Scenario -based Black-Box Testing in COSMIC-FFP

Manar Abu Talib, Olga Ormandjieva, Alain Abran, Luigi Buglione

Abstract

A functional size measurement method, COSMIC-FFP, which was adopted in 2003 as the ISO/IEC 19761 standard, measures software functionality in terms of the data movements across and within the software boundary. It focuses on the functional user requirements of the software and is applicable throughout the development life cycle, from the requirements phase up and including to the implementation and maintenance phases. This paper extends the use of COSMIC-FFP for testing purposes by combining the functions measured by the COSMIC-FFP measurement procedure with the black box testing strategy. It leverages the advantage of COSMIC-FFP, which is its applicability during the early development phase once the specifications have been documented. This paper also investigates the applicability of Entropy measurement in terms of its use with COSMIC-FFP for assigning priorities to test cases.

1. Introduction

Testing represents a major effort within the whole of the software development cycle. The Guide to the Software Engineering Body of Knowledge (SWEBOK) [7] provides an overview, including references, of the basic and generally accepted notions underlying the Software Testing Knowledge Area. It describes testing as an activity performed for evaluating product quality, and for improving it, by identifying defects and problems. The definition of testing provided in [7] is as follows: "*Software testing consists of the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified expected behaviour.*"

The underlined terms are key concerns in software testing and we must explain them briefly prior to introducing COSMIC-FFP [1] into the testing context.

The term *dynamic* means that, when we want to test a program, we can execute it with differently valued inputs. The valued input not only means the input value alone, but also the specified input state. The input value alone is not sufficient to determine the outcome of a test. For example, a non deterministic system may react to the same input with different behaviors, depending on the system state. The non deterministic system described in Figure 1 may go from state *S2* to either state *S3* or *S4* while reading *a* as an input value. The input state is also necessary in order for the system to decide where to go. However, this is a design issue and outside the scope of this paper.

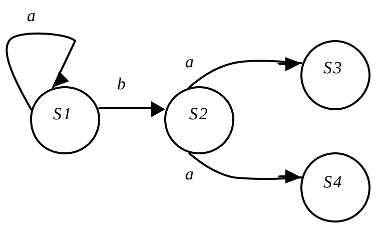

Figure 1: Example of a Non Deterministic System

Finite means having a test set (which includes test cases) while testing the system. In practice, an exhaustive test set can generally be considered infinite, even in simple programs. For example, a small program comparing two integers and returning the smaller number may require that the set of integers constitute the infinite test set. This is what makes testing a long and expensive and process. Testing implies a trade-off between limited resources and schedules, and inherently unlimited test requirements. As a result, we need a finite test set with which enough testing is conducted to obtain reasonable assurance of acceptable behavior.

The term *selected* refers to the way in which the finite test set has been chosen. The most difficult problem in generating the test cases is finding a test selection strategy that is both valid and reliable [8]. The power of a test case generation technique for detecting faults in an implementation is referred to as fault coverage [10]. Many different test methods exist (e.g. formal methods based on Finite State Machines (FSMs) and Extended Finite State Machines (EFSMs) [6]), which are all assumed to generate test suites containing test cases especially likely to reveal failures. These test methods can be compared according to their respective fault coverage. One method is considered more powerful than the other if it has better fault coverage.

Finally, to make the testing process useful, it must be possible, even though not always easy, to decide whether or not the observed outcomes or observed outputs of program execution are acceptable. This describes the term *expected* in the testing definition. It must be possible to determine whether or not the observed behavior is in conformity with user expectations, specifications, anticipated behavior requirements or reasonable expectations. The test pass/fail decision is, in the testing literature, commonly referred to as the "oracle problem" [6], [7] (see Figure 2).

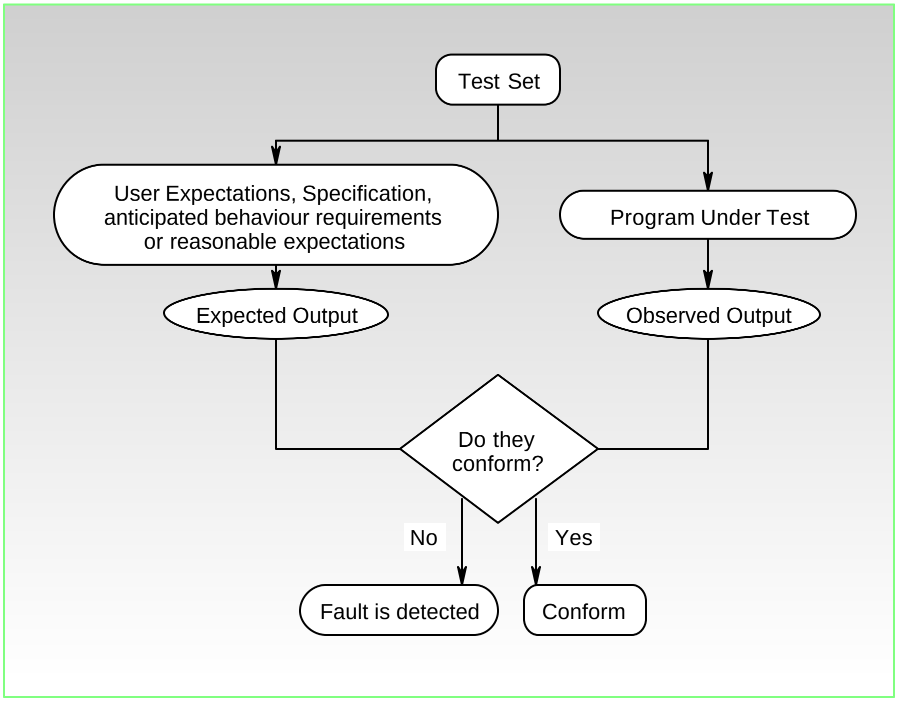

Figure 2: Oracle Methodology

In this paper, we investigate the use of the COSMIC-FFP method in the context of black-box use-case-driven testing using the Oracle methodology. In section 2, related work on scenario-based black-box testing is reviewed. Section 3 introduces the COSMIC-FFP method, and links it to the use-case technique. Our testing approach is introduced in section 4. The conclusions and directions for future work are outlined in section 5.

2. Related Work on Scenario-based Testing

There are three main testing strategies available: white-box testing, black-box testing and grey-box testing. In white-box testing, the test suite is generated from the implemented structures. In black-box testing, the structure of the implementation is not known, and the test cases are generated and executed from the specification of the required functionality at defined interfaces. In grey-box testing, the modular structure of the implementation is known, but the details of the programs within each component are not.

Scenario-based testing is a typical black box testing methodology at the system level, in which the scenarios depict the sequence of executions of the system, and the test cases can be derived from the use-case model and its corresponding UML diagrams [4, 5]. UML is the de facto industrial standard for modeling object-oriented software systems [13]. UML has 12 kinds of diagrams in 3 categories: structural diagrams (including class diagrams), behavioral diagrams (including use-case diagrams, interaction diagrams (e.g. Figure 5), activity diagrams, collaboration diagrams and state-chart diagrams) and model management diagrams.

In [4], a use case is defined as a collection of related scenarios describing actors and operations in a system, and these use cases can be organized hierarchically. Specifically, the root of the tree is the main use-case diagram, the middle branches are the low-level use-case diagrams, and the leaves are the sequence diagrams for each use case in the low-level use-case diagram. However, the main use cases may be at too abstract level to derive test cases. The authors propose transforming a use case into scenarios, then from a scenario into thin threads, and, finally, from thin threads into test cases – Figure 3.

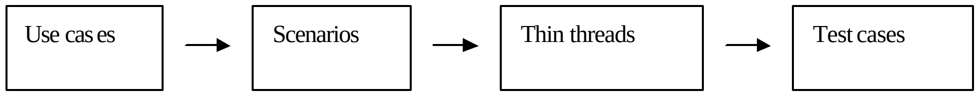

Figure 3: Testing with UML

A scenario is a specific sequence of actions and interactions between actors and the system [4]. A thin thread is a minimum-usage scenario in a software system. It is a complete data/message trace using a minimally representative sample of external input data transformed through an interconnected set of the system (architecture) to produce a minimally representative sample of external output data. The execution of a thin thread demonstrates that a method performs a specified function [4].

Our scenario-based testing approach differs from the related work in important ways, and is aimed at creating an optimal set of test cases in the context of the COSMIC-FFP method introduced in section 3.

3. COSMIC-FFP – ISO 19761

A functional size measurement method, COSMIC-FFP, which was adopted in 2003 as the ISO/IEC 19761 standard, measures software functionality in terms of the data movements across and within the software boundary. It focuses on the functional user requirements of the software and is applicable throughout the development life cycle, from the requirements phase up to and including the implementation and maintenance phases.

3.1. COSMIC-FFP overview

A key aspect of COSMIC-FFP [1] is the establishment of what is considered to be part of the software and what is considered to be part of the software's operating environment. Figure 4 below illustrates the generic flow of data from a functional perspective from which the following observations can be made [1]:

- Software is bounded by hardware in the "front-end" and "back-end" directions. In the first, or front-end, direction, the software used by a human user is bounded by I/O hardware such as mouse, keyboard, printer or display, or by engineered devices such as sensors or relays. In the second, or back-end, direction, the software is bounded by storage hardware like a hard disk and RAM and ROM memory.
- Four distinct types of movement can characterize the functional flow of data attributes. In the front-end direction, two types of movement (ENTRIES and EXITS) allow the exchange of data with the users across a "boundary". In the back-end direction, two types of movement (READS and WRITES) allow the exchange of data attributes with the storage hardware.

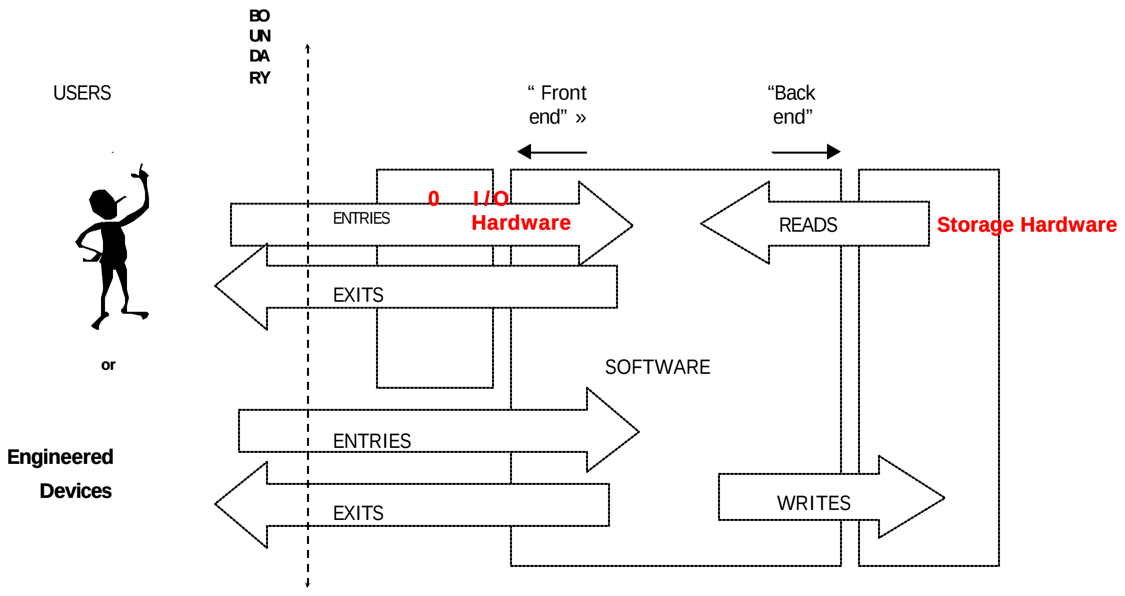

Figure 4: Generic flow of Data Attributes through software from a functional perspective [1]

3.2. COSMIC-FFP and scenarios

As with all functional size measurement methods, the design and rules for these methods are independent of technologies and development approaches. When measuring the functional size of software documented using a specific notation such as UML, it is necessary to establish how the generic measurement concepts are mapped into any notation. The mapping of COSMIC-FFP concepts into UML has been documented in [12]. Six COSMIC-FFP concepts (boundary, user, functional process, data movement, data group and data attribute) have direct UML equivalents (use-case diagram, actor, use case, operation, class and data attribute) – see Table 1.

COSMIC-FFP Concepts	UML Equivalents
Boundary	Use-case diagram
User	Actor
Functional process	Use case
Data movement	Operation (message)
Data group	Class
Data attribute	Class attribute

Table 1 : COSMIC-FFP concepts and their UML equivalents

In measuring software functional size using the COSMIC-FFP method, the software functional processes and their triggering events must be identified [1, 11]. The functional process is an elementary component of a set of user requirements that is triggered by one or more triggering events, either directly or indirectly, via an actor. As also discussed in [2], one functional process corresponds to a scenario (an instance of a use case) which has a set of events. For example, in Figure 5, the functional process includes the set of the following events {*e2, e3, e4, e4.1, e4.2, e6*}. The functional process is initiated by triggering events. A triggering event is an event that occurs outside the boundary of the measured software and initiates one or more functional processes. According to Figure 5, two events are triggered by the user which are outside the boundary of the software and initiate the above-mentioned functional process.

In COSMIC-FFP, the unit of measurement is a data movement, which is a base functional component moving one or more data attributes belonging to a single data group. Data movements can be of four types: Entry, Exit, Read or Write. The sub processes of each functional process are sequences of events, and the functional process comprises at least two data movement types: an Entry plus at least either an Exit or a Write. An Entry moves a data group, which is a set of data attributes, from a user across the boundary into the functional process, where an Exit moves a data group from a functional process across the boundary to the user requiring it. A Write moves a data group lying inside the functional process to persistent storage, and a Read moves a data group from persistent storage to the functional process. In Figure 5, the sub process for the functional process in this scenario is the set of the events.

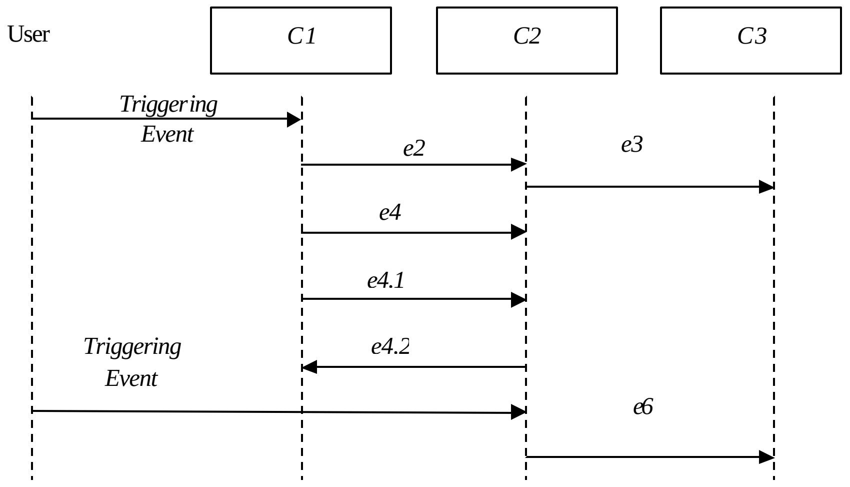

Figure 5: COSMIC-FFP overview

In black-box testing, the details concerning the way in which the data are transformed and manipulated in the scenarios are dropped, since this is a detail design issue (in design phase, the detailed information related to how the data are transformed corresponds to the data structures and algorithms). This data transformation information is equivalent to the data transformation information in COSMIC-FFP. The COSMIC-FFP measurement method recognizes only the data movement type of sub process and includes an approximation assumption whereby each data movement is associated with an average quantity of data transformed (and whereby its true value does not deviate significantly from such an average).

4. Our Testing Approach

One of the greatest benefits of the use-case technique is that it provides testers with a set of assets which can directly drive the testing process. An instance of a use case – a scenario – can be seen as a use-case execution that can be tested. Therefore, use cases are sources of potential test cases.

4.1. Test Case generation

The procedure for generating test cases takes the use-case model as an input. For each scenario identified in the use-case model, we derive a test case through mapping the scenarios to a sequence of events in time(or data movements in COSMIC-FFP), as described in the scenarios. Next, the specific conditions that would cause the test case to execute are identified, and real data values are supplied.

One of the most significant challenges with system testing is the large number of specific scenarios that must be tested to ensure that the system behaves in accordance with its requirements. Our testing approach targets this problem through reducing the number of test cases while keeping the highest test coverage within given budgetary constraints.

Effective management of the test coverage is the biggest challenge in the testing activity. For the application domains to which the COSMIC-FFP method is applicable, such as real-time, embedded and MIS software, the use cases can drive a significant number of test cases, testing all of which may not be feasible. In this paper, we propose to manage the test coverage by partitioning the generated set of test cases into equivalent classes, prioritizing the test cases within the equivalence classes based on the amount of information they process, and selecting the most critical test cases based on a balance between cost and priority.

4.2. Related work

Weiss & Weyuker [14] have partitioned the input domain into some equivalence classes with respect to the behavior of the system under test. This approach mainly reduces the number of test cases with respect to the input domain.

Davis & LeBlanc [9] have discussed partitioning of the input domain applying entropy-based software measures at a higher level by considering chunks of code. For example, a single statement, a block of code or a module itself can be considered as a chunk of code. Groups of chunks may form an equivalence class if they have the same number of in-degrees and out-degrees. In Figure 6, A and C belong to the same equivalence class. The entropy is therefore computed with respect to the chunks that are in the same equivalence class, as follows (FC being the Entropy-based functional complexity measure): $FC = FC1 + FC2 + \ldots + FCp$.

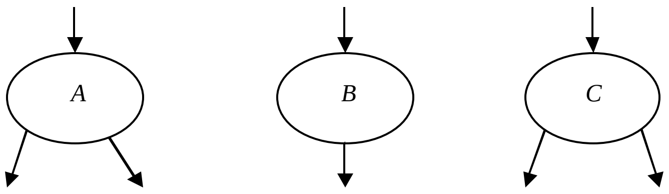

Figure 6: Example of an equivalence class

In the following subsections, we explain in detail our strategy for selecting the optimal set of test cases, given certain resource restrictions, characterized by the highest test coverage.

4.3. Partitioning into Equivalence Classes

One of the problems that accompanies the identification of equivalence classes in COSMIC-FFP is how to organize the scenarios into equivalence classes where each equivalence class is characterized by a distinct functionality. Determination of the equivalences classes themselves can be a tedious and rather time-consuming process. We propose an automatic extraction of equivalence classes from the scenario descriptions. The following research problems have been targeted by our approach:

1. Defining criteria for partitioning the set of scenarios into equivalence classes of test cases based on their similarity;
2. Determining the partitioning of the input domain into equivalence classes by the sets of input events corresponding to the set of scenarios in the equivalence classes E_i;
3. Prioritizing, within each equivalent class, the corresponding test cases based on their functional complexity values (in descending order) – see section 4.3 to calculate the functional complexity measurement.

Partitioning algorithm. We have adopted a strategy similar to that used in [3] with some changes. Our approach is summarized in Figure 7. This approach would allow the use of a metric-based algorithm to partition the set of generated test cases *STC* from the test selection domain *V*. Before applying such an algorithm, the test set *STC* is generated from *V* by using the test case generation algorithm. This algorithm is simply applied so that each scenario constitutes one test case, as described above.

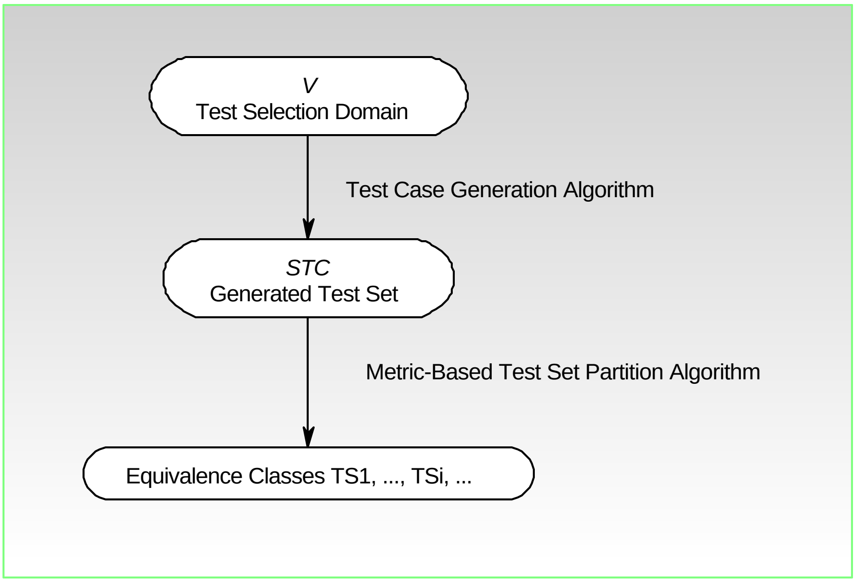
V
Test Selection Domain
Test Case Generation Algorithm
STC
Generated Test Set
Metric-Based Test Set Partition Algorithm
Equivalence Classes TS1, ..., TSi, ...

Figure 7: Test Set Partitioning Strategy

Next, in order to select the subset of test cases to be included in one equivalence class, the following constraint must be satisfied to execute the second algorithm:

The distance ε between any two selected test cases should not be greater than a given constant value $\varepsilon\, max$.

The metric-based test set partitioning algorithm described in Figure 8 will be executed to create the equivalence classes until $STC = \emptyset$. This algorithm divides *STC* into equivalent classes, where each equivalent class TS_i will include test cases with similar functionality and based on having short distances between test cases in one equivalence class. Every time the algorithm is executed, one equivalence class TS_i will be created. We stop executing the algorithm when *STC* contains no more test cases. It should be noted that we do not choose the final set of test cases here, we only partition *STC*.

```
Precondition :{
$STC \neq \emptyset \wedge$
$\varepsilon max > 0 \wedge$
" i · $TS_i = \emptyset$ }
```

```
-   sets $\varepsilon max$ (predefined value based on experimental work)

While $STC \neq \emptyset$
    {
        - selects the longest test case $t$ in set $STC$
            -
            -   sets $\varepsilon$ to 0
        - moves $t$ from the set $STC$ to the set $TSi$
        While ( $STC \neq \emptyset \wedge ( \varepsilon < \varepsilon max)$        {
            -   chooses a test case $t$ in set $STC$ whose distance to the set $TSi$ is minimal in order
                to put similar test cases into one equivalence class sets $\varepsilon$ to the distance between
                $t$ and $TS_i$
            -   if $\varepsilon < \varepsilon max$ then $t$ is moved from $STC$ to set $TS_i$
          } // equivalence class $TS_i$ is created
        i=i+1;
    }
```

Figure 8: Metric-based Test Case Partitioning Algorithm

Now, we have to explain how the distance between the two test cases, *t1* and *t2*, is calculated. The same formula that was used in [3] will also be used here:

$$td\ (t1, t2) = \text{similarity}\ (t1\ , t2) * \text{dissimilarity}\ (t1\ , t2) * \ e \wedge l$$

Remember that *t2* is the test case that will be chosen for the second algorithm. l can be calculated as follows: -(*length* (*t1*)/*length* (*ts*)).

The similarity (*t1* , *t2*) = 2 $^{(-\,length\,(LCP\ (t1,\,t2)}$), where *LCP* is the longest common prefix of the two test cases. The range of the similarity measure is between 0 and 1.

The dissimilarity measure between the two test cases, *t1* and *t2*, is calculated as the number of elementary transformations minimally needed to transform the string (*t1*/*LCP* (*t1*, *t2*)) into the string (*t2*/ (*LCP* (*t1*, *t2*)). For example, the distance between the following two test cases: e1.e2.e3.e4.e1 and e1.e1.e1.e2.e1 = ½ * 3 * e ^ -1. The distance formula *td* (*t1*, *t2*) indicates that the more distance there is between two test cases, the more they will differ.

4.4. Priority of Test Cases

For large systems, there may be a very large number of test cases, and a priority has to be assigned to them to help increase the testing coverage within the given budgetary constraints. Prioritization of test cases is an important issue in software engineering because of the limited testing budget, and is usually performed manually.

In our approach, the Functional Complexity (FC) measure is used to prioritize test cases [2]. Intuitively, more, and diverse, functionality of the system would lead to a bigger portion of the system being involved in that usage. The entropy calculated on a sequence of events abstracting a scenario quantifies the average information interchange for a given usage of the system. Therefore, it should correlate with the error spans during testing. The formula used to calculate functional complexity is as follows [2]:

$$FC = -\sum_{i=1}^{n} (f_i / NE)\log_2 (f_i / NE)$$

The probability of the ith most frequently occurring event is equal to the percentage of total event occurrences it contributes, and is calculated as $p_i = f_i$ / NE, where f_i is the number of occurrences of the ith event and NE is the total number of events in the sequence.

The priority assignment should be done automatically.

4.5. Test Selection Algorithm

It is to be noted that total testing process resource consumption is directly proportional to the number of the test cases selected. As a result, the total cost of the test set is calculated by multiplying the number of selected test cases with *C*, where *C* is a positive constant scalar denoting the cost of one test case. In order to select the optimal subset of test cases that will be characterized by the highest test coverage, we balance the budget and the priority of the test cases as follows:

```
For all non-empty equivalent classes $TS_i$

    Step 1. Choose the highest-priority test case from the equivalence class $TS_i$;
    Step 2. Add the chosen test case to the Optimal Set and remove it from the equivalence class $TS_i$
    Step 3. Increase the total testing cost in C

    If (The total testing cost exceeds a given budget Cmax)
        End the algorithm

End For
```

5. Conclusion and Directions for Future Work

The paper has explored an interesting linkage between the 2nd functional size measurement method, namely "COSMIC-FFP", and testing. It has described the development of a model for scenario-based black-box testing, in which the scenarios are derived using the COSMIC-FFP method for identifying functionality. Based on that model, the mechanisms of both test-case generation and their prioritization are elaborated to ensure high level fault coverage in black-box testing. First, the set of scenarios in the COSMIC-FFP context represents the set of test cases required to form the generated test set. Second, that set of test cases is partitioned into equivalence classes. Third, the test selection algorithm is run on the set of non-empty equivalence classes. The result is a set of selected test cases affording the best possible coverage (i.e. the test case with the highest FC value, from each equivalence class). However, we cannot claim the full fault coverage since this algorithm is maximizing the test coverage within the limits of a given budget.

The partitioning of the input domain into equivalence classes would also open up a new direction in the future towards enabling the reliability estimation of software. With each equivalence class comes its operational profile, that is, the probability that the inputs will really derive from normal operation of the system. By using the input reliability model, which stems from a functional view of software as operating on a certain input domain (sequences of events in our case) and producing results within a given output domain where we realize the software failures, we can select test cases over the input domain, perform software testing and then compute the reliability estimation.

In a future paper, it would be also be of interest to apply our testing strategy as described in this paper to real case studies in order to validate and verify the results of the strategy. Rational Unified Process (RUP) [15] case studies would serve as a good choice to begin with, since they provide a clear description of some software specifications and the detailed use cases and scenarios that correspond to them.

6. References

[1] ABRAN, A., DESHARNAIS, J.-M., OLIGNY, S., ST-PIERRE, D. AND SYMONS, C., *COSMIC FFP - Manuel de Mesures*, Université du Québec à Montréal, Montréal. 2003, URL: http://www.cosmicon.com

[2] ABRAN, A., ORMANDJIEVA, O. AND TALIB, M.A. *Functional Size and Information Theory-Based Functional Complexity Measures: Exploratory study of related concepts using COSMIC-FFP measurement method as a case study*, in Proceedings of the 14th International Workshop of Software Measurement (IWSM-MetriKon 2004), Springer-Verlag, Konigs Wusterhausen, Germany, 2004, pp. 457-471.

[3] ALAGAR, V.S., CHEN, M., ORMANDJIEVA, O. AND ZHENG, M., *Automated Generation of Test Suits from Formal Specifications of Real Time Reactive Systems*. Submitted to IEEE Transactions on Software Engineering Journal, 2004.

[4] BAI, X., PENG, L.C. AND LI, H., *An Approach to Generate Thin Threads from UML Diagrams*. Technical Report, TR-03-12, Software Engineering Research Group, School of Computer and Information Science, Edit Cowan University, 2002.

[5] BAI, X., TSAI, W.T., FENG, K. AND L.YU, *Scenario-based Modeling and Its Applications to Object Oriented Analysis Design and Testing*, Proceedings of IEEE Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002), San Diego (USA), 7-9 January 2002, pp. 253-270

[6] BEIZER, B. *Software Testing Techniques*. Van Nostrand Reinhold, 2/e, 1990, ISBN 1850328803

[7] BERTOLINO, A. *Knowledge Area Description of Software Testing Guide to the SWEBOK*. URL: http://www.swebok.org , 2004.

[8] CHOW, T.S. *Testing Software Design Modeled by Finite State Machines*, IEEE Transactions on Software Engineering, Vol. SE-4, No.3, 1978, pp. 178-187

[9] DAVIS, J.S. & LEBLANC, R.J., *A Study of the Applicability of Complexity Measures*, IEEE Transactions on Software Engineering, Vol. 14 No. 9, September 1988, pp. 1366-1372

[10] EN-NOUAARY, A., DSSOULI, R. & KHENDEK, F., *Timed Wp-Method: Testing Real Time Systems*, . IEEE Transactions on Software Engineering, Vol. 28 No. 11, November 2002, pp. 1023 -1038

[11] ISO/IEC19761. *Software Engineering - COSMIC-FFP - A functional size measurement method*, International Organization for Standardization - ISO, Geneva. 2003.

[12] JENNER, M., *Automation of Counting of Functional Size Using COSMIC-FFP in UML*, in Proceedings of the *12th International Workshop on Software Measurement (IWSM 2002)*, Magdeburg (Germany), October 2002, pp. 43-51.

[13] OMG Website: http://www.omg.org/technology/documents/formal/uml.htm

[14] WEISS, S.N. AND WEYUKER, E.J., *An Extended Domain-Bases Model of Software Reliability*, IEEE Transaction on Software Engineering, Vol.14 No.10, October 1988, pp.1512-1524

[15] KRUCHTEN, PHILIPPE, *The Rational Unified Process: An Introduction*, 2nd Edition, Addison-Wesley Pub. Co., 2000, ISBN: 0201707101