition, November 2006.

[27] IEC 61508. *Functional safety of electrical/electronic/programmable electronic safety-related systems*. April 2010.

[28] IEC. *Nuclear power plants. Instrumentation and control important to safety. General requirements for systems*. IEC 61513 Ed.2.0. 25/08/2011.

[29] RTCA Inc. *DO-178B. Software Considerations in Airborne Systems and Equipment Certification*. RTCA Inc. DO-178B. 1992.

[30] RTCA Inc. *Software Tool Qualification Considerations*. DO-330. 12/13/2011.

[31] RTCA Inc. *Model-Based Development and Verification Supplement to DO-178C and DO-278A*. DO-331. 12/13/2011.

[32] RTCA Inc. *Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A*. DO-332. 12/13/2011.

[33] RTCA Inc. *Formal Methods Supplement to DO-178C and DO-278A*

[34] CENELEC. *Railway applications - Communications, signalling and processing systems*. CENELEC. 2001.

[35] ISO *Road Vehicles - Functional Safety*. ISO-26262.11/11/2011.

[36] IEC *Medical Device Software*- IEC, May 2006.

[37] *Scitools Understand* Information available at: https://scitools.com/. [Last retrieved: 19/02/2015]

[38] *LDRA* Information available at: http://http://www.ldra.com/. [Last retrieved: 19/02/2015]

[39] *PC-Lint* Information available at: http://www.gimpel.com/html/index.htm. [Last retrieved: 19/02/2015]

[40] *Splint* Information available at: http://www.splint.org/. [Last retrieved: 19/02/2015]

[41] *PMD* Information available at: http://pmd.sourceforge.net/. [Last retrieved: 19/02/2015]

[42] *Maven* Information available at: http://maven.apache.org/. [Last retrieved: 19/02/2015]

[43] *Jenkins* Information available at: http://jenkins-ci.org/ [Last retrieved: 19/02/2015]

[44] Misra C http://www.misra.org.uk/forum/viewforum.php?f=179 (2015)

[45] *Misra C++.* http://www.misra.org.uk/forum/viewforum.php?f=184 (2015)

[46] *Vector Cast.* http://www.vectorcast.com/

[47] *Cantata ++.* http://www.qa-systems.com/cantata.html

[48] David Delmas, Jean Souyris. *Astrée: From Research to Industry.* Proc. of International Static Analysis Symposium (SAS), pp. 437–451. Lecture Notes in Computer Science, vol. 4634. 2007.

[49] Robert Seacord. *Secure coding in C and C++.* SEI Series in software Engineering, 2*nd* ed. Addison-Wesley Professional. 2013.

[50] CEA List, INRIA Saclay. *Frama-C.* http://frama-c.com (Last accessed June 2018)

[51] Alain Deutsch. *Static verification of dynamic properties.* Polyspace Technologies. 2014.

[52] John W. McCormick, Peter C. Chapin. *Building High Integrity Applications with SPARK 2014.* Cambridge University Press. 2015.