Investigating Automatic Static Analysis Results to Identify Quality Problems: an Inductive Study

Investigating Automatic Static Analysis Results to Identify Quality Problems: an Inductive Study

Antonio Vetro'[1,2], Nico Zazworka[1], Forrest Shull[1], Carolyn Seaman[1,3], Michele A. Shaw[1]

[1]Fraunhofer CESE
College Park, MD, USA
nzazworka@fc-md.umd.edu
fshull@fc-md.umd.edu
mshaw@fc-md.umd.edu

[2]Automatics and Informatics Dept.
Politecnico di Torino, Torino, Italy
antonio.vetro@polito.it

[3]UMBC
Department of Information Systems
Baltimore, MD, USA
cseaman@umbc.edu

ABSTRACT

Background: Automatic static analysis (ASA) tools examine source code to discover "issues", i.e. code patterns that are symptoms of bad programming practices and that can lead to defective behavior. Studies in the literature have shown that these tools find defects earlier than other verification activities, but they produce a substantial number of false positive warnings. For this reason, an alternative approach is to use the set of ASA issues to identify defect prone files and components rather than focusing on the individual issues.

Aim: We conducted an exploratory study to investigate whether ASA issues can be used as early indicators of faulty files and components and, for the first time, whether they point to a decay of specific software quality attributes, such as maintainability or functionality. Our aim is to understand the critical parameters and feasibility of such an approach to feed into future research on more specific quality and defect prediction models.

Method: We analyzed an industrial C# web application using the Resharper ASA tool and explored if significant correlations exist in such a data set.

Results: We found promising results when predicting defect-prone files. A set of specific Resharper categories are better indicators of faulty files than common software metrics or the collection of issues of all issue categories, and these categories correlate to different software quality attributes.

Conclusions: Our advice for future research is to perform analysis on file rather component level and to evaluate the generalizability of categories. We also recommend using larger datasets as we learned that data sparseness can lead to challenges in the proposed analysis process.

Categories and Subject Descriptors

D.2.8 [**Metrics**]: Product , D.2.0 [**General**]: Standards, D.2.4 [**Software/Program Verification**]: Statistical Methods

General Terms

Measurement, Experimentation, Verification.

Keywords

Automatic static analysis, software quality, defect prediction.

1. INTRODUCTION

Automatic program analysis is the process of extracting information about a software program from its source or artifacts (e.g., from byte or object code, or execution traces) using automatic tools [2]. Program analysis can be static (i.e. without executing the program) or dynamic (i.e. with executing the program): our work is focused on static analysis.

Automatic Static Analysis (ASA) tools analyze the source code or intermediate code (e.g. byte code) to determine defect patterns and violations of good programming practices, naming conventions, security flaws and coding standards. Violations are called "issues" and could cause defective behavior of the software system. ASA tools are able to evaluate code from early stages in development onward, and do not require a running version of the program. Also, contrary to popular verification techniques such as unit and system tests, ASA tools do not necessitate the specification of a test oracle. Since ASA tools are applicable while developers write code (i.e. they operate in close to real-time), their usage suggests a benefit in terms of identifying problems as early as possible when compared to other verification activities such as testing. As a consequence, given that the time between a fault insertion and its removal correlates with the cost of removing that defect [3], the introduction of ASA tools in the development phase could lead to important economic benefits.

Even if the usage of ASA tools promises benefits and high return on investment, the currently available tools and algorithms often have limitations when applied in practice. The most important and well-studied limitation is the large number of false positives returned when ASA is used to identify defects that would lead to faults in software applications. On realistic-sized applications ASA tools typically generate thousands of issues, and so the output needs further refinement and tailoring from developers to be useful. One of the main questions is how to prioritize the long list of issues in order to find the most important defects as soon as possible. This can typically be done in two different ways. First, one can try to understand which ASA issues are real indicators of defects that lead to faults. This assumes that for each observed fault the related defect is actually signaled by an ASA issue, which is not always the case.

Secondly, one can use the large set of ASA issues to understand the coexistence of issues and real defects in the same source code file, or software component. This broader approach will not assume a cause-effect relationship between ASA issues and software faults, and could therefore capture cases where, for example, sloppy programming or Technical Debt [7] leads to defects and ASA issues at the same time. However it is also less specific in guiding the developer to the specific location of the defect (i.e. the line of code).

Beyond simply predicting the occurrence of defects and faults, it is often also of interest to study how defects affect common software quality attributes, such as maintainability, functionality, usability, security, etc. For example, a security defect leading to an intrusion can be more expensive for an organization than a usability defect, and vice versa. What has not yet been studied is if ASA issues are able to effectively predict a defect (or a defect prone component) and to specify at the same time which software quality attribute this defect will affect.

This exploratory study uses the second approach of using ASA issues to identify defect prone files and components and takes a first leap towards understanding the feasibility of identifying more specific quality problems. The study is inductive in its character and the main aim is to understand key parameters for future model building and to generate a set of hypotheses and recommendations for future research.

2. RELATED WORK

The effort of the research community has focused on evaluating ASA tools in two main streams: I) looking at single ASA issues to identify defects in single lines of code or II) looking at large sets of issues as early indicators of the more defect-prone modules (e.g. classes, files, software components).

2.1 First research stream: looking at single ASA issues to find defects

Several studies in the literature have reported on the percentage of false positive ASA issues (i.e. issues not related to defects) of different tools and in different contexts. For instance, Wagner et al. [23] analyzed and classified with experienced developers issues from three ASA tools (FindBugs, QJPro and PMD) on four industrial projects and one university project, and they reported that the percentage of false positive was 47% for FindBugs, 31% for PMD and 96% for QJ Pro.

Weydan et al. [24] reported that more than 96% of FindBugs and IntelliJ issues did not relate to any fault or refactoring in two open source systems (jEdit and iText).

Similar findings were reported by Vetro' et al. [21] [20] applying the FindBugs tool to students' Java projects.

Lower percentages of false positives are reported by Ayewah et al. [1] running FindBugs on the JDK 1.6.0-b105; the authors report that almost 50% of medium/high priority issues related to correctness had impact on the functionality, and 10% had a serious impact. On the flip side, 160 issues out of 379 were trivial (i.e., no impact), while 5 issues were due to faulty analysis of FindBugs. A similar experiment with the same category of issues was performed at Google, with similar percentages of false positive issues, and a further validation conducted on Glassfish v2 showed an even better result: 50 defects out of 58 disappeared due to changes made to specifically address the issues raised by FindBugs.

Switching to the C languages, Boogerd and Moonen [4] [5] analyzed four industrial projects in C and C++ with an ASA tool for the MISRA standard [15], and they discovered that a small set of rule violations (12 out of 72 in [4] , and 10 out of 88 in [5]) were related to defects in source code. Finally, Nagappan et al. [16] reported that only 12.5% of defects fixed in Windows Server 2003 pre-release were found with two ASA tools (PREfix and PREfast). Precision was not reported in this study, so this figure is not directly comparable to the previously reported results.

Overall, except for one study [1], we conclude that the precision of the ASA tools is rather low, because high ratios of false positives (i.e. low precision) were reported in many studies.

2.2 Second research stream: using ASA issues to predict modules with more defects

The second approach is to investigate whether static analysis issues can be used as early predictors for the most defect prone modules in software systems, rather than identify the single issues that point to specific defects.

Nagappan et al. [16] discovered positive correlations (0.37 and 0.58) between issue densities from two ASA tools, PREfix and PREfast, and the pre-release defect density. Moreover, they successfully used the ASA issue densities to discriminate between components of high and low quality.

A similar approach was used by the same author in a study carried out at Nortel Networks [18], where automatic inspection defects found by ASA tools had a positive correlation with failures (0.40 and 0.49). Moreover, together with code churn, ASA issues were good discriminators in identifying fault-prone modules. The study at Nortel continued and one year later Zheng et al. [26] reported even higher correlations between the number of ASA issues in files and three different indicators of external quality, i.e. number of tests failures, number of customer reported failures and number of total failures (respectively 0.71, 0.60 and 0.73).

Other authors used a similar approach. For instance, Plosch et al. [19] studied the correlation between the number of FindBugs and PMD issues, and defects in Eclipse SDK 2.0, 2.1 and 3.0. They found positive correlations for both tools (0.34, 0.25 and 0.30 for PMD, and 0.20, 0.08, 0.20 for FindBugs). Excluding the LOC related metrics, PMD issues correlated better with defects than other static metrics (e.g., number of methods, number of fields, etc.)

Finally, Marchenko and Abrahamsson [14] used two tools, namely CodeScanner and PC-LINT, to analyze five projects in the Symbian C++ environment. They computed the correlation between issues and critical defects in two snapshots of the project (i.e. within 90 days after the release and within 180 days after the release) and they observed contradictory results: CodeScanner obtained very high positive correlations (0.70 and 0.90), while PC-LINT issues strongly correlated negatively (-0.90 and -0.70) with defects.

Overall, all the current results available in the literature but one [14] show that using ASA issues to find the most defect-prone files or modules is more effective than using individual ASA issues to discover individual defects.

2.3 Contributions of this study

We are helping an industrial partner in understanding the usefulness and effectiveness of the Resharper ASA tool in their development projects and we decided to adopt the second approach because it is more promising than the first one, as

summarized in the previous sections. Our long term aim is to provide our partner with models that use ASA issues to point to more specific quality problems. These models should be able to make recommendations for code inspections based on a set of quality characteristics of interest. For example, a security inspection should be able to use a prediction model pointing to software files and components with potential security flaws. Or, the user experience review should be able to use a model that selects parts of the software with potential usability problems.

The main novelties that we introduce with respect to the previously conducted related work are:

- We contribute to the body of evidence of the second research stream by adding a new tool/language/application combination (Resharper/ C#/ Web application). The Resharper tool has, to our knowledge, not yet been evaluated in past work.
- We perform the analysis at two granularity levels, i.e. software components and source code files. Components are high-level functional units encapsulating one or more main functionalities of the software system, such as: "User Login", "Database Access", or "Admin Backend". Source code files are low-level artifacts, usually containing classes that are the building blocks for components. Since past studies were done at only one of the two levels this study will give some insight into the comparison between the two levels.
- We investigate whether specific types of ASA issues can be linked to specific quality dimensions. This is helpful to understand if an increased importance of one quality dimension, such as usability, can help to pre-select the set of ASA issue types that will predict usability defects with the highest precision. Or more generally, the approach can be used to prioritize the set of ASA issues a reviewer would have to inspect, based on a prioritization of desired quality characteristics.

To our knowledge, no past work has yet studied the correlation between ASA issue types and quality characteristics. The most similar works we found were two studies that investigated instead the typology of defects found by ASA tools. The first one is a study conducted by Nagappan et al. [18], who classified defects found by the FlexeLint tool using the ODC classification schema [6] and found that defects associated with ASA fell into three ODC defect types: checking, assignment/initialization, and interface. Wagner et al. [23] also classified ASA issues, but they focused on their effect on code rather than their causes. The authors used a 5-point scale of severity to classify the true positive issues signaled by FindBugs, PMD and QJPro on five industrial projects. The highest category level was "Defects that lead to a crash of the application", while the lowest was "Defects that reduce the maintainability of the code". The authors found that most of the true positives were related to maintainability of the code (e.g., readability and changeability). They also compared ASA issues with defects found using code reviews and unit tests, and they discovered that all defects found by ASA tools were also found by the review, while testing activities found different categories of defects.

We adopt a different perspective from these two studies, and we focus on whether any ASA issues can predict defects relating to a very general set of software quality attributes, using the well-known ISO/IEC 9126 quality model [9] as a basis for defect classification. The ISO/IEC 9126 Software engineering Product Quality Model is an international standard for the evaluation of software quality. It defines a quality model with six main characteristics namely, functionality (F), reliability (R), usability (U), efficiency (E), maintainability (M), and portability (P), which are further broken down into 22 sub-characteristics. The standard was revised in March 2011 by the ISO/IEC 25010 standard committee [10]. Our defect classification based on the standard was created two months after the new standard was released, but we decided to keep the old standard because of its widespread use and because of the large overlap between the two.

We proposed our defect classification in a previous work [22]; it is complementary to already existing defect classifications because it helps in understanding the impact of the software on different quality attributes. Such classification might help programmers and managers with practical tasks, such as the prioritization of defects according to the different stakeholders' interests, the ease of process improvement measurement on specific quality dimensions or tuning verification activities according to specific quality dimensions. This work is specific to the latter point and it is a first step towards understanding whether different ASA issues could be related to specific quality dimensions.

3. GOAL AND STUDY DESIGN

The first goal of this study is to understand whether some predefined subsets of ASA issues (a.k.a. ASA issue categories[1]) are eligible as indicators of defect-proneness. The second goal is to understand whether and which categories of ASA issues are related to specific software quality dimensions. Both questions are analyzed at two levels of granularity: firstly with respect to components, and secondly source code files. The rationale behind the decision to perform analysis on different levels is to better comprehend if results would differ, be the same, or even contradict each other.

3.1 Study Context

The study was carried out at a software company that develops web-based applications in C# (using .NET and Visual Studio). The company uses the JIRA tracking system[2] to record defects.

Of the current projects at this company, we selected one for in-depth analysis based on data quality. Preliminary analysis showed that data quality varied considerably between available projects, reflecting the level of process conformance [25] with which developers recorded defects in JIRA. We chose the project with the best data quality (according to the three criteria below) to reduce the influence of incomplete or noisy data on the results:

A. Number of empty fields in defect reports (e.g. missing data).

[1] Issue categories vary depending on the ASA tool used. Typical categories for the Resharper tool used in this study are: Redundancies in Code, Common Practices and Code Improvements, Compiler Warnings, etc.

[2] http://www.atlassian.com/software/jira

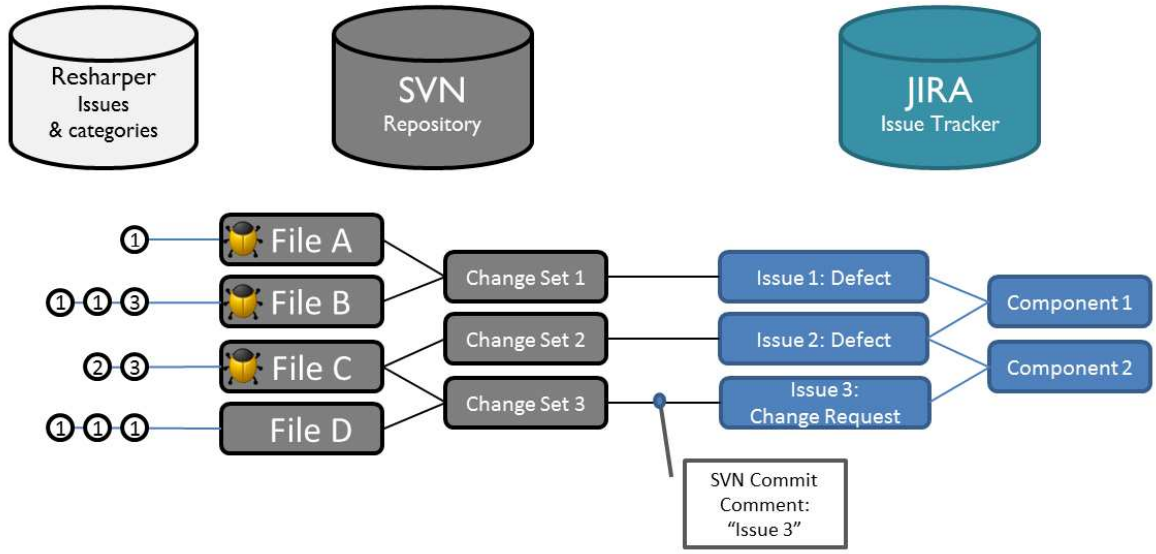

Figure 1: Linkage between Resharper issues, source code files, issue and defect fixes, and components. Yellow defects indicate that a file is linked to at least one defect issue in JIRA.

B. Number of defect report fields that were filled with the default value (which may indicate the default value was accepted rather than that the true value was investigated).

C. Percentage of components that could be bound to files (our approach for this is described below).

The selected application has about 35 KLocs and has been active in production since November 2009, with 4 developers working on it in parallel. At the time of the analysis, the JIRA system contained 78 fixed and closed[3] defects for the selected project (which we will call J).

3.2 Mapping between ASA issues, Defects, Files, and Components

Our methodology for performing the mapping between components, files, and ASA issues, as illustrated in Figure 1, is based upon the fact that JIRA systems can track not only defects but any other element that can be associated with software artifacts. Those elements are called "JIRA issues", and each project has its own set of issues. Example of JIRA issues are change requests, system incident reports, implementation tasks, etc. Moreover, developers establish links between files in the SVN code repository to JIRA issues by including ticket ids in their SVN commit comments. Finally, each JIRA issue is linked by the software developers to one or more software components.

With this information one can build a frequency table (see Figure 2) of files (rows) and components (cells) indicating how often files were changed (i.e. added, modified, or deleted) when working on a component. If a JIRA issue is related to one or more logical components, then the set of modified files belong to the respective components. Using this method a mapping is built based on evidence of how the system changed and evolved over time.

Since a file can belong to many logical components, we accept multiple classifications. Further we reduce some possible noise by mapping a file only to a component if it was linked to this component in at least 20% of all the files' changes. This percentage was set after an analysis of frequency distributions.

3.3 Study Execution

We derive from our first goal two research questions on component (C) and file (F) level:

RQ C1: Which ASA issue categories can identify defect-prone components?

RQ F1: Which ASA issue categories can identify defect-prone files?

	Component 1	Component 2
File A	1	0
File B	1	0
File C	1	2
File D	0	1

	Component 1	Component 2
File A	X	
File B	X	
File C	X	X
File D		X

Figure 2: Evidence-based binding of files to logical components

[3] JIRA defects with resolution "fixed" and status "closed" are the types of defects that were reported, found to be reproducible defects, fixed in the implementation, and validated as repairing the fault. Defects that were not considered in this analysis are, for example, "open and not yet fixed defects", "defects that were duplicates of other reported defects", "defects that could not be reproduced", and "defects that were fixed but not yet validated to solve the fault"

Additional research questions are derived from our second goal:

RQ C2: Which ASA issue categories can point to defect-prone components that impact various system quality characteristics?

RQ F2: Which ASA issue categories can point to defect-prone files that impact various system quality characteristics?

We address these questions inductively, investigating whether the detection of defect-proneness was possible and if so, which types of ASA issues were useful for doing so. We discuss the metrics and the methodology separately for each research question below.

RQ C1: Which ASA issue categories can identify defect-prone components?

To answer RQ1-C1, we first performed the mapping as described in sub-section 3.2 to link Resharper issues to components. Secondly, we checked to see if the number of Resharper issues is correlated with software size. This step was necessary to investigate a possible bias from code size. If such a correlation exists, it is necessary to normalize the data (e.g. by using issue density instead of number of issues). The same analysis is done for defects.

In a third step we test for correlations between numbers of defects and numbers of Resharper issues in each Resharper category, per component. We use the Spearman coefficient correlation (a non-parametric statistic), since we observe a wide range of issues and defects that do not appear to follow any defined distribution (see Tables I and II).

RQ F1: Which ASA issue categories can identify defect-prone files?

To answer this research question we used again the mapping procedure from sub-section 3.2. We also checked for possible bias as described in RQ-C1. Lastly, we tested for correlation between Resharper issue categories and defects by using a two sample Mann-Whitney test [23] after running an unsuccessful Shapiro test for normality. This type of test was more appropriate than the Spearman correlation due the sparseness of the data; it has also been used in previous studies [5] [21]. As the results will show, only a small number of files (about 10%) were associated with defects. Therefore, we partitioned the sample into *non-defect-prone files* and *defect-prone files* in order to perform the Mann-Whitney test. This decision implies that the analysis will investigate if files with at least one defect can be identified by the Resharper issues residing in the same file.

RQ C2: Which ASA issue categories can point to defect-prone components that impact various system quality characteristics?

RQ F2: Which ASA issue categories can point to defect-prone files that impact various system quality characteristics?

For both of these research questions, we used the ISO/IEC 9126 quality model as a basis for classifying the defects according to different quality characteristics. The method for classifying defects in this way was developed and validated in a prior experiment [22], which also used the same project as the subject project. In that study, six different subjects, divided into two groups with respect to their expertise, classified the 78 defects using the ISO/IEC 9126 quality main characteristics and sub-characteristics. Subjects read the defect reports and assigned

TABLE I. RESHARPER ISSUES DETECTED

Resharper category	Number of issues
ASP.NET	2
Common Practices and Code Improvements	521
Compiler Warnings	36
Constraints Violations	445
Language Usage Opportunities	591
Potential Code Quality Issues	14
Redundancies in Code	645
Redundancies in Symbol Declarations	82
Unused Symbols	7
Sum of issues	2343

TABLE II. RESHARPER ISSUES ON COMPONENTS

Component	Sum of ReSharper issues	Defects	NCSS
Cmp 1	1407	43	3192
Cmp 2	324	13	961
Cmp 3	232	6	711
Cmp 4	29	5	97
Cmp 5	7	4	9
Cmp 6	29	4	97
Cmp 7	0	3	0
Cmp 8	119	2	246
Cmp 9	93	1	208
Cmp 10	0	0	0
Cmp 11	428	0	1392
Cmp 12	0	0	0
Cmp 13	0	0	147
Cmp 14	0	0	0
Cmp 15	0	0	0

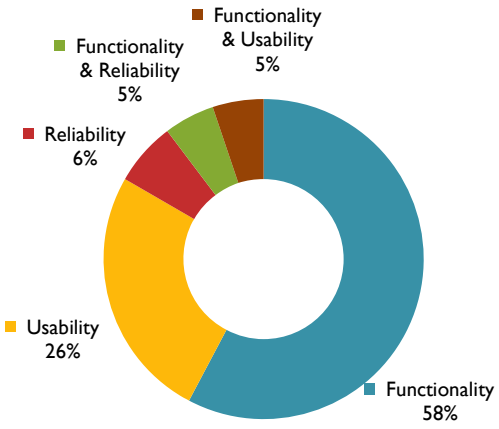

Figure 3: Defect Classification

each defect to one or more quality characteristics and sub-characteristics (the classification is not orthogonal). The underlying idea is that each defect reduces a software capability and impacts the corresponding characteristic and sub-characteristic.

We observed that more experienced software engineers produced classifications with less variability, and that the classification at characteristic level was more reliable than those at sub-characteristics level. As a consequence, we adopted as the final classification the one created by experts at the characteristics level.

Using that classification we were then able to check, in the work described in this paper, whether various types of Resharper issues are correlated to the defects related to specific quality characteristics.

4. RESULTS

We collected metrics on the revision of the target project preceding the first defect fix commit to include as many defects as possible. Resharper reported 2343 issues on the source code of the web application: Table I reports the issues per each Resharper category and Table II reports, for each logical component, total number of Resharper issues, number of defects and non-commented source statements. Some components have 0 NCSS for two reasons: a component was built after the version of the software analyzed, or the files-component mapping produced zero files for a component, or in some cases both. Resharper reported issues on files with extension .aspx, .xaml, .csproj, .cs (including .xaml.cs, .ascx.cs, .aspx.cs, .ashx.cs, .Master.cs). .

Among the 78 fixed and closed defects, 65 had commits linked to them. According to the experts' classification [22] (Figure 3), the majority of defects (58%) impacted only functionality, followed by usability (26%) and reliability (6%). Mixed classifications (FR and FU) accounted for 5% each, while no defects had impact in the remaining three categories.

The total number of files with at least one defect fix is 58. However, excluding those files that were out of scope of the Resharper analysis (e.g., .sql files, .css files) and those files that were added after the revision we analyzed, only 11 of the 58 remained. These files are listed in Table III. As with components, the data indicates that there is not a clear relationship between number of defects and Resharper issues: the most defect prone file (C) has 35 issues whereas some of the less defect prone files (G,I,J) have up to twice the issue count.

As this is an exploratory study, when analyzing statistical significance we ran our tests at a 90% confidence level. As we are intending to discover relationships that can be later more rigorously examined, we would prefer to err on the side of finding false positives, rather than missing any relationship.

We now answer separately each research question.

4.1 RQ C1-C2: Which ASA issue categories can identify defect-prone components?

Table IV, first column, reports Spearman correlations between Resharper issues densities of specific issue categories and defects. Statistically significant values (i.e., p-value ≤ 0.10) are shown in bold.

We used issue densities (issues/NCSS) in the following computations because a positive Spearman correlation (rho=0.93, pval < 0.01) was found between NCSS and number of issues. We did not normalize the number of defects because the correlation between defects and size was not significant (rho= 0.42, pval= 0.15)

TABLE III. DEFECTS PER FILE

File ID	Component(s)	Resharper issues	Defects
A	C1,	29	1
B	C1,C2,	15	4
C		35	6
D	C1,	84	3
E		7	1
F	C1,C2,	73	4
G	C3,C1,C2,	73	2
H		1	2
I	C1,	45	1
J	C1,	65	2
K	C5,C9,	7	2

TABLE IV. CORRELATION BETWEEN DENSITY OF RESHARPER ISSUE TYPES AND DEFECT DENSITIES

Defect types:	All RQ1C1	F	FR	FU	R	U
ASP.NET						
Common Practicesand Code Improvements	-0.14	-0.13	-0.34	0.07	0	-0.2
Compiler Warnings	0.3	0.31	0.48	0.28	0.04	0.25
Constraints Violations	0.11	0.1	0.03	0.09	0.23	0.18
Language Usage Opportunities	**0.57**	0.53	0.55	0.5	0.2	0.43
Potential Code Quality Issues	0.54	0.5	0.51	0.44	0.22	0.44
Redundancies in Code	0.52	0.49	0.47	0.33	0.39	0.53
Redundancies in Symbol Declarations	0.42	0.45	0.01	0.28	0.17	0.14
Unused symbols	0.53	0.53	**0.75**	**0.57**	0.33	**0.56**
Sum of Resharper issues	0.19	0.18	0.1	0.09	0.23	0.23

The total number of Resharper issues has an insignificant but positive correlation with defect-proneness (0.19, p = 0.29) with all defects. Looking at Table IV, column "All RQ C1", we observe positive correlations for all but one category (Common Practices and Code Improvements), and one (Language Usage Opportunities, rho= 0.57) is significant at the 90% confidence level (in bold). Hence, the answer to RQ C1 is: Only a few issue categories, such as *Language Usage Opportunities* in this example, are positively correlated with defects at the component level. Issues in the category *Language Usage Opportunities* identify optimizations at code level based on specific characteristics of C#. The most frequent detections were:

- Convert 'if' statement to 'switch' statement
- Invert 'if' statement to reduce nesting
- Loop can be converted into LINQ-expression
- Use 'var' keyword when initializer explicitly declares type
- Use 'var' keyword when possible

Possible root causes for this correlation are that the usage of more advances language features leads to less defect (i.e. the more language usage opportunities, the less code features are used in the code). Or, it might be that junior developers use less advanced language features than their more experience peers, and also produce more defect prone code.

Table IV, columns 2-6, reports on the correlations between Resharper issue densities and defects, divided into the ISO\IEC 9126 quality characteristics. The only category with significant positive correlations (in bold) is *Unused Symbols*: 0.75 with FR defects, 0.57 with FU defects, 0.56 with U defects. All Unused symbols issues were type members never used. We answer the research question the following way: Only very few indicators can be mapped to defects on the component level, and these indicators point to a wider range of quality characteristics rather than on a single one.

We performed a follow-up analysis to see whether the two categories Language Usage Opportunities and Unused Symbols could be used as defect locators. We tested their capability to detect defects earlier than metrics of size and complexity, widely used in the defect prediction literature (e.g., [13], [17], [11], [8], [12]). Figure 4 shows the cumulative distribution of defects found ranking logical components with respect to the following indicators:

- An ideal indicator that perfectly rank logical components from the faultiest one to the ones with no defect.
- The density of issues of each of the following Resharper issues categories:
  - Unused Symbols
  - Language usage opportunities
- The density of all Resharper issues.
- The number of statements (NCSS).
- The average McCabe complexity.

In other words, the curves in Figure 4 represent how quickly defects would be found if components were tested in different orders, sorted by the criteria listed above. A horizontal line on the graph indicates the point at which 80% of defects have been found.

We observe in Figure 4 that the first 3 components contain 80% of the defects using the ideal locator. Language Usage Opportunities issue density and the total Resharper issue density find 80% of defects at the 5th component, and all the other indicators at the 6th (Unused Symbols, Complexity and Size). The figure also shows that the two selected Resharper categories are overall close to the "all issues" data line which does not consider the category of Resharper issues. This indicates that, at the component level, the distinction between issue categories might lead to small but not vast improvement compared to using all issues.

4.2 RQ F1-F2: Which ASA issue categories can identify defect-prone files?

Tables V and Table VI show, both for defect prone files and non-defect prone files and for each Resharper issue category, mean and standard deviation of Resharper issues densities, the

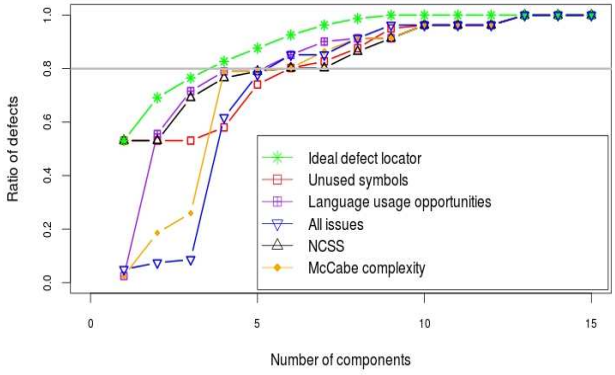

Figure 4. Cumulative distribution of defects in components and indicators

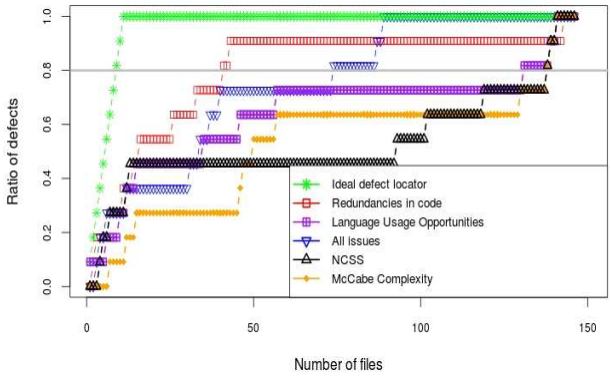

Figure 5. Cumulative distribution of defects in files and indicators

number of files for each set and the p-value of the Mann-Whitney test on the difference between the two sets. Bold percentages indicate p-values that are significant at our chosen confidence level of 90%. Table VI presents only combinations of Resharper categories and ISO\IEC 9126 defect classifications for which the null hypothesis was rejected.

The categories with highest differences on Resharper issues densities in defect prone/non defect prone files are Redundiancies in Code and Language usage opportunities. Redundancies in Code are related to Functionality and Usability defects, both separately and together. Constraints violations are related to Functionality and Functionality-Usability, while Language usage opportunities only with Usability.

We already presented examples of the issues of the category Language Usage. Examples of Redundancies in Code are:

- Assignment is not used
- Explicit delegate creation expression is redundant
- Expression is always 'true' or always 'false'
- Redundant boolean comparison
- Redundant cast
- Redundant 'else' keyword
- Redundant explicit type in array creation
- Redundant 'this.' qualifier

We performed the same follow up analysis that we did for components and we report in Figure 5 the cumulative distribution of defects found ranking files with respect to the following indicators:

TABLE V. RESEARCH QUESTION F1: RESULTS

Resharper issues	Defect prone files (11)		Non defect prone files (101)		Pval
	Mean Resharper issue/NCSS	*Sd Resharper issues/NCSS*	*Mean Resharper issues/NCSS*	*Sd Resharper issues/NCSS*	
ASP.NET	0	0	0	0	NA
Common Practices and Code Improvements	0.13	0.19	0.21	0.18	0.983
Compiler Warnings	0	0.01	0	0.01	0.333
Constraints Violations	0.13	0.05	0.08	0.05	**0.014**
Language Usage Opportunities	0.14	0.07	0.08	0.08	**0.026**
Potential Code Quality Issues	0	0.01	0	0	**0.021**
Redundancies in Code	0.27	0.20	0.08	0.11	**<0.001**
Redundancies in Symbol Declarations	0	0	0.06	0.1	0.969
Unused.Symbols	0	0	0	0	NA
Sum	0.67	0.24	0.52	0.23	0.133

TABLE VI. RESEARCH QUESTION F2 (ONLY STATISTICALLY SIGNIFICANT RESULTS)

Quality characteristic – Resharper issue category	Defect prone files			Non defect prone files			Pval
	Mean Resharper issues/NCSS	*Sd Resharper issues/NCSS*	*Nr of files*	*Mean Resharper issues/NCSS*	*Sd Resharper issues/NCSS*	*Nr of files*	
F – Constraints Violations	0.14	0.06	6	0.08	0.05	94	0.013
F – Redundancies in Code	0.23	0.14	6	0.09	0.13	94	0.002
FR – Compiler Warnings	0.02	NA	1	0	0.01	99	0.001
FU – Constraints Violations	0.18	0.04	3	0.08	0.05	97	0.002
FU – Redundancies in Code	0.35	0.05	3	0.09	0.13	97	0.004
FU - Sum	0.74	0.09	3	0.53	0.24	97	0.062
R – Redundancies in Code	0.39	0.39	2	0.09	0.12	98	0.033
R - Sum	0.93	0.21	2	0.53	0.23	98	0.029
U – Constraints Violations	0.13	0.07	4	0.08	0.05	96	0.085
U – Language Usage Opportunities	0.15	0.07	4	0.09	0.08	96	0.042
U – Potential Code Quality Issues	0.01	0.01	4	0	0	96	<0.001
U – Redundancies in Code	0.16	0.12	4	0.09	0.13	96	0.033

- an ideal indicator that perfectly rank logical components from the faultiest one to the ones with no defect;
- the density of issues of each of the following Resharper issues categories:
  - Language Usage Opportunities
  - Redundancies in code
- the density of all Resharper issues;
- the average McCabe complexity ;
- the number of statements (NCSS).

A horizontal line in the graphs indicates the point at which 80% of defects are found.

Results at file level are more diverse than at component level: Selecting files based on the density of *Redundancies in code* issues outperforms all the other indicators, reaching 80% of defects at the 41st file (compared to the 9th file of the ideal locator). The second best indicator is the sum of Resharper issues: however, it reaches the threshold at the 74th position. NCSS and McCabe complexity are less precise indicators at file level: they are able to identify the 80% of defects only very late: a user will have to examine at 90% of all files before capturing 80% of all defect prone ones.

Overall we answer the research questions on file level the following way:

1. Multiple Resharper categories are good candidates for building predictive models for defect prone modules.
2. There is a set of promising candidates of Resharper categories that is able to predict the quality impact of defect more precisely.

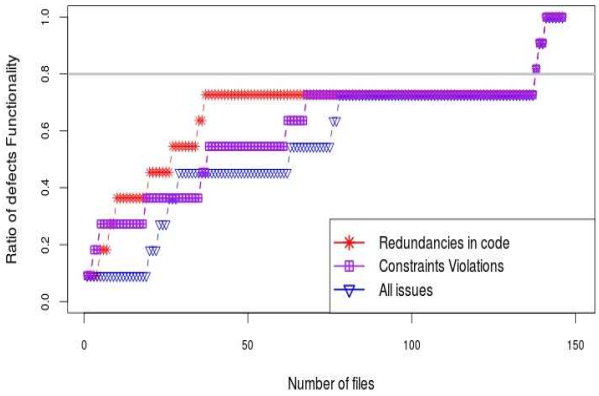

Figure 6: Predictor Performance for Functionality

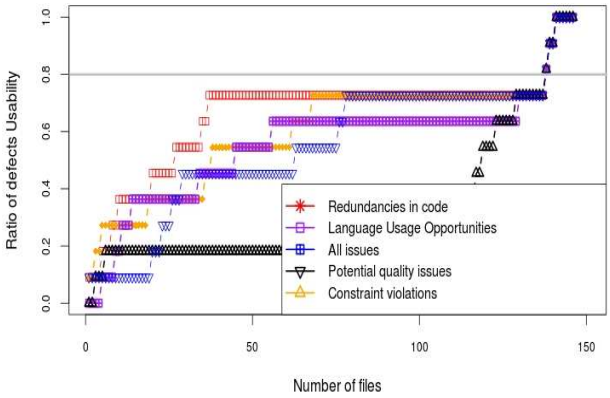

Figure 7: Predictor Performance for Usability

In a follow up analysis we picked two quality characteristics of interest, Functionality (F) and Usability (U), and plotted the same graphs as before (see Figure 6 and 7) for the respective significant issue categories from Table VI. In both cases *Redundancies in Code* is a more efficient predictor than the sum of all issues.

5. DISCUSSION

The presented data indicates that the answer to the research questions is not straight forward in all cases. Most statistics on component level were rather inconclusive and showed only small correlations or a small set of useful issue categories. We believe that this indicates the high-level component view is perhaps not the right perspective for future research direction. The more promising results showed on file level, even if we had to deal with a sparse data set. The results indicated that number of promising indicators is larger, and this also holds for the number of categories pointing to specific quality problems.

On both analysis levels we could improve the defect prediction quality by using selected single predictors, e.g. as Figures 4-7 show. Results also indicate that ASA issues are more promising to be good defect predictors than traditional software metrics, such as complexity or size.

Some of the inspected issue categories, such as redundancies in code and unused symbols (both components and file level) indicate problems regarding memory waste. Vetro' et al. [20] also found a correlation between a similar category of FindBugs issues (unused variables) and defects in students' projects. The authors commented that this correlation could be the consequence of the programmers' difficulties in the design of the class, because they planned to use more/different variables that indeed were not necessary. A similar explanation could be extended for these categories of Resharper.

Further, some of the issues of category Language Usage Opportunities can also be an indicator of the level of programmers' knowledge on the language.

6. THREATS TO VALIDITY

We identify a first construct threat in the mapping files-components. Even though this heuristic eliminates the subjectivity of the manual mapping, 18% of the files were not assigned to any component.

Another threat is subjectivity in the ISO 9126 defect classification. We controlled this threat selecting the most reliable classification made by the experts. A more comprehensive discussion of this threat is found in the original study [22].

The small number of components and of files with defects (11) make statistical significance and a definitive answer to our research questions hard to obtain. We were aware of this threat and also for this reason we performed an explorative study and findings will be evaluated and better investigated in future work.

As in any inductive study, the generalization of these findings is debatable because they are tied to the specific context of the analysis. Our research design reflects this concern: in this study we were focused on identifying whether there was any evidence that Resharper issues could be used as early indicators of defect-prone parts of the system, and especially whether estimates could be made regarding the type of quality impacted by those defects. Having obtained an initial indication that this is in fact a feasible approach, further study is necessary to determine whether the specific correlations found in this study can be replicated elsewhere.

7. CONCLUSIONS

Recent work in the literature ([23] [24] [4] [5] [15] [21] [20]) showed that automatic static analysis tools signal too many false positive issues, i.e. issues not related to any defect. As a consequence, looking at the single issues can be time consuming and not efficient. For this reason, researchers recently investigated whether using the ASA issues can help technical managers and developers to identify faulty modules: several studies ([16] [18] [26] [19] [14]]) reported a positive answer. The study presented in this paper is in the second stream of research, adding the following contributions:

- We evaluate a combination tool-language (Resharper,C#) not yet evaluated in past works, up to our knowledge.
- We performed and compared the analysis at two granularity levels, i.e. logical components and files.
- We investigate whether ASA issues are able to identify specific categories of defects belonging to specific quality dimension.

We found that few Resharper categories had positive correlations with defects at component level, while several categories were more efficient at file level. The issues with higher correlations identify problems regarding code readability, performance, and more in general related to maintainability problems.

Moreover, classifying the defects according to the ISO 9126 quality characteristics, different ASA issues categories were positively correlated to different quality characteristics.

We compared the capability of Resharper issues to detect the faultiest modules, both at components and files levels with the result that specific ASA issues were more efficient than the sum of them or traditional indicators (i.e. software metrics).

Based on the experience of this study, we provide future researchers with the following set of recommendations:

- Analysis on file level might lead to more promising results than on component level.
- The size of the project should be at least, but preferably larger than our medium sized project, to avoid data sparseness problems as we found in our study.

Considering future research directions, we suggest to better understand if results for specific categories are useful in other environments (e.g. if redundancies in code also predict usability problems when using other ASA tools), or if this approach will always require a process of exploration, data analysis, and tailoring towards a specific software environment. In latter case, the contribution of future research should focus on building practitioner-oriented *methods* to build such prediction models rather than building new models.

8. REFERENCES

[1] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '07, pages 1–8, New York, NY, USA, 2007. ACM.

[2] D. Binkley. Source code analysis: A road map. In *Future of Software Engineering, 2007. FOSE '07*, pages 104 –119, may 2007.

[3] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34:135–137, January 2001.

[4] C. Boogerd and L. Moonen. Assessing the value of coding standards: An empirical study. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 277 –286, 28 2008-oct. 4 2008.

[5] C. Boogerd and L. Moonen. Evaluating the relation between coding standard violations and faultswithin and across software versions. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 41 –50, May 2009.

[6] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.-Y. Wong. Orthogonal defect classification-a concept for in-process measurements. *Software Engineering, IEEE Transactions on*, 18(11):943 –956, nov 1992.

[7] Ward Cunningham. The wycash portfolio management system. In *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*, OOPSLA '92, pages 29–30, New York, NY, USA, 1992. ACM.

[8] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Softw. Eng.*, 26:797–814, August 2000.

[9] ISO/IEC. Iso/iec 9126. software engineering – product quality, 2001.

[10] ISO/IEC. Iso/iec 25010. systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models, 2011.

[11] A. Günes Koru, Khaled El Emam, Dongsong Zhang, Hongfang Liu, and Divya Mathew. Theory of relative defect proneness. *Empirical Softw. Engg.*, 13:473–498, October 2008.

[12] A. Günes Koru and Hongfang Liu. An investigation of the effect of module size on defect prediction using static measures. *SIGSOFT Softw. Eng. Notes*, 30:1–5, May 2005.

[13] A. Gunes Koru, Dongsong Zhang, and Hongfang Liu. Modeling the effect of size on defect proneness for open-source software. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 10–, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Artem Marchenko and Pekka Abrahamsson. Predicting software defect density: a case study on automated static code analysis. In *Proceedings of the 8th international conference on Agile processes in software engineering and extreme programming*, XP'07, pages 137–140, Berlin, Heidelberg, 2007. Springer-Verlag.

[15] MIRA Ltd. MISRA-C:2004 Guidelines for the use of the C language in critical systems, October 2004.

[16] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 580–586, New York, NY, USA, 2005. ACM.

[17] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006. ACM.

[18] Nachiappan Nagappan, Laurie Williams, John Hudepohl, Will Snipes, and Mladen Vouk. Preliminary results on using static analysis tools for software inspection. *Software Reliability Engineering, International Symposium on*, 0:429–439, 2004.

[19] R. Plosch, H. Gruber, A. Hentschel, G. Pomberger, and S. Schiffer. On the relation between external software quality and static code analysis. In *Software Engineering Workshop, 2008. SEW '08. 32nd Annual IEEE*, pages 169 –174, oct. 2008.

[20] A. Vetro', M. Morisio, and M. Torchiano. An empirical validation of findbugss issues related to defects. *IET Seminar Digests*, 2011(1):144–153, 2011.

[21] A. Vetro', M. Torchiano, and M. Morisio. Assessing the precision of findbugs by mining java projects developed at a university. In IEEE CS Press, editor, *Proceedings of MSR 2010*, pages 110–113, 2010.

[22] A. Vetro', N. Zazworka, C. Seaman, and F. Shull. Using the ISO/IEC 9126 product quality model to classify defects : a controlled experiment. In *Proceedings of the 16th International Conference on Evaluation & Assessment in Software Engineering (EASE 2012)*, 2012.

[23] Stefan Wagner, Jan Jï¿½rjens, Claudia Koller, and Peter Trischberger. Comparing defect finding tools with reviews and tests. In *IN PROC. 17TH INTERNATIONAL CONFERENCE ON TESTING OF COMMUNICATING SYSTEMS (TESTCOM 2005), VOLUME 3502 OF LNCS*, pages 40–55. Springer, 2005.

[24] F. Wedyan, D. Alrmuny, and J.M. Bieman. The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, pages 141 –150, april 2009.

[25] Nico Zazworka, Kai Stapel, Eric Knauss, Forrest Shull, Victor R. Basili, and Kurt Schneider. Are developers complying with the process: an xp study. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 14:1–14:10, New York, NY, USA, 2010. ACM.

[26] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J.P. Hudepohl, and M.A. Vouk. On the value of static analysis for fault detection in software. *Software Engineering, IEEE Transactions on*, 32(4):240 – 253, april 2006.