Prediction of the Testing Effort for the Safety Certification of Open-Source Software: A Case Study on a Real-Time Operating System

Domenico Cotroneo, Domenico Di Leo, Roberto Natella, Roberto Pietrantuono
DIETI department, Università degli Studi di Napoli Federico II, Via Claudio 21, 80125, Napoli, Italy
Critiware s.r.l., Via Carlo Poerio 89/A, 80121, Napoli, Italy
{cotroneo, roberto.natella, roberto.pietrantuono}@unina.it, domenico.dileo@critiware.com

***Abstract*—The reuse of Open Source Software (OSS) for safety-critical systems is seen with interest by industries, such as automotive, medical, and aerospace, as it enables shorter time-to-market and lower development costs. However, safety certification demands to supply evidence about OSS quality, and a gap analysis is needed to assess if the cost to produce certification evidence is worthwhile.**

This paper presents an empirical study on an open-source RTOS (RTEMS). The study investigates the relationship between software complexity and the effort to achieve a high test coverage, which is one of the most impacting activity for certification. The objective is to figure out if, and to what extent, it is possible to predict such effort preventively, by looking at software complexity metrics. This would enable a preliminary screening and benchmarking of OSS items, supporting strategic decision making. The study shows that combining metrics with classifiers can achieve a good prediction accuracy.

***Index Terms*—Real-Time OS; Open Source Software; Effort prediction; Testing; Safety Certification; Software Complexity Metrics**

I. Introduction

Open Source Software (OSS) has attracted much industrial interest over the last decades, and has enabled new business models. Companies can share OSS code not in their core business, and focus their efforts on value-added products and services. Moreover, customers can benefit from reduced costs and time-to-market, community and professional support, license flexibility, increased number of suppliers, market innovation, and avoid vendor lock-in. For these reasons, industries in safety-critical domains, such as automotive, medical, and aerospace are considering the adoption of OSS components [1], [2], [3], [4]. Examples are operating systems (OS), image processing libraries, development tools, middleware and data management software [5], [6], [7].

Despite the benefits and the large availability of OSS components, *safety certification* poses new, and still open, challenges for OSS. Certification requires evidence about the software development process, to show that software functions have been carefully designed, implemented and verified with safety concerns in mind. Given their importance to safety certification, this evidence is planned at the beginning of the software lifecycle. However, in most cases OSS components are not developed for safety-critical contexts, and lacks certification evidence: therefore, before reusing OSS, developers must perform a gap analysis to determine whether certification requirements are fulfilled, and must close any pending gap by supplying the missing evidence.

As an example of how challenging can be to reuse OSS in safety-critical systems, we can consider the case of the DO-178B recommendations for the avionic domain [8]. These recommendations allow the reuse of "previously-developed software", provided that safety evidence is produced from alternative sources such as development data, service history, additional testing, reverse engineering, and wrappers [9], [10, chap. 24]. However, this can take a significant amount of time and resources.

The reuse of OSS would be simpler if companies knew how much effort is needed before undertaking the task of producing safety evidence:

- A company could decide whether to go for OSS components if the effort to produce evidence is small enough, or otherwise it could develop its own component in-house;
- If several alternative OSS components are available (for example, different OS with similar functionalities), the company could select the one that requires less effort to produce safety evidence;
- The company could allocate an appropriate amount of time and resources to produce safety evidence.

Predicting the certification effort for OSS components is thus an important challenge. In the framework of this general objective, this paper considers the problem of *predicting the testing effort* for OSS components: the goal is to estimate in advance how much testing effort, in terms of amount and size of test cases, is needed to achieve a high statement coverage, which is a typical requirement imposed by safety standards.

We present an empirical analysis in the context of a well-known open-source real-time OS (RTOS), namely RTEMS [11]. We analyze, from a retrospective point of view, the test cases that were developed for RTEMS, and we evaluate how much testing effort has been spent for achieving a high statement coverage for this RTOS. Moreover, we propose an approach, based on software complexity metrics and machine learning, to predict the testing effort. We evaluate this approach on RTEMS, by predicting which components of RTEMS require a high testing effort. The experimental analysis shows that high-effort components cannot trivially be predicted by a simple visual analysis of software complexity metrics, and that combining several metrics with machine learning can achieve a good prediction accuracy.

The paper is structured as follows. In sections II and III, we provide background and an architectural analysis of the RTEMS case study, and on the software complexity metrics used in this study. In section IV, we analyze the test cases and the testing effort for RTEMS. In section V, we describe the effort prediction approach. In section VI, we present experimental results. Section VII discusses related work. The paper concludes with section VIII.

II. The RTEMS case study

RTEMS (Real Time Executive for Multiprocessor system) is an open-source real-time operating system, which has been developed by the OAR Corporation since the late 1980s. It was designed for embedded systems and provides a high performance environment for real-time applications in safety-critical domains (military systems, medical devices, space flight and so on). Well-known projects that adopted RTEMS are the Herschel and Planck satellites of the European Space Agency (ESA), and the Mars Rover Curiosity of the National Aeronautics and Space Administration (NASA).

It is worth mentioning that the ESA and the NASA, with the support of several contractors and the RTEMS developers community [12], [13], [14], [15], invested a significant effort for the *space qualification* of RTEMS. Space qualification means that this RTOS has been thoroughly revised, documented, and tested according to the guidelines of certification standards. When the RTOS is integrated in a safety-critical system, the evidence produced by these efforts are re-used to ease the safety certification process of the system. Qualification activities for RTEMS included the improvement of the testing toolchain; code re-engineering, in order to improve testability and remove dead code; and the addition of several new test cases, in order to achieve a high test coverage.

This RTOS supports both Ada and C applications, and several standard APIs (Application Programming Interface), including POSIX, ARINC 653, and ITRON. It has been ported to several processor families (including ARM, Intel x86, MIPS, PowerPC, Atmel AVR, and several others). It is a feature-rich RTOS, including: multitasking capabilities; homogeneous and heterogeneous multiprocessor systems; event-driven, priority-based, and preemptive scheduling; optional rate monotonic scheduling; intertask communication and synchronization; priority inheritance; responsive interrupt management; dynamic memory allocation. In this paper, we refer to version 4.11.

The architecture of RTEMS is composed by two main layers: the *Resource Managers*, and the *Core* (Figure 1). The Core consists of *Handlers*, that is, collections of routines for scheduling, dispatching, object management and other basic functions. These Handlers are not meant to be used by applications; instead, the *Managers* provide higher-level primitives for accessing and controlling resources, by using the functions provided by the Core. RTEMS is modular, and allows the user to configure which Managers to include at build time, in order to reduce memory consumption.

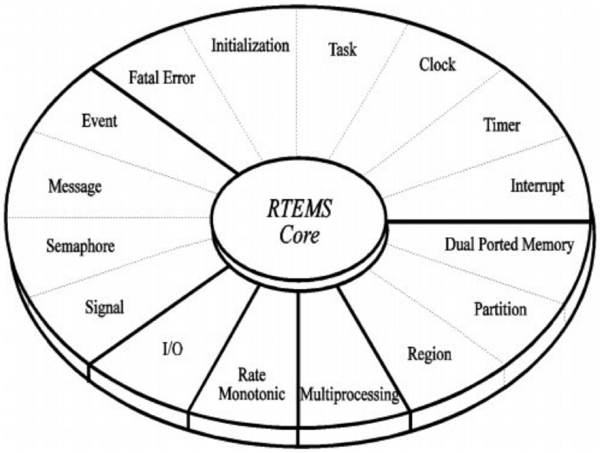


Fig. 1. The RTEMS architecture.

Since RTEMS, like any OS, includes several heterogeneous parts with distinct features and responsibilities (such as memory management, task management, etc.), we divide it in a set of *components*. We analyze components as separate entities, and investigate the relationship between the testing effort and the nature of the component under test. Each component (listed in Table I) exposes an API, which is exercised by a set of test cases. Each component includes a Manager, and one or more Handlers that are related to the Manager, according to the architecture and documentation of RTEMS (e.g., we group the Manager and the Handlers that provide task-related functions).

The test cases are available on the RTEMS source code repository. Each test case is a C program, that consists of one or more source files. Typically, it has a source file that contains an *Init* (initialization) procedure, and one or more files with a definition of RTEMS tasks, which invoke the functions of RTEMS Managers and (indirectly) the Core Handlers used by the Managers. For example, the test case in Figure 2 creates and executes a set of tasks, including a task called *TA1*; in turn, this task calls a routine of the Task Manager, namely *rtems_task_self*.

A test case is compiled and linked to RTEMS, obtaining a binary executable. The executable is then loaded on the execution environment (in our case, the QEMU emulator), and executed. RTEMS provides a set of tools to automate the execution of test cases, and the generation of test reports. We adopt the *QEMU-Couverture* framework [16] to analyze the coverage of test cases. This framework is non-intrusive and does not change neither the source nor binary code of the program. Instead, the QEMU emulator is modified to collect an execution trace. The trace is analyzed off-line and mapped back to the original source code, by using the source-to-object-code mapping information provided by the debugging information in the executable program.

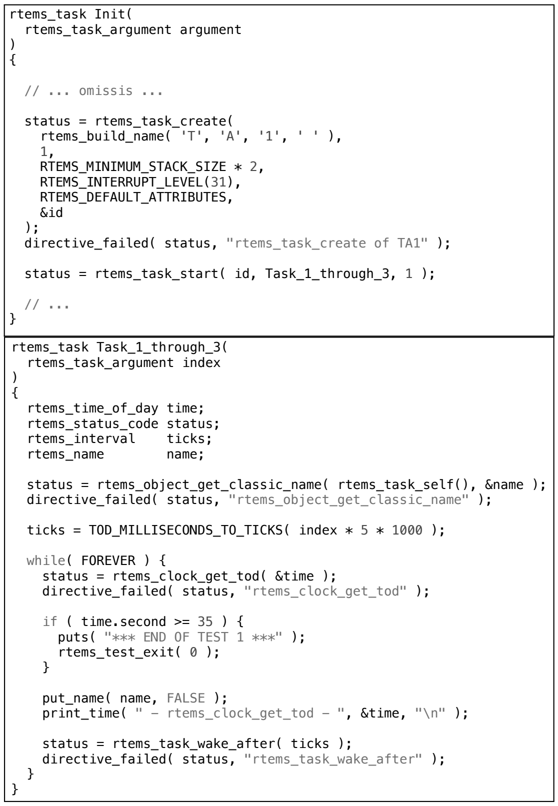
TABLE I
COMPONENTS ANALYZED IN THIS STUDY. A COMPONENT INCLUDES A MANAGER AND ONE OR MORE HANDLERS FROM THE RTEMS CORE.

Component	**Managers and Handlers**
Task	Task Manager, Thread Handler, Thread States Handler, Context Handler.
Event	Event Manager, Event Handler.
Clock	Clock Manager, Time Of Day Handler, Timestamp, Watchdog Handler.
Message	Message Manager, Message Queue Handler.
Initialization	Initialization Manager, System State Handler.
Interrupt	Interrupt Manager, ISR Handler.
Semaphore	Semaphore Manager, Semaphore Handler, Mutex Handler, API Mutex, Handler, RWLock Handler, Spinlock Handler.
Barrier	Barrier Manager, Barrier Handler.
User Extensions	User Extension Manager, User Extension Handler.
Dual Ported Memory	Dual Ported Memory Manager, Address Handler.
Fatal Error	Fatal Error Manager, Internal Error Handler.
Region	Region Manager, Thread Queue Handler, Heap Handler.
Rate Monotonic	Rate Monotonic Manager, Scheduler Handler, Bitfield Handler, Priority Handler.
I/O	I/O Manager.
Signal	Signal Manager.
Partition	Partition Manager.
API Extension	API Extension Handler.
Timer	Timer Manager.
Memory Management	Stack Handler, Protected Heap Handler, Workspace Handler.
Low Level Services	RedBlack Tree Handler, Helpers, Chain Handler.

III. SOFTWARE COMPLEXITY METRICS

In this paper, we analyze the *testing effort* needed to achieve a high coverage of RTEMS components (according to Table I), and we investigate the relationship between testing effort and software complexity metrics of the components under test. The metrics considered in this paper are summarized hereafter[1]. We include several metrics that, in the past, turned out to be correlated with bug density in complex software [17]:

Lines of code: The number of lines of code is probably the simplest software metric, but despite its simplicity, there is often a confusion about which parts of the code should be counted (comments, declarative parts etc.). We consider a common definition of this metric, given by [18]: “A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements”. We consider both the total number of lines of code in a component (*CountLineCode*) and the average number per function within the component (*AvgCountLineCode*).



```
rtems_task Init(
  rtems_task_argument argument
)
{

  // ... omissis ...

  status = rtems_task_create(
    rtems_build_name( 'T', 'A', '1', ' ' ),
    1,
    RTEMS_MINIMUM_STACK_SIZE * 2,
    RTEMS_INTERRUPT_LEVEL(31),
    RTEMS_DEFAULT_ATTRIBUTES,
    &id
  );
  directive_failed( status, "rtems_task_create of TA1" );

  status = rtems_task_start( id, Task_1_through_3, 1 );

  // ...
}
```

```
rtems_task Task_1_through_3(
  rtems_task_argument index
)
{
  rtems_time_of_day time;
  rtems_status_code status;
  rtems_interval    ticks;
  rtems_name        name;

  status = rtems_object_get_classic_name( rtems_task_self(), &name );
  directive_failed( status, "rtems_object_get_classic_name" );

  ticks = TOD_MILLISECONDS_TO_TICKS( index * 5 * 1000 );

  while( FOREVER ) {
    status = rtems_clock_get_tod( &time );
    directive_failed( status, "rtems_clock_get_tod" );

    if ( time.second >= 35 ) {
      puts( "*** END OF TEST 1 ***" );
      rtems_test_exit( 0 );
    }

    put_name( name, FALSE );
    print_time( " - rtems_clock_get_tod - ", &time, "\n" );

    status = rtems_task_wake_after( ticks );
    directive_failed( status, "rtems_task_wake_after" );
  }
}
```

Fig. 2. Example of a test case in RTEMS.

Cyclomatic complexity: It indicates the program control flow complexity, as defined by McCabe in [19]. It measures the number of independent paths in the control graph of the program. We consider the overall complexity of the component (*Cyclomatic*) and the average complexity per function (*AvgCyclomatic*).

Halstead’s metrics [20]: These metrics are based on the number of paths in the code and the number of operands and operators, respectively. We hypothesize that they are connected to test effort, since the latter may be proportional to the complex structures of a program. Halstead’s metrics consider:

- *n1*: the number of distinct operators.
- *n2*: the number of distinct operands.
- *N1*: the total number of operators.
- *N2*: the total number of operands.

Variables and constants in the source code are considered operands. Operators include all the punctuation marks, arithmetic symbols, keywords (e.g., *if*, *while*, etc.), special symbols and functional names. The following metrics are then derived:

- $n = n1 + n2$: the program vocabulary.
- $N = N1 + N2$: the program length.
- $V = N \; log_2 n$: the program volume, it could be considered the required bit to represent the program.

[1] Metrics are computed using the Understand tool: *http://www.scitools.com/*

- *D=(n1/2)(N2/n2)*: the program difficulty.
- *E=D · V*: the programming effort. Halstead derives also the time T needed to write a program, assuming that the number S of mental discriminations per second that a human can do is known: *T=E/S*, where S is estimated to be between 5 and 20, and to be 18 for programming.

We consider the total and the average over functions of each of these metrics per component.

FanOut: this metric indicates the degree of coupling among modules of a modularized software system [21]. In particular, it specifies the number of modules called by the module under investigation. Typically, the modules with a high *FanOut* are the ones on the highest layers of the design structure. The component-level (*FanOut*) and the average over functions (*AvgFanOut*) is considered.

FanIn: this metric is also a measure of the degree of coupling among software modules [21]. It is a count of all other modules that call the module under investigation. Usually, the modules with a high *FanIn* are small modules which do some simple task needed by a lot of other modules. Again, the component-level (*FanIn*) and the average over functions (*AvgFanIn*) is considered.

IV. Analysis of test coverage

Our analysis requires to quantify the efforts needed to test software components. Ideally, the testing effort is represented by the amount of time and of resources that are spent on testing activities, to reach a given testing goal (e.g., achieving high statement coverage). These activities include the design of test inputs and of test oracles, the preparation of a test execution environment, the development of test case program, the interpretation of results, the iterative refinement of test cases, etc.. Unfortunately, quantifying the efforts behind these activities is difficult, especially for OSS projects, which follow a distributed and decentralized development process. However, intuition suggests that the higher the quantity and the size of the test cases of a component, the higher the amount of effort that has been spent for testing that component. Therefore, we consider the following two indicators as proxies for quantifying the testing effort: the *number of test programs* (*Ntests*), and the *number of lines of test code* (*TestLoC*). The number of lines of code is widely used in software engineering, such as in COCOMO [22] and FPA [23], to estimate the software development effort. Moreover, the number of test cases is often used in studies about test coverage, such as [24]. The definition of line of test code is the same of section III, where the program is a test case (such as in Figure 2).

We study the relationship between testing effort, and the coverage that results from the testing effort. However, we do not limit the analysis to the coverage obtained from the full test suite, for two reasons. The first reason is that some of the test cases in the test suite can be redundant, that is, the test suite may include test cases that are not strictly required towards the goal of covering the code (for example, test cases may overlap each other, and they may have been introduced for the sake of additional confidence about the tested component). The second reason is that the test suites often do not all achieve full coverage (e.g., 100% statements), and different test suites of different components may have achieved a different coverage of components' code. In order to make comparisons between the test suites (and the testing effort behind them) of different components, we would need to have the same coverage level.

Therefore, we obtain a more comprehensive characterization of the test suite of each component. We analyze how the coverage of the component grows as more and more test cases are added in the test suite. By analyzing the trend of the coverage growth (e.g., by fitting a model), we can estimate how much effort is needed to achieve a fixed coverage goal. For example, if the component is easy-to-test, then the coverage quickly grows even with a small test effort (e.g., using few and/or small test cases). We identify the relation between testing effort and coverage, we perform the following steps (starting from $k = 1$ to $k = n$):

1) We consider the possible (unordered) subsets of test cases with cardinality k, among the n test cases of the test suite of the component.
2) We compute the average *cumulative test coverage* that would be obtained by executing k test cases.
3) We estimate the effort for developing k test cases, by computing average *number of lines of code* of k test programs.

To explain the above steps, we refer to a hypothetical example of Figure 3 of a component test suite with three test cases (t_1, t_2, t_3). The three test cases have, respectively, 10, 30, and 65 LoC, and achieve a coverage equal to 15%, 35%, and 55% of the component. Therefore, the average size of a test suite with $k = 1$ test cases is 35 LoC (i.e., the average of the three test cases), and the average coverage of $k = 1$ test cases is 35%. If we consider $k = 2$, we have again three possible combinations: $< t_1, t_2 >$; $< t_1, t_3 >$; $< t_2, t_3 >$. In this example, the average size of a pair of test case is 70 LoC (i.e., the average of the sums of pairs of test cases), and the average coverage is 56%. For $k = 3$, we have only one possible combination: the test suite size is 105 LoC, and the cumulative coverage is 75%.

The coverage contribution of k test cases is strictly dependent on "which tests" are taken to construct a subset. We avoid this dependency (that could potentially bias the results) by considering several possible combinations of k test cases, and we compute the average of both the coverage and the size of the possible combinations of k test cases.

Figures 4, 5, 6, 7, and 8 show the cumulative test coverage as a function of the amount of test cases, in terms of number of lines of code. As showed in Figure 3, there is one data point (the average test cases size, and cumulative test coverage) for each cardinality k.

These plots point out that test coverage quickly increases with the initial tests, then the growth gradually slows down. The test cases exhibit diminishing returns as the amount of the tests is increased: this is an usual phenomenon in software testing, since the core paths of the software are easily covered by few test cases, while the remaining paths (that represent

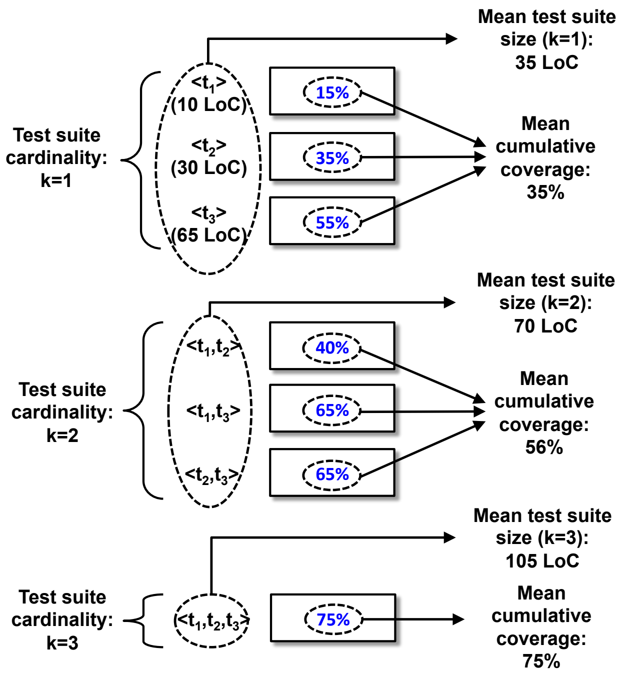

Mean test suite size (k=1): 35 LoC
Test suite cardinality: k=1
<t1> (10 LoC)
<t2> (30 LoC)
<t3> (65 LoC)
15%
35%
55%
Mean cumulative coverage: 35%
Mean test suite size (k=2): 70 LoC
Test suite cardinality: k=2
<t1,t2>
<t1,t3>
<t2,t3>
40%
65%
65%
Mean cumulative coverage: 56%
Mean test suite size (k=3): 105 LoC
Test suite cardinality: k=3
<t1,t2,t3>
75%
Mean cumulative coverage: 75%

Fig. 3. Example of test complexity metrics and test coverage.

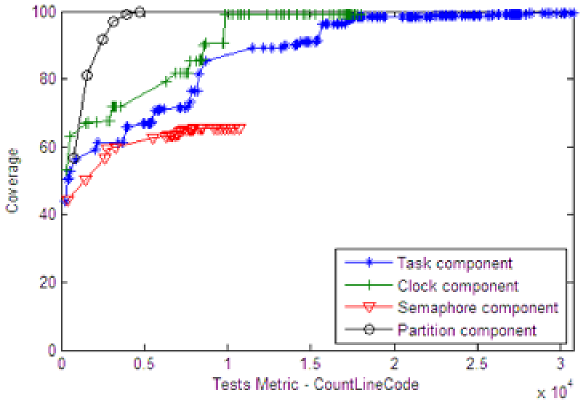


Fig. 4. Test coverage as a function of the amount of test case LoCs (Task, Clock, Semaphore, Partition).

corner cases of the program) each require specific test cases to covered them. The test coverage converges to a high percentage (90% or higher) for almost all RTEMS components. The amount of tests to converge to a high percentage ranges between 1 kLoC (for example, Initialization, Message, and Barrier) to about 10 kLoC (for example, Clock and Task). However, achieving 100% coverage for some components (again, Clock and Task) can take a significant effort (between 20 and 30 kLoC) due to diminishing returns.

From the analysis of these plots, there are few components that seem to be much more difficult to test than other components, since they require a higher amount of test cases. Ideally, it would be useful to identify these "difficult" components before performing any testing activity. However, predicting the

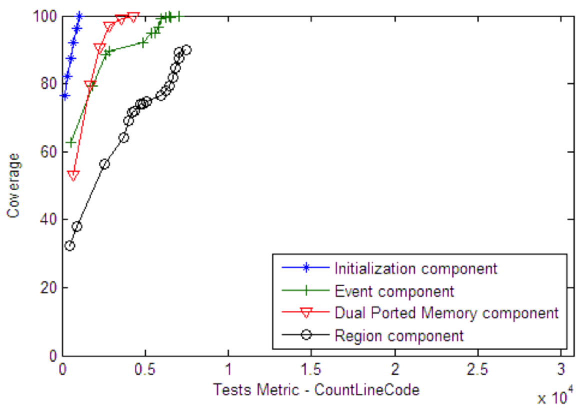


Fig. 5. Test coverage as a function of the amount of test case LoCs (Initialization, Event, Dual Ported Memory, Region).

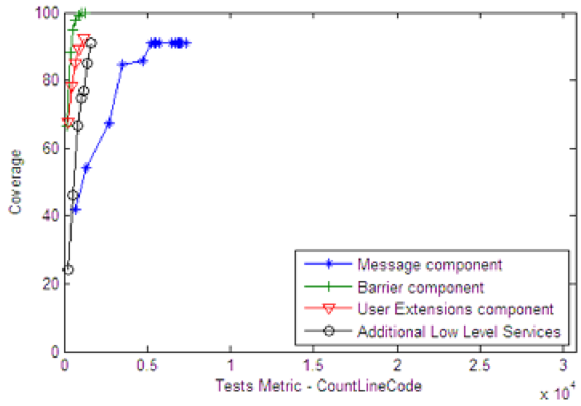


Fig. 6. Test coverage as a function of the amount of test case LoCs (Message, Barrier, User Extensions, Additional Low-Level Services).

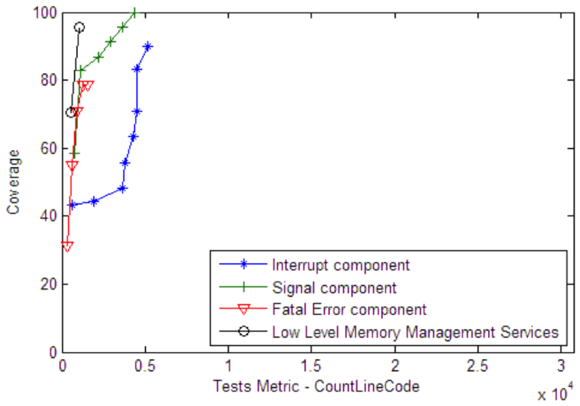


Fig. 7. Test coverage as a function of the amount of test case LoCs (Interrupt, Signal, Fatal Error, Low-Level Memory Management).

testing effort is a challenging task, since a cursory inspection of software complexity metrics does not reveal the difficult-to-test components. This can be seen in Figures 9 and 10,

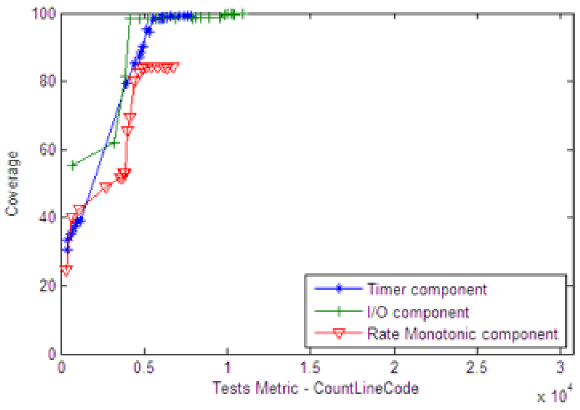


Fig. 8. Test coverage as a function of the amount of test case LoCs (Timer, I/O, Rate Monotonic).

which show, for each RTEMS component, respectively the distribution of the number of LoC and of the cyclomatic complexity of functions in the component. The complexity metrics for difficult-to-test components (such as Clock and Task) are similar to other, easier-to-test components (such as Dual Ported Memory and Event). Thus, these components cannot be identified only by a visual analysis or by setting a "cautionary" threshold on some software complexity metric. This result can be explained by observing that in complex software, such as RTEMS (which is a concurrent software, and provides a rich set of inter-dependent features and data structures), the testing effort is not only influenced by the number of lines of code or paths, and these metrics do not immediately point out the tricky parts of the system.

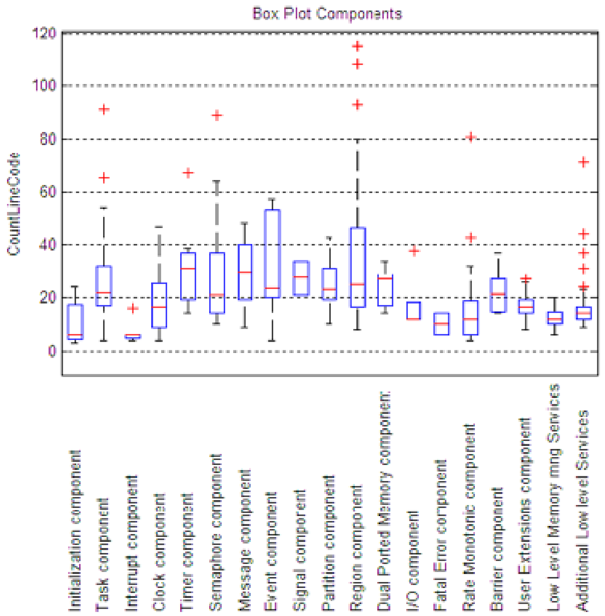


Fig. 9. Distribution of number of Lines of Code (LoC) of functions of RTEMS components.

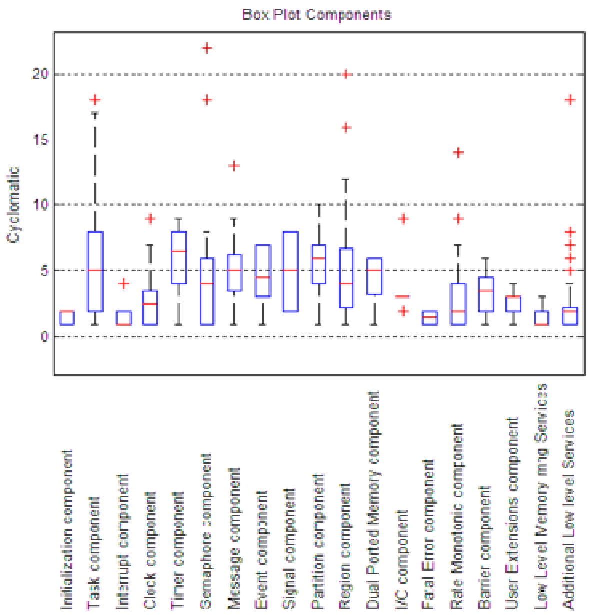


Fig. 10. Distribution of cyclomatic complexity of functions of RTEMS components.

V. Prediction approach

The previous analysis provided insights on the testing effort for a RTOS. We found that several components require similar testing efforts (roughly 1 kLoC of test code), but few components require an order of magnitude higher amount of effort (roughly 10 kLoC of test code). We also found that components have differences in terms of software complexity metrics, but difficult-to-test components cannot be identified by a simple visual analysis of these metrics. In the following, we address this problem by combining software complexity metrics using statistical prediction models.

A. *Effort estimation as a statistical prediction problem*

To estimate the testing effort, two approaches are possible. The first approach is to use a **classifier** to discriminate between difficult-to-test components and the remaining ones. This type of statistical prediction provides a *categorical* indication of the testing effort for a software component, which can be used for decision making. For example, components that exhibit a "high" difficulty can be assigned to teams with more resources or better skills; or an equivalent component with "low" difficulty could be preferred instead of a difficult one.

The second approach is to use **regression** to get a quantification of the expected testing effort, such as to predict the number of test cases or the amount of testing code. This type of prediction provides a *numerical* indication of the testing effort. For example, the components can be rated according to the predicted effort indication; or, the amount of resources can be proportional to the predicted effort indication.

Both these approaches build a model from a *training* set of samples (for which both the prediction outcome and the attributes of the samples are known), using a *learning*

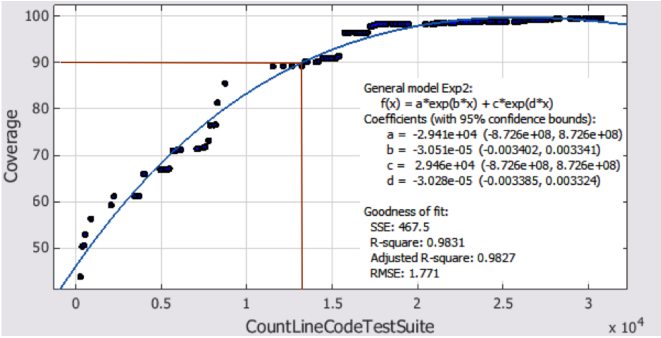


Fig. 11. Example of computation of test effort (in LoC) required to achieve 90% coverage.

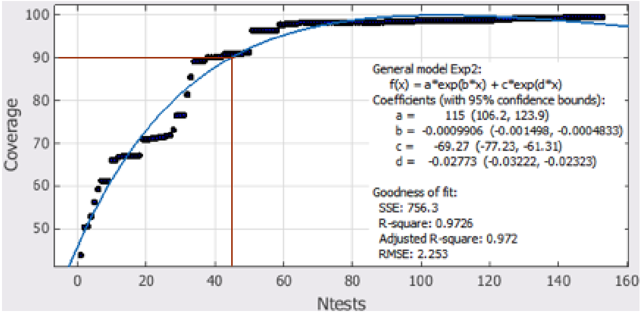


Fig. 12. Example of computation of test effort (in number of test programs) required to achieve 90% coverage.

TABLE II
DEFINITION OF EFFORT CLASSES, USING THE NUMBER OF LINES OF TEST CODE.

Component	Effort	2-Classes (50%)	2-Classes (75%)
Task	13290	BigEffort	BigEffort
Clock	9037	BigEffort	BigEffort
Region	7406	BigEffort	BigEffort
Message	6272	BigEffort	BigEffort
Rate Monotonic	6155	BigEffort	LittleEffort
Timer	5791	BigEffort	LittleEffort
Interrupt	5148	BigEffort	LittleEffort
I/O	4487	BigEffort	LittleEffort
Event	3450	LittleEffort	LittleEffort
Dual Ported Memory	2933	LittleEffort	LittleEffort
Signal	2629	LittleEffort	LittleEffort
Partition	2232	LittleEffort	LittleEffort
Fatal Error	1640	LittleEffort	LittleEffort
Low-Level Services	1443	LittleEffort	LittleEffort
User Extensions	978,2	LittleEffort	LittleEffort
Initialization	627,8	LittleEffort	LittleEffort
Barrier	488,3	LittleEffort	LittleEffort

algorithm. The model can make predictions for new samples outside the training set (for which only the attributes are known). In our context, a sample represents a component under test; the attributes are the software complexity metrics of the component; the prediction outcome is either the class of effort (for classification) or a numerical indication (for regression).

For regression, we consider two alternative effort indicators as dependent variable, namely the *amount of testing code* (in terms of number of lines of code of test programs), and the *number of test programs*. As for classifiers, we need to divide the components among a set of classes. Again, we consider the number of lines of test code and the number of test programs, and classify the components using thresholds.

In order to apply a classifier to RTEMS, we introduce two classes. The choice of using two classes is based on the previous results, where we observed that many components require a similar testing effort, while few components require a much higher effort.

To assign components to classes, we consider the test effort (respectively, number of lines of test code, and number of test cases) needed to achieve 90% coverage. We do not use the effort to achieve 100% coverage, since the final test coverage of some components is less than 100%, and the achieved test coverage is not the same for all components. The 90% coverage threshold is a relatively high coverage level, and represents a significant part of the effort spent for testing. Moreover, in the case of RTEMS, most of components achieve this level of coverage. We only excluded the Low Level Memory Management Services and Semaphore components from the analysis, since the former had a very small number of test cases, and the latter does not reach 90% coverage.

To get the test effort required for 90% coverage, we fitted the curves of the cumulative test coverage (Figures 4, 5, 6, 7, and 8), and computed the testing effort at 90%. For example, Figures 11 and 12 show the fitting for the Task component with respect to the number of lines of test code, and of the number of test programs, and the estimated testing effort to reach 90% statement coverage of the component (respectively, 13,290 LoC and 45 test programs).

Finally, we split components in two classes, according to the testing effort spent to achieve 90% coverage. Tables II and III rate the components with respect to the number of lines of test code, and to the number of test programs. We consider two possible thresholds to identify difficult-to-test components, namely 50% (i.e., half of the components are labelled as difficult) and 75% (i.e., the top quarter of the components are labelled as difficult). These classes represent the datasets on which we will evaluate the effectiveness of classifiers at predicting the testing effort.

B. Prediction algorithms

Regarding binary classification, the following machine learning algorithms are considered.

Naive Bayes (*NB*): this classifier estimates *a posteriori* probability of the hypothesis H (e.g., "a module is prone to require high effort"), given that an evidence E has been

TABLE III
DEFINITION OF EFFORT CLASSES, USING THE NUMBER OF TEST PROGRAMS.

Component	Effort	2-Classes (50%)	2-Classes (75%)
Task	45	BigEffort	BigEffort
Clock	28	BigEffort	BigEffort
Region	19	BigEffort	BigEffort
Rate Monotonic	16	BigEffort	BigEffort
Timer	13	BigEffort	LittleEffort
Interrupt	8	BigEffort	LittleEffort
Message	7	BigEffort	LittleEffort
Low-Level Services	6	BigEffort	LittleEffort
Fatal Error	5	LittleEffort	LittleEffort
Initialization	4	LittleEffort	LittleEffort
I/O	4	LittleEffort	LittleEffort
Event	4	LittleEffort	LittleEffort
Signal	4	LittleEffort	LittleEffort
User Extensions	4	LittleEffort	LittleEffort
Partition	3	LittleEffort	LittleEffort
Dual Ported Memory	3	LittleEffort	LittleEffort
Barrier	2	LittleEffort	LittleEffort

observed. The evidence E consists of any piece of information that is collected and analyzed for classification purposes. Many sources of information are typically considered, which correspond to the *features* of interest, e.g., complexity metrics in our case. Let E_i be a software complexity metric. A fundamental assumption of a Naive Bayes classifier is that each feature E_i is conditionally independent of any other feature E_j, $j \neq i$. Given this assumption, the *a posteriori* probability can be obtained as:

$$P(H|E) = \left[\prod_i P(E_i|H)\right] \frac{P(H)}{P(E)} \qquad (1)$$

since $P(E|H)$ can be obtained from the product of $P(E_i|H)$. This assumption is apparently oversimplifying, since features usually exhibit some degree of dependence among each other. Nevertheless, the Naive Bayes classifier performs well even when this assumption is violated by a wide margin [25], and it has been successfully adopted in several domains [26].

Bayesian network (*BayesNet*): it is a directed acyclic graph model representing a set of random variables (i.e., the graph nodes) and their conditional dependency (i.e., the graph edges). In a Bayesian network, each node is associated with a *conditional probability distribution* that depends on its parents.

The joint probability distribution for a set of random variables $X_1, \ldots, X_n$ of the Bayesian network is expressed as:

$$\begin{gathered} P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i|X_{i-1}, \ldots, X_1) = \\ \prod_{i=1}^{n} P(X_i|X_i\text{'s parents}) \end{gathered} \qquad (2)$$

Equation 2 can be used to compute the probability of a hypothesis H represented by a node of the network, given the conditional probability distributions of each node, and given a set of observed values. In this study, nodes represent complexity metrics and the hypothesis is "a component requires a big testing effort". A Bayesian network is a more accurate model than a naive Bayes classifier, since it makes weaker assumptions about independence of random variables: in a Bayesian network, the structure of the network and the conditional probability distributions can model complex relationships between random variables. On the other hand, it is considerably more complex to build than a naive Bayes classifier. In this work, we consider a common algorithm to build Bayesian networks, namely *K2* [26].

Decision tree: it is a hierarchical set of questions that are used to classify an element. Questions are based on attributes of elements to classify, such as software complexity metrics (e.g., "is Cyclomatic greater than 10?") and are placed as nodes of the tree. A decision tree is obtained from a dataset using the C4.5 algorithm [26]. To classify a component, a metric is first compared with the threshold specified in the root node of the tree, in order to choose one of the two children nodes; this operation is repeated for each selected node, until a leaf is reached.

Logistic regression. Regression models represent the relationship between a dependent variable and several independent variables by using a parameterized function. In the case of *logistic regression*, the relationship is given by:

$$P(Y) = \frac{e^{c+a_1X_1+\ldots+a_nX_n}}{1+e^{c+a_1X_1+\ldots+a_nX_n}} \qquad (3)$$

where the features $X_1, \ldots, X_n$ are independent variables, and c and $a_1, \ldots, a_n$ are parameters of the function. This function is often adopted for binary classification problems, since it assumes values within the range $[0, 1]$ that can be interpreted as probability to belong to a class. The model can be trained using one of several numerical algorithms: a simple method consists in iteratively solving a sequence of weighted least-squares regression problems until the likelihood of the model converges to a maximum.

Support vector machine (SVM): SVMs were developed for classification problem by Vapnik in the late 1960s, and are known for their good generalization and easy adaptation at modeling non-linear functional relationships. They are based on the concept of decision planes that define decision boundaries. A decision plane is one that separates between a set of objects having different class memberships. We use the Weka implementation of the Sequential Minimal Optimization (SMO) algorithm [27] to train the SVM, which address the large quadratic programming (QP) problem for SVM training, by breaking it into a series of smallest possible QP problems.

Besides classification, we applied regression to try predicting the exact value of the testing effort depending on complexity metrics. We use the SVM implementation for regression to this aim. In fact, the original SVM was extended in 1996 by the introduction of the so-called ϵ-intensive loss function [28], introducing the ability of solving linear and non-linear regression estimation problems. We used the *SMOreg* algorithm, which is the Weka implementation of the SMO algorithm for regression problems.

VI. Evaluation

We first evaluate the effectiveness of effort prediction based on classification. We evaluate the ability of a classifier to correctly predict difficult-to-test components among a set of "unknown" components, using the *k-fold cross-validation* approach [29], [30], with $k = 3$. In this approach, the dataset is randomly divided in *k folds*, where $k - 1$ folds are adopted as training set, and the remaining fold is adopted as test set. This operation is repeated k times, by varying the fold adopted as test set at each repetition. Moreover, since the evaluation is influenced by the random split, the k-fold cross-validation process was repeated 10 times using different random splits.

The quality of prediction is then evaluated by classifying each sample in the test set using the classifier, and by comparing the predicted class with the actual class of the sample. We compute the following set of performance indicators, that are commonly adopted in machine learning studies [26]:

- **Precision**: Percentage of components that are correctly classified as difficult-to-test (*true positives*, TP) among components that are classified as difficult-to-test (both *true positives* and *false positives*, FP):

$$Precision = TP/(TP + FP)\ . \tag{4}$$

- **Recall**: Percentage of *true positives* among all components that are actually difficult-to-test (*true positives* and *false negatives*, FN):

$$Recall = TP/(TP + FN)\ . \tag{5}$$

- **F-measure**: Harmonic mean of *precision* and *recall*:

$$F - measure = (2 \cdot Pr \cdot Re)/(Pr + Re)\ . \tag{6}$$

- **Accuracy**: Percentage of modules correctly classified (either as *true positives* or as *true negatives*) among all components in the dataset:

$$Accuracy = (TP{+}TN)/(TP{+}TN{+}FP{+}FN)\ . \tag{7}$$

Tables IV, V, VI, VII provide the metrics of quality of prediction for the 5 classifiers, both in the case of the plain dataset, and in the case of the dataset processed with the logarithmic transform ("+log"). This transform replaces each numeric value with its logarithm, and extremely small values ($< 10^{-6}$) with $\ln(10^{-6})$. This transformation can improve, in some cases, the performance of classifiers when attributes exhibit many small values and a few much larger values [29]. Finally, we adopted the *Wilcoxon signed-rank test* [31] to assess whether the differences between classifiers are statistically significant; the best classifiers are highlighted in gray.

TABLE IV
Quality of prediction (effort by LoC, threshold at 50%).

Algorithm	Accuracy	Precision	Recall	F-measure
SVM	0.71	0.81	0.71	0.76
SVM+log	0.59	0.59	0.59	0.59
NB	0.71	0.71	0.71	0.71
NB+log	0.59	0.59	0.59	0.59
BayesNet	0.53	0.28	0.53	0.37
BayesNet+log	0.53	0.28	0.53	0.37
DecTree	0.71	0.71	0.71	0.71
DecTree+log	0.71	0.71	0.71	0.71
Logistic	0.59	0.6	0.59	0.59
Logistic+log	0.71	0.71	0.71	0.71

In the first type of prediction, effort classes are based on the number of lines of test code, with a threshold at 50% (Table II, "2-classes (50%)"). The quality of prediction (Table IV) was high in some cases, and comparable to results in previous literature on effort and defect prediction [29]. The best case is represented by the SVM classifier, with 0.71 accuracy, 0.81 precision (i.e., 8-out-of-10 components signaled by the classifier are actually difficult-to-test), and 0.71 recall (i.e., 3-out-of-10 difficult-to-test components are missed by the classifier). In the case of Naive Bayes, Decision Trees, and Logistic with logarithmic transformation, we obtain results close to the SVM classifier. However, we also get noticeable differences across classifiers, including cases of poor precision for Bayesian Networks, and a general decrease of quality with the logarithmic transform.

Figure 13 shows an example of decision tree, that was trained on the full dataset of Table IV. This figure confirms that the prediction of difficult-to-test components can benefit from the combination of several software complexity metrics. In this case, difficult-to-test components are denoted by a high fan-out, or by a small fan-out and number of lines of code but high cyclomatic complexity.

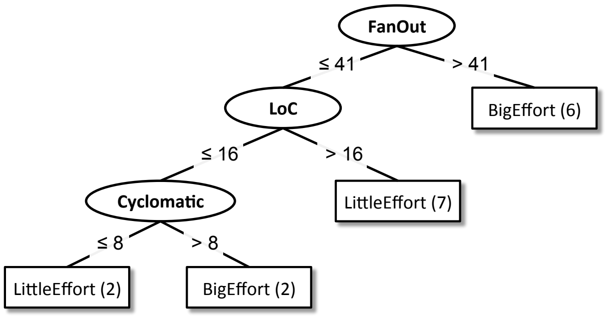


Fig. 13. Example of decision tree classifier (effort by LoC, threshold at 50%).

Table V shows the results on a similar analysis, but performed on the dataset obtained with a threshold of 75%

TABLE V
QUALITY OF PREDICTION (EFFORT BY LOC, THRESHOLD AT 75%).

Algorithm	Accuracy	Precision	Recall	F-measure
SVM	0.88	0.9	0.88	0.89
SVM+log	0.82	0.81	0.82	0.81
NB	0.88	0.9	0.88	0.89
NB+log	0.82	0.81	0.82	0.81
BayesNet	0.71	0.68	0.71	0.69
BayesNet+log	0.71	0.68	0.71	0.69
DecTree	0.59	0.68	0.59	0.63
DecTree+log	0.59	0.68	0.59	0.63
Logistic	0.71	0.73	0.71	0.72
Logistic+log	0.71	0.73	0.71	0.72

TABLE VI
QUALITY OF PREDICTION (EFFORT BY NUMBER OF TEST PROGRAMS, THRESHOLD AT 50%).

Algorithm	Accuracy	Precision	Recall	F-measure
SVM	0.65	0.65	0.65	0.65
SVM+log	0.76	0.78	0.76	0.77
NB	0.94	0.95	0.94	0.94
NB+log	0.76	0.78	0.76	0.77
BayesNet	0.88	0.9	0.88	0.89
BayesNet+log	0.94	0.95	0.94	0.94
DecTree	0.88	0.88	0.88	0.88
DecTree+log	0.88	0.88	0.88	0.88
Logistic	0.59	0.59	0.59	0.59
Logistic+log	0.65	0.65	0.65	0.65

on the number of lines of test code (Table II, “2-classes (75%)”). In this case, the set of difficult-to-test components is smaller. However, it seems that the quality of prediction can benefit from a more specific selection of the difficult-to-test components. In this case, the best classifiers were SVM and Naive Bayes, with a general improvement over the results from the analogous cases of Table IV. We believe that this result is due to the distribution the effort across components, in which the top-25% components exhibit a much higher effort than the other ones (see Tables II and III). Instead, the components between the 25% and the 50% of the ranking require a testing effort that is comparable to the components below the 50% of the ranking. This distribution makes it difficult for a classifier to discriminate mid-rank components as difficult-to-test.

Tables VI and VII show the quality of prediction in the case of effort measured in number of test programs, with a threshold set respectively on 50% and 75% (Table III). This effort metric exhibits even better prediction compared to the previous results. In the best cases (Decision Trees, Naive Bayes, and Bayesian Networks with logarithmic transformation), the measures reach 0.94. These results are explained by noting that the distribution of effort is even more skewed towards top-25% components, which reflects in a marked distinction between classes in terms of software complexity metrics.

Finally, we evaluate the quality of effort prediction by regression. We consider the following evaluation measures:

- The *mean* and the *relative absolute error* (MAE and RAE). The MAE represents the average difference between the absolute value of a prediction, and the actual absolute value of the effort. The RAE divides the error by the absolute value of the effort.
- The *mean* and the *relative root square error* (RMSE and RRSE). The RMSE reflects the quadratic error between the value of a prediction, and the actual value of the effort. The RRSE divides the quadratic error by the absolute value of the effort.

Again, the regression was evaluated with k-fold cross-validation, with $k = 3$. Table VIII shows the results of the evaluation. In the case of the prediction of the number of lines of test code, the average error is higher than 2 kLoC, and the relative error is close to 87%. In the case of the prediction of the number of test programs, the average error is higher than 6 test programs, and the relative error is close to 80%. These are negative results that discourage the adoption of regression for effort prediction, since the margin of error is comparable to the absolute value of the actual effort, which would lead to a high prediction uncertainty. We note that predicting the absolute value is a much challenging goal than classification, since prediction is not limited to few classes. This problem is also common in other prediction contexts [32], [33]: for example, in defect prediction, it is typically not feasible to predict the exact number of bugs in software modules, and the quality of prediction is much higher when it is based on binary classification (i.e., predicting fault-prone components among non-fault-prone ones). This evaluation confirms that

TABLE VII
QUALITY OF PREDICTION (EFFORT BY NUMBER OF TEST PROGRAMS, THRESHOLD AT 75%).

Algorithm	Accuracy	Precision	Recall	F-measure
SVM	0.82	0.81	0.82	0.81
SVM+log	0.82	0.81	0.82	0.81
NB	0.88	0.81	0.82	0.81
NB+log	0.82	0.81	0.82	0.81
BayesNet	0.88	0.88	0.88	0.88
BayesNet+log	0.88	0.88	0.88	0.88
DecTree	0.94	0.94	0.94	0.94
DecTree+log	0.94	0.94	0.94	0.94
Logistic	0.82	0.84	0.82	0.83
Logistic+log	0.76	0.76	0.76	0.76

this problem also holds for testing effort prediction, and points out the current limitations and the possibilities for future improvements in this context.

TABLE VIII
PREDICTION QUALITY WITH REGRESSION.

Proxy of Effort	**MAE**	**RMSE**	**RAE (%)**	**RRSE (%)**
Test LoC	2401.39	2938.61	87.39	87.79
Num. tests	6.55	9.01	80.39	80.58

VII. Related Work

The idea to use software metrics for prediction purpose is used in many contexts, such as to estimate the software development effort (COCOMO [22], FPA [23]), to predict software faulty modules [17], to predict types of faults [34], [33], etc.. This approach was introduced also in the context of software systems used in safety critical domains. For instance, the paper [35] proposes a framework, named PreCertification Kit (PK), that adopts software metrics to predict if an operating system could be selected for certification in a safety context. In order to build the PK, the authors define a reference model of the OS, they define also some properties (such as coverability, robustness, spatial and temporal partitioning) that could be used to establish whether the OS is suitable for the certification process, and then specify the method to collect the target values for software metrics. This work also adopts software metrics to predict features that can support the selection of an OS for certification in a safety context; hence, it can be seen as a contribution to the PK. In [36], Nagappan proposes to create and validate a metric suite, named Software Testing and Reliability Early Warning (STREW), that provides feedback on the quality of the testing effort and represents an early indicator of software reliability, for Object-Oriented (OO) languages. In the literature, there are also attempts to predict the effort needed to test a software system. The work in [37] shows an approach based on the Use Case Points (UCP), where the effort is calculated by a linear combination of several weights: actor weights, use case weights, and complexity factor weights, but the author conceived this method for web-based applications, thus it is not suitable for safety critical domains. In [38], Aranha and Borba introduce a different approach to the effort estimation, based on the Execution Points (EPs), which is a measure for the size and execution complexity of tests, they derive it from the test specifications. However, the proposed approach is closely related to mobile application domains, characterized by limited test automation technology. In [39], the authors conducted an analysis on two dissimilar industrial software projects (Avaya and Microsoft), they found in both cases that the test effort increases exponentially with the test coverage, using the number of changes to the test case classes as the proxy of effort. In [24], the authors investigate the effect of program structure on MC/DC coverage, and they confirmed the presence of this dependency studying six systems from the civil avionics domain and two toy examples. To assess the sensitivity of MC/DC to program structure, they considered two different implementations (named non-inlined and inlined) of each software system behaviorally equivalent, and they measured the coverage for increasing sets of tests (in a cumulative way). Finally, using the number of test cases as metric, they noted that the inlined implementation, on average, needs a greater number of tests to achieve the same level of coverage than non-inlined implementation. In this work, we measure coverage information in a cumulative manner, using an approach almost similar to [24]. We also proposed a proxy of test effort, but in contrast to [37], [38], [39], it is based on software metrics. Furthermore, we verified the exponential relationship between coverage and test effort, according to [39], and we introduce a methodology to measure and to predict the test effort.

VIII. Conclusion

Open-source is a valuable opportunity to promote software reuse and to improve the efficiency of software development in safety-critical domains. However, reusing OSS is a critical step that requires careful planning, and evidence that a high level of software quality has been achieved. In this paper, we considered the problem of predicting the effort to achieve a high test coverage in OSS software, in terms of number of test cases and amount of test code, and we considered RTEMS, a well-known RTOS adopted in safety-critical domains. To this goal, we leverage several software complexity metrics, that can be easily collected from a static analysis of the source code, and that provide insights on the testing efforts for a software component. We combined these metrics using machine learning, and got encouraging results on the prediction of difficult-to-test components through classification algorithms.

We remark that this case study is an initial effort towards a methodology for predicting the testing effort for OSS. As any empirical study, further studies are needed to confirm the feasibility of testing effort prediction in other RTOS and other kind of OSS components. Moreover, in this work we focused on statement coverage and on generic indicators of the testing effort, but other coverage criteria (e.g., branch and condition coverage) and effort indicators (e.g., component-, system-, or company- specific indicators) are also possible. The research field of testing effort prediction for OSS is still open to several further developments, including the following topics:

- To analyze at a fine-grain the effort required for the several testing sub-activities, such as: preparing the test execution environment (either on real hardware, or on a simulator); writing code for automating test execution; define test-oracles and test-inputs; collecting and report coverage information; writing “boilerplate” code to prepare test execution; remove dead-code, and refactor difficult-to-cover code in order to improve testability.
- To use testing effort prediction in combination with economic models, in order to provide quantitative evaluations of efforts for decision makers.
- To take into account the nature of sub-activities in effort prediction. Some of the sub-activities require a fixed

effort (e.g., "preparatory" activities), while other require a variable amount of efforts depending on factors such as the goal coverage level, the complexity and size of the component, etc.

- To analyze the effort for non-functional kinds of testing, such as stress testing, robustness and fault injection testing, and performance testing.
- To analyze the efforts for other verification activities beyond testing, such as code reviews, static and dynamic analysis, and model checking.
- To take into account in the effort prediction that testing must not necessarily started from scratch, and that OSS projects can provide an initial set of test cases and test tools to improve on.
- To evaluate the accuracy of testing effort prediction on different kinds of OSS components of potential interest for safety-critical systems. Examples are image processing libraries, development tools, middleware, and data management components.
- To benchmark alternative (but functionally similar) OSS projects, and evaluate the trade-off between the testing effort and the features and the maturity of alternative components.
- To investigate the impact of different test coverage criteria on the testing effort and on the accuracy of effort prediction.
- To improve the accuracy of "numerical" prediction of the testing effort, by exploring more robust techniques for data regression.

Acknowledgment

This work was supported by the *ICEBERG* (grant n.324356) and *CECRIS* (grant no. 324334) IAPP projects funded by the European Commission.

References

[1] J. S. Norris, "Mission-critical development with open source software: Lessons learned," *Software, IEEE*, vol. 21, no. 1, pp. 42–49, 2004.

[2] J. Zemlin, "The Next Battleground for Open Source Is Your Car," http://go.linuxfoundation.org/e/6342/attle-between-open-and-closed-/f8wh3/382522852, 2012.

[3] L. Morgan and P. Finnegan, "Open innovation in secondary software firms: an exploration of managers' perceptions of open source software," *ACM SIGMIS Database*, vol. 41, no. 1, pp. 76–95, 2010.

[4] The Economist, "Open-source medical devices: When code can kill or cure," http://www.economist.com/node/21556098, 2012.

[5] S. M. Sulaman, A. Orucevic-Alagic, M. Borg, K. Wnuk, M. Host, and J. L. de la Vara, "Development of Safety-Critical Software Systems Using Open Source Software–A Systematic Map," in *40th EUROMICRO Conf. on Software Eng. and Adv. Applications (SEAA)*, 2014.

[6] A. Corsaro, "CARDAMOM: A Next Generation Mission and Safety Critical Enterprise Middleware," in *IEEE Workshop on Software Tech. for Future Embedded and Ubiquitous Systems*, 2005.

[7] P. Gai, "Automotive and open-source: Current solutions and future developments," in *10th Automotive SPIN Italia Workshop*, 2012.

[8] RTCA, "DO-178B Software Considerations in Airborne Systems and Equipment Certification," *Requirements and Technical Concepts for Aviation*, 1992.

[9] J. Krodel, "Commercial Off-The-Shelf (COTS) Avionics Software Study," DTIC Document, Tech. Rep., 2001.

[10] L. Rierson, *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2013.

[11] The RTEMS Project, "Project Website," https://www.rtems.org/, 2014.

[12] H. Silva, J. Sousa, D. Freitas, S. Faustino, A. Constantino, and M. Coutinho, "RTEMS Improvement-Space Qualification of RTEMS Executive," *1st Simpósio de Informática-INFORUM, U. of Lisbon*, 2009.

[13] R. Barbosa, R. Maia, J. Esteves, L. Henriques, and D. Costa, "Robustness and Stress Testing RTEMS 4.5.0 - Software Dependability and Safety Evaluations," https://ftp.rtems.org/pub/rtems/publications/esa_validation_report_450/RTEMS_DependabilityTesting/, 2003.

[14] J. Seronie-Vivien and C. Cantenot, "RTEMS operating system qualification," in *Data Systems in Aerospace (DASIA)*, 2005.

[15] ECSS Secretariat, "Software product assurance," ESA-ESTEC, Tech. Rep. ECSS-Q-ST-80C, 2009.

[16] M. Bordin, C. Comar, T. Gingold, J. Guitton, O. Hainque, and T. Quinot, "Object and source coverage for critical applications with the Couverture open analysis framework," in *Embedded Real Time Sofware and Systems Conference (ERTS)*, 2010.

[17] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert systems with applications*, vol. 36, no. 4, 2009.

[18] S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc., 1986.

[19] T. J. McCabe, "A complexity measure," *IEEE Trans. on Software Eng.*, no. 4, pp. 308–320, 1976.

[20] M. H. Halstead, *Elements of software science*. Elsevier, 1977.

[21] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. on Software Eng.*, vol. SE-7, no. 5, 1981.

[22] B. W. Boehm, R. Madachy, B. Steece *et al.*, *Software cost estimation with COCOMO II*. Prentice Hall PTR, 2000.

[23] D. Garmus and D. Herron, *Function point analysis: measurement practices for successful software projects*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[24] A. Rajan, M. W. Whalen, and M. P. Heimdahl, "The effect of program and model structure on MC/DC test adequacy coverage," in *30th Intl. Conf. on Software Eng. (ICSE)*, 2008.

[25] P. Domingos and M. Pazzani, "On the Optimality of the Simple Bayesian Classifier under Zero-one Loss," *Machine Learning*, vol. 29, no. 2–3, 1997.

[26] I. Witten, E. Frank, and M. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers, 2011.

[27] J. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines," Tech. Rep. MSR-TR-98-14, 1998.

[28] H. Drucker, C. J. C. Burges, L. Kaufman, A. J. Smola, and V. Vapnik, "Support vector regression machines," in *Advances in Neural Information Processing Systems*, 1996.

[29] T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Trans. on Software Eng.*, vol. 33, no. 1, pp. 2–13, 2007.

[30] N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," in *28th Intl. Conf. on Software Eng. (ICSE)*, 2006.

[31] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. CRC Press, 2003.

[32] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. on Software Eng.*, vol. 31, no. 4, pp. 340–355, 2005.

[33] G. Carrozza, D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Analysis and prediction of mandelbugs in an industrial software system," in *Intl. Conf. on Software Testing, Verification and Validation (ICST)*, 2013, pp. 262–271.

[34] D. Cotroneo, R. Natella, and R. Pietrantuono, "Predicting aging-related bugs using software complexity metrics," *Performance Evaluation*, vol. 70, no. 3, 2013.

[35] D. Cotroneo, D. D. Leo, N. Silva, and R. Barbosa, "The PreCertification Kit for Operating Systems in Safety Domains," in *First Intl. Workshop on Software Certification (WoSoCER)*, 2011.

[36] N. Nagappan, "Toward a software testing and reliability early warning metric suite," in *26th Intl. Conf. on Software Eng. (ICSE)*, 2004.

[37] S. Nageswaran, "Test effort estimation using use case points," in *Quality Week*, 2001, pp. 1–6.

[38] E. Aranha and P. Borba, "Estimating manual test execution effort and capacity based on execution points," *International Journal of Computers and Applications*, vol. 31, no. 3, pp. 167–172, 2009.

[39] A. Mockus, N. Nagappan, and T. T. Dinh-Trong, "Test coverage and post-verification defects: A multiple case study," in *3rd Intl. Symp. on Empirical Software Engineering and Measurement (ESEM)*, 2009.