Empir Software Eng (2007) 12:295–310
DOI 10.1007/s10664-006-9029-x

Empirical analysis on the correlation between GCC compiler warnings and revision numbers of source files in five industrial software projects

Raimund Moser · Barbara Russo · Giancarlo Succi

Published online: 28 November 2006

Editor: F. Shull

Abstract This article discusses whether using warnings generated by the GNU C++ compiler can be used effectively to identify source code files that are likely to be error prone. We analyze five industrial projects written in C++ and belonging to the telecommunication domain. We find a significant positive correlation between the number of compiler warnings and the number of source files changes. We use such correlation to conclude that compiler warnings may be used as an indicator for the presence of software defects in source code. The result of this research is useful for finding defect-prone modules in newer projects, which lack change history.

Keywords Compiler warnings · Revision numbers · Software defect prediction · Bootstrap · Meta-analysis

1 Introduction

In software development, early detection of defects is of paramount importance. A defect found later in a software system is more expensive and difficult to fix (Basili et al. 2002). Maintenance costs decrease if a software system has fewer bugs: Every bug fix implies a modification of the system and can introduce new bugs, or even a malfunction (Shull et al. 2002). Early detection of a defect-prone class may lead to a redesign of it; thus guide to a more reliable and clearer design (Williams et al. 2003). Customer satisfaction significantly grows with quality code (Krishnan 1993). And the final goal of software engineering is to make the customer happy with the delivered product.

R. Moser (✉) · B. Russo · G. Succi
Center for Applied Software Engineering, Free University of Bolzano-Bozen,
Piazza Domenicani 3 Dominikanerplatz, 39100 Bolzano-Bozen,
Bolzano-Bozen, Italy
e-mail: raimund.moser@unibz.it

G. Succi
e-mail: Giancarlo.Succi@unibz.it

The GNU C++ compiler produces automatically warning messages during software compilation. There is no extra effort to collect them. Hence, if we could find a correlation between the number of compiler warnings of a source file and software defects located in it, we could use them for predicting defect-prone files already in an early stage of software development. In this research we study whether such a link exists by analyzing the correlation between warnings generated by the GNU C++ compiler and the revision number of corresponding source code files.

Our research strategy is the following. **First**, we extract from the final release of five industrial projects written in C++ the warnings generated during compilation. **Then**, we count the number of revisions of each file, as they appear analyzing the version control system used by the company; there is some evidence that the number of modifications together with the age of a file are related to its number of defects (Graves et al. 2000; Ostrand et al. 2004)—we assume such relationship in this research as we use the number of modifications of a file as a proxy for the number of defects (for a discussion of this hypothesis see Section 3).

Last, we study the presence of a relationship between compiler warnings and revision numbers using the non-parametric Spearman's correlation and cumulative plots.

The results obtained by our analysis confirm the assumption that warnings are correlated with the number of changes (revisions) of files. Therefore, GNU C++ compiler warnings seem to be useful for helping developers and testers to identify the set of source files that are likely to contain most defects.

The paper is organized as follows. In Section 2, there is a brief overview of work related to the problem. In Section 3, we explain our research methodology and the data collection process. In Section 4, we give a brief description of the statistical analysis we apply. In Section 5, we present the analysis of the data and in Section 6 the results. Section 7 contains some threats to validity of our approach and in Section 8 we finally draw some conclusions of this research.

2 Background

One of the key questions in software engineering is how to develop and discover models that could predict the presence of defects as early as possible (Fenton and Neil 1999). There exists an extensive literature dealing with this problem and many models and techniques have been proposed (Stringfellow and Andrews 2002). Some empirical results are addressing these issues and making them to a key point in developing quality and cost models for software projects (Boehm and Basili 2001). These results clearly give high motivation to determine the files with the highest defect rates as early as possible (Tian 2000). Many different techniques for data collection and analysis have been proposed to achieve this goal (Hall and Fenton 1997). Most of the proposed techniques advocate the collection of software size and complexity measures to predict defect-prone files. However, manual collection of product metrics is a time consuming and often-quiet invasive task (Johnson and Disney 1998). New ideas have been proposed to collect software engineering data in an automatic and non invasive way (Johnson et al. 2002; Sillitti et al. 2003), which could be able to solve some of the problems concerning the data collection part and provide reliable data for automatic feedback systems. However, such ideas are still preliminary.

In this paper, we study if it is possible to use information easily available to developers to estimate the presence of faults in source code. Needless to say, compiler warnings are easily available—they are generated automatically without even asking for them, and they can be easily collected and inspected.

The key question we ask is: “are compiler-generated warnings useful in predicting the presence of defects?”

Despite the wide presence of compiler warnings even in early compilers, as far as we know there has been very limited investigation to answer our research question. In the following we give a first answer by providing evidence that warnings generated by the GNU C++ compiler, hereafter referred to as GCC, could be used to predict the presence of defects in source code. Details on the compiler can be found at http://gcc.gnu.org). Our primary goal is not to derive a causal model for the relationship between compiler warnings and software defects, but to assess whether it is possible to provide developers with a very simple directive such as: “look for defects in files with a high number of compiler-generated warnings.” Such directive is very simple to state and to follow: it is likely to be implemented.

Subsequent analysis could then verify if more refined analysis could yield better results, paying off the extra effort to accomplish them.

3 The Research Methodology and Data Collection

Some research (Graves et al. 2000; Ostrand et al. 2004) has been done analyzing the relationship between the number of changes of a file and the defects it contains: Results from such work advocate that the number of modifications together with the age of a file can be used to predict its defect proneness. In particular, Graves *et al.* provide evidence that—again in the telecommunication domain—“deltas”, which are number of changes of files caused by Initial Modification Requests (IMR) perform similar for predicting defect prone files than a so called “stable model” based on the assumption that fault generation dynamics remain stable across modules over time. Moreover, the authors find that adding size measure (lines of code) does not improve a simple model based on change history.

In this research we use the results obtained by *Graves et al.* as we assume that files that have been frequently changed (i.e., with a high revision number in the version control system) are more defect prone than those with a low revision number. However, due to lack of data we analyze only two variables, namely number of compiler warnings and revisions of a file. Without controlling the impact of other variables like size we cannot use the findings of Graves *et al.* for deriving a causal model for the relationship between compiler warnings and software defects; at best we can provide some evidence for an empirical correlation of the two.

Since it appears that the number of revisions of a file is an indicator for its defect proneness one could ask why do compiler warnings add any additional information? Compiler warnings are available already after the first compilation of source code: Therefore, if compiler warnings are correlated with the presence of defects this information can be used at a much earlier time than the information gathered by a retrospective view on the evolution of a software system. However, we have to keep in mind that this conclusion is only valid if we assume that the distribution of faults is stationary over time (Graves et al. 2000).

To summarize, in this paper we analyze the correlation between GCC compiler warnings, W, and the revision number of C++ source code files, D. Using the latter for identifying defect prone files we then conclude whether compiler warnings can be used efficiently for predicting files, which are likely to contain a high number of defects.

3.1 Research Hypothesis

We frame our study within the following research hypothesis:

The null hypothesis H_0 is:
There is no correlation between W and D.
The alternate hypothesis H_1 is:
There is a correlation between W and D.

The goal of this research is to determine if it is possible to reject H_0.

To this end we set an α value equal to 0.01. Such value is smaller than the common α value 0.05 used in software engineering (El Eman et al. 2001; Mišić and Tešić 1997; Nesi and Querci 1998), providing additional confidence in the results.

3.2 Development Environment

The five projects we analyze belong to the telecommunication domain. The software applications have been developed for embedded systems in the telecommunications and are written entirely in C++. The development platform was Sun Solaris http://www.sun.com) and the version control system used for all five projects was CVS http://www.cvshome.org). The company acts as a main international provider for embedded systems in the telecommunication sector. The development team varied from three to five people: All of them are experienced software engineers.

The development process in place at the company was incremental. It prescribed that after the development of each increment, there was a testing phase. The warnings and the revision numbers (extracted from the CVS system) have been collected in the files submitted by developers to testers for unit and integration testing (Fig. 1) after the final release of the software.

3.3 Data Collection

As stated before the data collected for our analysis come from five C++ projects in the telecommunication domain. Due to business confidentiality reasons we are not allowed to reveal the specific features of these projects. We refer to them as project A, B, C, D, and E.

For each project we have the complete list of the source code filenames; and for each file we know the number of warnings *W* generated during compilation and its revision number *D*. The source code has been compiled with the GNU GCC compiler using the command line option "-Wall" for enabling all common warning messages. A detailed description of which type of warnings are reported during compilation using the "-Wall" option can be found on the GCC home page http://gcc.gnu.org). Repetitions of the same warning

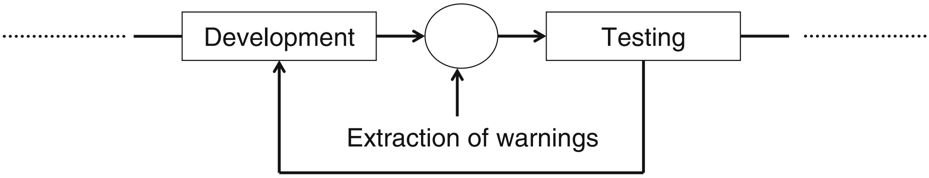

Fig. 1 Software process

messages referring to different source code lines but within a single compilation of a file are counted separately. For this analysis we have collected all warning messages reported during the compilation of the final release of the software. We did not have access to the version control system in order to collect the warnings after each new version. Thus, we do not have any knowledge about the time evolution of warnings: For example, we do not know if certain warnings have been present from the beginning of compilation of a unit or have been introduced only later in the development when a source file has been modified. Maybe developers have ignored throughout development some warnings while they eliminated others by correcting the code. We did not take into account such differences in this study.

Moreover, in this first approach we give all compiler warnings the same weight. We do not make any distinction on the different kinds of warnings. Our second variable is the revision number *D* of every source file as it appears by analyzing the CVS system again after the final release of the software. All projects used CVS as version control system; therefore, revision numbers have been generated in the same way for all projects. In the change data from one revision to the next the following types of changes are included: changes due to new or changed requirements and changes due to bug fixes. However, we are not able to distinguish between different types of changes. Thus, we could not analyze if specific changes (for example only the ones due to bug fixes) are related to the number of compiler warnings in a different way than others.

As said before the total number of compiler warnings and versions of all files contained in a project has been collected only after the final release of the software. Thus, we do not take into account any history of these data but we could take just one snapshot in time. Obviously in this way some useful and interesting information could be lost: For example it is not clear a priori that using the final release as snapshot leads to the same results as if we would have taken an earlier snapshot in time. We refer this issue to a more detailed, future analysis. Moreover, we do not have size information of single files; size may be seen as confounding factor in our analysis as it could account for the correlation we find. We address this issue in the threats to validity Section.

Table 1 shows a descriptive statistics of the raw data we have collected.The columns represent the different projects (project A to project E): Row number 2 contains the number of source code files per project, whereas rows number 4 to 6 contain the minimum, maximum, and mean value of the number of warnings (*W*) and the number of revisions (*D*) of all files corresponding to one project.

A box-plot (Fig. 1) of the number of warnings and revisions reveals that for almost every project the data contain a lot of outliers. To limit the influence of outliers we have to use techniques, which are resilient and robust for computing reliable correlation coefficients.

Table 1 Descriptive statistics for the five projects: W=number of warnings, D=number of revisions

Project	A		B		C		D		E	
Number of files	13		26		112		40		80	
	W	D	W	D	W	D	W	D	W	D
Minimum	1	1	1	1	1	1	1	1	1	1
Maximum	14	8	4	21	69	38	18	8	400	44
Median	3	1	1.5	2	4	3	2.5	2	9	4

4 Approach in Analyzing the Data

The first step of our statistical analysis involves a careful visual inspection of the empirical data and calculation of a robust correlation coefficient.It is evident that our samples do not have a normal distribution. Moreover, the number of data points within each dataset is limited. Therefore, we cannot apply the Pearson correlation coefficient, which is based on a linear model (Weber 2003). Instead, we adopt Spearman's correlation to investigate our research hypothesis.

For a basic statistical analysis we use Matlab http://www.mathworks.com). For the computation of the statistical significance of the Spearman's coefficient for small samples we apply the Bootstrap method provided by Matlab. By bootstrapping our samples we are able to compute the population mean and the corresponding 99% confidence interval for the sample correlation coefficient without assuming any statistical properties needed for analytical calculations (for example normal distributions).

4.1 Steps of the Analysis

In our analysis we proceed as follows.

(1) **Computation of Spearman's correlation coefficient**: As mentioned before, we use Spearman's coefficient to determine the correlation between number of warnings W of a file and the corresponding number of revisions D. Spearman's correlation coefficient is more robust to atypical values and to non-linearity of the underlying relationship (Siegel and Castellan 1988).

(2) **Computation of the confidence interval of Spearman's correlation coefficient using Bootstrap**: Since our samples are small, for determining the confidence interval of Spearman's correlation coefficients we use the *Bootstrap* method, in particular Efron's percentile confidence limit (Bontempi 2003; Efron and Tibshirani 1993). We set the α-level at 0.01.

(3) **Meta-analysis**: We use statistical meta-analysis to find a common correlation coefficient across the different projects and to check whether the different data samples are homogeneous in respect to Spearman's correlation coefficient (Hedges and Olkin 1985).

(4) **Development of cumulative function plots**: We draw cumulative function plots to discover to which amount compiler warnings are useful in predicting fault proneness of source code files.

4.2 Statistical Meta-Analysis and "Weighted Estimators of a Common Correlation"

Statistical meta-analysis is a branch of statistics that studies means to draw common conclusions from several experiments. Its use is rather limited in software engineering (Miller 2000). Therefore, in this section we summarize the meta-analytical approach we follow (Hedges and Olkin 1985). The basic problem we want to address with meta-analysis is the following.

If we would like to estimate a common correlation coefficient across different projects (samples) we cannot simply average correlations of samples. Studies have shown that the correlations of samples are biased estimators for the correlation of the original population. In addition, the range of a correlation is $[-1, 1]$. Therefore, the distribution of correlations is not normal. To address these issues, we employ the meta-analytical technique called “weighted estimators of a common correlation” (Hedges and Olkin 1985).

The method of the “weighted estimators of a common correlation” is based on the following assumptions:

- The sample data come all from normally-distributed populations with the same correlation;
- The sample correlations are computed according to Pearson’s definition of correlation.

To check normality we perform a Kolmogorov–Smirnov test (Siegel and Castellan 1988) for normality on the five samples: The result indicates that we are not allowed to accept the hypothesis that they are distributed according to a normal distribution. Therefore, we cannot compute in a meaningful way the Pearson correlation coefficient. In order to use meta-analysis techniques, which require the Pearson correlation coefficient we have to approximate it by Spearman’s correlation coefficient using the following formula (Pearson 1907):

$$r_c = 2\sin\left(\frac{\pi}{6}r_s\right) \tag{1}$$

r_s is the Spearman’s correlation coefficient and r_c is the approximated Pearson’s correlation coefficient.

To compute the “weighted estimators of a common correlation” we follow the steps described in (Succi et al. 2005).

4.3 Bootstrap Method

We use the Bootstrap method on one hand to determine the confidence interval for the Spearman’s correlation coefficient and on the other hand to approximate its population mean. Furthermore, we provide a confidence interval, the so-called Efron’s percentile confidence limit (Bontempi 2003; Efron and Tibshirani 1993) for the computed population mean. Our analysis strongly evidences that this interval is located on the positive axis; this means that the mean of the Spearman’s distribution of the population, out of which our samples are taken, is greater than zero. We use this result to reject the null hypothesis.

4.4 Use of Cumulative Function Plots

Cumulative function plots are a simple and yet powerful instrument to visualize any distribution of data; in our case we use them to report the distribution of revision numbers of source files across a software project. They have been extensively used in Medicine for analyzing frequency patterns of medical treatments (Rønning and Guldvog 1998) or for example in Brain Research (Southard et al. 2000).

Using cumulative plots gives as an immediate understanding whether the files containing most warnings are also the ones that have been changed most often during development. The idea of cumulative plots is the following:

We sort all files belonging to a project by their corresponding number of warnings: File k precedes file l, if the number of warnings W in file k is larger than in file l. V stands for the sum of the revision numbers of all files in a project. Let z be the smallest integer such that the sum of the number of revisions *D* from file *1* to file *z* is larger than *0.5*V* (i.e., *z* is the number of files in a project that contain at least half of the sum of all revision numbers in it):

$$z \textit{ is such that} \sum_{k=1}^{z} D_k \geq 0.5 \cdot V \tag{2}$$

Let *n* be the total number of files per project. Then $E=z/n\times 100\%$ is a good indicator for the efficiency of predicting files with high revision numbers using compiler warnings. The smaller *z* and closer *E* is to *0* the more restricted is the set of files with frequent changes (high revision numbers). The goal of this research is to show that compiler warnings are useful to identify such set.

5 Analysis of the Data

Box plots of our data samples (Fig. 1) reveal that there are outliers both for the independent variable number of warnings *W* and the dependent variable number of revisions *D*. The samples we use are small; therefore, we do not exclude any outliers from correlation analysis apart from one item in project E which has a value of *W=400.* Its distance from the median of the sample is four times larger than the distance of all other points, and it is likely that it is an erroneous value that has been introduced during data collection. However, since we use robust correlation techniques the influence of outliers on the estimation of the correlation coefficients is limited.

5.1 Analysis of the Correlations

Table 2 presents a summary of the calculation of Spearman's correlation coefficient.

The first row contains Spearman's correlation coefficients we obtain by computing them directly on our data samples. The second row shows the approximated population mean of the Spearman's correlation distribution for each project. We compute this approximated population mean by using the Bootstrap method with 1,000 resampling (Efron and Tibshirani 1993). The third row indicates the 99% confidence interval for these approximated population means. All Spearman's correlations are statistically significant at the 0.01 level.

Table 2 Spearman's correlation coefficients and Efron's confidence interval

Index\Project	A	B	C	D	E
Spearman's correlation coefficient	0.534	0.337	0.308	0.218	0.310
Approximated population mean of Spearman's correlation coefficient	0.514	0.331	0.307	0.215	0.309
Efron's 99% confidence interval for the population mean of Spearman's correlation coefficient	[0.497 0.528]	[0.316 0.347]	[0.300 0.314]	[0.201 0.227]	[0.299 0.318]

Bootstrap sample: re-sampling number *N=1,000*; for the confidence interval estimation we used *k=1,000* Bootstrap samples

Altogether, we can reject the null hypothesis for each project individually, and state that within each project there is a statistically significant correlation between GCC warnings and revision numbers of files. However, the correlation is not high, and there is for sure no simple linear model for predicting the presence of defects using compiler warnings with such low correlation.

5.2 Generalization of the Results

We apply the meta-analytical approach of the "weighted estimator of a common correlation" to determine if we can generalize the results by computing an average correlation coefficient between compiler warnings and revision numbers for all projects under scrutiny. The meta-analytical Spearman's correlation coefficient across all five projects is 0.308. Such correlation is statistically significant at the 0.01 level, as the 99% confidence interval is given by [0.163, 0.418]. The homogeneity test for our data samples gives a value of $Q=1.236$ which is smaller than $Q_{0.01}=13.277$. This confirms that the different samples are homogeneous from a meta-analytical perspective.

Altogether, we can reject the null hypothesis by combining all five projects in a meta-analytical way at a 99% confidence level. Results from meta-analysis allow us to state that GCC warnings and revision numbers of files are correlated in a comprehensive, project independent way in the problem domain and for the development process under scrutiny.

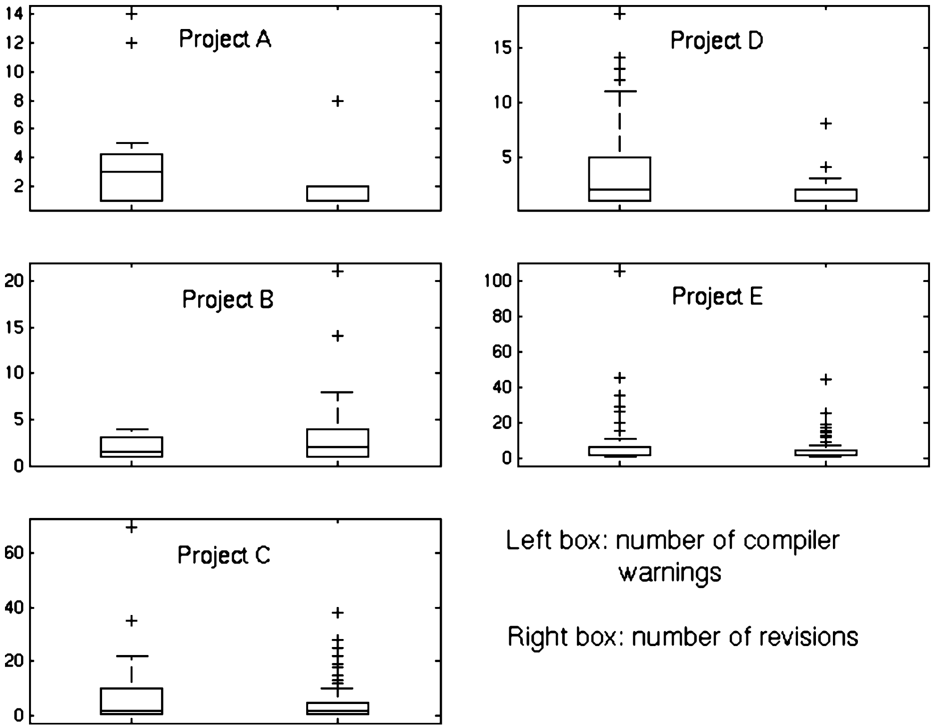

Fig. 2 Box plot of the number of warnings and revisions for the five projects

Table 3 Efficiency of using compiler warnings for predicting frequently changed, and thus defect-prone source files

Projects	A	B	C	D	E
Ideal	31%	15%	14%	30%	11%
E	31%	35%	29%	40%	33%

5.3 Analysis of the Cumulative Plots

Figure 2 shows three different cumulative plots for each project: The dashed line is derived using the assumption that every file in the project contains the same revision number. We use it as a benchmark. The solid line is the ideal curve we get by ordering all files of a project according to their revision numbers. The order is descending: Thirst come the files with the highest revision number. The dotted line finally is the one we obtain by using the number of compiler warnings of files instead of their revision numbers for arranging them. In project C for example the first 29 files account for 50% of the total number of revisions of all files in that project.

From Fig. 2 it is evident that compiler warnings are useful for predicting frequently changed files. The dotted line gives always a better estimation for predicting files with a high revision number than the dashed one, obtained by assuming that within one project all files have the same revision number. But, the plot also shows that we are not that close to the ideal curve: GCC warnings are a first indicator for frequently changed and thus error prone source files, but they lack some efficiency in their predictive power.

These results are confirmed by a straightforward calculation of the efficiency *E* defined in Section 4.4. Table 3 contains the results of such calculation. The first row of Table 3 shows the percentage of files we need to look at to catch half of the sum of all revision numbers contained in a project. The distribution reflects fairly well—as we should expect—the Pareto principle of distribution of faults and failures (Ohlsson and Fenton 2000): A small number of source files contain most of the faults discovered during development and system testing.

The second row of Table 3 shows the same kind of indexes, but instead of revision numbers we use the number of compiler warnings for the computation of *E*. It emphasizes that following this strategy for identifying frequently changed, and thus error-prone source files, is much more efficient than ignoring compiler warnings and picking files randomly.

Table 4 Percentage of files needed to locate revision numbers in the ideal case and using GCC warnings

	Project A		Project B		Project C		Project D		Project E	
X% of V	Ideal	GCC	Ideal	GCC	Ideal	GCC	Ideal	GCC	Ideal	GCC
10%	*8%*	*8%*	*4%*	*12%*	*2%*	*8%*	*3%*	*10%*	*1%*	*4%*
20%	*8%*	*8%*	*4%*	*15%*	*4%*	*14%*	*8%*	*13%*	*4%*	*5%*
30%	*8%*	*8%*	*8%*	*15%*	*5%*	*19%*	*15%*	*23%*	*5%*	*10%*
40%	***15%***	***15%***	***12%***	***23%***	***9%***	***22%***	***23%***	***33%***	***9%***	***20%***
50%	**31%**	**31%**	**15%**	**35%**	**14%**	**29%**	**30%**	**40%**	**11%**	**33%**
60%	39%	46%	23%	39%	20%	40%	38%	53%	16%	44%
70%	46%	54%	31%	54%	28%	55%	53%	65%	24%	61%
80%	69%	69%	46%	62%	41%	64%	70%	75%	35%	76%
90%	*85%*	*85%*	*65%*	*89%*	*61%*	*83%*	*85%*	*88%*	*58%*	*86%*

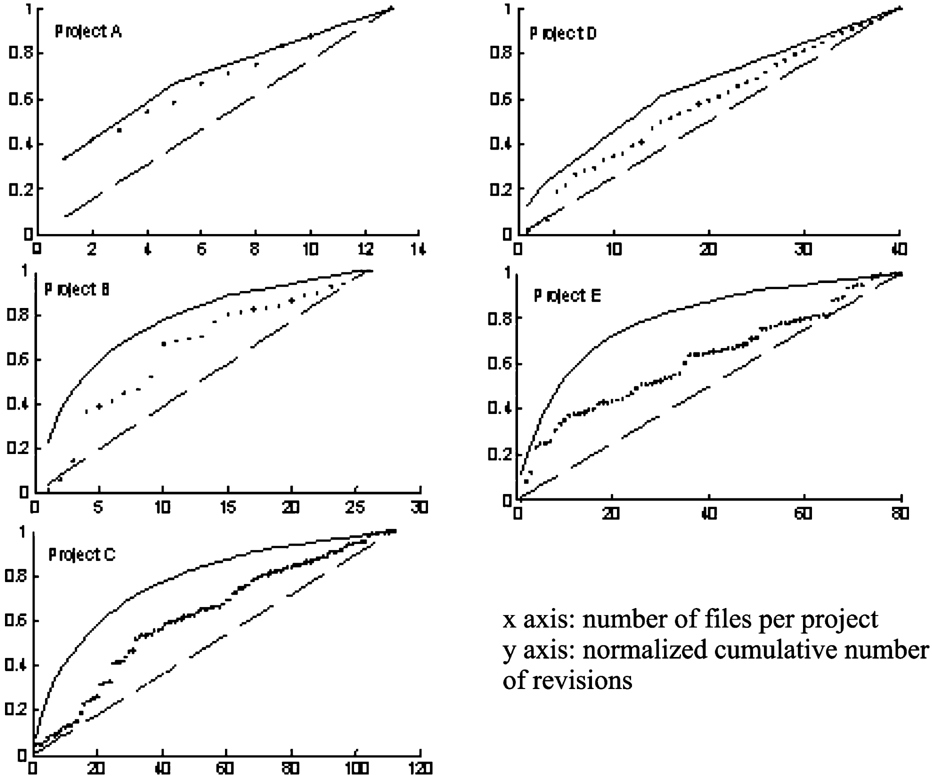

Fig. 3 Cumulative plots of the number of revisions per project

Table 4 is a numerical representation of the plots in Fig. 2. It shows the percentages of files per project that are needed to locate files that contain a given percentage of revision numbers (*V* stands for the sum of the revision numbers of all files in a project). For each project we have both the ideal (lowest possible) number and the number obtained by using GCC warnings. Even though the percentages obtained by GCC warnings are not always very close to the ideal ones, they are a good first indicator for finding the files that contain about half of the revision numbers (and presumably also have of the defects) in a project. Fig. 3.

Table 5 Summary of the results

	Average	99% confidence interval	
Cross correlation coefficient between *W* and *D* (α=0.01)	0.308	[0.163. 0.418]	
		Lower bound	Upper bound
Percentage of files in a project which contain half of the revision numbers	20%	11%	31%
Percentage of files in a project which contain half of the revision numbers predicted using compiler warnings	34%	29%	40%

6 Discussion of the Results

Table 5 summarizes the results of our study. It is evident that GCC warnings are correlated with revision numbers of files; thus, assuming that frequently changed files will show a higher fault incidence, for the projects under scrutiny compiler warnings are useful for early identification of defect prone source files. The main finding of this research is:

- GCC warnings are correlated in a project-independent way with revision numbers of source files. This implies that, given the relationship between number of changes and defects, if developers identify source files with a high number of compiler warnings then these files are more likely to contain defects.

A more indirect finding, which holds only if defect proneness of source files is correlated with version numbers, is the following:

- GCC warnings are useful to locate source files with a high presence of defects. This knowledge can be very useful during testing and maintenance, as GCC warnings provide testers the information which source files of a project should be tested first and more extensively. Several studies showed that these files would show fewest defects during operation and therefore reduce significantly maintenance effort and costs (Ohlsson and Fenton 2000).

Even after a software system is in operation it is very important to know *a priori* which part of the code could possibly cause failures, and should be inspected extensively. Compiler warnings give to the developers a clue where in the software system to inspect the code and possibly eliminate software defects which have not yet appeared as failures to the user.

We have to be well aware that **(a)** our data come from the telecommunication domain and that **(b)** warnings are generated by the GCC compiler. In principle, we cannot generalize our results to a different domain and to non-GCC compilers. This has to be addressed by subsequent analyses.

It is true, though, that most application domains dealing with embedded systems, like the telecommunication domain, pose similar issues to developers; it is our conjecture that similar results are likely to be obtained in other projects dealing with embedded systems and developed in C++.

7 Threats to Validity

In this research there are several threats to validity that have to be addressed. First of all, we base our analysis on the assumption that the revision number of a file is a good predictor for its number of defects: Prior research (Graves et al. 2000; Ostrand et al. 2004) supports this assumption. In particular, Graves *et al.* come the conclusion that the number of changes of a file is at least as good for defect prediction than size measures (lines of code or complexity) or a so called “stable model”, which is based on and extrapolates historic defect data. Since the experimental settings of the work of Graves *et al.* show some similarities (telecommunication domain, C language, changes of a file are due to bug fixes and addition of features) with the context of this study it seems reasonable to transfer their findings to this work. However, we are well aware that in doing so we could be misled and arrive at wrong conclusions. Only by repeating this experiment and using the real number of defects as dependent variable we can gain further confidence in our results.

Another threat to validity is that we are not able to control the influence of other variables, in particular size, on our results. One may object that size plays an important role in our analysis, since larger files probably produce more compiler warnings and are subjected to more changes. Therefore, we should consider a kind of warning density metric for correlation with revision numbers. Moreover, it could happen that compiler warnings are correlated with size, and size itself is correlated with revision numbers. In this case the correlation of warnings with revision numbers would be merely artificial and only due to their common correlation with size. Since we do not have access to size data we are not able to address these issues using statistics and by controlling or blocking the variables of interest. However, in this research we do not aim at deriving a causal model for the underlying relationship between compiler warnings and software defects. We simply want to analyze whether practitioners can use compiler warnings for identifying defect prone files. Such information is still useful from a practical point of view, regardless of the true relationship between the variables. However, since we do not know the cause of the relationship, there may be other metrics, which are more efficient and as easy to collect as compiler warnings. For sure we have to address these issues, in particular the use of a warning density metric instead of the total number of warnings in a future experiment. Then, we plan also to collect the number of warnings after each version in a stepwise manner in order to see whether new warnings have been introduced due to the changes in the previous version and how they correlate with the number of previous changes. This would allow us to understand how such correlation evolves over time, for example if it is stable (as we assume this in this work). We plan also to analyze in depth the relationship between specific changes made to a source file and different kinds of warnings introduced or removed by such changes.

Another threat to external validity is that we do not have detailed knowledge about the skills of the developers involved in the projects: They correspond in general to a university degree in computer science and some years of experience in developing embedded systems. Differences in developers' background and experience are a factor we do not control within this case study and thus limit its generalizability.

Finally, it remains to be investigated which kind of defects compiler warnings account for. In particular, for developers it would be very interesting to know whether for example compiler warnings are merely a symptom of complex, sloppy written, and bad structured code or if they indicate most of the time code issues that are likely to develop into actual defects.

8 Conclusions and Future Work

Detecting early the most defect prone files of a software project is critical for reducing the cost of non-quality in software development. In this paper we show that in five C++ projects there is a significant correlation between GCC warnings and the revision number of files, which we use as an indicator for software defects. Moreover, such correlation can be generalized with meta-analysis across all projects still maintaining its significance.

Using cumulative graphs based on compiler warnings we are able to localize early—beginning with the first compilation of source code—defect prone source files in a project. Such cumulative plots are useful and intuitive instruments for the developer and tester to get an early and immediate impression where in the code base to look for files to inspect and test extensively.

Collection of compiler warnings costs almost no effort and time. Therefore, they can be used effectively as a first indicator of which part of the code to inspect and test extensively. This can save both time and effort, for rewriting and redesigning code during development and most of all for delivering reliable, high quality code.

To summarize our results, in the target application domain it appears to be effective to give a very simple directive to software developers "Try to avoid compiler warnings!" and to software testers "Test first files with a high number of compiler warnings!"

This may save you a lot of time for debugging and help you in developing more reliable code.

As mentioned several times the results obtained in our study are valid only in the specific context of the application domain and for the GCC compiler. Further studies are needed to see whether extensions to different domains and compilers lead to similar conclusions.

For simplicity in this analysis we have only considered the warnings as a whole. We have not made any distinction on the different kinds of warning messages. A subsequent analysis could verify if a more refined analysis, which takes into account the different kinds of compiler warnings could yield better results, paying off the extra effort to accomplish them.

Acknowledgment The authors would like to thank the three anonymous referees for their valuable comments and ideas.

References

Basili VR, Lindvall M, Shull F (2002) A light-weight process for capturing and evolving defect reduction experience. Proceedings of 8th IEEE International Conference on Engineering of Complex Computer Systems, Los Alamitos, California, IEEE, 129–132

Boehm B, Basili, V (2001) Software defect reduction Top 10 list. Comput 34(1):135–137

Bontempi G (2003) Resampling techniques for statistical modeling. Université Libre de Bruxelles, Département d'Informatique, Boulevard de Triomphe-CP 212 http://www.ulb.ac.be/di/map/gbonte/ecares/boot1.pdf

Efron B, Tibshirani RJ (1993) An Introduction to the bootstrap. Chapman and Hall ch. 12–14

El Emam K, Melo W, Machado JC (2001) The prediction of faulty classes using object-oriented metrics. J Syst Softw 56:63–75

Fenton N, Neil M (1999) Software metrics and risks. Proceedings of FESMA 2nd European Software Measurement Conference. GNU GCC home page. 2004. Manual for GCC http://gcc.gnu.org

Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Trans Softw Eng 26(7):653–661

Hall T, Fenton N (1997) Implementing effective software metrics programs. IEEE Softw 14(2):55–65

Hedges LV, Olkin I (1985) Statistical methods for meta-analysis. Academic, Orlando, 230–235

Johnson PM, Disney AM (1998) Investigating data quality problems in the PSP. Proceedings of Sixth International Symposium On The Foundations Of Software Engineering (SIGSOFT'98), Orlando

Johnson PM, Kou H, Agustin J, Chan C, Moore C, Miglani J, Zhen S, Doane W (2002) Beyond the personal software process: metrics collection and analysis for the different disciplined. Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon

Krishnan MS (1993) Quality engineering: cost, quality and user satisfaction of software products: an empirical analysis. Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering—volume 1. Toronto, Ontario, Canada, 400–411

Miller J (2000) Applying meta-analytical procedures to software engineering experiments. J Syst Softw 54:29–39

Mišić VB, Tešić DN (1997) Estimation of effort and complexity: an object-oriented case study. J Syst Softw 41(2):133–143

Nesi P, Querci T (1998) Effort estimation and prediction of object-oriented systems. J Syst Softw 42(1):89–102

Ohlsson N, Fenton, N (2000) Quantitative analysis of faults and failures in a complex software system. IEEE Trans Softw Eng 26(8):797–814

Ostrand TJ, Weyuker EJ, Bell RM (2004) Using static analysis to determine where to focus dynamic testing effort. Proceedings of the Second International Workshop on Dynamic Analysis (WODA 2004), Edinburgh, Scotland

Pearson K (1907) Mathematical contributions to the theory of evolution. XVI. On further methods of determining correlation. Drapers' Company Research Memoirs (Biometric Series 4), Cambridge University Press

Rønning OM, Guldvog B (1998) Stroke units versus general medical wards, I: twelve- and eighteen-month survival. Stroke 29:58–62

Shull F, Tesoriero R et al (2002) What we have learned about fighting defects. Proceedings of the International Symposium on Software Metrics (Metrics 2002), Ottawa, Canada

Siegel S, Castellan NJ (1988) Nonparametric statistics for the behavioral sciences. McGraw-Hill, ch. 9.3, ch. 6

Sillitti A, Janes A, Succi G, Vernazza T (2003) Collecting, integrating and analyzing software metrics and personal software process data. Proceedings of the EUROMICRO 2003

Southard CR, Haggard J, Crider ME, Whiteheart SW, Cooper RL (2000) Influence of serotonin on the kinetics of vesicular release. Brain Res 871:16–28

Stringfellow C, Andrews AA (2002) An empirical method for selecting software reliability growth models. Empir Softw Eng 7(4):297–318

Succi G, Pedrycz W, Djokic S, Zuliani P, Russo B (2005) An empirical exploration of the distributions of the Chidamber and Kemerer object-oriented metrics suite. Empir Softw Eng. Kluwer 10(1):81–104

Tian J (2000) Risk Identification Techniques for Defect Reduction and Quality Improvement. Software Quality Professional, 2(2)

Weber R (2003) Statistics http://www.statslab.cam.ac.uk/~rrw1

Williams L, Maximilien EL, Vouk M (2003) Test-driven development as a defect-reduction practice. Proceedings of the 14th International Symposium on Software Reliability Engineering, Denver, Colorado

Raimund Moser is an assistant professor in Software Engineering at the Faculty of Computer Science, Free University of Bolzano-Bozen, Italy. He received an MSc in Theoretical Physics from University of Innsbruck, Austria, in 2000. His main research interests include experimental Software Engineering and software metrics, in particular static analysis and process metrics for creating quality and effort estimation models targeted to agile development projects. Currently, he is involved in several projects that aim at monitoring and improving agile and open source software development. At the same time he is pursuing a PhD degree in Software Engineering at the University of Genoa, Italy.

Barbara Russo received her PhD degree in Mathematics from the University of Trento, Italy, in 1991 and 1996, respectively. She was visiting researcher at the May Plank Insitute für Mathematik in Bonn (D) and at the University of Liverpool (UK). She is currently an associate professor of the Faculty of Computer Science at the Free University of Bolzano-Bozen, Italy. Her current research interest focuses on various topics of Software Engineering specifically related to software metrics, software reliability, data analysis, and prediction systems with particular focus on Open Source software and agile development.

Giancarlo Succi is Professor with Tenure at the Free University of Bolzano-Bozen, Italy, where he directs the Center for Applied Software Engineering. Before joining the Free University of Bolzano-Bozen, he has been Professor with Tenure at the University of Alberta, Edmonton, Alberta, Associate Professor at the University of Calgary, Alberta, and Assistant Professor at the University of Trento, Italy. The research interest of Giancarlo Succi involve multiple areas of software engineering, including open source development, agile methodologies, experimental software engineering, software engineering over the Internet, and software product lines and software reuse. Giancarlo Succi is a Fulbright Scholar.