2014 IEEE International Conference on Software Maintenance and Evolution

A Slice-Based Estimation Approach for Maintenance Effort

Hakam W. Alomari
Faculty of Information Technology
Jerash University
Jerash 26150, Jordan
halomari@jpu.edu.jo

Michael L. Collard
Department of Computer Science
The University of Akron
Akron, OH 44325, USA
collard@uakron.edu

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, OH 44242, USA
jmaletic@kent.edu

***Abstract*—Program slicing is used as a basis for an approach to estimate maintenance effort. A case study of the GNU Linux kernel with over 900 versions spanning 17 years of history is presented. For each version a system dictionary is built using a lightweight slicing approach and encodes the forward decomposition static slice profiles for all variables in all the files in the system. Changes to the system are then modeled at the behavioral level using the difference between the system dictionaries of two versions. The three different granularities of slice (i.e., line, function, and file) are analyzed. We use a direct extension of srcML to represent computed change information. The retrieved information reflects the fact that additional knowledge of the differences can be automatically derived to help maintainers understand code changes. We consider the hypotheses: (1) The structured format helps create traceability links between the changes and other software artifacts. (2) This model is predictive of maintenance effort. The results demonstrate that the approach accurately predicts effort in a scalable manner.**

Keywords—effort estimation; program slicing; software metrics; software maintenance

I. Introduction

Systems must be maintained so as to remain useful [1] and estimating the amount of effort for particular maintenance tasks is a key aspect for any system (closed or open). As systems grow, maintenance typically becomes more complicated and costly. Thus, the maintenance process should be well planned in advance through an accurate effort estimation of the maintenance tasks [2, 3]. Traditionally, maintenance effort is calculated using historical process and coarse-grained system information such as person-hours, number of tasks, and system size [4]. The predictor variables used to estimate this value typically compose measures of the system size and complexity, productivity factors, as well as size and number of maintenance tasks [5].

Typically, an estimation process for maintenance effort contains three steps: (1) Extract maintenance data, such as maintenance effort (person-hours), number of maintenance tasks, system size. (2) Build and validate the maintenance-effort model. Conventionally, this is a mathematical model that represents the maintenance effort as a function of other software measures. The model should be validated against additional maintenance data. (3) Predict future maintenance effort using the maintenance-effort model.

While using maintenance-task information is very attractive for managers of a typical closed-source system, who have to estimate the effort required to maintain the system in terms of the number of developers, this approach is not that useful for larger corrective, adaptive, or perfective tasks during the system evolution of open-source system [3]. In this case, the effort of a maintenance period greatly depends on the amount of source-code changes made to generate a new software version from an earlier operational version. For open-source systems, this data is not recorded or documented [2, 4, 6]. Additionally, because of the nature and complexity of the maintenance tasks in open-source systems, there are many negatives to directly using effort-estimation models built on closed-source data. Hence, we cannot follow the same process to estimate maintenance effort. However, the availability of the source code and history allow for other measures that are related to the maintenance effort. To this end we introduce a maintenance effort estimation based directly only on source code. It entails computing the slice for all the variables in a system and modeling how the slice changes over time.

Specifically, we identify and validate slice-based software measures and a corresponding process that can represent maintenance effort in open-source systems. We analyze 974 versions of Linux kernel, and construct and validate the indirect maintenance-effort model. The estimation approaches of maintenance effort are built and evaluated using residual-analysis statistics. Statistical measures include R^2, adjusted-R^2, $PRED_{25}$, $PRED_{50}$, MMRE, MdMRE, and SPR [7, 8]. The prediction results are encouraging and the production of the estimate is very scalable.

The remainder of this paper is organized as follows. Section II presents the maintenance effort estimation process in open-source systems. Section III discusses the indirect maintenance-effort measures. Section IV describes program-slicing process. Section V introduces the slice-based metrics. Section VI estimates slice-based metrics on the Linux kernel. The approach is evaluated in Section VII. Section VIII reviews related work followed by Section IX with the paper conclusions and some directions for future research.

II. Maintenance Effort Estimation Process

Building an accurate maintenance-effort estimation model should be derived from accurate maintenance-effort data, which is rarely recorded for open-source, and many closed-source, systems [2, 3]. Therefore, we cannot apply an effort-estimation model built from a closed-source system directly to

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.30

an open-source system because the absence of maintenance-effort data prevents validation. Alternatively, we take the following approach:

Phase 1: Identify measures that are theoretically related to and can indirectly represent maintenance effort. The candidate measures should be available for most systems, both closed and open source. If such measures can be found and validated, we can construct an indirect model for maintenance effort and use it to predict the indirect effort of open-source systems.

Phase 2: Extract the maintenance data. The data includes indirect maintenance-effort identified and validated in previous phase (aka *dependent variables*) and the data of other related measures that can be used to predict the indirect maintenance effort (aka *independent variables*). For example, if we identify source-code changes from version k to version $k+1$ as the indirect maintenance effort, then LOC change between both versions is a measure of source-code change.

Phase 3: Validate the correlation between the dependent variables and independent variables. We used Spearman's rank-correlation coefficient since there are no assumptions regarding the underlying distribution of the data, and its use is recommended for hypothesis testing when the number of data points exceeds 30 [5].

Phase 4: Multiple linear regression analysis is used to build the effort-prediction approach. Specifically, the indirect maintenance-effort is represented as a function of other related measures. We validate this approach against collected maintenance data from the Linux kernel.

Phase 5: Predict the indirect maintenance effort based on the models built in Phase 4.

III. Indirect Maintenance Effort Measures

In this section, we identify two software measures that are related to maintenance effort and could possibly be used to represent maintenance effort of open-source systems [2, 9-11]. They are lag-time and source-code change. To test this hypothesis, we need to consider whether lag-time and/or source-code change are a valid indirect maintenance effort measures.

1) Lag-time: Each version of the system has its own release date. The lag-time (*measured in days*) includes the duration from the date when a base version is released, until the date the evolved version is released. The assumption here that the maintenance requests start when the base version is released, and the tasks are completed when the evolved version is released. That is, the lag-time is the sum of the individual times for each maintenance task in a version of a system. Lag-time data is available for most closed-source systems as well as some open-source systems. For example, the lag-time data can be extracted from the defect tracking system, Concurrent Versions System (CVS), or change log [2]. Obviously, lag-time is related to maintenance effort. That is, an increase in lag-time is expected to indicate an increase in maintenance effort.

However, there could be some problems with using lag-time as an indirect maintenance effort indicator, including a risk of over reporting maintenance effort. For example, if a developer is sick for three weeks during the maintenance period and no one bothered to work on the system, the lag-time is then over reported. In addition, the importance of the bug controls the lag-time period. Important bugs are usually handled immediately after the assignment of the task, while less important bugs may be ignored until the next version. Therefore, using lag-time to represent the maintenance effort is not 100% accurate. Though, our empirical research on the Linux kernel will show that versions tend to cluster around two main roles: stable versions differ from development versions in terms of releases rate and activity. Such analysis could be used to determine how the effort is distributed in a given period, and to estimate future needs with respect to major versions.

2) Source-code change: We note that the maintenance effort for open-source systems is not given as the number of person-hours expended as the case in closed-source systems [2, 4]. However, it has been argued [2, 10, 12-14] that source-code change in open-source system can be used as an indirect measure for estimating maintenance effort.

A number of researchers have observed that the source-code change can be found using textual, syntactic, or semantic differencing [15]. For example, previous studies [2, 3, 10, 16], determine the source-code change between two consecutive versions either from CVS logs, using some computer aided software (CASE) tools, or system utilities such as *diff*. When source-code changes are submitted using the Software Configuration Management (SCM) tools (e.g., Subversion, CVS, and ClearCase) best practice is for developers to commit a brief explanation of the change into the change log, which is saved collectively with the source-code deltas in the SCM repository. Unfortunately, the quality of change logs varies greatly. That is, it depends on the developer that submits the changes: how well she understands the source code, and how well she writes change log messages [17]. Imprecise or blank change log entries make it hard for system maintainers to understand the source code. For example, Chen et al. [18] discussed the limitations of using the change logs to detect source-code changes in three open-source case studies. He shows that up to 78% of changes made to the source code are omitted from the system's change logs. Additionally, this tracking data is not always available. For example, the change logs for Linux kernel only started to be released after the major version 2.4 (version 2.4.1, January 29, 2001). That's why Yu [2] in his study of the Linux kernel built two models to estimate the maintenance effort using the change logs for major versions 2.4 and 2.5 only, with a total of 121 versions. These facts make it difficult to build effort estimation models and traceability links, based on the information found in the change logs.

Many existing software metrics are computed only using syntactic information of the code and use that to model semantic information. For example, cyclomatic complexity is computed by counting the number of branch (i.e., conditionals) to infer semantic complexity. Semantic information is much more difficult to derive and model. For example, a semantic change in one function might create a ripple effect among other functions. In a maintenance context, the effort estimation is a function of the code that is to be (was) changed. To help identify such problems, program slicers are often applied and are a valuable tool in determining side effects. It is possible to determine the parts with different behaviors by comparing the slices of the base and the evolved versions with respect to corresponding points.

IV. Program Slicing and Program Analysis

Program slicing is a widely used, and well-known, approach for understanding and detecting the impact of changes to software. The concept of program slicing was originally identified by Weiser [19] as a debugging aid. The calculation of a program slice is, with few exceptions, based on the notion of a Program Dependence Graph (PDG) [20, 21] or one of its variants, e.g., a System Dependence Graph (SDG) [22]. Unfortunately, building the PDG/SDG is quite costly in terms of computational time and space. As such, slicing approaches generally do not scale well.

Forward static program slicing [23-25] refers to the computation of program points that are affected by other program points. The forward slice from program point *p* includes all the program points in the forward control flow affected by the computation at *p*. Here we use the initial variable declaration as the starting point. The approach varies from the traditional definitions in two ways. First, a PDG is not computed for the entire program. Second, the slicing criterion does not require a precise reference to a location in the source (only a variable). Specifically, the approach taken here computes a *forward, static, non-executable (closure), inter-procedural program slice* for each variable in a system.

Our slicing approach [26, 27] addresses this limitation by eliminating the time and effort needed to build the entire PDG. In short, it combines a text-based approach, similar to Cordy's [28], with a lightweight static analysis infrastructure that only computes dependence information as needed (aka *on-the fly*) while computing the slice for each variable in the program. The slicing process is performed using the srcML [29, 30] format for source code. Source code is first converted to srcML and then a stream-oriented approach to compute the slice is performed. srcML augments source code with abstract syntactic information. This syntactic information is used to identify program dependencies as needed when computing the slice. srcML (SouRce-Code Markup Language) is an XML format used to augment source code with syntactic information from the AST to add explicit structure to program source code. The srcML format is supported with a toolkit, including *src2srcml* and *srcml2src*, which supports conversion between source code and the format.

We implemented our approach in a tool called *srcSlice*[1]. The approach was first introduced in [26], and there we conducted a small comparison study to the *CodeSurfer*[2] tool from GrammaTech, after that in [27], we extended this evaluation to a total of 18 open source systems. A system dictionary instead of PDG/SDG represents the program slice generated by the *srcSlice*. To detect system changes of two program slices, we compare their corresponding dictionaries. To make the comparison process efficient and make the comparison results reusable, we developed a system slice encoding (SSE) algorithm that encodes a program slice to a hash value, thereby allowing the slice hashes of a system to be used to detect behavioral changes across versions. The slice hashes for versions are stored in the system change information to identify behavioral changes across versions. This information serves as complement to the change logs to help maintainers understand changes better. We used an extension format to the srcML representation (denoted by sliceDiff) for representing slice-based differences in XML. The sliceDiff permits traceability links to be built between the change information and other software artifacts (e.g., design and requirement changes, test cases, bug reports, designs, and requirements).

A. Slice Profile and System Dictionary Construction

The approach computes a slice profile that contains all the relevant statements, from all possible slices, over a given slicing variable. We define our slicing criterion to consist of a file name, a function name, and a variable name. This slicing criterion is the triple (*f, m, v*) where *f* is a file in the system, *m* is a function/method in the file *f*, and *v* is a variable in the given function *m*. This definition of a slicing criterion does not require a precise reference to a statement number. This concept of slicing is used by Gallagher et al. [31] and is referred to as a decomposition slice. Rather than just a single variable of interest within the original program, our definition can retrieve the slices for all the variables inside a given function by modifying the slicing criterion to (*f*, *m*). Moreover, the slicing criterion (*f*) can be used to find all the slices of all variables in all functions in a given file. A system dictionary is built, referred to as (*F*, *M*, *V*), and includes all files in the system, all functions in each file, all variables in each function, and all global variables in the system. Each entry of the system dictionary is a slice profile with the following structure:

- *file*, *function*, and *variable* names;
- *@index*, an index of each variable as declared in order in the function;
- *slines*, a list of lines that comprise the slice;
- *cfunctions*, a list of functions called using the slicing variable;
- *dvariables*, a list of variables that are data dependent on the slice variable;
- *pointers*, a list of aliases of the slicing variable; and
- *controledges*, a list of all possible control-flow edges of the slicing variable.

We now present a definition of our slicing criterion and how a slice is computed using the criterion.

Definition 1 A forward decomposition slice *ds* of a program *p* is constructed with respect to a given file *f*, a given function *m* in *f*, and a given variable *v* in *m*. It consists of the union of the static forward slices (*denoted by sfs*) constructed for the criteria $\{(\{v\}, s_1), \ldots, (\{v\}, s_k)\}$, where $\{s_1, \ldots, s_k\}$ is the set of statements in *p* that assign to *v*. It is defined as:

[1] Available for download at www.srcML.org under General Public License.

[2] CodeSurfer is a produced of GammaTech Inc. www.gammatech.com.

$$ds(f,m,v) = \bigcup_{s \in \{s_1,\dots,s_k\}} sf_{(v,s)}(p).$$

This definition can be generalized to cater to a set of variables, functions, and files. This yields a definition 2.

Definition 2 A general forward decomposition slice of a program p is constructed with respect to the following slicing criteria (f, m), (f), and (F, M, V), where $F = \{f_1, f_2, \dots, f_j\}$ is the finite set of files in p, $M = \{m_1, m_2, \dots, m_y\}$ is the finite set of methods for each $f \in F$, and $V = \{v_1, v_2, \dots, v_d\}$ is the finite set of variables for each $m \in M$. The general decomposition slice for all variables (i.e., set V) inside a given function m is formed by:

$$mds(f,m) = \bigcup_{i=1}^{d} ds(f,m,v_i),$$

The general decomposition slice for all variables in a given file f is given by:

$$fds(f) = \bigcup_{i=1}^{y} mds(f,m_i),$$

The general decomposition slice for all variables in all the files F, and all global variables in the system is given by:

$$gds(F,M,V) = \bigcup_{i=1}^{j} fds(f_i).$$

(a)

```
1.   int main(){
2.       int sum = 0;
3.       int i = 1;
4.       while (i<=10){
5.          sum = sum + i;
6.          i++;
7.          }
8.       cout<<sum;
9.       cout<<i;
10.  }
```

(b)

Slice Profile(sum)= @index(1), slines={2, 5, 8},

Slice Profile(i)= @index(2), slines={3, 4, 5, 6, 9}, dvars={sum}

Figure 1. (a) Sample source code, (b) system dictionary with two slice profiles for the source code in (a). The final slice for *sum* = {2, 5, 8} and the final slice for *i* ={3, 4, 5, 6, 8, 9} after considering dependencies.

Let us now look at a simple example. The approach works much like a programmer would compute a slice in their head. Figure 1 presents a small program (a) along with the final system dictionary (b). The dictionary includes two slice profiles, one for each of the variables *sum* and *i*. The *@index* represents the position of variables as declared in the function. In this way, we can deal with variables of the same name within the same scope. The slice profiles are computed by examining each line starting from the beginning (line 1) and determining the forward slice. Definition-use chains are followed along with forward control dependencies. The profile for *sum* is created first as it is encountered in line 2 (*slines*(*sum*) = {2}). Then the profile for *i* is created in line 3 (*slines*(*i*) = {3}). The two profiles are updated as follows for the given line number:

4: slines(sum)={2}; slines(i)={3, 4}, controledges(i) = {(3,4)}

5: slines(sum)={2, 5}, controledges(sum) = {(2,5)}; slines(i)={3, 4, 5}, dvariables(i) = {sum}, controledges(i) = {(3,4),(4,5)}

6: slines(sum)={2, 5}, controledges(sum) = {(2,5)}; slines(i)={3, 4, 5, 6}, dvariables(i) = {sum}, controledges(i) = {(3,4),(4,5),(5,6)}

8: slines(sum)={2, 5, 8}, controledges(sum) = {(2,5),(2,8),(5,8)}; slines(i)={3, 4, 5, 6}, dvariables(i) = {sum}, controledges(i) = {(3,4),(4,5),(5,6)}

9: slines(sum)={2, 5, 8}, controledges(sum) = {(2,5),(2,8),(5,8)}; slines(i)={3, 4, 5, 6, 9}, dvariables(i) = {sum}, controledges(i) = {(3,4),(4,5),(4,9),(5,6),(6,9)}

These are the slice profiles for each variable, and the complete slice is then computed by finding the control-flow edges and then taking the union of the *slines* with the slice profiles of the *dvariables*, *cfunctions*, and *pointers*, minus any lines that are before the initial definition of the slice variable (i.e., the set {1, … ,*def*(*v*) – 1}). Thus, because *sum* is data dependent on *i*, the complete slice for *i* = *slines*(*i*) ∪ *slines*(*sum*) – {1, 2}. This comes out to {3, 4, 5, 6, 8, 9}. This final computation can be carried out for all variables via a single pass through the dictionary.

B. Encoding slicing information

Our system slice encoding (SSE) algorithm works on slice profiles represented by the system dictionary. The basic process of the SSE algorithm starts with a single pass through the system dictionary, encoding each slice profile to a string value, which is then fed to a hash algorithm to produce the final results, the hashed slice encoding. There are two steps in the SSE algorithm.

Step 1 of the SSE algorithm: Encode the slice profiles. The complete slice for a slicing variable after taking the union of all related slice profiles will have the following encoding string value (denoted by *dsES*): *variableName; @index;* {*ds*(*f, m, v*)}. For a given function, the SSE algorithm encodes the slice profiles into a string value (denoted by *mdsES*). This string consists of two parts, the *functionName* and the *slines* defined by the *mds* equation (*definition 2*). The file encoding string (denoted by *fdsES*) is equal to *fileName;* {*fds*(*f*)}. Finally, the system encoding string (denoted by *gdsES*) is equal to *systemName*; {*gds*(*F, M, V*)}. For the example program in Figure 1, the *dsES(sum) = sum; @1;* {*2, 5, 8*}, the *dsES(i) = i; @2;* {*3, 4, 5, 6, 8, 9*}, and the *mdsES(main) = main;* {*2, 3, 4, 5, 6, 8, 9*}.

Step 2 of the SSE algorithm: Hash the string value. This step maps the encoding string from Step 1 to a hash value using the MD5 hash algorithm [32]. For example, the MD5 for the *dsES(sum)* is 6c9eed3c2a88b623c05347aee687d289, the MD5 for the *dsES(i)* is *e20426ade1655eaaaccc2a9c09429261*, and the MD5 hash for the *mdsES(main)* is f65571d34f742bf9a65e53e9a6640d2b.

C. System Behavioral Change Information

To compute behavioral change information across the entire version history of a system, we check out every pair of consecutive versions of the system from its subversion repository, use *src2srcml* to convert the source code into srcML format, use *srcSlice* to build the system dictionary with slice profiles for all the slicing variables in each version, and apply the SSE algorithm on them. We compare the slice hashes for the *dsES*, *mdsES*, and *fdsES* in the later version with the corresponding hashes in the prior version to find the behavioral changes. Finally, we save the system behavioral change information for each version in a database.

```
<changeInfo systemName="linux" versionNumber="2.2.23"
              changeKind="changed" deltafdsES ="1" >
   <sourceFileChange sourceFilePath="linux/fs/read_write.c"
                              changeKind="changed" deltamdsES="1">
      <sliceHash>3769c57d417347bb9c0d74a0db637744</sliceHash>
      <functionChange functionName="do_readv_writev"
                              changeKind="changed" deltadsES="2"\>
         <sliceHash>a7df45bf6022cdef77cf49667aa6428b</sliceHash>
         <sliceChange changeKind="changed">
            <sliceLabel>tot_len</sliceLabel>
            <sliceHash>ce2d52d65f33eb611a6030735ebe9262</sliceHash>
         </sliceChange>
         <sliceChange changeKind="changed">
            <sliceLabel>retval</sliceLabel>
            <sliceHash>3a2289b2f656d5569ea0110d07f8a1c5</sliceHash>
         </sliceChange>
      </functionChange>
   </sourceFileChange>
</changeInfo>
```

Figure 2. A partial example of the *changeInfo* data for Linux kernel version 2.2.23 in the sliceDiff format.

The database includes a *SystemChange* table. This table has three fields, *systemName*, *versionNumber*, and *changeInfo*. Each version of the system has one record in the database. The *systemName* and *versionNumber* fields record the name of the system and the version number of the system, respectively. The *changeInfo* field contains the behavioral change information of this version compared to its prior version, represented in sliceDiff format. An example of the representation for the *changeInfo* data can be found in Figure 2.

In our extension to the srcML representation, the element *changeInfo* represents all the changes to the system at this version and the number of *fdsES* hashes changed (denoted in the sliceDiff representation by *deltafdsES*). The *changeInfo* element contains multiple *sourceFileChange* elements, which represent all of the source code files contain modified slices and the number of *mdsES* hashes changed (denoted by *deltamdsES*). A *functionChange* element records the function name, change kind, number of *dsES* hashes changed (denoted by *deltadsES*), and the change information for the variable slices. The *sliceChange* element records change in a slice profile of the variable. The *sliceLabel* element stores a label that indicates the name of the slicing variable. Finally, the *sliceHash* element contains the 32-character hash value for the encoding string computed by the SSE algorithm.

Due to the sliceDiff representation of the system change information, we open the door to locating components in the change information and associating them with other software artifacts. Once in sliceDiff, XML tools and technologies can be used for fact extraction. For example, use of XPath and XQuery for change extraction. The expressions that used to locate components in the change information make it possible to create traceability links between the change information and other artifacts. A full explanation of this is left for future work.

V. SLICE – BASED METRICS

In the context of effort prediction, Ramil et al. [33] stated that one may start the investigation of building an effort model by obtaining empirical data and by estimating from such data a productivity function *f()*. The final empirical data involved in the estimation of *f()* are represented in the following equation: $E(t, t+1) = f(act(t, t+1)) + error(t, t+1)$, where, $E(t, t+1)$ represents the estimated effort. That is, the effort required evolving the system from interval t to $t+1$. The $act(t, t+1)$ represents the amount of work accomplished over the time interval. Finally, $error(t, t+1)$ is the modeling error. In addition, Ramil mentioned that the appropriate way to measure the $act(t, t+1)$ in the continuing evolution context is by measuring some indicators of source-code change, e.g., lines of source code (LOC) or function points (FP) [34]. However, other metrics can also be extracted from source code with different degrees of granularity. Once the productivity function *f()* is determined, the resultant model may be used to predict future maintenance effort requirements.

We use the information on the sliceDiff generated above to calculate slice-based metrics. Here we compare three different granularities of the slice hashes (*dsES-level*, *mdsES-level*, and *fdsES-level*). Consequently, different levels of the number of hashes changed (*function-level*, *file-level*, and *system-level*) can be computed. In order to build the slice-based maintenance-effort model, for each of the 974 versions of the Linux kernel, we extract six measures from the source-code repository and the changes between slice hashes. These measures are described in Table I. In these measures, *lagTime* could be used to indirectly represent maintenance effort (for the reasons explained in Section VI).

TABLE I. CODE AND SLICE BASED EXTRACTED MEASURES.

Measure	Description
lagTime	Indirect maintenance effort on the system, time-intervals between versions measured in the number of days
sliceSize	Total slice size measured in LOC
hashSize	The number of slice hashes modified
locSize	Total size of the system measured in LOC
fileSize	Total size of the system measured in number of files
sCoverage	The slice coverage, the slice size relative to LOC

The first measure that we introduce is *sliceSize*, the slice size measured in LOC. For an individual slice this is just the *ds* value measured at Section IV. For a function and a file, the *sliceSize* is computed using the *mds* and *fds*, respectively. For the system level, the *sliceSize* is computed using the *gds* equation. Additionally, the number of modified hash slices between two versions is used to introduce *hashSize*. For a function and a file, the *hashSize* value is the *deltadsES* and *deltamdsES* values, respectively, as measured in the *changeInfo* data. For the system level, the *hashSize* is calculated in the *changeInfo* as a *deltafdsES* value. The metric *hashSize* at the function-level is the number of functions that contain modified slices and for the file-level is the number of files that contain modified slices. These two metrics indicate how much the changed statements in a slice profile depend on each other by intra-procedural or inter-procedural control or data dependencies. A high function-level value may indicate more logically complex code, and a high file-level value may indicate that the changes in the system were very broad.

In addition, we also extract the size of the system measured in LOC (i.e., *locSize*) and number of files (i.e., *fileSize*). By comparing the slice size (*sliceSize*) to the system size (*locSize*), we can measure the slice coverage using the *sCoverage* metric [19]. This metric represents the active portion of the system and is included as a factor of maintenance activity.

Considering two consecutive versions of Linux kernel, *base* version and *evolved* version, the measures of the maintenance data of base version are extracted as follows: *locSize*, *fileSize*, *sliceSize*, and *sCoverage* are determined from the source code of the base version. *hashSize* is determined from the change information of the evolved version (because changes made to the base version are recorded in the *changeInfo* of the evolved version). *lagTime* is determined according to the date differences between the base version and the evolved version.

VI. Slice-Based Estimation On The Linux Kernel

As a way of showing the application of our indirect maintenance-effort metrics on a real system, we have applied the metrics to the Linux kernel. These metrics are then compared to traditional measures of code effort, e.g., LOC. The Linux versions are classified as stable or development versions. Each major version includes several releases identified with either a three or four digit numbering scheme. The first digit represents the generation, i.e., Linux has three generations, initially with generation 1 released in 1994, generation 2 released in 1996, and generation 3 started in 2011 (not part of the dataset). The second digit represents the major kernel versions either even or odd. Up until major version 2.4 even digits (e.g., 1.0, 1.2, 2.0, etc.) corresponded to stable versions, whereas odd numbers (e.g., 1.1, 1.3, 2.1, etc.) corresponded to development versions. The third digit is the minor kernel version. However, in August 2004 this numbering scheme was changed affecting all the versions released after this date. A fourth digit number was added starting with version 2.6.8.1, after that the third number in a version indicates the development of new functionality, and the presence of a fourth number represents bug fixes [35].

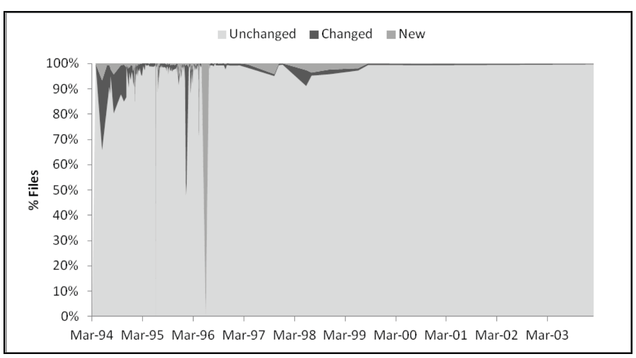

Figure 3. Files change evolution in first four versions (1.1, 1.2, 1.3, and 2.0) of Linux kernel. This graph illustrates change property captured by slicing.

According to law number 4 of Lehman's laws of software evolution [36], the average work rate on an evolving system is statistically invariant over the system life time. In order to examine this law we should study the maintenance effort spent on the system. Again, reliable data about person-hours or number of developers is hard to get in closed-source systems, and much harder in open-source systems. Additionally, person-hours are inaccurate measure of work to begin with [37]. Lehman et al. [36] suggests using the number of elements handled as a proxy. However, he mentioned this also has methodological difficulties.

We start by considering the number of elements handled. As an example, we will study the average rate of hash slices changed for files. That is, the likelihood that a file will change (slice-based) from one release to the next. To assess the likelihood of a file changing, we gather the ratio of files that are unchanged, ratio of files that are changed, and ratio of files that are added or removed, by comparing successive releases. Figure 3 shows the number of files that were added, deleted, and modified (divided into those that grew) between consecutive releases. As may be expected, the fraction of files that are handled seems to be relatively stable, except perhaps for some decline in the first years. On average across all versions we observed that 96% of the files are unchanged, 3% are modified, and 1% are a added/removed file. Thus if we interpret rate to mean the fraction of source code that is modified in each release, then the data supports the claim that the work rate is almost constant.

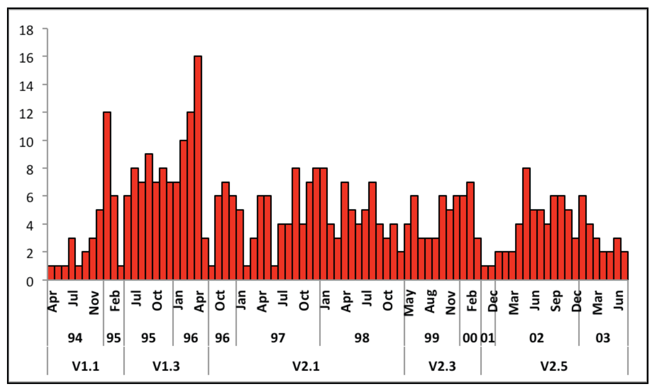

Figure 4. Number of releases per month for development versions, in x-axis (2) = v1.1, (4) = v1.3, (6) = v2.1, (8) = v2.3, and (10) = v2.5.

Invariant work rate can also be interpreted with regard to the release rate itself, i.e., how often releases occur. Based on the structure of the Linux kernel, it seems that the growth trend follows a consistent pattern: a new development version is released after a number of releases of a stable version, and after there are no more releases in that development version, a new stable version is released. However, during the time interval and releases of the stable version there are still releases of the previous stable version. For example, version 2.0 had continuous releases until the end of version 2.2.

We start analyzing the number of releases per month for the development versions as shown in Figure 4. In the *x-axis* each stub represents a year, and each bar represents a month. The vertical lines with the version label (i.e., V1.1, V1.3, V2.1, V2.3, and V2.5) represent the start of that new major version. It is obvious that since the mid of 1997 the rates seem stable (around 3 – 6 release per month) and with a minimum is equal to 1 and the maximum is 8. From Figure 5, we can see that the stable versions are released less frequently than the development versions. That is, usually there was one release per month, and the maximum is 10. Starting with version 2.6 the versions are timed to be released once every ~ 3 months. It is important to remember that the Linux releases are organized into major releases (e.g., 1.1, 1.2, etc.) and minor releases (e.g., 1.1.13, 2.2.3, etc.). Therefore, one should consider the intervals between major releases independently from minor releases.

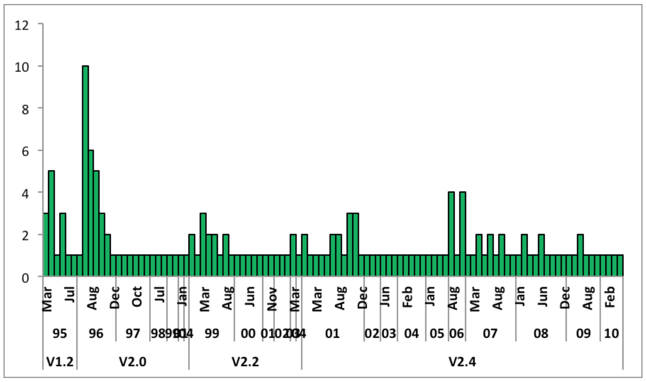

Figure 5. Number of releases per month for stable versions, in the x-axis (3) = v1.2, (5) = v2.0, (7) = v2.2, and (9) = v2.4.

In this context, we examined the intervals of time between consecutive releases (*measured in days*) inside the same major version. Figure 6 displays the raw data of the intervals for each version, and Figure 7 shows the statistics (median and 25th, 75th, and 95th percentiles) of the intervals for each version. Notice that in Figure 7 versions 2.0, 2.2, and 2.4, the 95th percentile values exceed the top of the graph and their true values appear on the labels above their respective up-bar boxes.

Looking back at Figure 6 we can see an interesting pattern. Generally the development versions (1.1, 1.3, 2.1, 2.3, and 2.5) have very low values, and so does version 2.6, while the stable versions (1.2, 2.0, 2.2, and 2.4) have much higher values. We also notice that at the end of any development version (red line) there is almost a simultaneous beginning of a new stable version (blue line), the exception here we do see a significant gap between versions 2.3 and 2.4, and between versions 2.5 and 2.6. Those two gaps are results of the structural changes in the Linux kernel numbering scheme.

Looking at Figure 7 we can notice that the bodies of the up-bars of the stable versions are usually higher than those of the development versions. In addition, the high values and the variance are usually higher in stable versions. In other words, we can see that the stable versions are released less frequently (usually on a weekly to monthly basis), while development versions are released quite often (are on a daily to weekly basis). For example, all medians, 25th, and 75th percentiles in development versions are lower than 10 days.

Based on the above results, we can see that the release rate performed in the development versions accounts for ~40% of the overall amount of versions (398 versions out of 974). 15% of the overall versions are released during the stable versions. Finally, 45% of the versions are released during the version 2.6. Thus when expressing the effort as a function of the performed activity (e.g., lines added, lines modified, files added, etc.) in the Linux kernel, and taking into account the three types of versions (development, stable, and version 2.6), and weighing them appropriately, then the effort estimation equation should be tailored to reflect such differentiation in both the kind of version, and consequently the type of activity performed (e.g., corrective, perfective, etc.), as follows:

$$E(t) = w_d * f(act_d(t)) + w_s * f(act_s(t)) + w_{2.6} * f(act_{2.6}(t))$$

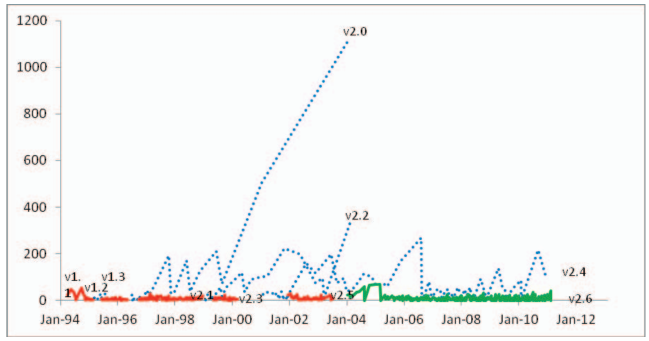

Figure 6. Intervals between version's releases measured in days.

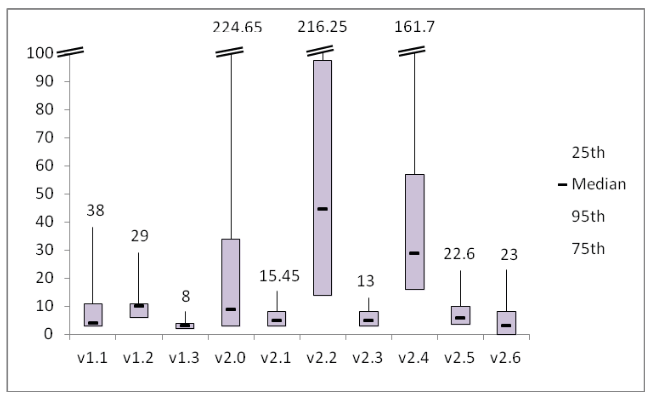

Figure 7. Statistics of intervals between releases, within major version measured in days (ordered as: 25th, median, 75th, and 95th percentile).

Where, *E(t)* is the maintenance effort in the Linux kernel during a period *t* (daily, monthly, etc.), w_d is the weight given to the activity observed within the development versions $(act_d(t))$; w_s the weight to the stable versions, $(act_s(t))$; and $w_{2.6}$ the weight to the version 2.6, $(act_{2.6}(t))$. In the case of the reported Linux kernel, the overall activity observed in this project, based on the number of versions released, produces the following weights: $w_d = 0.40$, $w_s = 0.15$, and $w_{2.6} = 0.45$. These results suggest that we should differentiate between the three types of versions during the calculation process of the slice-based metrics. Since the data used to build our models represents the three types of versions, therefore we omit the weights during the building process.

We analyzed 11 major versions containing 974 separate releases, covering a period that exceeds 17 years of software evolution. The models were built on data from 783 versions, and then validated on the maintenance data of 191 versions from version 2.6. Table II shows Spearman's rank correlations between the dependent variable and independent variables.

The correlation coefficients that are statistically significant at the 0.01 level (2-tailed) are shown in bold. From Table II, we can distinguish multiple significant linear correlations between the dependent variable and all of the independent variables. Based on this observation, we built the indirect maintenance-effort estimation models. The correlation could serve as guidelines to assess maintenance effort from two viewpoints; code-based and slice-based. Therefore, we chose to build the first model using the code-based metrics (E_{code}), and the second model, using the slice-based metrics (E_{slice}), as follows:

$E_{code} = c1 + c2\ (locSize) + c3\ (fileSize).$

$E_{slice} = c1 + c2\ (sliceSize) + c3\ (hashSize) + c4\ (sCoverage).$

TABLE II. THE CORRELATIONS BETWEEN DEPENDENT VARIABLE AND INDEPENDENT VARIABLES BASED ON THE TRAINING DATASET (783) VERSIONS, SIGNIFICANT AT 0.01 LEVEL IS SHOWN IN BOLD.

Variable	Effort	p-value
sliceSize	**0.768**	*0.008*
hashSize	**0.757**	*0.005*
locSize	**0.767**	*0.008*
fileSize	**0.662**	*0.003*
sCoverage	**0.338**	*0.001*

The c_1 variable represents the constant factor or the intercept, which characterizes the height of the regression line when it crosses the *y-axis* where the dependent variable is plotted, or we can say that the c_1 represents the predicted value of the dependent variable when all the independent variables are equal to zero. The c_i (where i = 2 to 4) represents the slope of the line regression which indicates the sensitivity of the dependent values to the changes in the independent values. That is, c_i represents the change in y for each unit change in x.

TABLE III. LINEAR REGRESSION ANALYSIS OF THE INDIRECT EFFORT ESTIMATION MODELS.

Model	Independent variable	ci	p-value	R^2	adjusted-R^2
E_{code}	*locSize*	*0.012*	*0.004*	*0.619*	*0.613*
	fileSize	*4.396*	*0.000*		
E_{slice}	*sliceSize*	*0.030*	*0.006*	*0.744*	*0.739*
	hashSize	*4.521*	0.002		
	sCoverage	*0.554*	0.000		

Table III shows the linear regression analysis of the model. The p-value demonstrates the ability of the independent variable to have a significant predictive capability in the presence of other variables. The R^2 coefficient of determination value is important to determine whether or not the regression model was helpful. If the regression line provides an estimate of the predictable values that closely match the observed values, then the R^2 value will be close to one, and with zero indicating no relation between independent and dependent variables. The adjusted-R^2 that adjusts for the number of independent variables in a model is also calculated. From Table III, we can see that both models have a moderate both R^2 and adjusted-R^2 values, which means, based on the data of 783 versions, the model is by some means accurate in predicting the indirect maintenance effort.

VII. EVALUATING MODEL PERFORMANCE

To study the quality of the proposed models for future predictions, we apply the models to predict the indirect maintenance effort of 191 versions from major version 2.6. These versions range from version 2.6.25.3 released May, 10 2008 to version 2.6.37.1 released Feb, 17 2011. The predicted results and the actual observed measurements are compared to study the accuracy of predictions. Model validation is the most important step in the model building process. The validation of a model often consists of the analysis of residuals [2, 3, 10]. The residual represents the difference between the predicted value estimated by the model and the observed value of the dependent variable. Our analysis includes the following.

SPR statistics: is the sum of absolute value of the residuals (e.g., prediction errors). That is, the SPR = ∑ k |Observed k – Predicted k|.

MRE statistics: the magnitude relative error, which includes the MMRE (mean magnitude relative error), and MdMRE (median magnitude relative error). The MRE is defined as: MRE k = (|Observed k – Predicted k|) / Observed k. The MdMRE is calculated, since the MMRE is known to be very sensitive to the extreme values, such as a few very high relative error MRE values could influence the overall result.

Other indicators commonly used to evaluate the prediction model based on MRE are the percentage of prediction at specific level PRED, which measures the percentage of predicted values within X% of the observed values. The value of X is suggested in [38] to be at least 25% and a good prediction model should predict 75% of the observed values. The two variants of the measure PRED we calculated are: $PRED_{25}$: the number of predicted values for which MRE was less than or equal to 25%. $PRED_{50}$: the number of predicted values for which MRE was less than or equal to 50%.

TABLE IV. MODELS PREDICTIVE PERFORMANCES OVER 191 RELEASES.

Measure	Code-based Model	Slice-based Model
$PRED_{25}$ %	*33.91*	*49.31*
$PRED_{50}$ %	*64.72*	*82.66*
SPR	*33520*	*25596*
MdMRE %	*37.56*	*25.35*
MMRE %	*42.53*	*31.25*

The results of the application of these measures over the 191 versions test dataset are shown at Table IV. It is clearly evident that the slice-based model performs better than the code-based model, although the performance of the code-based model can also be considered good. In particular, the values of the PRED measures for slice-based model are very promising: it predicts almost 50% of the cases within a relative error less than 25% ($PRED_{25}$) and about 83% of the cases with a relative error less than 50% ($PRED_{50}$). In addition, the relative mean error is ~32% and can be considered outstanding. These results suggest that the slice-based model using the slicing information reflects both the type and the size of the maintenance process more accurately.

VIII. Related work

Many approaches to the effort-estimation problem have been derived using different assumptions, data sources, and methods to process the data to estimate the effort in the context of maintaining strictly managed and closed-source systems [2, 10]. These models can be categorized into three main categories: *analogy*, *delphi*, and *parametric* [39]. The first two categories derive the estimation models based on the past experience of similar systems, or using expert opinions. Parametric effort estimation models involve the construction of statistical models from empirical data, e.g., using regression analysis. Moreover, the parametric models mathematically relate the effort and duration (e.g., days) to the variables that influence them.

Boehm [40] was the first to presents an algorithmic software cost estimation model, the constructive cost model COCOMO. Boehm et al. [41] extended the COCOMO model to estimate maintenance effort by using a size-change factor. This factor represents the estimation of the size of changes expressed as the fraction from the total size of the system in LOC, this factor over a year period. De Lucia et al. [3] called this factor the “annual change of traffic”. Another work based on the size of changes is presented by Hayes et al. [42] who built a model for adaptive-maintenance effort using the changed LOC and the number of operators changed.

Belady and Lehman [43] suggest a model to approximate the cost and effort of releasing a new version from an old one. The suggested model estimates the efforts that are related to both the functionality updating and anti-regressive activities. The maintenance-effort estimation that involves the convention of linear regression analysis was introduced by De Lucia et al. [16]. In this research, the authors claimed that the types of the different maintenance tasks should be considered to improve the outcomes of the estimation model being used.

Jorgensen [8] derived different estimation models for maintenance effort using log linear regression, neural networks, and pattern recognition. He compares the prediction accuracy of these models using an industrial dataset. All the models estimate the size of the system measured in the summation of added, deleted, and modified LOC during the maintenance phase. Another linear model based on the size and the number of maintenance tasks is proposed in [3], furthermore, other work done by Niessink et al. [13] use linear regression analysis to extract estimation based on function points.

Coarse granularity measures have an impact on predicting required changes during the maintenance activities of the software project. For example, Lindvall [44] demonstrates that the number of classes outperform the finer grained metrics in change prediction. In contrast, non-linear cost estimation models were proposed by several researches. For example, in [45] a code decay and a related number of measurements were illustrated to construct a non-linear changes prediction model.

Because of the nature and complexity of the maintenance tasks in open-source systems, there are many negative aspects to using existing effort estimation models directly for open-source projects. Little work of maintenance-effort estimation has been conducted for open-source systems. The major guidelines and tips to build an estimation model in these crucial systems are reported in [4]. Yu [2] derived two indirect maintenance-effort models for the Linux kernel system using multiple linear regression. Nevertheless, these estimation models are based on and used factors which are derived from the closed-source software projects. In addition the validation process determined using the recorded maintenance information from closed-source systems, i.e., both estimation models depend on the number of maintenance tasks for the next version of the system. Therefore, the models are not applicable if the maintenance tasks for the next version are not included.

IX. Conclusions and Future Work

In this paper, we presented a large-scale empirical study aimed at building indirect maintenance-effort estimation models for open-source systems. The dataset was obtained from the Linux kernel and used as a case study to build and validate the models performance using multivariate linear regression. Our proposed maintenance-effort estimation models are able to accurately determine the source-code changes based only on the source code, and estimate the maintenance effort based at the amount of changes made maintaining the system. It is worth noting that we did not construct a direct maintenance-effort model (person-hours) for open-source systems. However, we decided to use the available source code, because: (1) there is limited direct maintenance-effort data available for open-source systems and we therefore cannot validate the correctness of such a model; and (2) maintenance effort represented as person-hours is less meaningful for open-source systems.

The major threat in building the indirect maintenance-effort models comes from the difference between the closed-source and open-source systems. Our prediction model depends on source-code measurement to predict the volume of changes as an indication of the maintenance effort. However, this is not the case for closed-source systems that use person-hours as a metric for maintenance effort.

In order to perform slicing for multiple versions of large systems, we used a lightweight forward static slicing approach. That is the main reason that the analysis over all the Linux versions was even possible. Our future research will study other open-source systems to determine more measures that can be used to indirectly represent maintenance effort and construct new more accurate prediction models.

References

[1] Lehman, M. M., "Programs, Life Cycles, and Laws of Software Evolution", *Proc of the IEEE*, vol. 68, no. 9, 1980, pp. 1060-1076.

[2] Yu, L., "Indirectly Predicting the Maintenance Effort of Open-Source Software", *Journal of Software Maintenance and Evolution*, vol. 18, no. 5, September 2006, pp. 311-332.

[3] De Lucia, A., Pompella, E., and Stefanucci, S., "Assessing effort estimation models for corrective maintenance through empirical studies", *Information and Software Tech*, vol. 47, no. 1, 2005, pp. 3-15.

[4] Asundi, J., "The Need for Effort Estimation Models for Open Source Software Projects", *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, 2005, pp. 1-3.

[5] Binkley, A. B. and Schach, S. R., "Inheritance-Based Metrics for Predicting Maintenance Effort: An Empirical Study", 97-05, T. R., Ed. Nashville, TN: Computer Science Department, Vanderbilt Univ, 1997.

[6] Yu, L., Schach, S. R., and Chen, K., "Measuring the Maintainability of Open-Source Software", in *International Symposium on Empirical Software Engineering (ISESE '05)*, vol. 0., 2005, pp. 297-303.

[7] Kendall, M. G., Stuart, A., and Ord, J. K., *Kendall's Advanced Theory of Statistics*, New York, Oxford University Press, Inc., 1987.

[8] Jorgensen, M., "Experience With the Accuracy of Software Maintenance Task Effort Prediction Models", *IEEE Transactions on Software Engineering (TSE '95)*, vol. 21, no. 8, August 1995, pp. 674-681.

[9] Kula, R. G., Fushida, K., Yoshida, N., and Iida, H., "Experimental Study of Quantitative Analysis of Maintenance Effort using Program Slicing-based Metrics", in Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference, 2012, pp. 50-57.

[10] Ramil, J. F. and Lehman, M. M., "Effort Estimation from Change Records of Evolving Software", in *Proceedings of International Conference on Software Engineering (ICSE '00)*. Limerick, Ireland: ACM, 2000, pp. 777-787.

[11] Robles, G., González-Barahona, J. M., Cervigón, C., Capiluppi, A., and Izquierdo, D., "Estimating Development Effort in Free/Open Source Software Projects by Mining Software Repositories: A Case Study of OpenStack ", in Proceedings of 14th International Conference on Mining Software Repositories (MSR '14).

[12] Alshayeb, M. and Li, W., "An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Processes", *IEEE Trans. Softw. Eng.*, vol. 29, no. 11, 2003, pp. 1043-1049.

[13] Niessink, F. and Vliet, H. V., "Predicting Maintenance Effort with Function Points", in Proceedings of the International Conference on Software Maintenance (ICSM' 97). Bari, Italy, 1997, pp. 32-39.

[14] Niessink, F. and Vliet, H. V., "Two Case Studies in Measuring Software Maintenance Effort", in Proceedings of the International Conference on Software Maintenance (ICSM '98). 1998, pp. 76-86.

[15] Maletic, J. I. and Collard, M. L., "Supporting Source Code Difference Analysis", in Proceedings of the International Conference on Software Maintenance (ICSM '04). Chicago, IL, USA, 2004, pp. 210-219.

[16] De Lucia, A. and Pompella, E., "Effort Estimation for Corrective Software Maintenance", in Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE '02). Ischia, Italy: ACM, 2002, pp. 409-416.

[17] Pan, K., James Whitehead, E., and Ge, G., "Textual and Behavioral Views of Function Changes", in Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering, Long Beach, California, 2005, pp. 8-13.

[18] Chen, K., Schach, S. R., Yu, L., Offutt, J., and Heller, G. Z., "Open-Source Change Logs", *Empirical Software Enginering*, vol. 9, no. 3, 2004, pp. 197-210.

[19] Weiser, M. D., "Program slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method. PhD thesis". Ann Arbor, MI, USA: University of Michigan, 1979.

[20] Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B., and Wolfe, M., "Dependence graphs and compiler optimizations", in Proceedings of the 8th ACM SIGPLAN-SIGACT, 1981, pp. 207-218.

[21] Ferrante, J., Ottenstein, K. J., and Warren, J. D., "The Program Dependence Graph and its Use in Optimization", *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, 1987, pp. 319-349.

[22] Liang, D. and Harrold, M. J., "Slicing Objects Using System Dependence Graphs", in Proceedings of the International Conference on Software Maintenance (ICSM), 1998, pp. 358-367.

[23] Bergeretti, J.-F. and Carre', B. A., "Information-flow and data-flow analysis of while-programs", in *ACM Trans. Program. Lang. Syst.*, vol. 7: ACM, 1985, pp. 37-61.

[24] Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs", *SIGPLAN Not.*, vol. 23, 1988, pp. 35-46.

[25] Horwitz, S., Reps, T., and Binkley, D., "Interprocedural Slicing Using Dependence Graphs", *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 1, 1990, pp. 26-60.

[26] Alomari, H. W., Collard, M. L., and Maletic, J. I., "A Very Efficient and Scalable Forward Static Slicing Approach", in Proceedings of the IEEE International Working Conference on Reverse Engineering (WCRE'12). Kingston, Ontario, Canada, 2012, pp. 425-434.

[27] Alomari, H. W., Collard, M. L., Maletic, J. I., Alhindawi, N., and Meqdadi, O., "srcSlice: very efficient and scalable forward static slicing", *Journal of Software Evolution and Process*, DOI: 10.1002/smr.1651, 2014.

[28] Cordy, J. R., Eliot, N. L., and Robertson, M. G., "TuringTool: A User Interface to Aid in the Software Maintenance Task", *TSE*, vol. 16, no. 3, 1990, pp. 294-301.

[29] Collard, M. L., Maletic, J. I., and Robinson, B. P., "A Lightweight Transformational Approach to Support Large Scale Adaptive Changes", in Proceedings of the IEEE International Conference on Software Maintenance (ICSM), 2010, pp. 1-10.

[30] Collard, M. L., Decker, M., and Maletic, J. I., "Lightweight Transformation and Fact Extraction with the srcML Toolkit", in Proceedings of 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'11), Sept 25-26 2011.

[31] Gallagher, K. B. and Lyle, J. R., "Using Program Slicing in Software Maintenance", *TSE.*, vol. 17, no. 8, 1991, pp. 751-761.

[32] Rivest, R., "The MD5 Message-Digest Algorithm", http://www.ietf.org/rfc/rfc1321.txt., 1992.

[33] Ramil, J. F. and Lehman, M. M., "Metrics of Software Evolution as Effort Predictors - A Case Study", in Proceedings of the International Conference on Software Maintenance (ICSM'00), 2000, pp. 163-172.

[34] Albrecht, A. J. and Gaffney, J. E., "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation", *IEEE Trans. Softw. Eng.*, vol. 9, no. 6, 1983, pp. 639-648.

[35] Koren, O., "A study of the Linux kernel Evolution", *ACM SIGOPS Operating Systems Review*, vol. 40, no. 2, 2006, pp. 110-112.

[36] Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M., "Metrics and Laws of Software Evolution - The Nineties View", in Proceedings of the 4th International Symposium on Software Metrics, 1997, pp. 20.

[37] Israeli, A. and Feitelson, D. G., "The Linux kernel as a case study in software evolution", *J. Syst. Softw.* no. 3, 2010, pp. 485-501.

[38] Conte, S. D., Dunsmore, H. E., and Shen, V. Y., *Software Engineering Metrics and Models*, Redwood City, CA, USA, Benjamin-Cummings Publishing Co., Inc., 1986.

[39] Shepperd, M., Schofield, C., and Kitchenham, B., "Effort Estimation Using Analogy", in Proceedings of the International Conference on Software Engineering (ICSE '96). Berlin, Germany, 1996, pp. 170-178.

[40] Boehm, B. W., "Software Engineering Economics", in *Software Pioneers*, Springer-Verlag New York, Inc., 2002, pp. 641-686.

[41] Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R., and Selby, R., "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0", *Annals of SE*, vol. 1, no. 1, 1995, pp. 57-94.

[42] Hayes, J. H., Patel, S. C., and Zhao, L., "A Metrics-Based Software Maintenance Effort Model", in Proceedings of the Working Conference on Software Maintenance and Reengineering (CSMR '04). Tampere, Finland, 2004, pp. 254-259.

[43] Belady, L. and Lehman, M. M., "An Introduction to Program Growth Dynamics", in *Statistical Computer Performance Evaluation*, 1972, pp. 503-511.

[44] Lindvall, M., "Monitoring and Measuring the Change-Prediction Process at Different Granularity Levels: An Empirical Study", *Software Process Improvement and Practice 4* 1998, pp. 3-10.

[45] Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., and Mockus, A., "Does Code Decay? Assessing the Evidence from Change Management Data", *TSE*, vol. 27, no. 1, 2001, pp. 1-12.