Using Inspection Results for Prioritizing Test Activities

Frank Elberzhager[1], Robert Eschbach[1,2], Jürgen Münch[1,2]
[1]Fraunhofer IESE, [2]University of Kaiserslautern
Kaiserslautern, Germany
{frank.elberzhager, robert.eschbach, juergen.muench}@iese.fraunhofer.de

***Abstract*—Today's software quality assurance techniques are often applied in isolation, without any consideration for the synergies resulting from systematically combining and integrating different techniques. Such combinations promise benefits, such as a reduction in quality assurance effort or higher defect detection rates. The integration of inspection and testing techniques, for instance, promises different synergy effects, especially reduced testing effort. Existing approaches for reducing testing effort are widely based on the use of metrics in order to predict fault-prone parts of a product or determine test exit criteria. However, they do not make systematic use of the results from inspections. In this article, a rule-based approach is presented that uses inspection results to prioritize test activities; i.e., test activities are focused on specific parts of the code or on specific defect types, which leads to reduced testing effort. Thus, an overall reduction in quality assurance effort is expected. Results from a case study show that a reduction in testing effort of between 6% and 34% was achievable at a quality level comparable to a non-integrated approach. In addition, related work is described and an outlook on future research is given.**

Keywords – inspection; testing; test prioritization; quality assurance strategy

I. INTRODUCTION

To ensure software products of high quality, well-established quality assurance (QA) techniques exist today, such as various inspection and testing techniques [3][28][47]. However, the effort for performing such QA, especially testing, can amount to a significant percentage of overall development costs [1][2]. Thus, it is often important to reduce the overall QA effort while preserving defect detection effectiveness at least at a comparable level.

In order to reduce QA effort, several approaches for prioritizing QA activities exist. Strooper and Wojcicki [52][53], for example, describe how to select and combine different QA techniques in order to systematically find the most cost-effective combination. A different approach often applied is to prioritize QA effort to certain parts of a product, for example, by using metrics such as code size or code complexity [4][10][11]. Such metrics are used to focus QA activities on those parts of a system that are expected to have the most defects. Another source for prioritizing QA activities is historical data [17][18]. Defect types, for example, that were found in previous releases, are also expected to be relevant in a current release. Furthermore, prediction approaches [30][32] are used to estimate the number of remaining defects based on gathered defect data, which is then used to suggest when to stop a certain QA activity. Moreover, in order to reduce testing effort, static analysis tools are sometimes applied before the test activities start. However, most of the approaches do not use defect data from current QA activities to prioritize subsequent QA activities (e.g., testing).

A multitude of different inspection and testing techniques have been developed in the past 30 years. A number of studies and experiments have been performed to analyze their effectiveness and efficiency. Though it is suggested to apply them in an integrated way in order to obtain higher quality at lower overall costs [42][54], this suggestion is rarely followed. Instead, if both inspection and testing techniques are applied, this is done in isolation in many cases, i.e., no synergy effects are exploited.

Thus, this article presents an integrated inspection and testing approach that mainly uses defect data from an inspection to prioritize test activities in order to reduce testing effort and overall QA effort. In addition, certain metrics and historical data combined with the inspection data can be used to further improve the test prioritization. The approach is based on explicitly described assumptions and concrete selection rules derived from them. These selection rules support choosing specific parts of the product or defect types to consider and thus, help to focus test activities. An initial case study was performed to evaluate the applicability of the approach and assess the extent of effort reductions that might be achievable.

The remainder of this article is structured as follows. Section 2 gives an overview of related work. Section 3 presents the integrated inspection and testing approach. Section 4 describes the performed case study. Finally, Section 5 concludes the article and gives an outlook on future research.

II. RELATED WORK

When considering approaches that reduce QA effort, making a distinction between local and global approaches is useful. Local approaches concentrate on the enhancement of a single QA activity (i.e., in this context, test effort reduction), while global approaches refer to a mix of different QA activities to reduce overall QA effort.

A. Approaches to Reduce Local QA Effort

In general, knowledge about the product to be checked and the environment (e.g., project, process, or context

information) can be used to prioritize a test activity. One approach is to use one or multiple metrics to predict defect-prone parts of a system. Basili and Perricone [4] empirically investigated size and complexity metrics and their relation to defect-prone modules. They stated that these metrics can show areas of potential defects. Emam et al. [5] empirically showed that the larger the code modules are, the more defects are contained in them. Hatton [6] suggests an optimal module size (i.e., modules with the least number of defects) of between 200 and 400 lines of code (LoC), as proven in several studies. Later studies [7][8][9] showed results inconsistent with Hatton's suggestion. Moreover, different studies showed either a strong correlation between McCabe complexity [10] and defect rates [12][13], or no correlation [14]. Basili et al. [11] were able to show that different object-oriented complexity metrics appear to be useful to predict classes that are more defect-prone than others.

Although metrics are worthwhile, the application of one single metric to predict defect-prone modules can lead to quite inaccurate predictions. Fenton and Neil [15] suggest not to use complexity as sole predictor of defects and Ostrand and Weyuker [7] state that "various other factors beside size might be responsible for the fault density of a particular file". Thus, another strategy is to combine metrics to predict the defect-prone parts of a product [16]. Nevertheless, it is still unclear which combinations of metrics are the best ones.

Besides gathering metrics from a current project, historical data are another source for predicting defect-prone parts in order to prioritize test activities. Illes-Seifert and Paech [18] stated that historical data are a good indicator of quality and that, among others, the number of historical defects is an indicator of future defects, though not the best one. A number of studies were performed where different historical data were used to estimate defect-prone modules in order to improve test effort allocation [19][20][57]. However, it often remained unclear which kind of historical data is appropriate in a certain context. Turhan et al. [9] propose using cross-company data together with nearest-neighbor sampling or static call graph-based sampling to prioritize software testing and allocate resources. A study based on 25 large projects in a telecommunication environment resulted in a reduced amount of code to be checked. Zimmermann and Nagappan [21] propose using network analysis on dependency graphs to prioritize test activities for programming units that are more likely to contain defects. Suitable results could be shown, but the authors stated that this is just one more predictor of defects.

Some hybrid approaches are able to predict the number of remaining defects resulting in suggested test exit criteria, which facilitates the allocation of test effort. One example is the hybrid estimation model HyDEEP for estimating defect content and QA effectiveness [22][23]. The approach combines historical project data and expert knowledge about relevant influencing factors in order to predict the defect content and thus, to estimate test exit criteria.

Some other approaches reducing test effort include, for example, using static analysis tools [24], performing cumulative test analysis [25], or optimizing a test profile using a Markov decision model [26]. However, detailed prioritization on concrete parts of the product to be tested is rarely proposed. Finally, defect classifications (e.g., Orthogonal Defect Classification (ODC) by IBM [27]) of current test defects, historical test defects, or domain-specific test defects can be used to prioritize certain defect types.

When information or data from prior inspection activities is considered for prioritizing test activities, the main approaches are defect prediction and defect classification. One prediction approach, which uses inspection data, is the capture-recapture method [30]. The number of remaining defects can be predicted with statistical methods in a software artifact (including code). Though the approach is mainly used to control the inspection, the results may also be used to suggest test exit criteria. The detection profile method [31] is an alternative prediction approach using a linear regression method. Another prediction approach is subjective estimations, which are investigated by El Emam et al. [32] and Biffl [33], among others. Studies performed by them resulted in better predictions compared to objective prediction approaches. While such approaches are mainly used to decide if a re-inspection should be performed, a decision on how many tests to perform is also conceivable. Harding [34] explicitly suggests using inspection data to forecast test defects and proposes some pragmatic calculation rules. Finally, defect classifications used to classify current or historical inspection and test defects (e.g., [27][35][36]) can be used to prioritize defect types for testing based on classified inspection defect types.

However, although it can sometimes be observed in industry that inspection data are used in an informal way to prioritize test activities, no systematic approach could be found by the authors in the existing literature. A lot of inspection research has been done in the past, but the focus often was on improving the inspection itself rather than on combining or integrating the results with further QA activities in order to reduce effort [28][29].

B. Approaches to Reduce Overall QA Effort

Defect flow models capture the number of defects per QA activity and the origin of each defect (i.e., the activity in which they were injected) [38]. This is done in order to identify those development steps where QA effort should be a priority. Risk-based approaches can prioritize certain parts of artifacts to be checked. Lamersdorf et al. [17], for instance, propose a model for identifying risks early in a project for globally distributed software development. The model is based on project characteristics and could be used to identify quality risks, resulting in a prioritization of QA activities. HyDEEP [21] can be extended to comprise different QA activities. In addition, holistic inspection and test process control approaches [39] can be applied to direct QA activities. Moreover, approaches supporting the selection of a mix of different QA techniques have been developed [52][53]. Finally, improvement approaches such as the well-established Quality Improvement Paradigm [37] can be applied for continuous optimization of QA strategies.

A number of studies and experiments have been performed to compare inspection and testing techniques

(e.g., [35][40][41]) and have shown the benefits of both techniques. Laitenberger suggested applying both techniques in combination rather than in isolation [42]. Some studies investigated the extent to which a combined application of inspection and testing is beneficial: Laitenberger performed an experiment in which first inspection and then testing were performed on the corrected code, but the inspection result was not used to prioritize the test activity [42]. Wood et al. [43] calculated the effectiveness of combined inspection and testing techniques based on an experiment where the combination was much more effective than single usage. Gack [44] used a simulation approach to compare five scenarios where different QA technique combinations were selected to calculate an appropriate combination. It could be shown that a mixed strategy might outperform a test-only strategy in terms of cost and found defects. Nevertheless, none of the mentioned approaches and studies used inspection results to prioritize the test activities in a more fine-grained manner.

III. APPROACH

The basic idea of the integrated inspection and testing approach is to use inspection results (i.e., a defect profile) for prioritizing specific parts of the product that are expected to have the most defects or for prioritizing defect types that are expected to be especially relevant. Consequently, test activities are focused on such prioritized parts or defect types. In order to focus test activities, it is necessary to describe context-specific assumptions about the relationship between findings in the inspection and defect distribution in the product under test. Such an assumption can be, for instance, that in code classes where many defects are detected during the inspection a higher probability for remaining defects exists than in code classes where few defects were found, i.e., an accumulation of defects is expected (of course, such an assumption needs to be context-specific). Ideally, assumptions to be applied in a concrete context are based on empirically validated hypotheses that are valid in the given environment. If such evidence is not available, assumptions have to be described explicitly and analyzed with respect to their suitability for the specific context (e.g., a post-testing analysis could show if an assumption was wrong). As a next step, the assumptions need to be quantified if they are not already defined in measurable terms. This means that concrete metrics need to be derived that make the assumptions measurable. For instance, the number of defects detected in a code class could be measured as defect content or defect density. Finally, so-called selection rules need to be formulated that operationalize assumptions in a way that parts of the product or specific defect types are prioritized and thus, testing activities can be focused. An example of such a selection rule is "Focus your test activity on all code classes where the defect density is higher than five major defects per 1000 LoC based on the inspection defect profile". When prioritizing on defect types for a test activity, an exemplary selection rule might be "Focus your test activity on those two defect types which are found most often with the inspection." In this case, it has to be ensured by the defect classification that the defect types can be found by both inspection and testing activities.

In addition to the inspection results (i.e., the inspection defect profile), further data can be used to make assumptions and derive selection rules. Different metrics for predicting defect-prone classes or historical data are valuable and established concepts, which can be used in combination with the inspection data. For instance, an assumption could be the following: "For code classes with a high complexity value in combination with a high defect density based on inspection results, a high probability exists that further defects will be found during testing within such code classes".

While these steps describe the configuration of the approach, Fig. 1 shows its application for code inspection and testing. First, the inspection (step 1) has to be performed. The inspection defect profile (containing, for instance, the number of inspection defects found per class or per defect type) is obtained after the inspection, and, in some cases, additional information is gathered (such as metrics or historical data which is stored in an experience database, short: EDB). In order to be able to rely on the inspection data, the inspection results have to be monitored [28][46]. Thus, inspection quality monitoring (step 2) should be performed in order to analyze and determine the quality of the inspection results, which is done by comparing context-specific historical data and certain inspection metrics, such as reading rate or number of defects found per inspector. If no historical data is available, data from the literature can give first recommendations.

Next, test prioritization (step 3) is done. Certain assumptions, respectively selection rules, can be applied to prioritize parts of the code or defect types. Fig. 2 gives an overview of the test prioritization steps based on assumptions and selection rules (on the left); two simplified examples are sketched on the right. The exemplary assumption A1 claims that parts of the code where a significant number of inspection defects are found indicate remaining defects to be found by testing (i.e., an accumulation of defects is assumed). It is particularly based on the empirical observation that a high number of defects are often contained within a small number of modules [50]. The number of defects is expressed as defect content (absolute number of

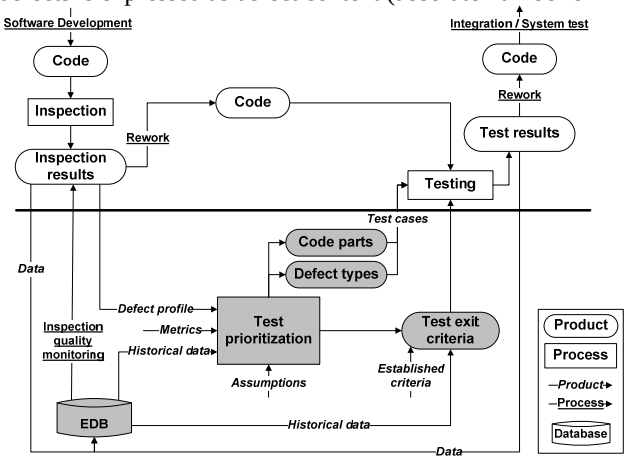

Figure 1. Overview of the integrated approach

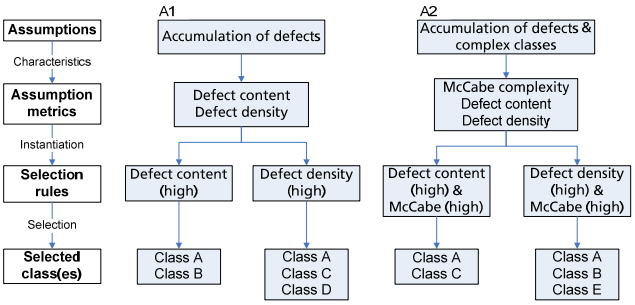

Figure 2. Steps and two examples for the prioritization of code classes to be tested based on assumptions and refined selection rules

defects) and defect density (absolute number of defects divided by inspected LoC). According to assumption A1, code classes that have high defect content based on the inspection results are selected for testing. The second selection rule chooses code classes for testing that have a high defect density based on the inspection results. The exemplary assumption A2 claims that parts of the code where a significant number of inspection defects are found and which are complex indicate remaining defects to be found by testing. This assumption uses the inspection defect profile and one complexity metric, expressed as McCabe complexity. Consequently, two concrete selection rules combine, defect content with McCabe complexity on the one hand and defect density with McCabe complexity on the other hand, resulting in different code classes being selected. The selection rules may prioritize different code classes to be tested, which depends on the concrete context. Ideally, evidence is already obtained in a concrete environment which selection rules lead to the best selection of code classes (i.e., highest effort reduction at a comparable quality).

The approach does not define fixed values as to what high defect content means, for example. From our point of view, this is highly dependent on certain context factors and thus, has to be defined in a concrete environment before the approach is applied. Finally, established techniques for deriving concrete test cases for the code classes can be used, such as equivalence partitioning or boundary-value analysis.

Based on the test prioritization, testing the selected code classes (step 4) can be conducted. Here, the test prioritization also has an influence on the test exit criteria (i.e., when to stop testing). Again, historical context-specific data from the EDB can give valuable hints on when to stop testing. In addition, established criteria such as branch coverage can be considered. However, test exit criteria are not the focus of this article. Finally, data from the current inspection and testing should be stored together with the data in the EDB for future use.

IV. CASE STUDY

A. Goals of the Study

The primary goal of the study was to evaluate whether the integrated inspection and testing approach leads to effort reduction for testing and, consequently, for the overall QA. For this, a comparison with a non-integrated inspection and testing approach (i.e., a "traditional approach") is necessary, which does not use the inspection results as a means for prioritizing the test.

A second main goal of the study was to evaluate the quality of the integrated approach with respect to the number of defects found, i.e., an answer to the question of how many defects are found with the integrated approach compared to a traditional one.

Finally, different assumptions and their derived selection rules, which are used to prioritize the test activity, are compared.

These goals result in the following research questions:

RQ1 – Effort reduction: Does the proposed integrated inspection and testing approach lead to effort reduction for testing compared to a non-integrated approach (i.e., the inspection and the test activity are performed sequentially without exchanging data to prioritize the test activity)?

RQ2 – Quality preservation: Does the proposed integrated inspection and testing approach find a comparable number of defects compared to a non-integrated approach?

RQ3 – Assumptions: Which assumptions and derived selection rules are best suited with respect to effort reduction at comparable quality for the given environment?

The following hypothesis can be derived from the research questions:

H1: If the integrated inspection and testing approach is used, it takes less test effort than a non-integrated inspection and testing approach, while comparable quality is achieved.

B. Context of the Study

A pilot study and a more comprehensive case study were conducted in the same environment.

The artifact to be checked was a Java prototype tool, which had mainly been developed by one developer. The tool supports practitioners in performing sequence-based specifications. Currently, it consists of 76 classes, over 650 methods, and about 8500 lines of code (LoC). The critical code parts were inspected. In the pilot study, these comprised four classes with a total of about 1000 LoC. In the case study, four classes of about 2400 LoC were inspected. Due to continuous development of the tool, the inspected classes were different between the pilot study and the case study.

In the pilot study, one inspector had very good inspection knowledge, but only limited programming experience, while the remaining inspectors were mainly testers or developers with some inspection knowledge, but high programming experience. In the case study, one developer was replaced by an experienced inspector. The test activity was performed by one developer who was not involved in the inspection.

In the given context, the integrated inspection and testing approach was only applied to prioritize certain code parts of the product (i.e., no defect types were prioritized).

C. Design of the Study

The pilot study and the more comprehensive case study followed the same design.

In order to answer RQ1-RQ3 and to check the hypothesis, several variables were taken into account.

Regarding the variables of the study, inspection and testing effort was measured in minutes. An approach A is of comparable quality to another approach B if at least the same number of defects are found. For applying assumptions, respectively the selection rules, different quantifications of variables were used. For example, regarding the number of defects, the absolute number and the defect density were taken. In addition, three severity classes are used. Size was measured in lines of code and different complexity measures such as McCabe complexity were applied (see Table IV).

First, an inspection (step 1) was performed by four computer scientists. The participants of the study were selected systematically with respect to the knowledge needed to perform the inspection in an effective way.

To cover the quality needs of the prototype tool, context-specific checklists were developed, of which each inspector used a different one as suggested in [47][48]. The checklists were partly similar to suggestions from Burnstein [45], but also contained additional context-specific questions. Overall, five checklists were used, containing between four and nine questions focusing on requirements fulfillment, functional aspects, extensibility, performance, and reliability. Each inspector searched individually for defects, using one or two checklists. Two experienced quality assurance engineers put together the defect profile from the inspection results.

Afterwards, the inspection results were analyzed by those two quality assurance engineers (step 2), i.e., they performed the inspection quality monitoring based on certain inspection metrics such as reading rate and defect detection rate.

Next, the prioritization of certain code classes was performed based on the inspection defect profile in order to focus the test activity (step 3).

According to the integrated inspection and testing approach, a focused test activity (i.e., a test of prioritized parts of the code) would be the next step. However, in order to be able to evaluate the achievable effort reduction and the quality of the integrated approach, one developer followed a traditional testing approach (step 4) first, i.e., an experience-based test activity including equivalence partitioning was performed without using information from the inspection.

Finally, the defect results obtained from the traditional test (i.e., number of test defects found per code class) were used as a baseline and compared with the test prioritization (see step 3) from the integrated approach, i.e., an analysis of the achievable test effort reduction and the resulting quality was conducted.

D. Preparation (Pilot Study)

One goal of the pilot study was to evaluate the general applicability of the integrated approach. In addition, further goals to be explored to the extent possible included an initial identification of effort reduction and an initial evaluation of the number of defects found with the integrated approach compared to that found using a non-integrated inspection and testing approach (where the inspection is followed by a test and uses equivalence partitioning, but no inspection data for prioritizing the test) (RQ1 and RQ2).

TABLE I. INSPECTION DEFECT PROFILE OF THE PILOT STUDY BASED ON DEFECT DATA BY FOUR INSPECTORS

Code class	SBSTree State	SBSTree Comperator	SBSTree	Main	Sum
Total	26	6	27	8	**67**

After four inspectors performed the inspection using focused checklists (i.e., each inspector checked the code with a different perspective), the inspection defect profile could be put together. In total, 67 issues were found by the inspectors. Table I shows an overview.

One assumption was used in the pilot study (no definition of selection rules was initially done):

A1. Parts of the code where a large number of inspection defects are found indicate more defects to be found with testing.

Context-specific historical data were not available for monitoring the quality of the inspection results. Hence, an analysis of at least the reading rate was performed, which was 137 lines of code per hour and was consistent with general suggestions found in [46]. Furthermore, 67 issues found per 1000 LoC also seemed reasonable based on the experience of the inspection experts (though this was just a gut feeling).

Based on the inspection results (see Table I) and assumption A1, a prioritization of the classes *SBSTreeState* and *SBSTree* was done, as those two contained by far the largest numbers of the issues found.

An experience-based test activity of all four code classes including equivalence partitioning was performed by one person. Overall, seven defects were found, with three defects being found in *SBSTreeState* and four defects in *SBSTree* (see Table II). About 70% of these defects were critical ones (i.e., classified as major or crash). The prioritization of the code classes fits exactly to those code classes where the defects occurred and consequently, assumption A1 could be confirmed in the given context.

The inspection defect profile in combination with the assumption was suitable for prioritizing certain parts of the code and for focusing the test activity, and thus, the applicability of the approach could be shown. The main results of the pilot study with respect to our research questions can be summarized as follows:

- Effort reduction (RQ1): It was not possible to determine a concrete test effort reduction. This was mainly due to continuous development, rapidly changing code, and an unsystematic test.
- Quality preservation (RQ2): Based on assumption A1, all defects could be found with the integrated approach, just like with the traditional approach.

TABLE II. TEST DEFECTS FOUND BY ONE TESTER IN THE PILOT STUDY CLASSIFIED ACCORDING TO THREE SEVERITY CLASSES

Severity / Class	minor	major	crash	Sum
SBSTree	1	3	0	**4**
SBSTreeState	1	1	1	**3**

E. Execution of the Case Study

Due to the general applicability of the integrated inspection and testing approach as seen in the pilot study, a case study was performed to evaluate RQ1-RQ3, and H1.

Performing the inspection: The first step was the performance of the inspection by four inspectors, which took about three to four hours per inspector, resulting in 835 minutes in total for all inspectors. Each inspector checked the four code classes with different checklists and documented the issues found in a defect-tracking system, recording a description of the issue, the place where it was found, and a severity classification (minor, major, crash).

After the inspection had been performed, all issues were analyzed by two experienced quality assurance engineers in order to eliminate duplicates and comments (e.g., improvement suggestions) and put together the defect profile. A total of 100 defects were found by all inspectors. Table III shows the total number of defects found (i.e., defect content) and the defect density per code class. In addition, the defect content and defect density for each severity class per code class are shown.

Monitoring the inspection results: As in the pilot study, a quality monitoring of the inspection results was performed. Aurum et al. [28] state that "historical data may help to determine the quality of the current inspection". Thus, a comparison of the current inspection results with those from the pilot study was performed (which was justified since the context was the same). The average number of defects found was a bit lower (42 to 67 defects per 1000 LoC). The same is true for the average time needed to inspect the code (347 to 435 minutes per 1000 LoC). The reading rate in the case study was therefore slightly higher (172 LoC per hour). Finally, the distribution of minor, major, and crash defects is similar. Thus, all in all, the results seemed reasonable, though the inspection performance was a bit lower compared to the pilot study. One reason for this might be a smaller amount of annotated code, which prevented the inspectors from checking all code parts in detail due to unclarities.

Prioritizing the test: Prioritization of the test activity was mainly based on the inspection profile and metrics. Three assumptions were used, where A1 was the same as in the pilot study. Some rationales and empirical evidence for each assumption are presented next.

A1. Parts of the code where a large number of inspection defects are found (i.e., an accumulation of defects is observed) indicate more defects to be found with testing.

A large number of different studies performed in various environments showed that an accumulation of defects can be observed rather than an equal distribution of defects. Basili and Perricone [4] already documented that about 60% of the defects stem from 35% of the modules. Later observations [14][49] resulted in the rule of thumb that 80% of the defects can be found in 20% of the modules [50]. Some recent studies have confirmed these results [8][51][57]. Based on the published observations, assumption A1 was defined with respect to the integrated inspection and testing approach.

A2. Parts of the code where a large number of inspection defects are found (i.e., an accumulation of defects is observed) and which are of small size indicate more defects to be found with testing.

As mentioned in Section 2, a size metric is often used to prioritize the test activity. Though this metric is often applied, a number of studies showed inconsistent results when size is applied as the sole metric for predicting defect-prone modules. Emam et al. [5] stated that if models are built to predict fault-proneness, more variables than just size should be used. Thus, inspection results were combined with a size metric. A number of studies were identified in which small code modules tended to be more defect-prone (though some studies showed the opposite). Consequently, this was added to assumption A2.

A3. Parts of the code where a large number of inspection defects are found (i.e., an accumulation of defects is observed) and which are of high complexity indicate more defects to be found with testing.

Besides size, complexity is another popular metric often used to prioritize the test activity (see Section 2). However, Schröter et al. [16] noted that new or combinations of existing metrics should be used to study the relationship between complexity and the presence of bugs. Thus, in order to improve the test prioritization, the inspection results were combined with the mean of two different complexity metrics, namely *method length* and *McCabe's cyclomatic complexity*.

Overall, 23 selection rules were initially derived from the three assumptions manually, mainly based on a brainstorming session (see Tables VI, VII and VIII). Following the assumptions, selection rules with respect to the inspection results were derived first, resulting in a focus on the most defect-prone parts. Second, common metrics, identified during related work, were considered and combined with the inspection results. Table IV shows the values of the assumption metrics for each code class (the last two metrics were calculated using *metrics* [56]). Based on the inspection defect profile and the metrics, the derived selection rules were applied to prioritize code modules for the test activity.

Performing the test: To evaluate the integrated inspection and testing approach, a traditional test activity was performed first. One developer of the prototype tool performed the test activity using equivalence partitioning. Besides the four inspected classes, three additional connected code classes were tested.

TABLE III. INSPECTION DEFECT PROFILE OF THE CASE STUDY BASED ON INSPECTION DATA BY FOUR INSPECTORS

	Code class	Sequence	SequenceTableModel	SimpleKeyedTableModel	SimpleOrderedKeyedT.	Sum
	defect content (dc)	**14**	**40**	**39**	**7**	**100**
	defect density (dd)	**0.061**	**0.029**	**0.056**	**0.061**	-
severity dc	minor defects	10	31	25	5	71
	major defects	3	9	14	1	27
	crash defects	1	0	0	1	2
severity dd	minor defects	0.043	0.023	0.036	0.044	-
	major defects	0.013	0.007	0.020	0.009	-
	crash defects	0.004	0.000	0.000	0.009	-

TABLE IV. VALUES OF ASSUMPTION METRICS

Class	size (LoC)	mean method length (LoC)	mean McCabe complexity
Sequence	231	3.28	1.78
SequenceTableModel	1364	13.54	3.90
SimpleKeyedTableModel	701	8.11	2.91
SimpleOrderedKeyedT.	115	7	2

F. Case Study Results and Interpretation

During the traditional test activity, six defects were found, with five defects being found in the class *SimpleKeyedTableModel* (class three) and one in *TableUtil*, which is called by that class, among others. Overall, 16 hours of test effort were needed to test all seven classes with the traditional approach, including correction and documentation (see upper part of Table V). Based on the test result, the integrated inspection and testing approach was applied to evaluate the achievable effort reduction and the quality preservation of the different prioritizations.

RQ1 (Effort reduction): Overall, an effort reduction of between 6% and 34% was achieved by the integrated inspection and testing approach. The achievable effort reductions depend on different assumptions and the concrete selection rules.

RQ2 (Quality preservation): In the bottom part of Table V, only those selections of code classes and the resulting test effort reduction are shown in which the defect-prone class *SimpleKeyedTableModel* (class three, including one calling class) is contained. Consequently, this class has to be tested with the integrated approach by all means in order to achieve comparable quality. However, ten of the initially defined selection rules led to quality preservation, while 13 of them did not select the defect-prone class three.

RQ3 (Assumptions): Next, a detailed analysis of each assumption and the derived selection rules with respect to effort reduction and quality preservation is given in order to analyze which ones led to appropriate selections of code classes in the given context. Note that the current integrated approach is only able to prioritize code classes that were also inspected. Before the application of the selection rules, which was done by two experienced quality assurance engineers, a clarification of what "high", "low", "small", and "large" meant in the given context was done. In most cases, the definition was obvious or discussed until the same understanding was gained.

A1: Those selection rules led to suitable selections of classes for testing. Table VI shows each applied selection rule for assumption A1, the selected classes, and the achieved effort reduction. If the defect-prone class three is selected, a '+' marks a suitable quality Q of the selection rule, otherwise a '-' is chosen and no effort reduction is calculated (expressed as "/" in Tables VI - VIII) because no comparable quality is achieved. For example, using the selection rule 'defect content (high)' resulted in choosing the classes *SequenceTableModel* (40 inspection defects found) and *SimpleKeyedTableModel* (39 inspection defects found) and consequently, in an effort reduction of nine percent. The same selection of classes was done when choosing classes containing the largest numbers of minor and major defects (see Table III for concrete values). The two selection rules

TABLE V. TEST EFFORT OF THE TRADITIONAL TEST AND DIFFERENT EFFORT REDUCTIONS OF THE PRIORITIZED TEST

	Tested code classes	Test Effort	Tested classes	Effort reduction
1	Sequence	4.0 h		
2	SequenceTableModel			
3	SimpleKeyedTableModel	5.5 h		
4	SimpleOrderedKeyedT.	0.5 h		
5	MultipleKeyedTableLinkSet	1.0 h		
6	SimpleKeyedTableLinkSet			
7	SimpleKeyedTableLink			
	Test documentation	1.0 h		
	Test correction	4.0 h		
	Non-integrated test effort	**16 h**	1-7	
	Prioritized code classes:	15 h	1+3+4	6.25%
	Test effort reduction with quality preservation	14.5 h	1+3, 2+3	9.38%
		10.5 h	3	34.38%

TABLE VI. SELECTION RULES, SELECTED CODE CLASSES AND EFFORT REDUCTIONS OF ASSUMPTION A1

Assumpt.	Selection rule	Selected classes	Effort reduction	Q.
A1	defect content (high)	2, 3	9.38%	+
	defect density (high)	1, 3, 4	6.25%	+
	crash severity (defect content (high))	1, 4	/	-
	major severity (defect content (high))	2, 3	9.38%	+
	minor severity (defect content (high))	2, 3	9.38%	+
	crash severity (defect density (high))	1, 4	/	-
	major severity (defect density (high))	1, 3	9.38%	+
	minor severity (defect density (high))	1, 3, 4	6.25%	+

using crash severity (both defect content and defect density) led to class selections that did not contain any defects or, more precisely, did not select the defect-prone class three. Due to the high criticality of crash defects, already one such defect within a code class was interpreted as high defect content or high defect density, for this severity class. However, due to the very low number of crash defects found, a different interpretation is conceivable. With respect to defect density in general, code classes one, three, and four were selected (see Table III, where the defect density of the three mentioned classes is about twice as high as that of code class two). The resulting effort reduction was six percent. When prioritizing code classes with a high defect density for defects classified as minor or major, an effort reduction of between six and nine percent is achievable. Based on the described results, six of eight selection rules led to an appropriate selection of code classes. Consequently, based on those selection rules, assumption A1 could be confirmed in our context.

A2: Those selection rules led to inappropriate selections of code classes (see Table VII). None of the five selection rules chose the defect-prone class three (either no code class fulfills both criteria or class one and four were selected). To further analyze the combination of inspection defects and size, an alternative assumption A2* was defined, which combines defect accumulation with code classes of large size (instead of small size). As mentioned in the rationales for A2, some studies showed that large-sized modules are more defect-prone than small-sized modules (e.g., Emam et al. [5]). In the given context, this assumption led to suitable results, and four of five derived selection rules included the

TABLE VII. SELECTION RULES, SELECTED CODE CLASSES AND EFFORT REDUCTIONS OF ASSUMPTION A2

Assu mpt.	Selection rule	Selected classes	Effort reduction	Q.
A2	defect content (high) & size (small)	-	/	-
	defect density (high) & size (small)	1, 4	/	-
	crash severity (defect content (high)) & size (small)	1, 4	/	-
	major severity (defect content (high)) & size (small)	-	/	-
	minor severity (defect content (high)) & size (small)	-	/	-
A2*	defect content (high) & size (large)	2, 3	9.38%	+
	defect density (high) & size (large)	3	34.38%	+
	crash severity (defect content (high)) & size (large)	-	/	-
	major severity (defect content (high)) & size (large)	2, 3	9.38%	+
	minor severity (defect content (high)) & size (large)	2, 3	9.38%	+

TABLE VIII. SELECTION RULES, SELECTED CODE CLASSES AND EFFORT REDUCTIONS OF ASSUMPTION A3

Assu mpt.	Selection rule	Selected classes	Effort reduction	Q.
A3	defect content (high) & McCabe (high)	2, 3	9.38%	+
	defect content (high) & mean method length (high)	2	/	-
	defect density (high) & McCabe (high)	3	34.38%	+
	defect density (high) & mean method length (high)	-	/	-
	crash severity (defect content (high)) & McCabe (high)	-	/	-
	crash severity (defect content (high)) & mean method length (high)	-	/	-
	major severity (defect content (high)) & McCabe (high)	2, 3	9.38%	+
	major severity (defect content (high)) & mean method length (high)	2	/	-
	minor severity (defect content (high)) & McCabe (high)	2, 3	9.38%	+
	minor severity (defect content (high)) & mean method length (high)	2	/	-
A3*	defect content (high) & mean method length (low)	3	34.38%	+
	defect density (high) & mean method length (low)	1, 3, 4	6.25%	+
	major severity (defect content (high)) & mean method length (low)	3	34.38%	+
	minor severity (defect content (high)) & mean method length (low)	3	34.38%	+

defect-prone class three. When using the selection rule 'defect density (high) & size (large)', only the defect-prone class three was selected (i.e., 'defect density (high)' is true for class one, three, and four, 'size (large)' is true for class two and three, resulting in selection of class three), resulting in an effort reduction of 34%. Combining the large number of defects, and defects classified as major or minor in terms of size, an effort reduction of about nine percent was achieved (see Tables III, IV, and VII). Again, using crash severity in combination with size led to an inappropriate selection. One reason might be the very low number of such defects found. Overall, assumption A2 must be rejected in our context in the way initially stated, but could be confirmed when redefined as A2*.

A3: The results for the different selection rules of assumption three were inconsistent (see Table VIII). An effort reduction of 9% was achieved when combining 'defect content (high)' with 'McCabe (high)'. A combination of high defect density and a high McCabe value led to an effort reduction of 34%. An effort reduction of also 9% was achieved for large numbers of major and minor defects combined with a high McCabe value. These four selection rules confirmed A3 (see Tables III, IV, and VIII). However, a combination of large numbers of crash defects and different complexity metrics again led to inappropriate selections. With respect to combinations with the complexity metric 'mean method length (high)' (here a mean method length > 10 LoC), all selection rules led to inappropriate selections of code classes. In order to check the combination of mean method length and defect accumulation, an alternative assumption A3* was defined and four selection rules were derived. In the bottom part of Table VIII, excellent results for effort reduction can be observed for the alternative selection rules. In summary, combining different complexity metrics with defect metrics led to inconsistent results. Assumption A3 thus can neither be confirmed nor rejected for all selection rules in our context, but an indication of appropriate and inappropriate selection rules in the applied environment was gained.

To recap the results with respect to RQ3, many useful selection rules were identified for our context and all three assumptions are valuable, though more evaluation across a number of QA cycles is necessary to identify the most beneficial selection rules and obtain more evidence in the given context in order to enable application of the integrated inspection and testing approach. For the integrated approach to be applied, evidence is needed on which assumptions and derived selection rules lead to appropriate selections of code classes to be tested. Our analyses can give initial answers, with the assumptions and selections rules being able to serve as a starting point for applying and analyzing them in a different context.

H1: With respect to hypothesis H1, it is not possible to provide statistically significant results due to a single tester only. This drawback came from project and effort restrictions. However, based on the test results of the single tester, it could be shown that different test effort reductions of between 6% and 34% could be achieved with the integrated approach depending on which assumptions and concrete selection rules were used. A comparison of 23 initially defined selection rules was conducted. 10 selection rules led to an appropriate selection (i.e., no undetected test defect), while 13 ones led to inappropriate selections. Changing assumption A2 to combine inspection results with large-sized classes (instead of small-sized ones, which was initially taken from identified related work), the ratio changed to 14 good ones and nine bad ones. Finally, removing those rules using *crash* defects (due to only two such defects in our context), the ratio changed to 19 adequate selections and only four inadequate selections for the given context.

G. Limitations of the Case Study

As in any empirical study, there are threats to the validity of the study results [55]. Next, a discussion of what we consider to be the most relevant threats in our case study is presented.

Conclusion validity: Due to the low number of test effort data of the testers, it was not possible to perform statistical tests (i.e., low statistical power). However, the initial results confirm that the approach is promising.

Construction validity: A decision on how to treat, e.g., "low" and "high" was done to be able to apply the selection rules. As mentioned above, concrete values have to be chosen depending on the context because fixed values are not necessarily valid in each environment. However, a different determination could have led to different test prioritizations. A representative number of different selection rules were derived in order to compare them. Some of them are correlated. Still, additional ones may further support prioritization. In addition, no standard checklists were used, which could have affected the inspection performance.

Internal validity: The selection of the subjects was done systematically, but another selection might have led to different defects being found for inspection and testing. Regarding testing, the effect was slightly reduced by using equivalence partitioning and regarding inspections, the effect was slightly reduced by using checklists that focused the inspectors on certain aspects in the code.

External validity: The prototype tool is one small example used to apply and evaluate the integrated approach, which was developed by only one developer. The number of found test defects was rather small. Thus, the conclusions drawn have to be considered with caution. However, the tool is large enough to get an initial impression of the approach. Furthermore, assumptions and selection rules have to be analyzed always in each new environment in order to find the optimal ones, i.e., they cannot be generalized without adaptation.

V. Conclusion and Outlook

Using inspection results can be one additional predictor of defect-prone parts or expected defect types in a product. An approach that mainly uses inspection data (i.e., defect data from a current QA cycle) and certain assumptions to prioritize the test activity and thus, to reduce test effort and overall QA effort was introduced. Established concepts, such as size and complexity metrics and historical data, were integrated to enhance the integrated approach.

A case study was described including a pilot study initially performed to check the general applicability of the approach, which could be shown. To summarize the main results of the case study: An effort reduction of the integrated approach between 6% and 34% could be achieved compared to a non-integrated approach, with comparable quality being obtained. All three assumptions and many derived selection rules led to appropriate selections of code classes at comparable quality. However, as expected, some selection rules did not lead to an appropriate selection of code classes (i.e., the defect-prone class was not selected). Thus, when the integrated approach is applied in a different context, it has to be validated which assumptions and corresponding selection rules lead to appropriate selections over a number of QA cycles, i.e., evidence for applied assumptions and selection rules must be obtained. The shown assumptions and selection rules can be a starting point for this and are generic enough to make them easy to reuse.

We have several plans for future work. Regarding evaluation, we want to evaluate the integrated approach in different contexts (e.g., industrial environments, academia, or open-source projects) where inspection and test data are available. We also want to apply the approach across several releases in order to identify the most valuable assumptions and selection rules in a given environment and to gain statistically significant results. This also includes comparing different assumptions and selection rules. Furthermore, we plan to extend the integrated approach. One idea is to combine different selection rules, in order to further improve prioritization (e.g., identify the most defect-prone parts (assumption 1) and for those parts, focus on defect types which are found most often by inspection (assumption 2)). Second, no complete parts should be omitted for testing, but prioritization on a percentage basis should be suggested for certain parts of the product. Third, from our experience, inspections are often only performed on limited parts of the product. However, the limited inspection results should also be used to prioritize those parts of the product to be tested that were not inspected (e.g., based on characteristics of the inspected parts that are similar to suggested parts of the product). Finally, the approach should be studied in terms of the extent to which it is applicable to different development phases such as requirements inspections used for prioritizing system test activities.

Acknowledgment

Parts of this work have been funded by the Stiftung Rheinland-Pfalz für Innovation project "Qualitäts-KIT" (grant: 925). We would like to thank Sabine Nunnenmacher for the initial review of the article and Sonnhild Namingha for proofreading.

References

[1] R. Pressman. Software engineering: a practitioner's approach. McGraw-Hill, London, 5th edition, 2000

[2] Health, Social, and Economic Research, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, 2002

[3] N. Juristo, A.M. Moreno, S. Vegas, "Reviewing 25 years of testing technique experiments," Emp. Software Engineering, pp. 7-44, 2004

[4] V.R. Basili, B.T. Perricone, "Software errors and complexity: an empirical investigation," Com. of the ACM, pp. 42-52, 1984

[5] K.E. Emam, S. Benlarbi, N. Goel, W. Mela, H. Lounis, S.N. Rai, "The optimal class size for object oriented software," IEEE Transactions on Software Engineering, pp. 494-509, 2002

[6] L. Hatton, "Reexamining the fault density – component size connection," pp. 89-97, IEEE Software, 1997

[7] T.J. Ostrand, E.J. Weyuker, "The distribution of faults in a large industrial software system," International Symposium on Software Testing and Analysis, pp. 55-64, 2002

[8] C. Andersson, P. Runeson, "A replicated quantitative analysis of fault distributions in complex software systems," IEEE Transactions on Software Engineering, pp. 273-286, 2007

[9] B. Turhan, G. Kocak, A. Bener, "Data mining source code for locating software bugs: A case study in telecommunication industry," Expert Systems with Applications, pp. 9986-9990, 2009

[10] T.J. McCabe, “A complexity measure,” IEEE Transactions on Software Engineering, pp. 308-320, 1976

[11] V.R. Basili, L.C. Briand, W.L. Melo, “A validation of object-oriented design metrics as quality indicators,” IEEE Transactions on Software Engineering, pp. 751-761, 1996

[12] S. Henry, D. Kafura, K. Harris, “On the relationship among three software metrics,” ACM workshop/symposium on measurement and evaluation of software quality, pp. 81-88, 1981

[13] N. Nagappan, T. Ball, A. Zeller, “Mining metrics to predict component failures,” International Conference on Software Engineering, pp. 452-461, 2006

[14] N.E. Fenton, N. Ohlsson, “Quantitative analysis of faults and failures in a complex software system,” IEEE Transactions on Software Engineering, pp. 797-814, 2000

[15] N.E. Fenton, M. Neil, “A critique of software defect prediction models,” IEEE Transactions on Software Eng., pp. 675-689, 1999

[16] A. Schröter, T. Zimmermann, R. Premraj, A. Zeller, „If your bug database could talk,“ 5th International Symposium on Empirical Software Engineering, pp. 18-20, 2006

[17] A. Lamersdorf, J. Münch, A. Fernández-del Viso Torre, C. Rebate Sánchez, M. Heinz, D., “A Rule-based Model for Customized Risk Identification in Distributed Software Development Projects,” Int. Conference on Global Software Engineering, 2010, in press

[18] T. Illes-Seifert, B. Paech, “Exploring the relationship of a file’s history and its fault-proneness: An empirical method and its application to open source programs,” Information and Software Technology Journal, pp. 539-558, 2009

[19] N. Nagappan, T. Ball, B. Murphy, “Using historical in-process and product metrics for early estimation of software failures,” 17th Int. Symposium on Software Reliability Engineering, pp. 62-74, 2006

[20] T.J. Ostrand, E.J. Weyuker, R.M. Bell, “Predicting the location and number of faults in large software systems,” IEEE Transactions on sofware engineering, pp. 340-355, 2005

[21] T. Zimmermann, N. Nagappan, “Predicting defects using network analysis on dependency graphs,” 30th International Conference on Software engineering, pp. 531-540, 2008

[22] M. Klaes, H. Nakao, F. Elberzhager, J. Münch, “Predicting Defect Content and Quality Assurance Effectiveness by Combining Expert Judgment and Defect Data - A Case Study,” 19th International Symposium on Software Reliability Engineering, pp. 17-26, 2008

[23] M. Klaes, F. Elberzhager, J. Muench, K. Hartjes, O.v.Graevemeyer, “Transparent combination of expert and measurement data for defect prediction – an industrial case study,” 32nd Int. Conference on Software Engineering, pp. 119-128, 2010

[24] N. Nagappan, T. Ball, “Static analysis tools as early indicators of pre-release defect density,” 27th international conference on Software engineering, pp. 580-586, 2005

[25] I. Holden, D. Dalton, “Improving testing efficiency using cumulative test analysis,” Testing: Academic & Industrial Conference on Practice and Research Techniques, pp. 152-158, 2006

[26] D. Zhang, C. Nie, “A markov decision approach to optimize testing profile in software testing,” 9th International Conference for Young Computer Scientists, pp. 1205-1210, 2008

[27] Orthogonal Defect Classification, IBM, available: http://www.research.ibm.com/softeng/ODC/ODC.HTM

[28] A. Aurum, H. Petersson, C. Wohlin, “State-of-the-art: software inspections after 25 years,” Software Testing, Verification and Reliability Journal, pp. 133-154, 2002

[29] S. Kollanus, J. Koskinen, “Survey of Software Inspection Research: 1991-2005,” Comp. Science and Inf. Systems reports, 2007

[30] H. Petersson, T. Thelin, P. Runeson, C. Wohlin, “Capture–recapture in software inspections after 10 years research—theory, evaluation and application,” Journal of Systems and Software, pp. 249-264, 2004

[31] L. Briand, K.E. Emam, B. Freimut, “A comparison and integration of capture–recapture models and the detection profile method,” 9th Int’l Symp. Software Reliability Eng., pp. 32-43, 1998

[32] K.E. Emam, O. Laitenberger, T. Harbich, “The application of subjective estimates of effectiveness to controlling software inspections,” Journal of Systems and Software, pp. 119-136, 2000

[33] S. Biffl, “Using inspection data for defect estimation,” IEEE Software, pp. 36-43, 2000

[34] J.T. Harding, “Using inspection data to forecast test defects,” Software Technology Transition, pp. 19-24, 1998

[35] V.R. Basili, R.W. Selby, “Comparing the effectiveness of software testing strategies”, IEEE Trans. on Softw. Eng., pp. 1278-1296, 1987

[36] M.V. Mantyla, C. Lassenius, “What Types of Defects Are Really Discovered in Code Reviews?” IEEE Transactions on Software Engineering, pp. 430-448, 2009

[37] V.R. Basili, G. Caldiera, H.D. Rombach, “Experience factory,” Encyclopedia of Software Engineering. Volume 1. A-O. New York: John Wiley&Sons, pp. 511-519, 2002

[38] B. Freimut, C. Denger, M. Ketterer, “An industrial case study of implementing and validating defect classification for process improvement and quality management,” 11th International Software Metrics Symposium, pp. 19-31, 2005

[39] J.K. Chaar, M.J. Halling, I.S. Bhandari, R. Chillarege, “In-Process evaluation for software inspection and test,” IEEE Transactions on Software Engineering, pp. 1055-1070, 1993

[40] E. Kamsties, C.M. Lott, “An empirical evaluation of three defect detection techniques,” 5th European Software Engineering Conference, pp. 362-383, 1995

[41] N. Juristo, S. Vegas, “Functional testing, structural testing and code reading: what fault type do they each detect?” Empirical Methods and Studies in Software Engineering, pp. 208-232, 2003

[42] O. Laitenberger, “Studying the effects of code inspections and structural testing on software quality,” 9th International Symposium on Software Reliability Engineering, pp. 237-246, 1998

[43] M. Wood, M. Roper, A. Brooks, J. Miller, “Comparing and combining software defect detection techniques - a replicated empirical study,” 6th European Softw. Eng. Conf., pp. 262-277, 1997

[44] G.A. Gack, “An economic analysis of software defect removal methods,” based on book Managing the Black Hole: The Executive's Guide to Software Project Risk, Business Expert Publishing, 2010

[45] I. Burnstein, Practical Software Testing. Springer, 2002

[46] J. Barnard, A. Price, “Managing Code Inspection Information,” IEEE Software, pp. 59-69, 1994

[47] K.E. Wiegers, Peer Reviews in Software, Addison-Wesley, 2002

[48] T. Gilb, D. Graham, Software Inspections, Addison-Wesley, 1993

[49] N. Ohlsson, M. Helander, C. Wohlin, “Quality improvement by identification of fault-prone modules using software design metrics,” 6th Int. Conference on Software Engineering, pp. 1-13, 1996

[50] B. Boehm, V.R. Basili, “Software reduction top 10 list,” IEEE Computer, pp. 135-137, 2001

[51] M. Hamill, K. Goseva-Popstojanova, “Common trends in software fault and failure data,” IEEE Trans. on Softw. Eng, pp. 484-496, 2009

[52] P. Strooper, M.A. Wojcicki, “Selecting V&V technology combinations: how to pick a winner?” 12th IEEE Int. Conference on Engineering Complex Computer Systems, pp. 87-96, 2007

[53] P. Strooper, M.A. Wojcicki, “An iterative empirical strategy for the systematic selection of a combination of verification and validation technologies,” Int. Workshop on Software Quality, pp. 9-14, 2007

[54] A. Endres, D. Rombach, A Handbook of Software and Systems Engineering, Addison Wesley, 2003

[55] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, A. Wesslen, Experimentation in software engineering an introduction, Kluwer, 2000

[56] Metrics tool, http://metrics.sourceforge.net/

[57] T.J. Ostrand, E.J. Weyuker, R.M Bell, “We’re finding most of the bugs, but what are we missing?” Int. Conference on Software Testing, Verification and Validation, pp. 313-322, 2010