# Regression Testing of Multi-Tasking Real-Time Systems: A Problem Statement

Daniel Sundmark, Anders Pettersson and Henrik Thane
MRTC, Mälardalen University
Box 883, SE-721 23 Västerås, Sweden
{daniel.sundmark, anders.pettersson, henrik.thane}@mdh.se

## Abstract

*Regression testing is one of the most intricate parts of software development and maintenance. In complex multi-tasking real-time systems, task interleaving issues, deadlines and other factors further complicate this activity. In all software, however, the process of regression testing comes down to two basic activities: (1) selecting which test cases to re-execute and (2) actually performing the re-tests.*

*The twofold contribution of this paper is the definition of the following problems: The* ***regression test selection problem*** *and the* ***reproducibility problem*** *for multi-tasking real-time system regression testing.*

## 1. Background

In a perfect world, software defects are nonexistent and programmers in bliss and harmony build flawless systems. Unfortunately, in our somewhat less than perfect world, bugs constitute a part of the harsh reality we have to deal with. The discipline-inherent difficulty of producing error-free software has consolidated testing and debugging as major ingredients in the software engineering process.

### 1.1. Test Cases and Test Sets

Ideally, testing should be performed in an exhaustive manner, leaving no doubt that the produced and tested software is free of bugs. Sadly, very few programs exhibit such a low level of complexity that allows exhaustive testing to be performed. Since exhaustive testing in general is impracticable, test set creation must deal with the issue of restricting the number of test cases. The two most common pragmatic approaches for creating non-exhaustive test sets are (1) to derive these from the expected functional behavior (denoted functional test case selection, and used for *black-box* testing), and (2) by analyzing the internal structure of the software (denoted structural test case selection, and used for *white-box* testing).

It could be stated that a test case is best defined by its input and its expected output. However, to be of more practical use, a test case should comprise of <*Id*, *Input*, *Output*, *Configuration*>, where *Id* is a unique identifier used for traceability of a test case. *Input* is the set of values that must be passed as input parameters in order to traverse the desired execution path and/or test the desired functionality. *Output* is the set of expected delivered values, and *Configuration* is the description of how to set up the test environment, how to exercise the test case, etc.

### 1.2. Regression Testing

Testing is performed at different levels during the software development process. Throughout the entire development cycle and at every level, testing will reveal bugs. Most often, these bugs will be corrected via changes in the source code, leading to a necessity to re-test the software. Hence, a need for *regression testing* has evolved. IEEE software glossary defines regression testing as: *"Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements"* [7]. Based on the type of modifications, regression testing can be divided into two categories [4]: *Corrective* regression testing is triggered by changes of the source code, whereas *progressive* regression testing is triggered by specification changes.

Regression testing has been thoroughly exploited for single-tasking software (i.e. software executed in a sequential manner). The main research focus in this area has been on the regression *test selection* problem. This is because of the fact that a careful test selection can significantly reduce testing efforts. The basic idea is to select test cases such that only modified parts are re-tested (since the verifications of non-modified parts still are valid). This is a non-trivial problem because of the difficulty of determining how changes propagate and affect non-changed parts of the code.

As a piece of software evolves during its life cycle, the set of test cases used to verify the correct functionality of the software cannot remain static. Software test sets need to be

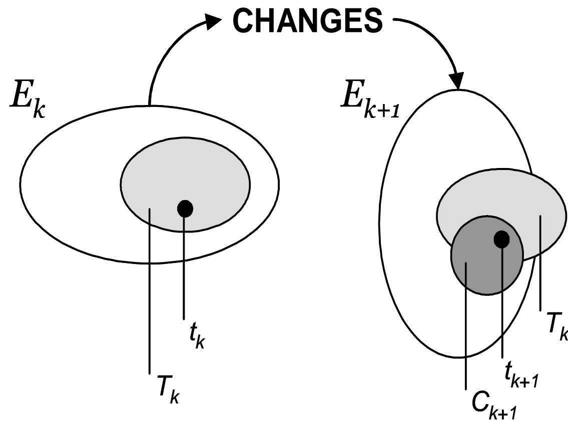


**Figure 1. Test case sets $E_k$ and $E_{k+1}$.**

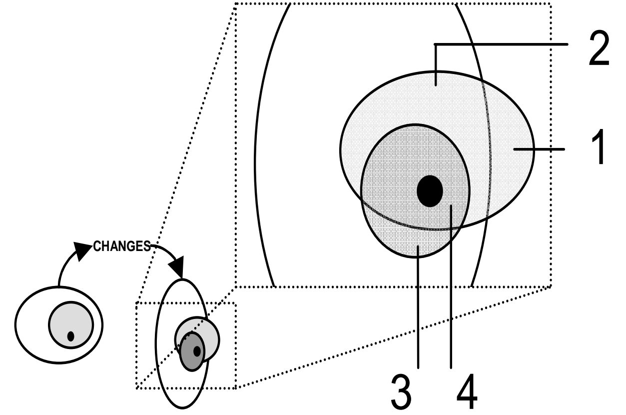


**Figure 2. Derived test case subsets.**

maintained and kept updated as the software evolves. Each change to the software will require a corresponding change in the test set. An initial set of test cases $T_0$ should evolve to a modified test set $T_1$ following software changes. Hence, after $n$ changes, the software is verified by exercising the test cases in $T_n$. In Figure 1, the process of test set evolution is shown. We assume that $E_k$ is a set of test cases, including all possible test cases of our system (i.e. an exhaustive test set). The smaller set $T_k \subset E_k$ contains all test cases that are selected for software verification by the functional- or structural test selection method of our choice. Furthermore, $t_k \in T_k$ is a specific test case, which results in an error. In order to correct this error, the source code of the software is changed. This change yields a new system, and hence a new test set $E_{k+1}$ (the exhaustive set of all possible test cases of the new system). Since the selected test set $T_k$ was created based on the system before the changes, it might not be a clean subset of $E_{k+1}$. Finally, the set $C_{k+1}$ defines all test cases that are affected by program changes.

From the above sets, we can derive four subsets of particular importance for test set maintenance (see Figure 2):

I $T_k \setminus (E_{k+1} \cap T_k)$ contains test cases that no longer are part of the behavior of the software. Note the possibility of $t_{k+1} \notin E_{k+1}$. By changing the system, we may have prohibited execution of $t_k$.

II $(E_{k+1} \cap T_k) \setminus C_{k+1}$ is is the set of unaffected, still valid test cases. These require no re-testing at this stage.

III $C_{k+1} \setminus (T_k \cap C_{k+1})$ holds untested test cases that should be tested due to the software changes.

IV $T_k \cap C_{k+1}$ contains still valid test cases, possibly affected by software changes. These need to be re-tested.

In short, set I needs to be identified, such that no effort is spent trying to exercise test cases that *cannot* be exercised. Similarly, a tight identification of set II saves us the effort of testing test cases that *need not* be tested. Identification of set III is also required in order to keep the test set up to date with the changed system. For this paper, however, set IV is of highest importance. This set contains all test cases that are possibly affected by software changes. The validity of these test cases must be re-verified by means of regression testing. In the next iteration of software and test set evolution, $(E_{k+1} \cap T_k) \cup C_{k+1}$ will serve as $T_{k+1}$.

## 2. Regression Testing of Real-Time Systems

While the regression test selection problem for sequential software has been thoroughly examined, regression testing for multi-tasking real-time systems has often been performed in an ad-hoc manner. Not surprisingly, regression testing for real-time systems implies differences in the test process compared to regression testing of single-tasking software.

### 2.1. Multi-Tasking Test Cases

The main concern for multi-tasking software is system-level testing (i.e., testing at the level of concurrent task execution). Concurrent execution may cause race conditions and interleaving of task statements. This calls for additional efforts in order to ensure the uniqueness of each test case, with respect to execution behavior. This requirement is posted by the fact that two or more test cases with identical input may traverse different task interleaving sequences. Basically, each test case that is exercised traverses a task interleaving sequence (i.e., an ordered sequence of task switches, interrupts and synchronization operations). Hence, we define a test case for testing of multi-tasking

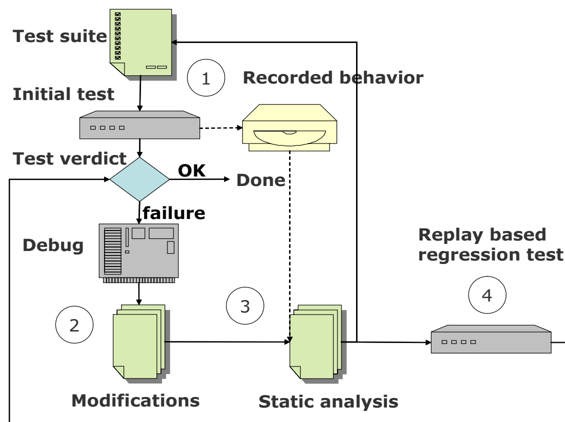


**Figure 3. The replay based regression testing process and its phases.**

software at system level as <*Id*, *Input*, *Output*, *Configuration*, *Interleaving Sequence*>. In our previous research, we have used System-Level Control Flow (SLCF) graphs for describing these sequences [9]. Other methods that uniquely define task interleaving sequences [1, 2] can be used for this purpose. However, most other methods settle for the synchronization sequence (useful for detecting "ordering" and "synchronization" failures), whereas the SLCF graphs also capture timing aspects required for detecting "preemption" and "timing" failures. In regression testing of real-time systems, such failure sources should not be ignored.

### 2.2. The Test Process

Depicted in Figure 3 is our proposal for a replay based regression testing process. Our work is concentrated to the static analysis for validity of test cases (regression test selection problem (3)) and the replay based re-test (reproducibility problem (4)). Starting in phase (1) an inital set, $A$, of test cases is created based on the chosen test technique for integration and system testing. In addition to $A$ there must also exists a set of interleaving sequences, $I$. $I$ can either be empty ($I = \emptyset$) or containing possible interleaving sequences that is analytical derived. After creating $A$ and $I$ the initial test case set $T_0$ is composed by mapping each sequence in $I$ onto the derived test cases in $A$. During initial test runs the behavior of executions (i.e., intereleaving sequences) is recorded. If failures are revealed, the test verdict is false and a debugging phase (2) is started. Via debugging the cause of the failure is pin-pointed. When the exact cause is established the fault(s) are removed by modifying the source code. Before the software is retested it is analyzed in order to establish the validity of the test cases (3). This is done in two steps, first the chosen test technique is responsible for creating the set of canditates for the re-test (to form $C_{k+1}$). Then, the temporal behavior of the modified program must be analyzed in order to establish the validity of the previously recorded interleaving sequence. Those test cases in $C_{k+1}$ with valid recorded interleaving sequences are re-tested using the deterministic replay technique [10]. And for test cases where replay cannot be used the system is re-tested with techniques used today.

### 2.3. The Regression Test Selection Problem

For efficient regression testing, the goal is to chose test cases from $T_{n-1}$ in order to establish the correctnes of the modification. However, there is a trade-off between running a large number of test cases (to be confident that the system is correct), and running a small number of test cases (to spend as little resources on re-testing as possible). This problem of finding the minimal sufficient set of test cases is denoted the *regression test selection problem*.

A major issue in the regression test selection problem for multi-tasking software is the act of finding *infeasible* task interleaving sequences (i.e. execution orderings that practically cannot re-occur). The order and timing of task interleavings are highly affected by code changes. Code changes may also affect the temporal behavior, leading to that some interleaving sequences become infeasible. Basically, when the number of statements to execute is changed, the location (in execution time) of task interleavings is also changed.

### 2.4. The Reproducibility Problem

An important test case attribute when performing regression testing is the test repeatability attribute. This attribute is defined as: *for a test case, the same output (and the same task interleaving sequence) is produced each time the test case is run* [7]. Test repeatability must be achieved in order to establish that encountered faults have been corrected and no further faults have been introduced. If the interleaving sequence of a test case definition and the interleaving sequence of its re-execution do not prove identical, the test case is not properly re-tested. Hence, a regression testing method for multi-tasking systems must ensure the proper reproduction of test cases. This is the *reproducibility problem* of multi-tasking real-time system regression testing.

## 3. Related Work

Previous research work in the regression testing domain has identified five main problem areas. These problem areas are: (1) The problem of analyzing the source code for

changes and their impact on the behavior of the software, (2) version management, (3) creation of test cases, (4) the problem of selecting test cases for regression testing and (5) the nondeterministic run-time behavior during exercise of test cases. (For completeness it should also be described how to deal with maintenance of test cases but this is considered to be included in area (2), (3) and (4).)

As for (1), it is important not only to be able to detect textual changes, but also semantic changes. Detection of semantic changes can be detected e.g. by using a Program Representation Graph, as shown by Horwitz et al. [5].

To our knowledge, there has been no work done on investigating the impact of changes on the temporal execution behavior in the sense of control-flow and data-flow at system level (i.e., flows that are dictated by system calls, task interleaving events, and by the task scheduling mechanism). As for (2), both industry and academia has learned to rely on tools that can manage different versions of source code.

In (3), an initial set of test cases must be created. Initially the test cases are selected based on coverage criteria. Furthermore, in (4), the regression testing subset must be chosen with the trade-off between the ability to reveal failures, and the cost of selecting and exercising the test cases in mind.

In (5), non-deterministic behavior during test runs makes it difficult or even impossible to determine if faults have been corrected. Multi-tasking software testing has focused on testing the order of synchronization events [2] or concurrent events in Java [3], but to the authors knowledge, there are very few research results on testing at the granularity of program statement interleaving [9].

## 4. Conclusion

In this paper, we have emphasized that regression testing of multi-tasking real-time systems includes two main problem domains: the *test selection* problem and the *reproducibility problem*. The test selection problem deals, in addition to selecting the appropriate set of inputs and outputs for re-testing of changed systems, also with the problem of selecting the appropriate set of interleaving sequences. As for the reproducibility problem, the issues discussed are the practical problems of actually repeating test cases that have been selected for re-testing. These problems include repeating task interleaving sequences, system-level- as well as task-level control and data flow.

Even if academic results make regression testing methods and techniques more efficient and less time-consuming, there are other benefits. More important for the industry might be an automation of the test activities such that they can be performed during non-working hours and more frequently, in order to capture faults more early in the software development [8]. Therefore, future proposed solutions must, in addition to solving the technical issues, also allow for automation. This combination will result in economical benefits for the industry, especially for severe faults that rarely propagate to failures [6].

## 5. Future Work

We intend to tackle the test selection problem, as well as the reprocibility problem in a pragmatic way, starting with small systems and software changes of low complexity.

We assume the need for some type of static analysis of code, run-time behavior and interleaving sequences for approaching the test selection problem. As for the reproducibility problem, we will extend the Deterministic Replay method [10] in order to be able to reproduce changed systems by means of interleaving sequence recording and execution replay.

## References


[1] S. Blaustein, F. Oliveto, and V. Braberman. Observing timed systems by means of message sequence chart graphs. *Proceedings of the 24rd International Conference on Software Engineering, ICSE 2002*, pages 707–, 2002.

[2] R. H. Carver and K.-C. Tai. Replay and Testing For Concurrent Programs. In *IEEE Software*, volume 8(2), pages 66–74, 1991.

[3] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[4] I. Granja and M. Jino. Techniques for regression testing: Selecting test case sets taylored to possibly modified functionalities. In *Proceedings of the Third European Conference., Software Maintenance and Reengineering*, pages 2–11, 1999.

[5] S. Horwitz. Identifying the Semantics and Textual Differences Between Two versions of a Program. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, June 1990.

[6] NIST. The Economic Impacts of Inadequate Infrastructure for Software Testing., May 2002.

[7] I. S. G. of Software :Engineering Terminology. IEEE Standards Collection, IEEE Std 610.12-1990. September 1990.

[8] A. K. Onoma, W. T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. In *Proceedings of IEEE Transactions on Software Engineering*, volume 22(8), pages 529–551, 1996.

[9] H. Thane and H. Hansson. Towards Systematic Testing of Distributed Real-Time Systems. In *Proceedings of The 20th IEEE Real-Time Systems Symposium*, pages 360–369, 1999.

[10] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay Debugging of Real-Time Systems Using Time Machines. In *Proceedings of Parallel and Distributed Systems: Testing and debugging (PADTAD*, pages 288–295. ACM, April 2001.