A Software Quality Predictive Model

Elisabetta Ronchieri and Marco Canaparo
INFN CNAF, Viale Berti Pichat 6/2, Bologna, Italy

Keywords: Quality, Model, Software Construction.

Abstract: Software development is facing the problem of how to improve the quality of software products. The lack of quality can easily lead to major costs and delays in the development and maintenance of the software. Its improvement can be guaranteed by both the definition of a software quality model and the presence of metrics that are designed and measured to plan and monitor productivity, effectiveness, quality and timing of software. Integrating the metrics into the model contributes to collecting the right data for the handling of the analysis process and to establishing a general view to the control of the overall state of the process. This paper aims at introducing a mathematical model that links software best practices with a set of metrics to predict the quality of software at any stage of development. Two software projects have been used to analyze the defined model as a suitable predictive methodology in order to evaluate its results. The model can improve the level of the software development process significantly and contribute to achieving a product of the highest standards. A replication of this work on larger data sets is planned.

1 INTRODUCTION

The software development life cycle is often very expensive because of the growing overall complexity and the average size of software products. Over the past decades software engineering researchers have put a lot of effort into software quality, being considered as important as the delivery of the product within scheduled budget and time. Quality, in fact, represents the degree of excellence that is measurable in a given product (IEEE90, 1990). Quality requirements are increasingly becoming determining factors in selecting between design alternatives during software development. In order to appraise the quality of any software project, quality estimation models are necessary, which help the development team to track and detect potential software defects during development process and to save effort that is later required for the maintenance of the product (Khoshgoftaar and Seliya, 2003). Furthermore, the presence of metrics is recommended in order to plan and monitor productivity, effectiveness, quality and timing of software. The continuous application of measurement-based techniques to the software development process supplies meaningful information to improve products and process (DeMarco, 1982). Integrating the metrics into the model contributes to the assessment and the prediction of software quality. In addition to that, metrics are input to control and management of general planning activities. In this paper, we propose a general approach and a particular solution to the problem of improving the software quality. The main idea is to connect software best practices with a set of metrics into a mathematical model in such way that the quality of software at any stage of development is well predicted. Best practices (Khoshgoftaar and Seliya, 2003) in this context refers to the software structure, the construction of the code (McConnell, 1996), deployment, testing, and configuration management (Wingerd and Seiwald, 1998) in order to obtain a maximum of maintainability, in terms of adaptability, portability and transferability, during the ongoing product life cycle. As concerns metrics (Coleman et al., 1994), they derive from both best practices and static analysis. The following categories have been taken into consideration: file and code conventions, software portability and static analysis. As this paper focused on the early phases of the software development life cycle, only static metrics have been analyzed (Chhabra and Gupta, 2010), leaving dynamic ones (such as feasibility and NPATH evaluation) for future work since they concern the late stage (Debbarma et al., 2012) and are based on the data collected during an actual execution of the system (Chhabra and Gupta, 2010). Two software projects that we build

Ronchieri E. and Canaparo M..
A Software Quality Predictive Model.
DOI: 10.5220/0004492001860197
In *Proceedings of the 8th International Joint Conference on Software Technologies* (ICSOFT-EA-2013), pages 186-197
ISBN: 978-989-8565-68-6

daily, one on storage management system (StoRM[1]) and another on virtual resource provisioning on demand (WNoDeS[2]), have been used to analyze the defined model with a predictive technology called discriminant analysis (Munson and Khoshgoftaar, 1992) and based on risk-threshold (Pighin and Zamolo, 1997). Established this work, the model has been proving so far to have all the capabilities of enhancing the development software process. We therefore are confident that in the future project managers and developers adopt this solution as particularly helpful for evaluating their projects and controlling the overall health of the process. This paper is an opportunity to expose our ideas and share our experience with researchers that think and try out things in the same area. We have reached a point where we need to involve others in a constructive manner in order to move forward in our understanding of software engineering. The paper is organized as follows: Section 2 describes some of the software best practices that have contributed to define the core of the model, whilst Section 3 summarizes the metrics considered. Section 4 provides the definition of the mathematical model that links software data entities to a set of well-known metrics. Section 5 illustrates the experimental results. Section 6 describes related works and Section 7 concludes with a brief of discussion of future work.

2 SOFTWARE BEST PRACTICES

Developers have been striving to improve software quality for decades. Despite this, projects keep failing from familiar causes as poor design and inadequate testing (Kopec and Tumang, 2007) as well as the lack of a widespread well-known recipe (Brooks, 1995). From software development experience in several projects (Ronchieri et al., 2009), a set of best practices have been selected in relation to their capabilities of determining projects' success and offering the greatest return, but that yet seem to be violated more often than not. Some of the identified best practices have been denoted earlier either in different contexts or with different pre-requisites.

The best practices considered in this paper are described below. Software Structure is the initial stage of developing an application. Best practice includes the usage of one of the existing software structures known in literature (Top et al., 2004) such as control flow, data flow, file and code conventions (Merlo et al., 1992), (Mengel and Tappan, 1995). Configuration management involves knowing the state of all artifacts that make up a project, managing the state of those artifacts, and releasing distinct versions of a system. Best practices for configuration management consider, for example, the application of change code on a new branch, the creation of a branch only when necessary, the application of change propagation, and the usage of common build tools (Wingerd and Seiwald, 1998). Construction of the code occupies the central role in software development and often represents the only accurate description of the software; hence, it is imperative that code be of the highest possible quality (McConnell, 1996). Best practices for the construction of the code include daily builds and continuous integration (Fawler et al., 1999). Testing is an integral part of software development. Best practices include the planning of test cases before coding starts and the development of test cases whilst the application is being designed and coded (Majchrzak, 2010). Deployment is the final stage of releasing an application for users. A best practice is the usage of a deployment procedure (Jansen and Brinkkemper, 2006), (Flissi et al., 2008), (Elbaum, 2005).

By following these best practices that seem obvious once used, a software project increases its chances of being completed successfully. However, adopting some of them can be very challenging, especially in relation to the construction of the code and testing: the former because it requires a certain amount of effort in order to perform a good initial design; the latter on the grounds that testing is time consuming, too inconsistent to be effective, error prone and inaccurate.

3 METRICS DESCRIPTION

Numerous empirical studies confirm that many software metrics can be used to evaluate quality aspects of general interest, like maintainability and correctness (Fenton, 1990). The metrics considered into this paper derives from the best practices and static analysis.

From the software structure best practice, the metrics of file and code conventions are considered. The main purpose of these metrics is measuring how well a project is organized, focusing on the files and directories structure. Every software project is characterized by a main directory underneath which a number of files and folders are located. Some file names well fit into every kind of project, such as `AUTHORS` that contains the names of the authors of the project with their roles such as developer, and project leader, and `CREDITS` that contains a set of acknowledgments.

[1] `http://storm.forge.cnaf.infn.it/`
[2] `http://web.infn.it/wnodes/`

Furthermore, the main directory can contain several types of subdirectories, the most common of which are named: `bin` for essential user command binaries, `doc` for documentation files, and `tests` for code to evaluate functionalities. Specific features can be added according to the language by which the project is implemented. For example, a software code that needs `autotools` for the building could have a directory called `m4` with all the customized macros. In a java web project the `WEB-INF` and `META-INF` directories might be found.

From the configuration management best practice, the software portability metric is considered that refers to the software capability of being installed in various platforms each of which is characterized by a combination of operating system, kernel architecture, and compiler version. Each platform can be identified as a string of the form `os_arch_compile`: the `os` substring is about the OS family e.g. `slc5`; `arch` stands for CPU architecture e.g. `ia32`, `x86_64`; finally, the `compile` substring provides information about the type and version of the compiler used, e.g. `gcc346`. The portability metric depends on how the software project is distributed and their objective is to measure the number of platforms on which a module can be installed.

Finally, metrics about the static analysis of the code are reckoned with (Chidamber and Kemerer, 1994): SLOCCount shows the number of lines of code; Findbugs indicates the number of bugs found during the build or test; Findbugs rate shows the percentage of modules that have successfully passed the threshold defined by the user; WMC (Weighted Methods per Class) provides an index of the total complexity of a class' methods; DIT (Depth of Inheritance Tree) provides for each class a measure of the inheritance levels from the object hierarchy top (e.g., in Java where all classes inherit Object the minimum value of DIT is 1); NOC (Number of Children) measures the number of immediate descendants of the class; CBO (Coupling Between Object classes) represents the number of classes coupled to a given class (efferent couplings) that can occur through method calls, field accesses, inheritances, arguments, return types, and exceptions; NMP (Number of Public Methods) counts all the methods in a class that are declared as public. It can be used to measure the size of an API provided by a package.

4 MODEL DEFINITION

The core of our approach is the model. In literature UML diagrams, code, textual documents and mathematical description (Harel and Rumpe, 2004) are formalisms to express models. This work has chosen to only use the mathematical description formalism to express various levels of abstraction the fundamental concepts of software engineering, best practices, and metrics, with a notation and concepts deriving mostly from the set and graph theories. In the following sections a subset of the software best practices and metrics that are described in Section 2 and Section 3 are taken into account for the construction of the model.

4.1 Fundamental Concepts

The fundamental concepts are file, directory, module, and component, the hierarchy of which is shown in Figure 1, according to which components can contain files that are not in directories and modules, modules can contain directories, and files cannot be contained in either directories or modules.

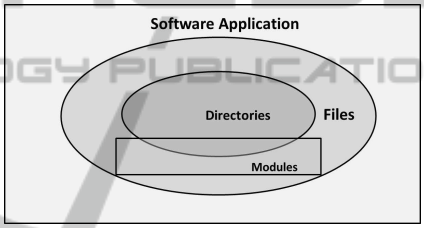

Figure 1: Fundamental concepts.

A file f is a block of information, the block of which is a set of text lines. Let L be a set of text lines of a given file. Consider the function $NumLines : SA \rightarrow \mathbb{N}$ such that

$$NumLines(f) = l \tag{1}$$

returns the number of lines l of the file f. A software application $SA = \{f_1, f_2, ..., f_m\}$ is a set of files with $m = |SA|$ and $m \in \mathbb{N}$.

Let $SD = \{d_1, d_2, ..., d_k\}$ be a set of directories with $k = |SD|$ and $k \in \mathbb{N}$, and where a directory d is defined as a collection of some files and of other directories identified by a name - is a triple $(name, D, \{f_1, f_2, ..., f_l\})$ where *name* is the directory identifier, and $\{f_1, ...f_l\}$ is a subset of SA, and $D \subseteq SD\{name\}$.

A module $m = \{d_1, d_2, ..., d_j\}$ - a logical collection of directories - is a subset of SD. Let $SM = \{m_1, m_2, ..., m_h\}$ be a set of modules with $h = |SM|$ and $h \in \mathbb{N}$.

A component $c \subseteq SD \cup SA$ - a logical portion of the overall software application - is a subset of directories or files. Let $SC = \{c_1, c_2, ..., c_b\}$ be a set of components with $b = |SC|$ and $b \in \mathbb{N}$.

The function $CompToFiles : SC \rightarrow \mathcal{P}(SA)$ such that

$$CompToFiles(c) = \{f_1, ..., f_q\} \tag{2}$$

gets the set of files $\{f_1,...,f_q\}$ that are in the component c. Whilst the function $CompToDirs : SC \rightarrow \mathcal{P}(SD)$ such that

$$CompToDirs(c) = \{d_1,...,d_{qq}\} \quad (3)$$

gets the set of directories $\{d_1,...,d_{qq}\}$ that are in the component c.

4.2 Best Practices Modeling

The selected best practices (as reported in Section 2) are related to software structure, configuration management, construction of the code, testing and deployment.

4.2.1 Software Structure

Amongst the software structures file and code conventions have been modeled.

The file structure expresses the structure of the software design. It recommends putting files that are associated with a component and work together into the same directory.

A component will often contain various file types for storing source code, object code, scripts, binary executables, data, and documentation. Let $FT = \{executable, object, source\ \ code, batch, text, work\ processor, library, archive\}$ be a set of file types.

File name extensions are commonly used to distinguish amongst different kinds of files (e.g., `.h`, `.c`, `.hpp`, `.java`, `.sh`). Let $SE = \{se_1,...,se_n\}$ be a set of standard extensions (shown in Table 1) with $n = |SE|$ and $n \in \mathbb{N}$ that are considered for the file types.

Table 1: Standard extensions of a set of file types.

File Type	Standard Extension
executable	bin, jar, none
object	obj, o
source code	c, h, py, java, wsdl, cpp, hpp
batch	sh, csh
text	doc, txt, pdf, ps
word processor	doc, tex, wp, rrf
library	a, so
archive	tar, rpm, deb

Let $FN = \{README, CHANGELOG, INSTALL, LICENSE, MAINTENANCE\}$ be a set of file names that are recommended (i.e., `README` file describes the module and its use; a `CHANGELOG` file lists what is finished and what needs to be done; an `INSTALL` file explains how to install the module; a `MAINTENANCE` file explains how to maintain the module files; a `LICENSE` file contains license module information), the type of which is `text` with `txt` as standard extension. The function $IsFileFnIdentified : FN \times SA \rightarrow \{0,1\}$ by taking

$$IsFileFnIdentified(fn, f) = \begin{cases} 1 & \text{if } f \in fn \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

determines if the file f is called fn.

Furthermore, a component should contain at least the following high-level directories with fixed names in relation to the used programming language: `src` containing source code, replaced by `lib` for Perl modules and by `<package name>` for Python modules; `test` containing test source code, replaced by `lib` for Perl modules; `interface` for public interface files such as files with suffixes `.wsdl`, or `.h`; `config` for configuration and scripting files such as files with suffixes `.conf`, `.ini`, `.sh`, `.csh`; `doc` containing documentation files such as release notes, and api references. The fixed structure allows the automation of tasks, such as directory creation, compliance monitoring, file collection, and packaging. Let $DN = \{dn_1, dn_2, ..., dn_{mm}\}$ be a set of directory names with $mm = |DN|$ and $mm \in \mathbb{N}$. The function $IsDirDnIdentified : DN \times SD \rightarrow \{0,1\}$ by taking

$$IsDirDnIdentified(dn, d) = \begin{cases} 1 & \text{if } d(0) \in dn \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

determines if the name of the directory $d(0)$ is called dn.

Let $PL = \{java, c, c++, perl, python, python\}$ be a set of programming languages that are considered in the paper.

The code structure expresses the structure of the software design. It recommends producing consistent, clear code by using effective coding style (Oman and Cook, 1988), following the conventions of the adopted programming language (Li and Prasad, 2005), (SunMicrosystems, 1997), (Butler, 2012), (Fang, 2001), (Rossum and Warsaw, 2001), (GCCTeam, 2012) and using formatting rules to display the structure of the code.

Let $ST = \{st_1, st_2, ..., st_e\}$ be a set of styles with $e = |ST|$ and $e \in \mathbb{N}$. Here the function $StyleOfLangToFiles : PL \times ST \rightarrow \mathcal{P}(SA)$ such that

$$StyleOfLangToFiles(pl, st) = \{f_1,...,f_y\} \quad (6)$$

gets the set of files $\{f_1,...,f_y\}$ with $y \in \mathbb{N}$ that follows the correct style st in accordance with the programming language pl.

Let $CT = \{ct_1, ct_2, ..., ct_w\}$ be a set of conventions with $w = |CT|$ and $w \in \mathbb{N}$. The function $ConvOfLangToFiles : PL \times CT \rightarrow \mathcal{P}(SA)$ such that

$$ConvOfLangToFiles(pl, ct) = \{f_1,...,f_o\} \quad (7)$$

gets the set of files $\{f_1,...,f_o\}$ with $o \in \mathbb{N}$ that follows the correct convention ct in accordance with the programming language pl.

Let $FR = \{fr_1, fr_2, ..., fr_t\}$ be a set of formatting rule with $t = |FR|$ and $t \in \mathbb{N}$. Finally the function $FruleOfLangToFiles : PL \times FR \rightarrow \mathcal{P}(SA)$ such that

$$FruleOfLangToFiles(pl, ft) = \{f_1,...,f_p\} \quad (8)$$

gets the set of files $\{f_1,...,f_p\}$ with $p \in \mathbb{N}$ that follows a formatting rule fr in accordance with the programming language pl.

4.2.2 Configuration Management

For this best practice the concepts of branch and build have been modeled.

A branch b - a variant of code lines - is an element of SA. Let $SB = \{b_1, b_2, ..., b_a\}$ be a set of branches with $a = |SB|$ and $a \in \mathbb{N}$.

Let $CC = \{cc_1, cc_2, ..., cc_s\}$ be the set of code changes with $s = |CC|$ and $s \in N$. The function $IsChangecodeOnNewBranch : CC \rightarrow \{0,1\}$ such that

$$IsChangecodeOnBranch(cc) = \begin{cases} 1 & \text{if } cc \text{ is on a branch} \\ 0 & \text{otherwise} \end{cases} \quad (9)$$

determines if the code change cc is on new branch.

Once branches are created to handle code changes, the change propagation across branches must be factored in. The function $ChangecodeOnBranchToBranches : CC \times SB \rightarrow \mathcal{P}(SB)$ such that

$$ChangecodeOnBranchToBranches(cc) = \{b_1, ..., b_{ss}\} \quad (10)$$

gets the set of branches $\{b_1, ..., b_{ss}\}$ with $ss \in \mathbb{N}$ that contains the code change cc.

A build is the business of constructing usable software from original source files. It is based on source files and the tools to which they are input, and characterized by producing the same result. The build tools, examples of which are shown in Table 2, are typically linked to the used programming language and are able to support several archive formats.

Table 2: Build tools.

Language	Tool
java	maven, ant
c++, c	autotool, Cmake, make
python, perl	autotool, Scons

Let $BT = \{bt_1, bt_2, ..., bt_{ff}\}$ be a set of build tools with $ff = |BT|$ and $ff \in \mathbb{N}$. The function $LangToBuildtools : PL \rightarrow \mathcal{P}(BT)$ such that

$$LangToBuildtools(pl) = \{bt_1, ..., bt_x\} \quad (11)$$

gets the set of build tools $\{bt_1, ..., bt_x\}$ with $x \in \mathbb{N}$ that are associated to the program language pl. The function $SupportsCompBuildtool : BT \times SC \rightarrow \mathbb{N}$ such that

$$SupportsCompBuildtool(bt, c) = \begin{cases} 1 & \text{if } c \text{ uses } bt \\ 0 & \text{otherwise} \end{cases} \quad (12)$$

determines if the component c uses the build tool bt. Furthermore, the function $BuildtoolOfLangToComps : PL \times BT \rightarrow \mathcal{P}(SC)$ such that

$$BuildtoolOfLangToComps(pl, bt) = \{c_1, ..., c_g\} \quad (13)$$

gets the set of components $\{c_1, ..., c_g\}$ with $g \in \mathbb{N}$ that use the given build tool bt with respect to the programming language pl, where $bt \in LangToBuildtools(pl)$ (see Eq. 11). The function $CompWithTargetArchiveToArchives : BT \times SC \rightarrow \mathcal{P}(SE)$

$$CompWithTargetArchiveToArchives(bt, c) = \{se_1, ..., se_h\} \quad (14)$$

gets the set of standard extensions $\{se_1, ..., se_h\}$ of the archive file type with $h \in \mathbb{N}$ that are provided by the component c with respect to the target archive supported by its build tool bt.

Let OS be the set of operating systems. Let CMP be the set of compilers. Finally, let MA be the set of machine architectures. $PLAT \subseteq OS \times CMP \times MA$ is a set of platforms. The function $SupportsCompPlat : PLAT \times SC \rightarrow \{0,1\}$ such that

$$SupportsCompPlat(plat, c) = \begin{cases} 1 & \text{if } c \text{ runs on } plat \\ 0 & \text{otherwise.} \end{cases} \quad (15)$$

gets the component c that supports the platform $plat$, whilst the function $CompToPlats : SC \rightarrow \mathcal{P}(PLAT)$ such that

$$CompToPlats(c) = \{plat_1, ..., plat_v\} \quad (16)$$

gets the set of platforms $\{plat_1, ..., plat_v\}$ with $v \in \mathbb{N}$ that are supported by the component c.

4.2.3 Construction of the Code

Here, the concepts of software dependency, class, method, function and procedure have been modeled.

The software dependencies considered in the paper are amongst components (see Figure 2), inside a given component (see Figure 3).

Dependencies amongst components DAC is a directed graph composed of a set of vertices that represent components and a set of edges. Each edge connects two components c_i, c_j and the sense of direction from c_i to c_j is specified by an ordered pair $< c_i, c_j >$. A path in DAC is a set of components $< c_1, c_2, ..., c_n >$ such that $< c_i, c_{j=i+1} >$ for each i from 1 to $n \in \mathbb{N}$ is an edge in DAC.

Dependencies inside a component DIC is a directed acyclic graph composed of a set of vertices that represent files and a set of edges. Each edge connects two files f_i and f_j where f_j is adjacent to f_i, and the

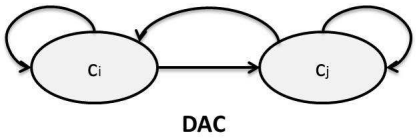

Figure 2: Dependencies amongst components: c_i depends on c_j with $j = i+1$.

sense of direction from f_i to f_j is specified by an ordered pair $< f_i, f_j >$. A path in *DIC* is a set of files $< f_1, f_2, ..., f_n >$ such that $< f_i, f_{i+1} >$ for each i from 1 to $n \in \mathbb{N}$, is an edge in *DIC*. In this cases cycles are not as problematic as in the previous case, due to the fact that all dependencies are internal and therefore do not increase the overall complexity of the system. Furthermore they are common; e.g. an I/O component may have a file $F1$ with routines that provide high-level interfaces and another $F2$ that contains the low-level implementation. In such a situation is it common that not only $F1$ depends on $F2$, but also that $F2$ depends on $F1$ to propagate common error situations.

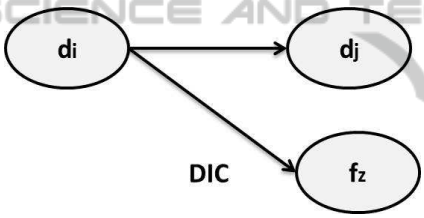

Figure 3: Dependencies inside a component: d_i depends on f_z and d_j.

Circular dependencies unfortunately do happen in real programs, and therefore they cannot be excluded, though they reduce maintainability of a software program due to increase interrelations among components.

4.2.4 Others

For the testing and deployment best practices the concepts of daily build, test cases and deployment procedures have been modeled.

Let $DH = \{0,1,2,...,24\}$ be a set of daily hours. The function $NumBuilds : DH \rightarrow \mathbb{N}$ such that

$$NumBuilds(dh) = \left\lfloor \frac{dh}{num} \right\rfloor \quad (17)$$

returns how many builds are run daily with $num \in \mathbb{N}$.

Let *DF* and *BU* a set of defined functionalities and a set of discovered bugs during the build or test activity respectively, then $TC = \{tc_1, tc_2, ..., tc_z\}$ is a set of test cases with $z = |TS| \geq |DF| + |BU|$ and $z \in \mathbb{N}$. The function $FileToBugs : SA \rightarrow \mathcal{P}(BU)$ such that

$$FileToBugs(f) = \{bu_1, ..., bu_{ww}\} \quad (18)$$

gets the set of bugs $\{bu_1, ..., bu_{ww}\}$ with $ww \in \mathbb{N}$ that are included in the file f. The function $IsTestcaseForBug : BU \times TC \rightarrow \{1,0\}$ such that

$$IsTestcaseForBug(bu,tc) = \begin{cases} 1 & \text{if } tc \text{ is for } bu \\ 0 & \text{otherwise.} \end{cases} \quad (19)$$

gets the test case *tc* that is for the bug *bu*. Furthermore the function $CompToBugs : SC \rightarrow \mathcal{P}(BU)$ such that

$$CompToBugs(c) = \{bu_1, ..., bu_{bb}\} \quad (20)$$

gets the set of bugs $\{bu_1, ..., bu_{bb}\}$ with $bb \in \mathbb{N}$ that are included in the component *c*, while the function $CompToTests : SC \rightarrow \mathcal{P}(TS)$ such that

$$CompToTests(c) = \{ts_1, ..., ts_{cc}\} \quad (21)$$

gets the set of test cases $\{ts_1, ..., ts_{cc}\}$ with $cc \in \mathbb{N}$ that are included in the component *c*.

Let *DP* be a set of deployment procedures. The function $SupportsCompProcedure : DP \times SC \rightarrow \{1,0\}$ such that

$$SupportsCompProcedure(dp,c) = \begin{cases} 1 & \text{if } c \text{ supports a } dp \\ 0 & \text{otherwise.} \end{cases} \quad (22)$$

gets the component *c* that supports the deployment procedure *dp*, whilst the function $CompToProcedures : SC \rightarrow \mathcal{P}(DP)$ such that

$$CompToProcedures(c) = \{dp_1, ..., dp_{vv}\} \quad (23)$$

gets the set of deployment procedures $\{dp_1, ..., dp_{vv}\}$ with $vv \in \mathbb{N}$ that are supported by the component *c*.

4.3 Metrics Modeling

Here, a subset of the metrics introduced in Section 3 have been modeled.

4.3.1 Software Structure

As concerns the software structure category, a set of metrics have been defined.

In relation to the file structure best practice two metrics have been defined: the former is the Total Number of File Names (TNFN) metric that returns the number of the recommended filenames included in *SC*; the latter is the Total Number of Directory Names (TNDN) metric that returns the number of the recommended directory names included in *SC*.

TNFN relies on the function *CompToFiles* (see Eq. 2) that returns the list of files included in a component, and the function $IsFileFnIdentified$ (see Eq. 4) determines if the name of a given file is amongst the recommended ones. The function $NumFilenames : SC \rightarrow \mathbb{N}$ such that

$$NumFilenames(c) = \sum_{k=1}^{|CompToFiles(c)|} \sum_{j=1}^{|FN|} IsFileFnIdentified(fn_j, CompToFiles(c)_k) \quad (24)$$

returns the number of the recommended filenames defined in FN that are included in the component c, the maximum value of which is $|FN|$, where $CompToFiles(c)_k$ is the k-th file of the component c, and fn_j is the j-th file name $\in FN$. Therefore the function $TNFN : \mathbb{N} \to \mathbb{N}$ such that:

$$TNFN = \sum_{i=1}^{|SC|} NumFilenames(c_i) \quad (25)$$

returns the total number of the recommended filenames that are in SC, where c_i is the i-th component that belongs to SC and $i = 1, ..., |SC|$.

TNDN relies on the function *CompToDirs* (see Eq. 3) that returns the list of directories included in a component, and the function *IsDirDnIdentified* (see Eq. 5) determines if the name of a given directory is amongst the recommended ones. The function $NumDirnames : SD \to \mathbb{N}$ such that

$$NumDirnames(c) = \sum_{k=1}^{|CompToDirs(c)|} \sum_{j=1}^{|DN|} IsDirDnIdentified(dn_j, CompToDirs(c)_k) \quad (26)$$

returns the number of the recommended directories defined in DN that are included in the component c, the maximum value of which is $\geq |DN|$, where $CompToDirs(c)_k$ is the k-th directory of the component c, and dn_j is the j-th directory name $\in DN$. Therefore the function $TNDN : \mathbb{N} \to \mathbb{N}$ such that:

$$TNDN = \sum_{i=1}^{|SC|} NumDirnames(c_i) \quad (27)$$

returns the total number of the recommended directory names that are in SC, where c_i is the i-th component that belongs to SC and $i = 1, ..., |SC|$.

In relation to the code conventions structure best practice the defined metrics focus on determining if files included in a component follow code styles (IsCST), conventions (IsCCT), and formatting rules (IsCFR). They relies on the function *CompToFiles* (see Eq. 2) that returns the list of files included in a component, the function *StyleOFLangToFiles* (see Eq. 6) that returns the files that follows a given style for a specified language, the function *ConvOFLangToFiles* (see Eq. 7) that returns the files that follows a given convention for a specified language, the function *FruleOFLangToFiles* (see Eq. 8) that returns the files that follows a given formatting rule for a specified language.

The function $IsCST : SC \to \{1,0\}$, given the programming language pl and the code style st, such that

$$IsCST(c) = \begin{cases} 1 & \text{if } CompToFiles(c) \in StyleOfLangToFiles(pl, st) \\ 0 & \text{otherwise} \end{cases} \quad (28)$$

determines if the files included in the component c follow the code style st in accordance with the programming language pl. Therefore the function $TST : \mathbb{N} \to \mathbb{N}$ such that:

$$TST = \sum_{i=1}^{|SC|} IsCST(c_i) \quad (29)$$

returns the total number of the files that follow the code style st in accordance with the programming language pl and that are in SC, where c_i is the i-th component that belongs to SC and $i = 1, ..., |SC|$.

The function $IsCCT : SC \to \{1,0\}$, given the programming language pl and the code style ct, such that

$$IsCCT(c) = \begin{cases} 1 & \text{if } CompToFiles(c) \in ConvOfLangToFiles(pl, ct) \\ 0 & \text{otherwise.} \end{cases} \quad (30)$$

determines if the files included in the component c follow the convention ct in accordance with the programming language pl. Therefore the function $TCT : \mathbb{N} \to \mathbb{N}$ such that:

$$TCT = \sum_{i=1}^{|SC|} IsCCT(c_i) \quad (31)$$

returns the total number of the files that follow the convention ct in accordance with the programming language pl and that are in SC, where c_i is the i-th component that belongs to SC and $i = 1, ..., |SC|$.

The function $IsCFR : SC \to \{1,0\}$, given the programming language pl and the formatting rule fr, such that

$$IsCFR(c) = \begin{cases} 1 & \text{if } CompToFiles(c) \in FruleOfLangToFiles(pl, fr) \\ 0 & \text{otherwise.} \end{cases} \quad (32)$$

determines if the files included in the component c follow the formatting rule fr in accordance with the programming language pl. Therefore the function $TFR : \mathbb{N} \to \mathbb{N}$ such that:

$$TFR = \sum_{i=1}^{|SC|} IsCFR(c_i) \quad (33)$$

returns the total number of the files that follow the formatting rule fr in accordance with the programming language pl and that are in SC, where c_i is the i-th component that belongs to SC and $i = 1, ..., |SC|$.

4.3.2 Configuration Management

As concerns the configuration management category, the total number of platforms (TNP) metric is defined relying on the function *CompToPlats* (see Eq. 16) that returns the list of platforms supported by a component. The function $TNP : \mathbb{N} \to \mathbb{N}$ such that:

$$TNP = \| \cap_{i=1}^{|SC|} CompToPlats(c_i) \| \tag{34}$$

returns the total number of platforms supported.

4.3.3 Static

For the static category, a subset of the metrics described in Section 3 are modeled below.

The Total SLOCCount (TSLOCCount) metric relies on the function *NumLines* (see Eq. 1) that returns the number of lines for a given file and *CompToFiles* (see Eq. 2) that returns the list of files included in a component. The function $SLOCCount : SC \to \mathbb{N}$ such that:

$$SLOCCount(c) = \sum_{k=1}^{|CompToFiles(c)|} NumLines(CompToFiles(c)_k) \tag{35}$$

returns the code lines of the component c, where $CompToFiles(c)_k$ is the k-th file of the component c: therefore the function $TSC : \mathbb{N} \to \mathbb{N}$ such that:

$$TSC = \sum_{k=1}^{|SC|} SLOCCount(c_i) \tag{36}$$

returns the total code lines that are in SC.

The Total Findbugs (TF) metric relies on the set BU that contains the discovered bugs, the function *CompToFiles* (see Eq. 2) that returns the list of files included in a component, and the function *FileToBugs* (see Eq. 18) that returns the list of bugs included in a file. The function $Findbugs : SC \to \mathbb{N}$ such that

$$Findbugs(c) = \sum_{k=1}^{|CompToFiles(c)|} |FileToBugs(CompToFiles(c)_k)| \tag{37}$$

returns the number of bugs found in the component c, where $CompToFiles(c)_k$ is the k-th file of the component c: therefore the function $TF : \mathbb{N} \to \mathbb{N}$ such that:

$$TF = \sum_{k=1}^{|SC|} Findbugs(c_i) \tag{38}$$

returns the total code lines that are in SC.

5 EVALUATION

The described model can be verified with the usage of a risk-threshold discriminant analysis, the starting point of which is the measurement of a set of parameters connected to the software products. In our study the parameters are basically best practices and metrics that can identify faults in software components and can be defined as risky parameters for that reason. The identification of these risky parameters and the components which have a high risk to contain faults can be used to process the components before their releasing. The validation steps of the model are specified below.

The process starts with a set of best practices SBP and metrics SMT.

Each best practice i and metric u have been verified in each component j as $x_{i,j}$ ($1 \le i \le t$, $1 \le j \le p$) and $y_{u,j}$ ($1 \le u \le r$, $1 \le j \le p$), being $t = |SBP|$ the number of best practices, $r = |SMT|$ the number of metrics and $p = |SC|$ the number of components.

For each best practice i, the mean value BPM_i and the standard deviation BPS_i estimated on values obtained for all components has been computed as

$$BPS_i = \sqrt{\frac{\sum_{j=1}^{p}(x_{i,j} - BPM_i)^2}{p}} \tag{39}$$

where $BPM_i = \frac{1}{p}\sum_{j=1}^{p}(x_{i,j})$.

For each metric u, the mean value MTM_u and the standard deviation MTS_u estimated on values obtained for all components has been computed as

$$MTS_u = \sqrt{\frac{\sum_{j=1}^{p}(y_{u,j} - MTM_u)^2}{p}} \tag{40}$$

where $MTM_u = \frac{1}{p}\sum_{j=1}^{p}(y_{u,j})$.

For each best practice i and each component j, the values

$$BPS_{i,j} = \frac{|x_{i,j} - BPM_i|}{BPS_i} \tag{41}$$

have been considered as the offset of the best practice evaluated on the j-th component from the best practice mean value BPM_i, normalized on the standard deviation of the best practice BPS_i.

For each metric u and each component j, the values

$$MTS_{u,j} = \frac{|y_{u,j} - MTM_u|}{MTS_u} \tag{42}$$

have been considered as the offset of the metric evaluated on the j-th component from the metric mean value MTM_u, normalized on the standard deviation of the metric MTS_u.

The risk level of best practice i is calculated as

$$BPRL_i = \sum_{j=1}^{p} R_j \cdot BPS_{i,j} \tag{43}$$

being R_j considered 1 if the component j has reported faults, 0 else. All the risk values have been normalized with respect to the sum of all *BPRL*.
The risk level of metric u is calculated as

$$MTRL_u = \sum_{j=1}^{p} R_j \cdot MTS_{u,j} \quad (44)$$

being R_j considered 1 if the component j has reported faults, 0 else. All the risk values have been normalized with respect to the sum of all *MTRL*.
The risk level of component j is computed as

$$CRL_j = \sum_{i=1}^{t} BPRL_i x_{i,j} + \sum_{u=1}^{r} MRL_u y_{u,j} \quad (45)$$

the sum of the best practice value with its risk level, and the metric value with its risk level. All the risk values have been normalized with respect to the sum of all *CRL*.

The risk levels for each metric $MTRL_u$ and best practice $BPRL_i$ are calculated on the basis of the examined components, whilst the risk level of component CRL_j is calculated on the basis of the risk levels of metric $MTRL_u$ and best practice $BPRL_i$. The risk-threshold RT has been taken in the middle of the average values of the risk level of components *CRL* with faults and components without faults, defined as

$$RT = avg(CRL_j)_{j \in faults} + \frac{avg(CRL_j)_{j \in non-faults}}{2}. \quad (46)$$

5.1 Experiment Data Sets Description

The experiments have been carried out processing sources from two software projects: StoRM (STOrage Resource Manager), an implementation of the standard SRM interface for generic disk based on storage system (Zappi et al., 2011), and WNoDeS (Worker Nodes on Demand Service), a solution to virtualize computing resources and to make them available through local, Grid and Cloud interfaces (Salomoni et al., 2011). These projects present files coming from the same environment of development and application fields (that are mainly related to the High Energy Physics community). StoRM is a medium sized system written in different programming languages (i.e., `java`, `c++`, `c`, `python`, and `sh`), whilst WNoDeS is a small system wholly written in `python` and `sh`. Both projects are composed of several software components included in EMI3 Monte Bianco distribution (Aiftimiei et al., 2012): 10 for WNoDeS and 21 for StoRM. For StoRM the following components have been considred: `tStoRM` (Ronchieri et al., 2012) that is a StoRM testing framework, and five `sensor` components that are StoRM monitoring framework. Whilst for WNoDeS there are: `hypervisor` that contains code to interact with the virtualization system, `bait` that requests the instantiation of virtual machines if enough resources are available, `nameserver` that is a sort of information management, `manager` that is an administrative command line, `accounting` that is responsible for providing accounting information of the provided resources, `cli` that is the cloud command line, `cachemanager` that takes care of the cloud resources provisioning, `site-specific` that is the site administrator resolver, and `utils` that contains common code shared amongst the other WNoDeS components.

Table 3: Measured best practices per component: the *bt* in Eq. 12 is *autotool*; the values of Eq. 14 are *rpm* and *tar*; the values of Eq. 16 are *sl*5, *sl*6 and *deb*; the values of Eq. 23 are basically *sl*5, and *sl*6 with the peculiarity of the component *cli* that also contains *deb*.

Components	Eq. 12	|Eq. 14|	|Eq. 16|	Eq. 17	|Eq. 21|	|Eq. 23|
tStoRM	1	2	3	5	10	2
sensor-api	1	2	3	2	2	2
sensor-common	1	2	3	2	0	2
sensor-host	1	2	3	2	2	2
sensor-run	1	2	3	2	0	2
sensor-service	1	2	3	2	0	2
hypervisor	1	2	3	2	0	2
bait	1	2	3	2	0	2
nameserver	1	2	3	2	0	2
manager	1	2	3	2	0	2
accounting	1	2	3	2	2	2
cli	1	2	3	4	8	3
site-specific	1	2	3	2	0	2
utils	1	2	3	2	1	2
cachemanager	1	2	3	2	1	2

Table 4: Measured Metrics per component.

Components	Eq. 26	Eq. ??	Eq. 28	Eq. 30	Eq. 32	Eq. 35	Eq. 37
tStoRM	4	3	1	1	1	14,011	97
sensor-api	3	0	1	1	1	233	0
sensor-common	3	1	1	1	1	158	3
sensor-host	3	1	1	1	1	166	1
sensor-run	3	1	1	1	1	192	0
sensor-service	3	1	1	1	1	191	6
hypervisor	3	2	1	1	1	1,635	12
bait	3	2	1	1	1	1,924	14
nameserver	3	2	1	1	1	1,094	12
manager	3	2	1	1	1	859	12
accounting	3	3	1	1	1	382	0
cli	4	3	1	1	1	1,386	1
site-specific	3	2	1	1	1	352	2
utils	3	3	1	1	1	2,265	26
cachemanager	3	3	1	1	1	2,558	20

The experimental data set (see Table 3, Table 4, and Table 5) have been collected with heterogeneous

Table 5: Parameters and basic statistical data.

Parameters	Total	Mean
TArchives	30	2
TNumBuild	35	2.333
TNumTestCases	26	1.733
TProcedures	31	2.066
TNFN	47	3.133
TNDN	29	1.933
TST	15	1
TCR	15	1
TFR	15	1
TNP	45	3
TSC	27,406	1,827.066
TF	206	13.733

source types in order to highlight similarities and differences amongst development scenarios. Therefore we have selected software components mainly written in `python` and `sh` for each project that have been produced over a period of five years. The analysis have been done on $1,513$ files in 15 components amounting to a total code lines *TSC* of $27,406$. The considered best practices and metrics to create the data sets have been estimated by using a prototype tool that codes the presented model. The data related to faults have been used as the dependent variable of the following study. Here, a software model has been considered fault if at least a fault has been recorded. On account of the complexity of the model, no further inspection on the relationships amongst faulty component, metrics and best practices have been carried out.

5.2 Experimental Results

In this section, a short description of the procedures adopted for the data analysis is reported. The objective has been to validate our model with existing software projects in order to estimate their fault-proneness. The main idea is to start from the analysis of the whole set of best practices and metrics so as to identify the most important ones, relying on their contribution in estimating component concentration of failures. At this stage, all the best practices and metrics measured in Table 3 and Table 4 have been considered. However, no limitation on their number has been adopted. Experiments have been performed by using the parameters mentioned in Table 5, for which the total number of occurrences, and the mean value have been calculated.

The followed steps are specified below: the risk-coefficient has been calculated for each component of the set; the mean values have been computed for fault and no-fault; the risk-threshold has been fixed between the two means with a "neutral" range centered in the threshold value so to exclude critical value from the classification; the set has been grouped according to the same nature on the basis of complexity and size; the discriminant analysis has been performed on best practice and metric groups and results have been produced; the model identified by discriminant analysis has been evaluated.

The best practices and metrics have been considered as the main subject of the analysis in order to produce an acceptable rate for fault-proneness estimation (about 85%). By using the whole best practices and metrics set contribution, the model classified all the components in the groups with a correctness of about 83%. In this case our set partly failed in producing a suitable fault proneness prediction due to both the number and low risk level of some metrics and best practices.

6 RELATED WORKS

A number of useful related projects have been reported in the literature. (Briand et al., 1993) proposed the construction of a modeling process aiming at predicting which components of a software project are likely to contain the highest concentration of faults. Such a modeling process is based on the Optimized Set Reduction (OSR) approach. With respect to this work, ours leverages not only on measurements related to the code but also on best practices. (Khoshgoftaar and Seliya, 2003) introduced two new estimation procedures for regression modeling, comparing their performance in the modeling of software quality in terms of predictive quality and the quality of fit with the more traditional least square and least absolute values estimation. The major difference between our research and this one is in the use of best practices, as already noticed in Section 2. (Kim et al., 2007) proposed a model to predict the most fault prone entities and files. Caching the location where faults are discovered a developer can detect likely fault-prone locations. This is used in order to prioritize verification and validation resources on the most fault prone files or entities. With respect to this work, ours uses best practices and metrics to build the predictive model.

7 CONCLUSIONS

Over the last years we have been gathering conceptual elements that helped us in facing the complexity of the activity of making software. We have distilled our insights and by trying to invent as less as possible we have stabilized relationships amongst some basic concepts of software engineering. In this paper we have presented a model to predict software quality, which has been designed leveraging on our experience on software development in European projects (Ronchieri et al., 2009). As a peculiarity of the model, we have combined best practices with metrics in order to contrive the improvement of software development process focusing on the early planning phases. The described best practices and metrics consider several aspects of the software life cycle such as configuration management and testing belonging to the best practices, and static analysis metrics. Our approach has consisted of evaluating a subset of those best practices and metrics that have been assessed as

crucial for achieving our fulfillment. Furthermore, we have analyzed the quality model by using similar modules of StoRM and WNoDeS projects characterized by having in common programming language and build tool. We have decided to present our work at this stage to share our thoughts with researchers interested in modeling. We hope that some parts of our works might help to understand the evolution of software engineering models. In the near future this work should be repeated by involving more heterogeneous modules of the stated projects, and, hence, increasing the validity of the described model. By doing this, larger data sets could be produced leading to a better estimation of our work. To enlarge the input data of the used predicting technology, on one hand the set of metrics will be extended with the dynamic one; on the other hand other best practices included in their set will be modelled.

ACKNOWLEDGEMENTS

This research was supported by INFN CNAF. The findings and opinions in this study belong solely to the authors, and are not necessarily those of the sponsors.

REFERENCES

Aiftimiei, C., Ceccanti, A., Dongiovanni, D., Meglio, A. D., and Giacomini, F. (2012). Improving the quality of emi releases by leveraging the emi testing infrastructure. *Journal of Physics: Conference Series*, 396(5).

Briand, L. C., Basili, V. R., and Hetmanski, C. J. (1993). Developing interpretable mmodel with optimized set reduction for identifying high-risk software components. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 19.

Brooks, F. P. (1995). *The mythical man-month*. Addison-Wesley Boston.

Butler, S. (2012). Mining java class identifier naming conventions. In *Software Engineering (ICSE), 2012 34th International Conference*, pages 1641–1643.

Chhabra, J. and Gupta, V. (2010). A survey of dynamic software metric. *Journal of Computer Science and Technology*, 25:1016–1029.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *Transactions on Software Engineering*, 20(6):476–493.

Coleman, D., Ash, D., Lowther, B., and Auman, P. (1994). Using metrics to evaluate software system maintainability. *IEEE Computer*, 27(8):44–49.

Debbarma, M. K., Kar, N., and Saha, A. (2012). Static and dynamic software metrics complexity analysis in regression testing. In *International Conference on Computer Communication and Informatics (ICCCI)*, Coimbatore, India. IEEE, IEEE.

DeMarco, T. (1982). *Controlling Software Project*. Prentice Hall.

Elbaum, S. (2005). Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions on Software Engineering*, 31.

Fang, X. (2001). Using a coding standard to improve program quality. In *Quality Software 2001. Proceedings Second Asia-Pacific Conference*, pages 73–78.

Fawler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code [Hardcover]*. Addison-Wesley Professional.

Fenton, N. (1990). Software metrics: theory, tools and validation. *Software Engineering*, 5(1):65–78.

Flissi, A., Dubus, J., Dolet, N., and Merle, P. (2008). Deploying on the grid with deployware. In *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium*, pages 177–184.

GCCTeam (2012). Gcc coding conventions.

Harel, D. and Rumpe, B. (2004). Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10):64–72.

IEEE90 (1990). *IEEE Standard Glossary of Software Engineering Terminology*, ieee std 610.12-1990, institute of electrical and electronic engineers, inc., new york, ny, edition.

Jansen, S. and Brinkkemper, S. (2006). Evaluating the release, delivery, and deployment processes of eight large product software vendors applying the customer configuration update model. In *WISER '06 Proceedings of the 2006 international workshop on interdisciplinary software engineering research*, pages 65–68.

Khoshgoftaar, T. M. and Seliya, N. (2003). Analogy-based practical classification rules for software quality estimation. *Empirical Software Engineering Journal*, 8(4):325–350.

Kim, S., Zimmermann, T., Jr, E. W., and Zeller, A. (2007). Predicting faults from cached history. pages 489–498. IEEE Computer Society Washington DC, USA.

Kopec, D. and Tumang, S. (2007). Failures in complex systems: case studies, causes, and possible remedies. *SIGCSE Bulletin*, 39(2):180–184.

Li, X. S. and Prasad, C. (2005). Effectively teaching coding standards in programming. In *SIGITE'05*, pages 239–244, Newark, New Jersey, US. ACM New York, NY, USA.

Majchrzak, T. A. (2010). Best practices for technical aspects of software testing in enterprises. In *Information Society (i-Society), 2010 International Conference*, pages 195–202.

McConnell, S. (1996). Who cares about software construction? *IEEE Software*, 13(1):127–128.

Mengel, S. A. and Tappan, D. A. (1995). Program design in file structures. In *Frontiers in Education Conference, 1995. Proceedings., 1995*, pages 4b2.11 – 4b2.16 vol.2.

Merlo, E., Kontogiannis, K., and Girard, J. (1992). Structural and behavioral code representation for program understanding. In *Computer-Aided Software Engineering, 1992. Proceedings., Fifth International Workshop*, pages 106–108.

Munson, J. C. and Khoshgoftaar, T. M. (1992). The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–432.

Oman, P. W. and Cook, C.-P. (1988). A paradigm for programming style research. *ACM SINGPLAN Notices*, 23(12):69–78.

Pighin, M. and Zamolo, R. (1997). A predictive metric based on discriminant statistical analysis. In *The 19th International Conference on Software Engineering, ICSE'97*, pages 262–270, Boston, Massachusetts, USA.

Ronchieri, E., Dibenedetto, M., Zappi, R., Aiftimiei, C., Vagnoni, V., and Venturi, V. (2012). T-storm: a storm testing framework. In *PoS(EGICF12-EMITC2)*, number 088, pages 1–11.

Ronchieri, E., Meglio, A. D., Venturi, V., and Muller-Wilm, U. (2009). Guidelines for adopting etics as build and test system.

Rossum, G. V. and Warsaw, B. (2001). Style guide for python code.

Salomoni, D., Italiano, A., and Ronchieri, E. (2011). Wnodes, a tool for integrated grid and cloud access and computing farm virtualization. *Journal of Physics: Conference Series*, 331(331).

SunMicrosystems (1997). Java code conventions.

Top, S., Nørgaard, H. J., Krogsgaard, B., and Jørgensen, B. N. (2004). The sandwich code file structure - an architectural support for software engineering in simulation based development of embedded control applications. In Press, A., editor, *IASTED International Conference on Software Engineering*.

Wingerd, L. and Seiwald, C. (1998). High-level best practices in software configuration management. In *Eighth International Workshop on Software Configuration Management Brussels*.

Zappi, R., Ronchieri, E., Forti, A., and Ghiselli, A. (2011). *An Efficient Grid Data Access with StoRM*. Lin, Simon C. and Yen, Eirc, Data Driven e-Schience. Springer New York.