Computer Standards & Interfaces 33 (2011) 152–158

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi

Software automated testing: A solution to maximize the test plan coverage and to increase software reliability and quality in use

Marcantonio Catelani, Lorenzo Ciani *, Valeria L. Scarano, Alessandro Bacioccola

University of Florence, Department of Electronics and Telecommunications, via S. Marta 3, 50139, Florence, Italy

ARTICLE INFO

Available online 30 June 2010

Keywords:
Software automated testing
Software reliability
Quality in use
Mean time to overflow

ABSTRACT

Software plays an increasingly important role in complex systems, especially for high-tech applications involved in important fields, such as transportation, financial management, communication, biomedical applications and so on. For these systems, performances such as efficient operation, fault tolerance, safety and security have to be guaranteed by the software structure, whose quality in use is assuming a growing importance from the industrial point of view. The basic problem is that the complexity of the task which software has to perform has often grown more quickly than hardware. In addition, unlike hardware, software cannot break or wear out, but can fail during its life cycle (dynamic defects) [1]. Software problems, essentially, have to be solved with quality assurance tools such as configuration management, testing procedures, quality data reporting systems and so on [2]. In this context, the paper proposes a new approach concerning the automated software testing as an aid to maximize the test plan coverage within the time available and also to increase software reliability and quality in use [3]. In this paper a method which combines accelerated automated tests for the study of software regression and memory overflow will be shown, in order to guarantee software with both a high quality level and a decrease of the testing time. The software will be tested by using test sequences reproducing the actual operating conditions and accelerated stress level. Moreover the research wishes to define some parameters of the software life and to show the generality of the proposed technique.

1. Introduction

A software system is often subjected to conflicting requirements; in fact it has to be reliable in its application and, at the same time, has to follow the needs of the market with competitive costs [1,2]. In this context the test process, through quantitative planning, tracking and automation, covers a fundamental role. Software reliability testing combines the use of quantitative reliability aims with operational profiles (profiles of system use), that guide developers in the testing implementation. An important innovation would be the introduction of the acceleration of the automated test in order to reduce both time and cost of development without inducing, in the system, different failures from the ones which we want to analyse.

Inadequate and ineffective testing is responsible for many problems regarding software reliability faced by computer users. On the other hand, the complexity of modern software packages makes exhaustive testing difficult. Nevertheless, automated testing can help to improve efficiency of the testing process in order to identify areas of a program that are prone to failure. Automated testing can be applied in large portions of many applications, with reduction of the workload on overburdened testers. Until a few years ago, developers considered software testing as a secondary activity, if compared with the development phase; nowadays the test represents, in many fields of application, the starting point for the development of the product and its cost is often comparable with the cost of the product development. In fact, it has been estimated that software testing, able to detect errors in source code, involves more than 50% of software development [4]. Time and cost can be significantly reduced through the use of automated test generators [4].

Among the activities that allow the detection of nonconformity and potential failures in different phases of the software product, implementation software Verification and Validation (V&V) plays a fundamental role. To this aim, some standards and guidelines have been issued about software as a key component which contributes to system behaviour and performance; some examples are represented by the International Standard ISO/IEC 9126 [5], which defines a quality model for a software product, in order to satisfy the customer optimizing the product and IEEE Standard 1012 [6] for Software Verification and Validation, which attempts to establish a common framework for all activities and tasks in support of software life cycle steps. In particular, amongst the different steps, the efforts on improving quality are concerned with V&V phases; these activities determine whether products of a given activity conform to the requirements and whether the software satisfies its intended use and

* Corresponding author.
E-mail addresses: marcantonio.catelani@unifi.it (M. Catelani), lorenzo.ciani@unifi.it (L. Ciani), valeria.scarano@unifi.it (V.L. Scarano).

0920-5489/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.csi.2010.06.006

customer needs. Verification and Validation is the activity in software production that allows production costs to be decreased and, at the same time, to increase software reliability [3,7]. Verification is the process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Validation is the process of checking in order to ensure compliance with software requirements [6]. V&V activities carry out the integration test that exposes defects in the interfaces and interaction between integrated components (modules); progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software will work as a system.

It is important to remember that memory leaks and memory corruption are considered critical software bugs that meaningfully influence availability and security performances of the system. Memory leaks and memory overflow in the 'C' language are referred to as an unwanted increase in the program memory occupation. The memory consumption of the program increases to a great extent by an unintentional consumption of it. It also means that the memory of the program is being corrupted and this results in an error. Some of these errors are not critical but can cause major problems to the program. If a memory leak is present, the system may stop functioning and may violate some of the files of the operating system. A memory overflow is also known as stack overflow or buffer overflow [8–10]. When this kind of error occurs, the program can terminate itself; particularly during saving beyond the limit. Consequently the program can produce wrong results and this leads to a fault in the execution of the program and the operating system. In this case, memory leaks, caused through the inaccessibility of some allocated memory, can cumulatively degrade overall system performances increasing memory paging or, in the worst case, generating the program crash. The system memory, allocated by the processes, is managed by the developer with set rules; these rules are often optimized for many applications but not for all [2]. In order to evaluate the memory leaks during test regression, the software automatic test proposed in this paper can be classified as a dynamic test [4], being able to stimulate the software under test with a stress level accelerated in time. To this aim, we considered a list of processes that have to be observed and, for each of them, two parameters to be analysed are selected: *Private bytes* and *Working set,* as defined in Section 2. In this paper an approach that takes into account, simultaneously, the regression tests execution and the memory overflow evaluation are proposed; the aim is to demonstrate a decrease of testing costs and an increase of software reliability in terms of Mean Time to Overflow (MTOF) [11–14]. MTOF parameter, that allows to estimate the failure of the system due to overflow, is defined in Section 2. Its evaluation for specific tasks and applications is shown in Section 3. The automatic monitoring of this parameter during regression test can be considered as a software product aid in order to plan the right maintenance operations and cost reduction.

With the aim of proving the general validity of the proposed approach, an industrial application, concerning an automatic distribution petrol station, is considered. For this complex plant the role covered by the software is fundamental; in fact it allows the global management of the petrol station, from the fuel sale to its supplying, from the warehouse management to the accounts department.

In the following, after the description of the main steps of the automated test method, the proposed improvements are shown, supported by detailed consideration for experimental data.

2. Proposed test methodology

Several well-known methodologies for testing general purpose software exist in literature [15,16]. Such techniques follow a structural or a functional approach, also identified as the white-box and the black-box approach respectively. The proposed methodology can be classified as a black-box approach, where the software under test has to be verified with a suitable studied set of inputs whose expected outputs are known only on the basis of the functional specifications. In addition to the black-box approach, we propose a test method with test sets well representative of the field behaviour of the system, according to the block diagram shown in Fig. 1. One can observe that "test parameters" and "field data statistical distribution" are inputs; "test results", "expected results", "MTOF" and "final results" are outputs. The "software under test" is the object of the analysis and "test cases generation", "test cases execution", "input data elaboration", "memory occupied monitoring" and "comparison" are activities.

The software testing approach is based on a dynamic pseudo-random generation which mixed the preliminary test parameters, which ones came from test results, and the statistical distribution of field data. The combination of possible variables and therefore the states evolution is random and represents the test case. The proposed methodology takes advantage of pseudo randomization voted to increase the number of test sequences and, at the same time, to simulate better the possible real conditions; it allows the test sequences to be modified without changing the programming code, for its high flexibility performances [17–19]. Test case generator creates quite unlimited sequences that can be iterated on the software under test. Both the inputs and the outputs can be *txt* file with dimension of about 100 kB. The test results, represented by the output of the software under test, are compared with the expected results obtained from the test cases, at the end of the testing phase; final results allow information to be deduced about the quality in use of the software under test and new data for dynamic test case generation

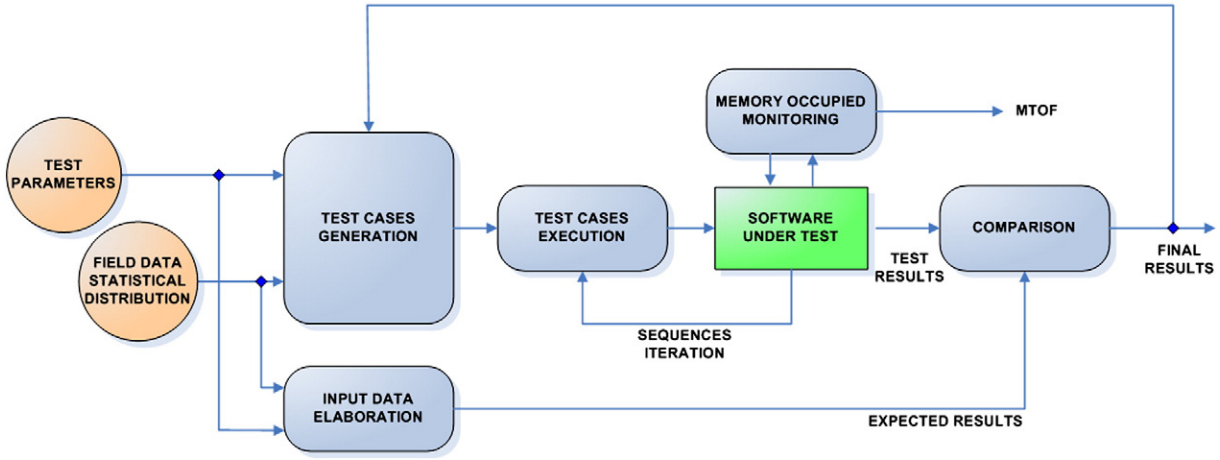

Fig. 1. Block diagram of the test method.

[20–23]. Detailed steps concerning the comparison between obtained and expected results are shown in Fig. 2. If no differences between obtained and expected values are present, the test is stopped; vice versa, a fault has to be tracked and the software has to come back to the development team in order to be analysed and correct the bug; successively the test has to be carried out again. There is also the possibility of locating unclear values, such as ambiguities that have to be solved manually by repeating ambiguous sequences.

The key-step introduced in the proposed method (Fig. 1), that allows important information to be obtained, is the simultaneous evaluation of the occupied memory of the system during the execution of the automated tests; in fact, we can estimate the filling up of the total memory both for a single module and the entire device on which the software under test runs. In Fig. 1 this step is denoted as memory occupied monitor. This step, that considers the evaluation of possible memory overflow on the tested products, is implemented by a flexible macro, that doesn't induce a decrease of the machine performances on which it is active.

Through the memory occupied monitor step lists of processes that have to be observed can be defined; for each of them the parameters to analyse, and observe also at a later date, are chosen. The potential memory overflow can be diagnosed through the following parameters:

- *Private bytes:* this parameter displays the number of bytes reserved exclusively for a specific process. If a memory leak occurs, this value will tend to rise steadily.
- *Working set:* represents the current size of the memory area used by the process for tails, threads, and date. The dimension of the working set grows and decreases as the VMM (Virtual Memory Manager) can permit. Since it shows the memory that the process has allocated and deallocated during its life, when the working set is too large and doesn't decrease correctly it usually indicate a memory leak.

The parameters above mentioned can be estimated, both on a single operating task and on the total occupation memory of the computer, through the software trial with a stress level comparable to the real use or by repeating accelerated series of normal use operations in order to reduce the testing time.

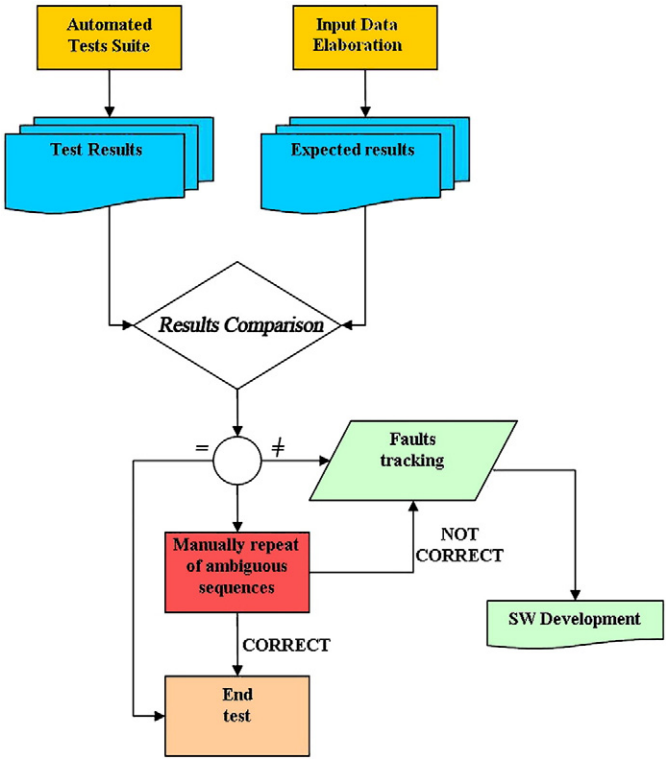

Fig. 2. Automated software testing process realized.

The trend of memory filling is monitored: so if the trends of chosen parameters increase we can deduce that a memory leak affects the software under test, in this case the program has to come back to the development team for the fault correction.

The stress produced by a test simulates the real operating conditions found in the field. At the same time, during a memory occupation test, a regression test can be performed, be+ing it based on continuous repetition of the same operations or test sequences [24]. Since it identifies, in a relatively short time, failures in the field before the product is realised and then to correct them through software redesign, the implementation of this kind of automated tests allows to reduce costs and working time.

It is important to underline that the saturation of available memory due to memory leaks is a common cause of software failure [25,26]. Memory leak is the phenomenon of permanent memory occupation that appears when a module allocates the memory without ever deallocating it. This behaviour, repeated through time, wears out the available memory size and induces the block of the system (fail stop).

A software module is characterized by memory leak if at least one of the operations (or sequence of operations), that it executes, suffers from memory leaks. After the test, the portion of memory occupied by the program, suffering from this failure cause, can be minimal (some kB); the phenomenon is generally not appreciable on the single test but can become considerable by repeating the same test sequences several times. This type of problem can be studied through the automated tests method proposed in this work; in fact, repeating a test sequence and monitoring the trend of the occupied memory during the test, an estimation of the presence of memory leaks is possible. It is useful, therefore, to introduce the parameter Mean Time To Overflow (MTOF) defined as:

$$MTOF = \frac{Mem_{size}}{\Delta_m} \tag{1}$$

where Mem_{size} denotes the available memory size and Δ_m represents the mean increase of memory occupation in an established time interval. According to the value of Δ_m, evaluated in kB per day, hour or sequence, we can explain the MTOF in number of days, hours or sequence to the breakdown of the memory resources and, therefore, to the collapse of the system (system fail stop).

For a complex system where independent memories are involved for the operating functions, Eq. (1) can be modified and the System Mean Time To Overflow is defined as:

$$MTOF_S = \min\left(\frac{Mem_{sizei}}{\Delta_{mi}}\right), \quad i = 1\ldots n. \tag{2}$$

3. Validation of the methodology

The validity of the proposed approach is verified by considering an industrial application, constituted by a multifunction distribution petrol station [27] as represented in Fig. 3.

For this application the software suite covers a fundamental role in all the activities involved in the management of the fuel station. In our case and in its complete version, approximately 40,000 modules and 5.000.000 code lines identify the complexity of the software suite able to manage important activities such as the fuel sale and storage by means of Dispenser Management System (DMS) and all operations that require the use of Card Payment System (CPS).

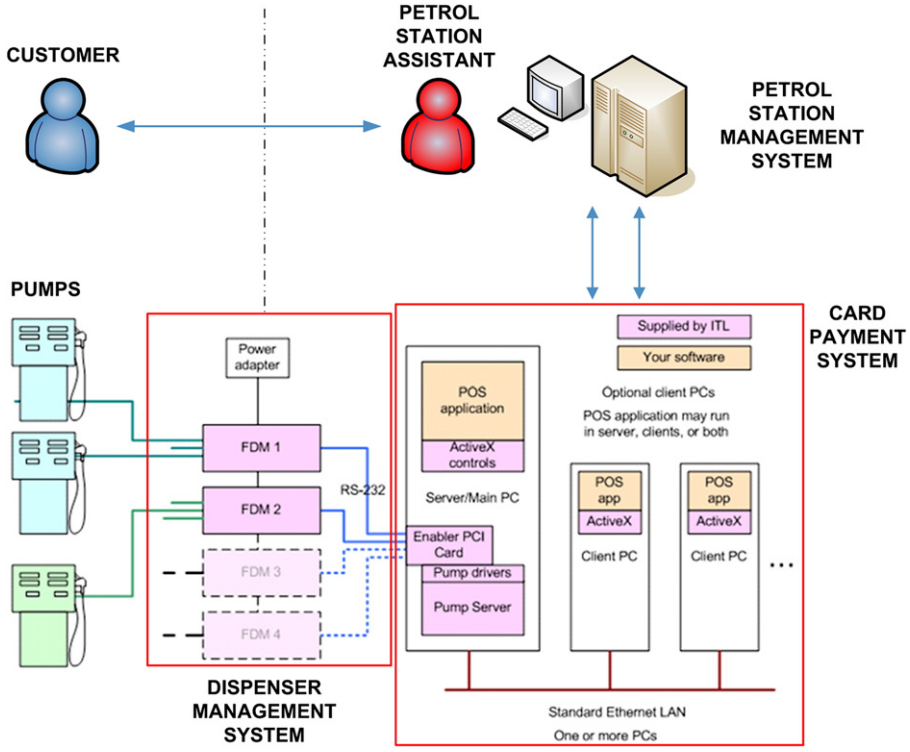

Fig. 3. Typical components of a standard automatic petrol station.

The test suite that reproduces the sequence that will recursively repeat on the service station has been planned and implemented. Its dimension is about 8 MB and it is composed of 590 code lines.

Such test suite, shown in Fig. 4, involves real operations with connection to the Card Payment System that can be summarized as:

a) System login, read card parameters, update or create log;
b) Check purchase restriction and card validity;
c) Random available pump selection;
d) Product selection and delivery (based upon the real products statistical distribution);
e) Random avoid operation or system logout, close log.

The test suite allows the reproduction of 13,100 test cases generated according to the proposed test methodology described in Section 2, without any modification of the system software under test.

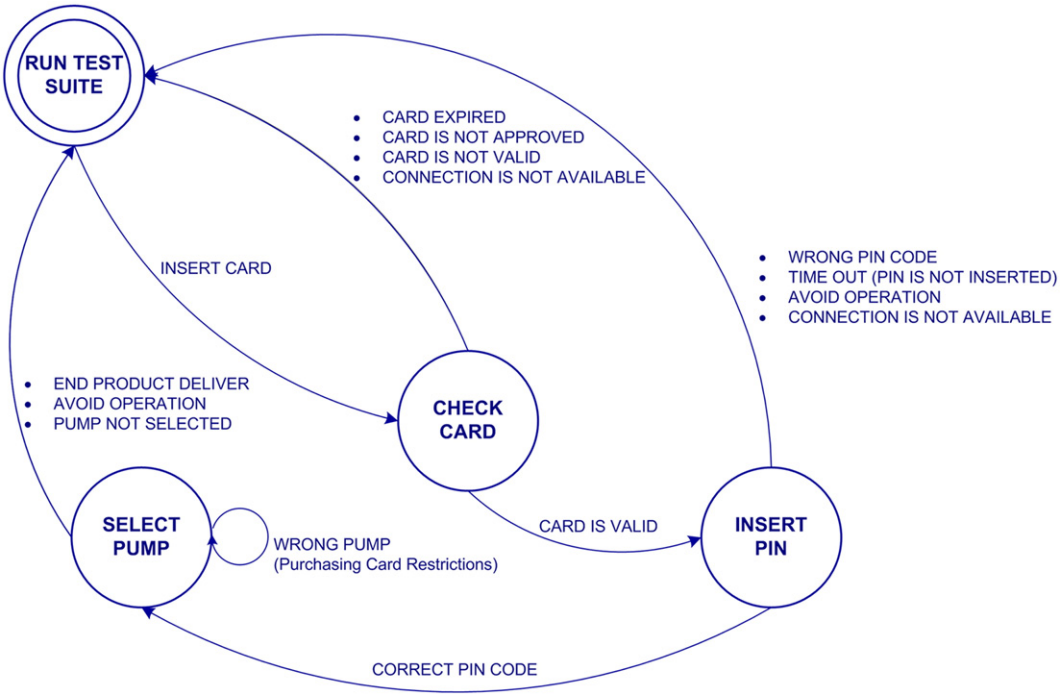

Fig. 4. Test suite flow chart.

The test suite is then repeated carrying out 2233 sales in 40 h, with an average of approximately 56 sales/h; four times greater than the conditions in the field (about 14 sales/h), that are carried out for every POS (Point of Sale). The test suite stimulates a lot of activities: the interaction between software and controller, the simulator of the pumps that recreates the interaction between the system, the customer and the connection to the service centre of the oil company for the management of the fidelity, credit and debit card. Only external hardware devices, pumps and DMS hardware subsystem, are simulated.

During the tests the filling of the total memory of the computer and five operations that field feedbacks of the application indicated as critic, in terms of memory requirements, are monitored.

The trend of the occupied memory grows linearly with time and sales number; we assume that the sales are uniformly distributed during the time interval considered for the test. If the performances of computers are known, the memory variation (Δ_{m_1}) can be evaluated in bytes/h as:

$$\Delta_{m_1} = \frac{(Mem_{max} - Mem_{min})}{t} = \frac{5.877.760}{40} = 146.944\ bytes/h \quad (3)$$

where Mem_{max} and Mem_{min} are respectively the maximum (15.609.856 bytes) and the minimum (9.732.096 bytes) allocated memory by the application under test; t denotes the sequence duration in hours ($t = 40$ h).

Considering the memory size equal to 256 MB (268.435.456 byte, denoted as Mem_{size}) and assuming that such memory is only dedicated to the exclusive use of this process, it is possible to calculate the Mean Time to Overflow, according to Eq. (1):

$$MTOF = \frac{Mem_{size}}{\Delta_{m_1}} = 1826,787h \cong 76 days \quad (4)$$

Considering that the distributor server is never off and that the management software stays on field for a mean time of 3 months before any maintenance activity, it appears that this process would lead to a crash of the server after 76 days of operation.

For a more accurate MTOF assessment we can observe that, when the system's memory value is achieved, the operating system executes a swap on the hard disk.

So we can consider the trend of the total occupied machine memory under investigation and the maximum dimension of the swap (Mem_{swap}) partition as 1 GB (1.073.741.824 byte). By means of the Performance Monitor we can observe that at login, before executing any stress, the memory already occupied by all the processes (named as Mem_{start}) is equal to 327.122.944 bytes; therefore, in the starting phase, the system is carrying out the swap on disk. The total memory, Mem_{tot}, that the process can allocate, is:

$$Mem_{tot} = Mem_{swap} - Mem_{start} + Mem_{size} = 970.054.336\ bytes \quad (5)$$

consequently

$$MTOF = \frac{Mem_{tot}}{\Delta_{m_1}} = 6601.524h \cong 275 days \quad (6)$$

Remembering the hypothesis that the process under investigation is the only one that occupies the memory, the swap on the disk allows the software to carry out nearly three life cycles (90 days is the typical useful life). However, from Eq. (4), the necessity to expand the device memory in order to assure a greater reliability appears.

For every observed task the private bytes and the working set are also monitored as shown in the trend plots of Fig. 5, where the red and the black lines represent the private bytes and the working set, respectively. By observing these trends, two transitory spikes can be noted. We assume that these phenomena are due to an incorrect process execution event and, in this sense, not considered as significant.

The software under test has also a database controller where all information inserted by the customer or employed user interface is collected. The database is saved on two different disks (main and secondary); and the Database Management System carries out its reconstruction. During the test, the trend of the occupied memory from the database controller is observed by means of measurement of the private bytes and the working set as shown in the plots in Fig. 6 in blue line and yellow line, respectively.

The trend of the memory appears constant except for the presence of some instantaneous peaks during the execution of the database reconstruction. Every time that the reconstruction is executed, the process occupies about 35 MB/day (Δ_{m_2}) of memory, as shown

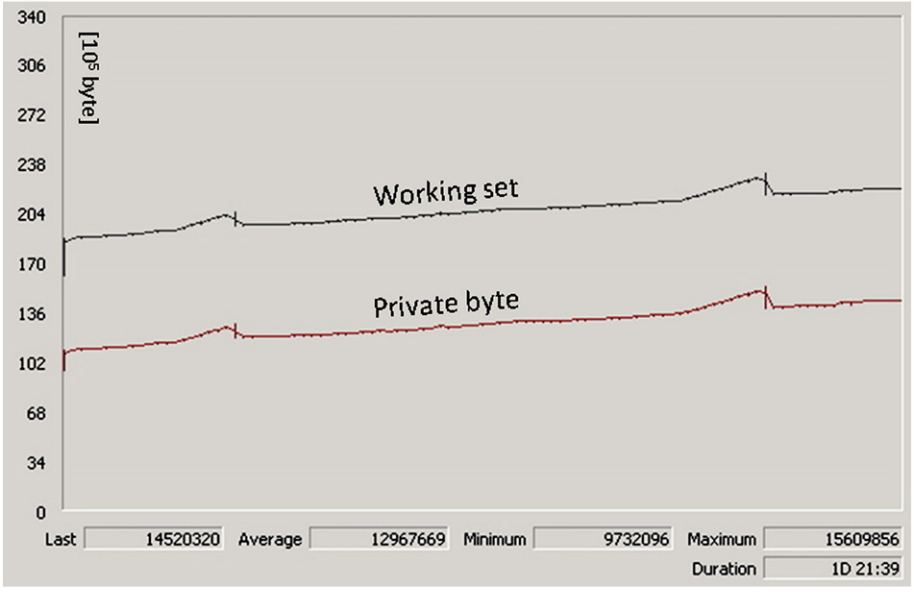

Fig. 5. Trend of memory occupation from the customer interface manager.

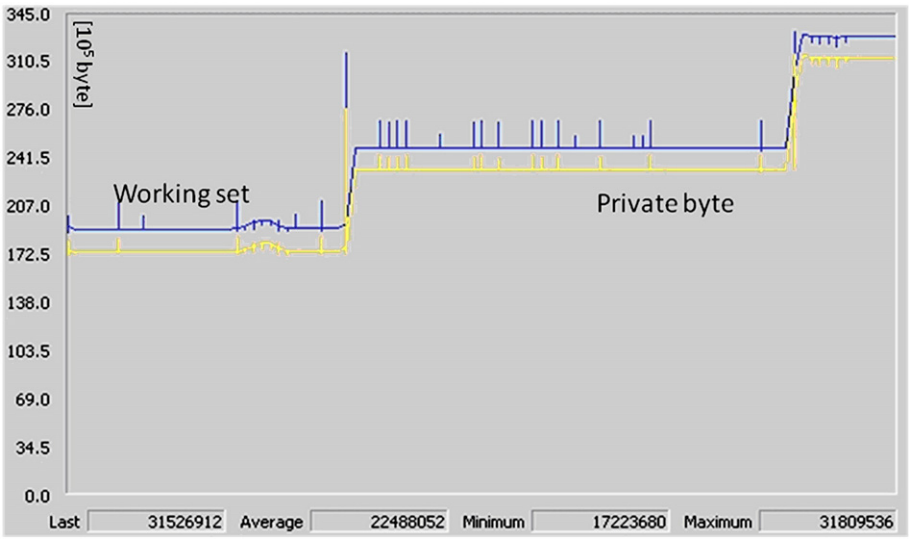

Fig. 6. Trend of memory occupation from the database controller.

in Fig. 6. The total memory value, that the process can allocate, is evaluated as:

$$MTOF = \frac{Mem_{tot}}{\Delta_{m_2}} = \frac{970.054.336}{36.700.160} \cong 26\,days \tag{7}$$

The server overflow is reached after 26 days and the server operates the reconstruction of database every day; consequently more than 3 crashes due to overflow in one life cycle may occur. As in the previous application, it's possible to assume that the memory is occupied by only one process under investigation. If the combined effect of two tasks were considered, the new occupied memory, Δ_{TOT}, could be expressed as:

$$\Delta_{TOT} = 24\,\Delta_{m_1} + \Delta_{m_2} = 40.226.816\,\frac{byte}{day} \tag{8}$$

In Eq. (8) Δ_{m_1} is multiplied by 24 h in order to obtain the same measurement unit of the parameterΔ_{m_2}. From the result obtained in Eq. (8), the new value of MTOF can be evaluated as:

$$MTOF = \frac{Mem_{tot}}{\Delta_{TOT}} \cong 24\,days \tag{9}$$

So, the software Mean Time To Overflow is now decreased to 24 days.

MTOF being an index able to evaluate the software crash, it can be considered a parameter to estimate the software reliability and, as consequence, to plan the correct system update and the quality in use according to ISO/IEC standard 9126.

As an additional advantage, the proposed approach allows the change of the stress level so testing the software to a high stress level in the smallest test time. This can lead to faults not detected with low stress levels increasing the test plan coverage. For example, in 15 working days we can realise 12.600 test cycles (35 test sequences/h) if we implement accelerated automated tests, instead of 5.040 cycles (14 tests sequences/h as suggested from field data) with no accelerated tests and 600 cycles (~5 test sequences/h) with manual tests. Such considerations are summarized in Table 1 and plotted in Fig. 7, where automated tests are compared to the traditional one (manual testing).

4. Conclusions

The proposed approach of dynamic software automated testing showed the importance of accelerated automated tests for the software debug and validation in a short time interval, before product distribution, with the aim of increasing software test plan coverage, quality in use and reliability. Moreover this research has defined some parameters of software life and has shown the generality of the proposed technique.

The application presented in this paper is able to stimulate the software under test with an established stress level, comparable in the sequence operations to the real use but accelerated four times compared to the manual tests. Information concerning the memory leaks and fault regression of the new software versions with respect to the old one can be deduced, as the experimental results proved. In particular, the automatic tests have been able to detect, locate and then correct some important software bugs that can be considered critical for particular industrial application such as, e.g., the management of an automatic petrol station. In this context some examples can be represented by errors in the visualization of the fidelity card amount, errors in product delivery, paying by credit or debit card, in the report printed after cash closing and errors in the mapping of the inter-bank centre services answers. Moreover we observed that the first task of the application under test had a linear increase of the occupied memory due to the customer interface manager. The second task, instead, had an increase of the memory occupation only when the reconstruction of data base was performed. In order to increase the software reliability, a hardware upgrade of the RAM memory becomes necessary. The Mean Time to Overflow parameter, calculated for the tasks and application, represents a fundamental parameter that allows the system crash to be estimated due to an overflow. At the same time, it will allow the estimation of the software availability in order to plan an effective maintenance operation plan, voted to induce not only an increase of software quality in use and customer satisfaction but also a decrease of maintenance costs.

Table 1
Test sequences implemented vs. test methods.

Days	Man-hours	Machine-hours	Manual tests	Automated tests	Accelerated automated tests
1	*8*	*24*	*40*	*336*	*840*
7	*56*	*168*	*280*	*2.352*	*5.880*
15	*120*	*360*	***600***	***5.040***	***12.600***

The use of terms in italic is only for a graphic choice.
The use of terms in bold is to put in evidence the data mentioned in text above the Table 1.

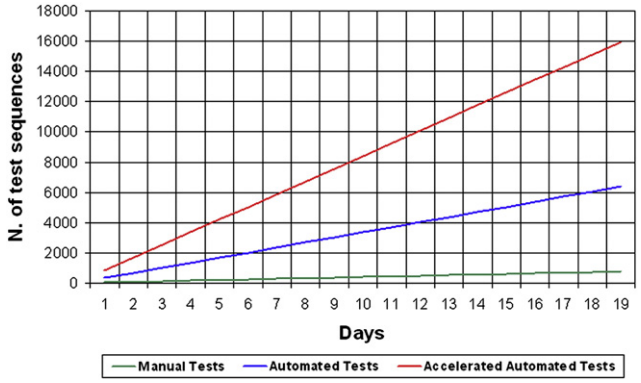

Fig. 7. Trend of number of tests vs. days.

Furthermore, the benefits of such approach based on accelerated automatic testing compared with the traditional one (manual testing) can be reached both at a lower cost and a decrease of the testing time of the software verification and validation. In addition, the possibility of replicating old test sequences on new future versions (with no testing cost) can also be considered an important benefit from the industrial point of view.

Other parameters, such as handle counts for each process and virtual bytes, could be useful in order to control memory leaks. If a memory leak is occurring, an application could create additional handles to identify memory resources, so a rise in handle count might indicate a memory leak. Virtual bytes, that are the current size of the virtual address space used by a process, don't necessarily imply corresponding use of either disk or main memory pages, but virtual space is however finite and, using too much, the process may limit its ability to load libraries.

References

[1] Reliability Analysis Center, Introduction to Software Reliability: A State of the Art Review, Reliability Analysis Center (RAC), 1996.
[2] J.D. Musa, Introduction to software reliability engineering and testing, Proceedings of the 8th International Symposium on Software Reliability Engineering, 1997.
[3] A. Birolini, Reliability Engineering — Theory and Practice, Springer-Verlag3-540-40287-X, 2004.
[4] E. Diaz, J. Tuya, R. Blanco, Automated software testing using a metaheuristic technique based on tabu search, Proceedings of 18th IEEE International Conference on Automated Software Engineering, 2003, pp. 310–313.
[5] ISO/IEC 9126: Information technology — software product evaluation — quality characteristics and guidelines for their use, 2001.
[6] ANSI / IEEE Std. 1012, IEEE Standard for Software Verification and Validation Plans, , 1986.
[7] ANSI / IEEE St. 829, Standard for software test documentation, , 1998.
[8] M. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors), Proceedings of the 20th Annual Computer Security Applications Conference, Tucson, Arizona, USA, December 6-10 2004.
[9] H. Güneş Kayacık, A. Nur Zincir-Heywood, M. Heywood, Evolving successful stack overflow attacks for vulnerability testing, Proceedings of the 21st Annual Computer Security Applications Conference, Tucson, Arizona, USA, December 5–9, 2005.
[10] Y. Wiseman, J. Isaacson, E. Lubovsky, Eliminating the threat of kernel stack overflows, IEEE IRI 2008, July 13–15, 2008, Las Vegas, Nevada, USA, 2008.
[11] V.V. Mazalov, M. Tamaki, S.V. Vinnichenko, Optimal computer memory allocation for the poisson flows, Automation and Remote Control (ISSN: 0005-1179) 69 (9) (2008) 1510–1511.
[12] R.B. Cooper, M.K. Solomon, The average time until bucket overflow, ACM Transactions on Database Systems 9 (3) (September 1984) 392–408.
[13] G. Copeland, T. Keller, R. Krishnamurthy, M. Smith, The case for safe RAM, Proceedings of the Fifteenth International Conference on Very Large Data Bases, Amsterdam, 1989.
[14] S.P. Meynt, M.R. Fraterni, Recurrence times of buffer overflows in jackson networks, Proceedings of the 29th Conference on Decision and Control Honolulu, Hawaii, December 1990.
[15] H. Freeman, Software testing, IEEE Instrumentation & Measurement Magazine 5 (3) (September 2002) 48–50.
[16] G. Betta, D. Capriglione, A. Pietrosanto, P. Sommella, A statistical approach for improving the performance of a testing methodology for measurement software, IEEE Transactions on Instrumentation and Measurement 57 (6) (June 2008) 1118–1126.
[17] M.A. Bailey, T.E. Moyers, S. Ntafos, An application of random software testing, IEEE MILCOM, Conf. Rec., vol. 3, November 1995, pp. 1098–1202.
[18] S.C. Ntafos, On comparisons of random, partition, and proportional partition testing, IEEE Transactions on Software Engineering 27 (10) (October 2001) 949–960.
[19] W. Lingfeng, K.C. Tan, Software testing for safety critical applications, IEEE Instrumentation & Measurement Magazine 8 (2) (March 2005) 38–47.
[20] I. Burnstein, Practical software testing: a process-oriented approach, Springer-Verlag, New York, 2003, ISBN:0-387-95131-8.
[21] D. Galin, Software Quality Assurance: From Theory to Implementation, Pearson Addison Wesley, Harlow, England, 2004.
[22] S.M. Phadke, Quality Engineering Using Robust Design, Prentice-Hall, Englewood Cliffs, NJ0137451679, 1989.
[23] S. Stoica, Robust test methods applied to functional design verification, Proceedings of IEEE International Test Conference, September 1999, pp. 848–857.
[24] M. Catelani, L. Ciani, V.L. Scarano, A. Bacioccola, A novel approach to automated testing to increase software reliability, Proceedings of IEEE International Instrumentation and Measurement Technology Conference, Vancouver, Canada, May 12-15 2008.
[25] S. Roohi Shabrin, B. Devi Prasad, D. Prabu, R.S. Pallavi, P. Revathi, Memory leak detection in distributed system, Proceedings of World Academy of Science, Engineering and Technology, 1307-6884vol. 16, November 2006.
[26] M. Grottke, K.S. Trivedi, Fighting bugs: remove, retry, replicate, and rejuvenate, Computer 40 (2) (February 2007) 107–109.
[27] M. Catelani, L. Ciani, V.L. Scarano, A. Bacioccola, An automatic test for software reliability: the evaluation of the overflow due to memory leaks as failure cause, Proceedings of 16th IMEKO Symposium TC4, Florence, Italy, September 2008.

Marcantonio Catelani received the degree in electronic engineering from the University of Florence, Italy. Since 1984, he has been with the Department of Electronics and Telecommunications, University of Florence, where he is currently a Full Professor on reliability and quality control. His current research interests include system reliability and availability, reliability evaluation test and analysis for electronic systems and devices, fault detection and diagnosis, quality control, instrumentation, and measurement, where his publications are focused.

Lorenzo Ciani was born in Florence, Italy, on July 11, 1977. He received, in 2005, the M.S. degree in Electronic Engineering and, in 2009, the Ph.D. degree in Industrial and Reliability Engineering from the University of Florence, Italy. His current research concentrates in the fields of system reliability, availability, maintainability and safety, reliability evaluation test and analysis for electronic systems and devices, fault detection and diagnosis, electrical and electronic instrumentation and measurement.

Valeria L. Scarano was born in Lucera (Fg), Italy, on June 7, 1980. She received, in 2005, the M.S. degree in Electronic Engineering and, in 2009, the Ph.D. degree in Industrial and Reliability Engineering from the University of Florence, Italy. Her current research interests include system reliability and availability, reliability evaluation test and analysis for electronic systems and devices, fault detection and diagnosis, quality control, electrical and electronic instrumentation and measurement.

Alessandro Bacioccola received, in 2008, the M.S. degree in Electronic Engineering from the University of Florence, Italy. His current research concentrates in the fields of system reliability, software reliability, testing and development.