Automated Testing of Timeliness : A Case Study

Robert Nilsson
School of Humanities and Informatics
University of Skøvde
Box 408, SE-54128 Skøvde, Sweden
robert.nilsson@ida.his.se

Jeff Offutt
Department of Information and Software Engineering
George Mason University
Fairfax, VA 22030 USA
offutt@ise.gmu.edu

ABSTRACT

A problem with testing timeliness of real-time applications is the response-time dependency on the execution order of concurrent tasks. Conventional test methods ignore task interleaving and timing and thus do not help determine which execution orders need to be exercised to test temporal correctness. Model based mutation testing has been proposed to generate inputs and determine the execution orders that need to be verified to increase confidence in timeliness. This paper evaluate a mutation-based framework for automated testing of timeliness by applying it on a small control system running on Linux/RTAI. The experiments presented in this paper indicate that mutation-based test cases are more effective than random and stress tests in finding both naturally occurring and randomly seeded timeliness faults.

1. INTRODUCTION

Current real-time systems must be both flexible and timely. There is a desire to increase the number of services that real-time systems offer while using standardized (hardware) components. This increases system complexity and introduces sources of temporal non-determinism that make it hard to predict the execution behavior of tasks [14]. Faults in such predictions may result in timeliness violations and costly accidents. Thus methods are needed to detect violation of timing constraints for computer architectures for which we cannot rely on accurate off-line assumptions. *Timeliness* is the ability for software to meet time constraints. For example, a time constraint for a flight monitoring system can be that once landing permission is requested, a response must be provided within 30 seconds [15].

When designing real-time systems, software behavior is typically modelled by periodic and sporadic tasks that compete for system resources (for example, processor-time, memory and semaphores). The response times of these tasks depend on the order in which they are scheduled to execute, the activation times, and the interaction between tasks. *Periodic* tasks are activated with fixed inter-arrival times, thus all the points in time when such tasks are activated are known. *Sporadic* tasks are activated dynamically, but assumptions about their activation patterns, such as *minimum inter-arrival times*, are used in analysis. Each real-time task typically has a *deadline*, but other time constraints are also possible. Tasks can also have an *offset*, which denotes the time before a task of that type is activated.

Timeliness is traditionally analyzed and maintained using off-line scheduling analysis techniques or regulated online through admission control [17]. These techniques use assumptions about the tasks execution behavior and activation patterns that must be correct for timeliness to be maintained. Further, doing schedulability analysis of non-trivial system models is complicated and requires specific rules to be followed by the run-time system. In contrast, testing of timeliness is general in the sense that it applies to all system architectures and can be used as a complement to analysis for gaining confidence in assumptions by systematically sampling among the execution orders that can lead to missed deadlines. In fact, only some of the possible execution orders typically reveal timeliness violations, and thus, input sequences capable of revealing such execution orders need to be generated for effective testing of event-triggered real-time systems. However, most existing testing techniques are purely black-box and ignore information about real-time design in test case generation [12].

Our framework for mutation-based testing of timeliness is inspired by a model based method for automatic test case generation presented by Ammann, Black and Majurski [2]. The main idea behind that method is to systematically "guess" what faults a system contains and then evaluate how such faults could affect a model of the system. Once faults are identified, test cases are constructed that try to reveal those faults in the system implementation. Heuristic-driven simulation and model-checking has previously been evaluated for automatically generating meaningful test cases within our framework [12]. This paper gives an overview of how the proposed framework is used and presents a case study where the framework is used to test a small real-time control application running on Linux/RTAI.

2. FRAMEWORK OVERVIEW

This section introduces central concepts for timeliness testing and gives a high-level overview of how the testing framework can be applied. Within the proposed framework, estimates or measurements of the temporal behaviors of application tasks are captured in a *real-time applications model*. This model also express the design assumptions about the behavior of the systems environment, for example, physical laws or causality constraints that limit certain task activation events from happening simultaneously. In particular, the testing framework uses a subset of Timed Automata with Tasks (*TAT*) [6, 13] to specify the assumptions about the application under test. This notation can be used to model a set of tasks

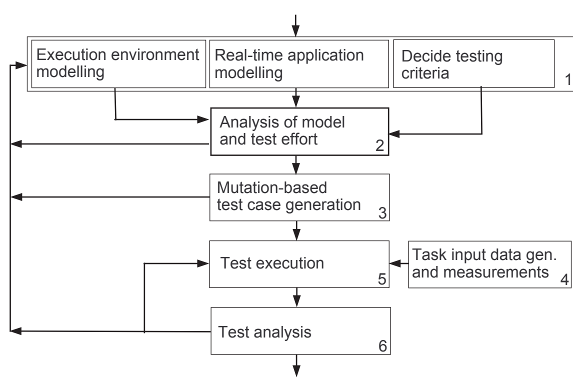

Figure 1: Overview of framework activities

with shared resources. In the framework, these models are used as a deterministic "snap-shot" of the assumed worst case behavior of each task. Task activations are modelled using timed automata [1]. This means that both sporadic and periodic tasks can be modelled, as well as more complex constraints on activation patterns.

Apart from the information about application tasks and environment assumptions, an *execution environment model* is used to capture the overall policies and protocols that are implemented in the target system. For example, the execution environment model may specify that application tasks are scheduled using EDF-scheduling, protect shared data using monitors with FIFO semantics, and that context switches impose a worst-case overhead of two milliseconds. The execution environment model can potentially be reused for several variations of real-time applications using the same type of platform and protocols.

When the real-time application model and the execution environment model are integrated, we refer to the combined model as the *real-time system model*. Figure 1 summarizes the activities within the proposed framework for testing of timeliness.

1. First, a real-time system model is built and test criteria are chosen.
   (a) An execution environment model is configured to correspond with the architectural properties and protocols that are present in the target system.
   (b) The temporal behaviors of real-time tasks that make up the tested real-time applications and the corresponding triggering entities are modelled (by estimations or measurements).
   (c) Suitable mutation-based test criteria are selected based on the required levels of thoroughness and the allowed test effort.
2. At this phase it is possible to perform analysis of the models and refine the application models or test criteria.
3. Test cases are generated automatically from the model to fulfill the test criteria.
4. Sets of input data for individual tasks are acquired using compiler-based methods or temporal unit testing.
5. Test cases are automatically and repeatedly executed to capture different behaviors of the potentially non-deterministic platform.
6. During test execution, the test harness produces logs that can be analyzed off-line to further refine the model or isolate timeliness faults.

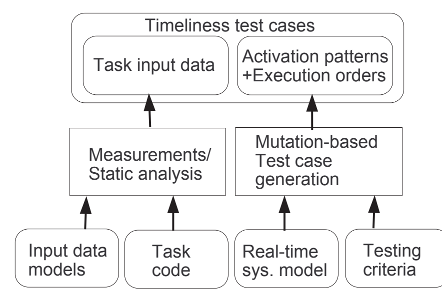

Figure 2: Timeliness tests overview

2.1 Timeliness Test Cases

Figure 2 shows an overview of the sources for data in timeliness test cases. *Activation patterns* are time-stamped sequences of requests for task activation. For example, an activation pattern may specify that task A should be activated at times 5, 10, and 14 while task B should be activated at times 12, 17 and 23. The *execution orders* that are part of test cases predicts how tasks are interleaved when timeliness can be violated for a particular activation pattern. The execution orders can optionally be used to derive a test prefix [9] for test execution; it could also be used during test analysis to determine if a test run has revealed an execution order that might lead to a timeliness violation.

Timeliness test cases should also contain relevant input data for the various tasks so that their execution behaviors can be partially controlled during timeliness testing. *Input data*, in this context, are the values that are read by tasks throughout their execution. For example, a task for controlling the temperature in a chemical process might read the current temperature from a memory mapped I/O port and the desired temperature from a shared data structure. Both values will influence the control flow of the task and, thus, decide the execution behavior of the task. It is common to use input data that cause long execution times or cause shared resources to be used for a long time. There are several ways of obtaining such input data for tasks running undisturbed. Wegener et al. [18] has applied a method based on genetic algorithms to acquire test data for real-time tasks. Petters [14] has used compiler-based analysis for the same purpose. Further, deriving task input data is similar to deriving input data for unit testing of sequential software; hence, methods from that domain can be adapted to make sure a wide range of execution behaviors is covered. Common to all such approaches is that they require the actual implementation and information about the input domain of tasks.

2.2 Mutation-based Test Case Generation

Figure 3 illustrates the automated test generation process. The inputs to mutation-based testing of timeliness are a real-time system model and a test criterion. A mutation-based testing criterion determines the level of thoroughness of testing and what kind of test cases should be produced by specifying what mutation operators to use. *Mutation operators* change some property of the real-time system model to mimic faults and deviations from assumptions that may lead to timeliness violations. Several mutation operators for testing of timeliness have been defined formally and evaluated in

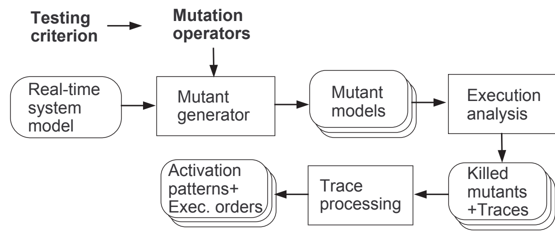

Figure 3: Mutation-based test generation

previous work [11]. The mutation operators used in the this case study are summarized here.

Execution time operators: Execution time mutation operators increase or decrease the assumed execution time of a task by a constant Δ. These mutants represent an overly optimistic estimation of the worst-case (longest) execution time of a task or an overly pessimistic estimation of the best-case (shortest) execution time. Note that the execution time of a task running concurrently with other tasks may be different than running the task uninterrupted; for example if there are caches and pipelines in the target platform.

Lock time operators: Lock time mutation operators increase or decrease the interval before a particular resource is locked relative to the start of that task. An increase in the time a resource is locked may increase the maximum blocking time for a higher priority task.

Unlock time operators: Similarly to the lock time operators, this operator changes the point in time when a resource is unlocked. For example, if a resource is to be locked from time 2 until 4 in the original model, a "+/-1 Unlock time" mutation causes the resource to be held from time 2 to 5 in one of the mutants and from 2 to 3 in another.

A mutant generator tool applies the specified mutation operators to the real-time model and sends each mutated copy to the execution order analysis engine. Execution order analysis determines if and how a specific mutation can lead to a timeliness failure, that is, finding a trace that shows how a task in the mutated model will break a time constraint. Execution order analysis can be performed in different ways. Within our framework, two complementary approaches are proposed. One is based on model-checking [11] and another is based on heuristic-driven simulation [12]. The mechanism that makes it possible to use model-checking for this purpose is the generic schedulability analysis using TAT models described by Fersman et al. [6]. When heuristic-driven simulation is used, each mutated model is simulated and fed with inputs so that the resulting execution order can be analyzed. Activation patterns are then iteratively modified using heuristics that stress the simulated mutant to miss a deadline. The simulation-based approach is more extendible and can be used to analyze larger systems, whereas the model-checking based approach is more reliable and always reveals if a deadline can be missed in a mutated model [12]. If execution analysis reveals non-schedulability or a missed deadline in a mutated model, the model is marked as *killed*. A mutated model that contains a fault that can lead to a timeliness failure is called a *malignant mutant*, whereas a mutated model that contains a fault that cannot lead to a timeliness failure is called a *benign mutant*. Activation patterns that can kill mutated models are later used to create timeliness test cases for the real target system. It is also possible to extract the execution orders that lead to deadline violations from the model traces.

2.3 Test Execution

Test execution is the process of running the target system and injecting stimuli according to the activation pattern part of the timeliness test case. Since the target platform may contain sources of nondeterminism, several execution orders may occur in the real system for the same activation pattern. Consequently, a requirement for testing timeliness is that each activation pattern can repeatedly be injected automatically. Each single execution of a test case is called a *test run*, that is, a test case execution may consist of one or more test runs. The outputs collected from the system during test execution are collectively called the *test outcome*.

The *test harness* incorporate all the software needed for controlling and observing a test execution on the target system. The design of the test harness is critical for real-time systems since parts of it typically must be left in the operational system to avoid probe effects.

During *test analysis* the test outcome produced by the system during test execution is analyzed with respect to the expected outcome. A *test result* indicates whether a test execution succeeded in revealing a fault or not. For timeliness testing, the most relevant test outcomes are the execution orders and temporal behavior observed during a set of test runs.

3. TIMELINESS TESTING CASE STUDY

In this case study, the proposed framework is used to test timeliness of a control system prototype running on a real-time operating system platform. The activation patterns generated are compared with a larger set of random activation patterns to establish confidence in the effectiveness of the method.

Random generation of activation patterns is not an optimal baseline for this research, but unfortunately there are no well established timeliness testing methods or tools available for this, and most methods used in practice appear to be domain specific or adhoc [9].

The application used in our experiments is a control system for a robot arm, hereafter referred to as the *slave robot*. The slave robot has the ability to move in a fixed operational plane, using three servo motors for adjusting angles of joints. The purpose of the slave robot in this application is to balance a ball on a beam while the other end of the beam is moved by an independent *master robot*. A camera captures images to determine the beam angle and the ball's position on the beam. A force sensor on the slave robot arm is used to help determine the movement of the other robot together with the camera images. This case study focuses on a sub-system that controls the movement of the slave robot.

3.1 Real-time Design

The system prototype consists of seven tasks; of these, four are periodic and three are sporadic. Figure 4 shows an overview of the relations between these tasks.

One sporadic task continuously maintains and updates the current trajectory based on orders and adjustments received from a remote node. This task updates a cyclic buffer containing the current trajectory and desired path for the end-point of the beam, hereafter referred to as the *hand*. New updates from the network may require

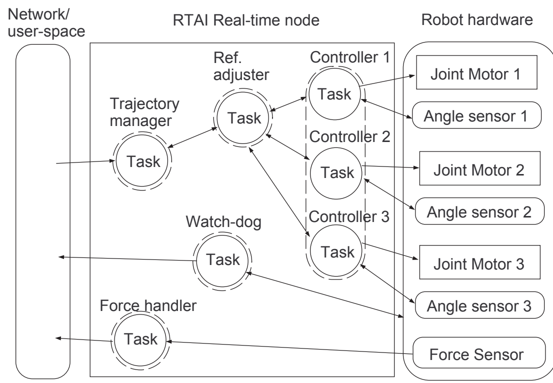

Figure 4: Tasks and their interactions

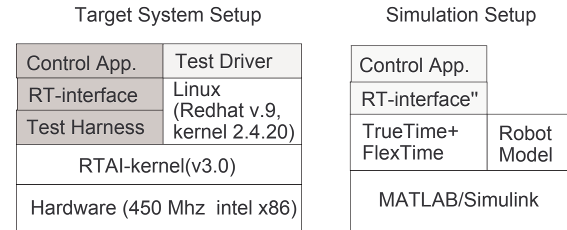

Figure 5: Overview of configurations

recalculation or extension of the planned trajectory. This task also translates trajectory coordinates of the form $< x, y, angle >$ to the corresponding set points for the three joints. Here x and y are the position of the hand relative the base point of the robot arm; $angle$ is the desired angle of the hand, relative the ground. A second sporadic task coordinates the motor controllers by adjusting the reference signal (set-points) to follow the precalculated trajectory. This sporadic task is activated when the robot arm is within bounds of certain intermediate via-points along the trajectory. In this prototype, the robot arm is assumed to be equipped with a sensor that sends interrupts when the amount of force applied on the robot hand is changed. The interrupts from the force sensor are serviced by a third sporadic task that filters the information and send notifications to a remote node if changes are above a certain threshold.

Three system tasks are periodic controllers with high sampling rate for accurately controlling the joint motors of the robot. These controllers get feedback through angle sensors that are read at the beginning of each task invocation. The implementation of controllers are of PID-type, parameterized according to the dynamics of the controlled system. The three controller tasks are implemented using a single thread similar to a cyclic executive (c.f. [5]). This means that the periods of the controllers share a common multiple and that the activation order of these periodic tasks relative each other is known.

A high priority watch-dog task is used to ensure that the hardware is always serviced within a certain interval of time. The reason for this is that the robot hardware requires the hardware interface to be serviced with a certain minimum rate. To monitor the polling, a shared time stamp structure is updated every time any task polls the hardware interface; if the watchdog detects that the hardware has not been polled within the interval, actions are taken to service the interface and report the failure. Further, the watchdog task is responsible for checking the integrity of the control system by regularly sending messages to remote nodes.

A *shared resource* is an entity that is needed by several tasks but only should be accessed by one task at a time. Examples of such resources are data structures containing state variables that need to be internally consistent and non-reentrant library functions. Mutual exclusion between tasks can be enforced by holding a mutex or executing within a monitor. The tested system has four resources that require mutual exclusive access; these are *(i)* the robot hardware device interface, *(ii)* the network send buffer, *(iii)* a trajectory buffer, and *(iv)* a setpoint data structure.

For an application such as a robot arm controller, there is a trade-off between the control performance and the periods (sampling frequency), hence, it is up to the designer to set suitable periods. In other applications involving the controlled robots, the sampling frequency of the joint controllers varies between 200 Hz (which is the lower limit from the closed-loop dynamics of the system) up to 4 KHz (to improve the disturbance rejection of the system) [8]. For the tested prototype, the sampling frequency of the controller is set at 2.5 KHz.

For sporadic tasks, minimum-interarrival times and deadlines are typically decided by properties of the environment, network protocols, or set by system designers to meet end-to-end deadlines in a distributed system [15]. For the testing experiments in this paper, the attributes of sporadic tasks are chosen so that the system remain timely in worst-case situations without wasting resources.

3.2 Target Platform

The target system for this application is an x86 based PC (a 450 MHz Pentium III) running Linux with RTAI (Real-Time Applications Interface). Linux/RTAI has previously been used for implementing real-time applications and is designed so that the Linux kernel and all the user-level processes can be fully preempted when a real-time task arrives [4]. This also means that the Linux operating system will run in the background reclaiming “wasted” cpu resources for its desktop applications. Theoretically, the only consequence this should have on the tested real-time system is that shared hardware caches may be flushed when the real-time system is assumed to be idle.

An overview of the system is shown in figure 5. As can be seen, the real-time application (shaded gray) has a layered structure. The motivation for the layered structure is to hide the test instrumentation from the code of the application tasks and to make the application prototype portable to other real-time platforms. The MATLAB/simulink target also makes it possible to test how the control application would interact with a simulated hardware robot given that the assumption of temporal behavior is correct.

3.3 Test Harness

The test harness incorporates all the software needed for injecting stimuli and observing a test run on the target system. The design of the test harness is critical for real-time systems since it typically must be left in the operational system to avoid probe effects [7]. The probe effect means that the probes used to observe the behavior of the system actually changes its behavior. Hence, if the probes are removed after testing, the results from testing are invalidated.

The activation of periodic tasks is handled by using the facilities provided by the RTAI native interface. Hence, the periods and start times of tasks are specified at system initialization and activation (stimuli) are generated by the system timer. For injecting sporadic stimuli at the points in time specified by the activation pattern part of timeliness test cases, a sporadic task is added to emulate the interrupt-handler in the system. This task reads an activation pattern specification from a FIFO-queue that has been populated by the test driver before testing starts. The event sequence contains the type of the next sporadic task to be activated and an absolute point in time when it should be activated. Since this task executes on the highest priority and does not share any resources with the application tasks, the execution disturbance from this task is assumed to be very similar to that of interrupt handlers. Further, the same type of mechanism are needed to execute all test suites and should therefore not affect the result of the experiments presented in this paper.

According to Schutz [16], the behavior of a real-time system can be captured by monitoring access to time, context switches and asynchronous interrupts. Apart from this, it is helpful to log the beginning and end of each task invocation so that it is possible to check that it executes within its time constraints. Currently all tasks log their own events in separate main-memory buffers. Thus, no synchronization between real-time tasks are necessary for this purpose. The events in the log are time-stamped so that they can be consolidated and analyzed after a test run has been completed. The main-memory buffers are also accessed. In the following experiments, information from these main-memory buffers are collected after a test run has ended, to avoid unnecessary competition for resources.

3.4 Unit Testing and Measurements

As mentioned in section 2.3, when performing system level testing of timeliness, tasks are activated according to the activation pattern part of timeliness test cases to cause a potential worst case interleaving to occur. During this process the individual tasks are given input data that are good candidates for causing long execution times. This section describe how task input data can be generated (step 4 in figure 1).

In the following experiments, the same classes of task input data will be used for all compared activation pattern test suites. Consequently, the thoroughness of this step is important for the result of all the methods. Unfortunately, the methods described in section 2.1 that are directly developed for automatically deriving this type of input data could not be used in this case. Instead, a black-box testing technique based on equivalence partitioning and pair-wise coverage was used to create input data for each task. In particular, the equivalence partitioning was designed to exercise a wide range of temporal behaviors of tasks. This means that parameters that change the execution time of tasks were identified. The values of each parameter were then pair-wise combined into a set of input data for each task. Each task was then run on the target system without any disturbances from other application tasks. For each input datum the task was activated periodically 20 times, and the procedure was repeated 5 times. The task's execution behavior for a particular input data, over these 100 invocations, was measured.

3.4.1 Combining Task Input Data

Before starting system level testing it is necessary to decide what input data for the different tasks should be combined when tasks run concurrently in system tests. The most intuitive choice is to combine the input data that causes the longest execution time for each task. However, anomalies might exist so that input data that cause a shorter execution time during measurements (when the task is executing undisturbed) cause a disproportionately longer execution time when a task is executed concurrently with other tasks.

One way to expand the testing while still limiting the number of tests is to use the input data that causes the longest execution times as a "base-choice" [3]. Using this test method, all tasks, except one, execute with the input data that cause the longest execution time. The inputs to the last task are varied among the other candidates for longest execution time.

For the case study we selected three candidate input data for each sporadic task, and two input data for each periodic task, and combined them using the input data with the longest execution time as the base-choice. The other two candidates were chosen based on the diversity of the paths covered, with bias for test cases that executed for a long time (overall, or inside critical sections). Since there are 3 sporadic tasks and 3 periodic tasks that depend on inputs during execution, we get 10 different sets of input data using the base-choice method. The reason only two input data were selected for each periodic task is that the code is identical for these three tasks (except for constant configuration differences), so in effect all different classes of input data were tested. The test harness was designed so that each activation pattern was run at least one time with each of the combined input data sets.

3.5 Modelling and Test Case Generation

To generate the most effective activation patterns to combine with the task input data, we used measurements collected during unit testing to construct a TAT model of the system under test. In particular, we used a TAT model automata notation where the n*th* task instance is modelled to behave as the worst case behavior observed for that particular task instance. One effect of this is that if, for example, the third invocation of a task always executes longer in all measurements, then it is modelled to have a longer execution time. The reason for using this kind of automata pattern was primarily that the caching on the target platform caused the first task invocation to execute significantly longer in all measurements. Further, an execution environment model encoding was built to correspond to the fixed priority scheduling and priority inheritance protocols defined by the open-source operating system documentation. The testing criterion was set to kill all mutants generated using the "+/-20 Execution time", "+/-20 Lock time" and "+/-20 Unlock time" mutation operators.

Using the tools we developed for mutation-based test case generation [10], 70 mutants were automatically created and analyzed in five separate trials. Table 2 lists the number of unique mutants that were killed in any of the five trials in the column marked "K_{GA}"; the average number of mutants killed in a trial is listed in column "$\overline{K_{GA}}$". For each trial, the activation pattern that killed a mutant model was added to a test suite. This resulted in five distinct test suites for each mutation operator type. For comparison, five test suites with 10 random activation patterns each were also created and run with the same sets of task input data. All the random activation patterns for this experiment were generated with uniform distribution so that the specified minimum interarrival time was maintained while the average system load was at the same level as for the mutation-based tests.

As a further comparison, we manually generated a test suite with simple activation pattern types that we intuitively believed would

Task	Period/MIAT	Offset	Deadline
TrajectoryMgr	600	600	800
RefAdjuster	700	1500	650
ForceHandler	600	300	600
Controller 1	400	0	400
Controller 2	400	200	400
Controller 3	800	0	800
Watchdog	600	0	500

Table 1: Task set attributes

Mutation operator	Δ	μ	K_{GA}	$\overline{K_{GA}}$
Execution time	20	14	7	7.0
Lock time	20	28	8	6.0
Unlock time	20	28	6	4.0
Total	-	70	21	17.0

Table 2: Results from test generation

reveal timeliness failures. The activation patterns were based on the ideas: *(i)* to activate all sporadic tasks with their minimal interarrival time, and *(ii)* let the first activation of all sporadic tasks occur simultaneously, with some offset, and then reactivate them at the assumed maximum frequency.

3.6 Test Execution and Evaluation

Each generated test suite was loaded by the automated test harness and executed on the target system. Each test execution was repeated 10 times using a different combination of the task input data, as described in section 3.4. The results of the test executions are presented in table 3. In this table, the columns marked 'T' contain the number of test cases in each test suite. The columns marked 'F' contain the number of effective test cases in each test suite, that is, a test case that revealed a timeliness failure.

As this table illustrates, all test suites generated with mutation testing (Exec., Lock, Unlock) found timeliness violations in the real system. Neither the manual nor the randomly generated tests cases found any timing faults in any of the trials. Since all the random tests cases were unique activation patterns, it can be concluded that any one of the five test suites complying to, for example, "+/-20 execution time" coverage is more effective than 50 random activation patterns for this particular system.

3.6.1 *Effectiveness in Finding Seeded Faults*

The previous experiment indicates that test suites with activation patterns generated using mutation-based testing can be significantly more effective than randomly generated activation patterns and manually created stress tests. However, since random testing could not reveal any timeliness failures, it is difficult to analyze how big the difference is. This experiment is designed to collect more data about the difference in test suite effectiveness.

Trial	Random		Manual		Exec.		Lock		Unlock	
	T	F	T	F	T	F	T	F	T	F
1	10	0	4	0	7	4	7	5	4	1
2	10	0	4	0	7	4	7	4	5	1
3	10	0	4	0	7	4	5	4	5	0
4	10	0	4	0	7	5	5	1	4	1
5	10	0	4	0	7	3	6	2	2	0
Total	50	0	20	0	35	20	30	16	20	3

Table 3: Results from test executions

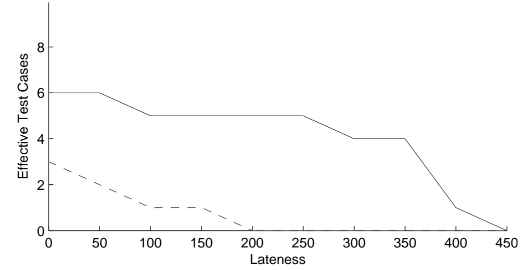

Figure 6: Failures revealed in version 7

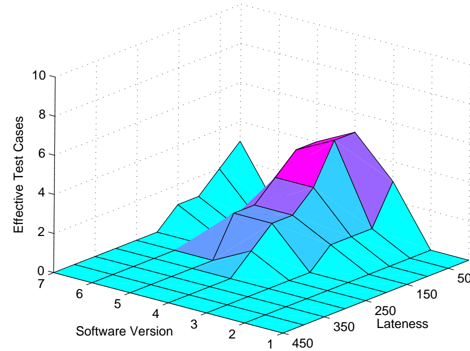

Figure 7: Results for random test-suite

For this experiment we created seven versions of the application with timeliness faults seeded into the code. Faults were seeded by adding a busy delay statement at a random point in each task. The fault seeding was controlled so that each task was affected by one fault. This means that the fault can occur within a condition statement or in a code segment where a shared resource is locked. The size of the busy delay added to the source code was 80 microseconds.

Since this experiment requires seven times as many test-runs as in the previous experiment, only one mutation-based testing criterion was compared with random testing. In particular, five "+/-20 execution time" test suites and five random test suites from experiment one were rerun on each of these versions of the target system. The amount of slack assumed in the system was iteratively increased in steps of 50 microseconds.

As an example, figure 6 show the number of effective test cases for two test suites executed on version 7. In this figure, the solid line is from the mutation-based tests and the dashed line is from the random tests. Point 0 on the x-axis shows the number of effective test cases, assuming the deadlines listed in table 1. The values along the x-axis represent an iterative increase of all deadlines in table 1, expressed in microseconds. This figure gives us a indication of how many test cases of each type would reveal timeliness violations as the system slack is increased.

An overview of all results is presented as surface plots in figures 7 and 8. These figures contain the results of the most effective test suites for each software version with seeded faults. The graphs contain the same information as in figure 6, but with each software version plotted on the Z-axis. From these figures it can be

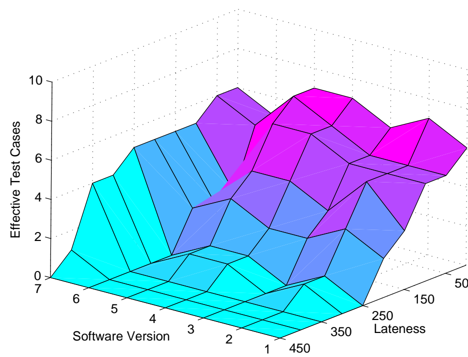

Figure 8: Results for mutation test-suite

seen that the test suites generated with mutation-based testing contain a larger fraction of effective test cases, and consistently reveal behaviors with longer response times than test suites with random activation patterns.

4. CONCLUSIONS

This paper has outlined a framework for testing timeliness in event-triggered real-time systems. To explain how the framework is used, and to evaluate its effectiveness, it was applied in a case study with a robot control application running on an open-source real-time operating system. The experiments indicate that the approach is significantly more effective in revealing timeliness faults than both randomly generated test suites and manually created test suites that try to maximize the load on the system. When increasing the system slack, the mutation-based test suites consistently remain more effective than random test suites in finding seeded timeliness faults. The results of the experiments indicate that if only random performance stress testing is used for testing timeliness, then significant timeliness faults can remain undetected.

The generation of mutation-based test suites is significantly more costly than the generation of random test suites. However, the importance to use meaningful activation patterns for testing timeliness, as demonstrated in this paper, justifies the difference in computational complexity off-line. To further decrease human test effort, the execution behavior measurement and model refinement steps can be automated and connected in a closed-loop so that the test execution harness can iteratively explore the behavioral boundaries with respect to timeliness.

We believe that one of the main reason that the mutation-based testing approach is effective is that the system models extracted and used for test case generation take internal system design into consideration while remaining deterministic. The deterministic models can systematically be mutated and analyzed, and the resulting activation patterns can then be run multiple times on the non-deterministic target system. In this way, the framework divides and conquers the testing problem.

5. REFERENCES

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods*, pages 46–54. IEEE Computer Society, December 1998.

[3] P. Ammann and J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS 94)*, pages 69–80, June 1994.

[4] E. Bianchi, L. Dozio, G. Ghiringhelli, and P. Mantegazza. Complex control systems, applications of DIAPM-RTAI at DIAPM. In *Realtime Linux Workshop*, Vienna, 1999.

[5] Burns and Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 2001.

[6] E. Fersman. *A Generic Approach to Schedulability Analysis of Real-Time Systems*. PhD thesis, University of Uppsala, Faculty of Science and Technology, 2003.

[7] J. Gait. A probe effect in concurrent programs. *Software - Practice and Experience*, 16(3):225–233, March 1986.

[8] D. Henriksson. *Resource-Constrained Embedded Control and Computing Systems*. PhD thesis, Department of Automatic Control, Lund University, 2006.

[9] R. Nilsson, S. Andler, and J. Mellin. Towards a Framework for Automated Testing of Transaction-Based Real-Time Systems. In *Proceedings of Eigth International Conference on Real-Time Computing Systems and Applications (RTCSA2002)*, pages 109–113, Tokyo, Japan, March 2002.

[10] R. Nilsson and D. Henriksson. Test case generation for flexible real-time control systems. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 723–731, Catania, Italy, September 2005.

[11] R. Nilsson, J. Offutt, and S. F. Andler. Mutation-based testing criteria for timeliness. In *Proceedings of the 28th Annual Computer Software and Applications Conference (COMPSAC)*, pages 306–312, Hong Kong, September 2004. IEEE Computer Society.

[12] R. Nilsson, J. Offutt, and J. Mellin. Test case generation for mutation-based testing of timeliness. In *Proceedings of the 2nd International Workshop on Model Based Testing*, pages 102–121, Vienna, Austria, March 2006.

[13] C. Norström, A.Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Proceedings of Ninth International Conference on Real-Time Computing Systems and Applications (RTCSA'03)*, Hong Kong, December 1999.

[14] S. M. Petters and G. Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Proc. 6th International Conference on Real-Time Computing, Systems and Applications (RTCSA'99)*, Hong Kong, 1999.

[15] K. Ramamritham. The origin of time constraints. In M. Berndtsson and J. Hansson, editors, *Proceedings of the First International Workshop on Active and Real-Time Database Systems (ARTDB 1995)*, pages 50–62, Skövde, Sweden, June 1995. Springer.

[16] W. Schütz. Fundamental issues in testing distributed real-time systems. *Real-Time Systems*, 7(2):129–157, September 1994.

[17] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo. *Deadline scheduling for real-time systems*. Kluwer academic publishers, 1998.

[18] J. Wegener, H. H. StHammer, B. F. Jones, and D. E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.