A Mutation-based Testing using Genetic Algorithm for Timeliness of Real Time Systems

G.N Purohit*
A.M Sherry**
Manish Saraswat***

Abstract

A problem when testing timeliness of event-triggered real-time systems is that response times depend on the execution order of concurrent tasks. Conventional testing methods ignore task interleaving and timing and thus do not help determine which execution orders need to be exercised to gain confidence in temporal correctness.

This paper presents and evaluates a framework for testing of timeliness that is based on mutation testing theory on uni processor systems. The framework includes two complementary approaches for mutation-based test case generation, testing criteria for timeliness, and tools for automating the test case generation process.

Keywords - Real Time Systems, Temporal Correctness Timeliness, Genetic Algorithm.

1. Introduction

Real-time systems must be dependable as they often operate in tight interaction with human operators and valuable equipment.

A trend is to increase the flexibility of such systems so that they can support more features while running on "off-the shelf" hardware platforms. However, with the flexibility comes increased software complexity and non-deterministic temporal behavior.

There is a need for verification methods to detect errors arising from temporal faults so that confidence can still be placed in the safety and reliability of such systems.

A problem associated with the testing of real-time applications is that their timeliness depends on the execution order of tasks. This is particularly problematic for event-triggered and dynamically scheduled real-time systems, in which events may influence the execution order at any time (Sch¨utz 1994). Furthermore, tasks in real-time systems behave differently from one execution to the next, depending not only on the implementation of real-time kernels and program logic, but also on efficiency of acceleration hardware such as caches and branch-predicting pipelines.

*Dept. of Mathematics & Computer Science, Banasthali University, Banasthali
**Institute of Management & Technology, Ghaziabad, (U.P)
***Department of MCA, Geetanjali Institute of Technical Studies, Udaipur, (Raj.)

Non-deterministic temporal behavior necessitates methods and tools for effectively detecting the situations when errors in temporal estimations can cause the failure of dependable applications.

In summary, problems with testing of timeliness arise from a dynamic environment and the vast number of potential execution orders of event-triggered real-time Systems. Therefore, we need to be able to produce test inputs that exercise a meaningful subset of these execution orders. Within the field of software testing, several methods have been suggested for model-based test case generation but few of them capture the behavior that is relevant to generate effective timeliness test cases.

Related Works

Liu & Layland 1973 presents scheduling analysis methods by algorithmic ways to derive worst case situations for their assumed system models. For example, a set of periodic tasks, without shared resources using rate monotonic priority assignment, experience their worst case response time when all tasks are released simultaneously and execute their longest time Hence, releasing all the tasks simultaneously at a critical instant would be the only meaningful test case for testing timeliness for such a system. However, not all real-time systems meet the assumptions in such a simple model.

Burns & Wellings 2001 presents a methods that analyzed timeliness of embedded real-time systems is traditionally using scheduling analysis techniques or regulated online through admission control and contingency schemes these techniques use assumptions about the tasks and load patterns that must be correct for timeliness to be maintained.

Chenyang Lu, John A. Stankovic, Sang H. Son, and Gang Tao presents Feedback Scheduling (FS) based on control theory treats the system scheduling as a control plant and uses a controller to adjust the scheduling parameters at runtime. This approach offers no temporal isolation. It is assumed that the system is composed of Model Predictive Control (MPC) tasks served by CBS. However, they do not provide dynamic reconfiguration. If a system has no idle time.

Random testing is significant method of real time system testing, however, does not detect certain value combinations that might be significant for the temporal behavior.

Consequently, for testing timeliness the application of new approaches is examined in this paper, namely Model based testing, Random Testing and genetic algorithms based testing using mutation concept of testing Here we also examine effectiveness of genetic algorithms to establish timeliness of complex real time systems.

2. Issues Related to Testing of Timeliness in Real-Time Systems

Timeliness is difficult to implement, especially if the system under test is regarded as a black box, which is a necessity for a commercial program. For instance, the time granularity of the tool needs to be as small or smaller than the time granularity of the system under test. Another reason for the difficulty of finding for automatic test execution of real-time systems is that many real-time systems are embedded and lacking standardized interfaces. Also many real-time systems have specialized application domains with very specific demands. Imagine for instance the different demands of mobile phones, brake-by-wire systems, and pacemakers. Still another reason is that some systems are built using state-of-the-art technology, which means that there hardly exist tools that support the new technology.

The term timeliness fault denotes a mistake in the implementation or configuration of a real-time application that may result in unanticipated temporal behaviors.

A timeliness error is when the system internally deviates from assumptions about its temporal behavior. A timeliness failure is an externally observable violation of a time constraint. In a hard real-time system, this typically has an associated penalty or consequence for the continued operation of the overall system.

3. Mutation Testing

Mutation testing is a method in which a program (DeMillo, Lipton & Sayward 1978) or a specification model (Ammann & Black 1999) is copied and mutated so that each copy contains an artificially created fault of a particular type. Each mutated copy is called a mutant. Test cases are specially generated that can reveal the faults in the copies of the program or model (manually, or using reachability tools). For example, if one occurrence of the operator '>' is mutated into the operator '<', then a test case must be selected that causes the mutated code to be executed and the corresponding failure to be detected. If such a test case is found, then the mutant is killed.

4. Generation of Test Cases using Model- Checking

Model-checking can be used to kill mutants, that is, finding a trace that shows how a task in the mutated model violates a time constraint.

The mechanism that makes this possible is the generic schedulability analysis using TAT model-checking described by Fersman et al (Fersman et al. 2002).

This mechanism is implemented in the Times tool, developed at Uppsala University (Amnell, Fersman, Mokrushin, Pettersson & Yi 2002).

Once a real-time system has been modelled in TAT, an implementation of the mutation operators, defined automatically produce each mutant and send it to the model checker for schedulability analysis. If the mutant is malignant the schedulablility analysis results in a trace of transitions over the automata model that leads to a violated timing constraint.

From a timeliness testing perspective, the points in time in which different sporadic tasks are requested are interesting to use as activation patterns hence, by parsing the automata trace it is possible to extract the points in time where stimuli should be injected in order to reveal a fault in a system represented by the mutated model.

In the trace from TAT model-checker, such points are characterized by transitions to task activation locations (that is, a location li where M(li) maps to an element in P). For each such transition, the task associated with the activation location, M(li), together with the current global time, are added to the activation pattern part of the derived test case.

The same trace contains information of how the system is assumed to behave internally for the timeliness failure to occur. This can be extracted by monitoring the state changes over the set of active tasks in the ready queue this information can be used during test case execution and test analysis.

The model-checking approach for generating test cases can be used with arbitrary complex automata modelling constraints on task activation patterns (in the environment of the real-time system) without changing the modelling notation or model-checker implementation. This makes it possible to model systems where some sporadic tasks always arrive in clusters and others never occur at the same time.

The main advantage with this approach is that, theoretically, if a mutant is malignant it is killed by the model-checker, given sufficient time and resources for the analysis, However, when system models increase in size and a larger fraction of triggering automata becomes more non-deterministic, the analysis performed by the model-checker suffers from reachable state space explosion.

A further limitation of the model-checking approach is the assumption that the target system complies with a limited set of execution environment models supported by the model-checker tool. If the execution environment model must be changed, that change must be done in the "back-end" of the model-checker. It is unclear how such changes affect the effectiveness of schedulability analysis.

5. Genetic Algorithms for Test Case Generation

Genetic algorithms are suggested for the heuristic search of TAT models, since they are configurable and are known for coping well with search spaces containing local optima (Michalewicz & Fogel 1998).

Genetic algorithms operate by iteratively refining a set of solutions to an optimization problem through random changes and by combining features from existing solutions.

In the context of genetic algorithms, the solutions are called individuals and the set of solutions is called the population. Each individual has a genome that represents its unique features in a standardized and compact format. Common formats for genomes are bit-strings and arrays of real values.

The fitness function in genetic algorithms is to evaluate the optimality or fitness of a particular individual.

Cross-over functions are applied on the selected individuals to create new individuals with higher fitness in the next generation. Traditionally, a function applied on a single individual is called a "mutation" but to avoid ambiguity we use the term cross-over functions for all the functions that result in new genomes for the next generation.

A genetic algorithm search typically continues for a predetermined number of generations, or until an individual that meets some stopping criteria has been found.

In general, three types of functions need to be defined to apply genetic algorithms to a specific search problem: (i) a genome mapping function, (ii) heuristic cross-over functions, and (iii) a fitness function.

The following subsections suggest such functions for mutation-based timeliness test case generation.

6. Test Case Generation Experiment

In particular, these real-time systems are mixed load of soft and hard real-time tasks that share resources. The periodic tasks are soft and implement adaptive controllers for three inverted pendulums and the sporadic tasks are modeled to have hard deadlines.

The genetic algorithm is implemented as in on following two real time systems

Base-line Real-time System
This experiment uses a small task set that has simple environment models but complex interactions between the tasks. Static priorities were assigned to the tasks using the deadline monotonic scheme, that is, the highest priority was given to the task with the earliest relative deadline. Arbitrary preemption was allowed.

The system used the immediate ceiling priority protocol to avoid priority inversion (Sha, Rajkumar & Lehczky 1990). That is, if a task locks a semaphore then its priority becomes equal to the priority of the highest priority task that might use that semaphore, and is always scheduled before lower prioritized tasks. The "first come first served" policy is used if several tasks have the same priority.

The base-line setup has five tasks (denoted A-D, and listed in table). Two tasks are sporadic and the three remaining tasks are strictly periodic.

The system has two shared resources and one precedence relation between tasks D and A. The precedence relation specifies that a new instance of task A cannot start unless task D has executed after the last instance of A. Table-1 describes the assumptions of the task set. The first column ("ID") indicates task identifiers, column 'c' gives execution times, and column'd'

Table 1: Tasks set for Base Line Experiment

ID	c	d	SEM	PREC	MIAT	OFS
A	3	7	{(S1, 0, 2)}	{D}	≥ 28	10
B	5	13	{(S1, 0, 4), (S2, 0, 5)}	{}	≥ 30	18
C	7	17	{(S1, 2, 6), (S2, 0, 4)}	{}	40	6
D	7	29	{}	{}	20	0
E	3	48	{(S1, 0, 3), (S2, 0, 3)}	{}	40	4

provides relative deadlines. The "SEM" column specifies the set of semaphores used

Table 2: Results from the Base Line System Experiment

Mutation operator	μ	μ_M	K_{MC}	K_{GA}	$\overline{K_{GA}}$	$\overline{GEN}$
Execution time	10	6	6	6	5.8	7.6
Lock time	8	1	1	1	1.0	2.2
Unlock time	11	2	2	2	2.0	1.3
Hold time shift	14	0	1	0	-	-
Precedence	20	14	15	14	14.0	1.2
Inter-arrival time	10	3	4	3	3.0	5.7
Pattern offset	10	3	5	3	3.0	2.5
Total	83	29	33	29	28.8	-

malignant after this comparison is listed in column "μM".

An interesting observation is that all the malignant mutants were killed within 10 generations on average. Furthermore, all the malignant mutants where killed in 7 of the 8 experiments and in which interval they are required. Column "PREC" reveals what other tasks have precedence over tasks of this type. For sporadic tasks, the "MIAT" column contains the minimum inter-arrival time assumptions. For periodic tasks the same column contains the fixed inter-arrival time. Column "OFS" denotes the initial offset constant.

Table 2 contains the results from mutation testing the task set in Table 1.

A Δ value of 1 time unit was used to generate the mutants. The number of mutants generated for each operator type is listed in column μ and the number of mutants killed by model-checking is contained in column "K_{MC}".

For the genetic algorithm setup, we used a population of 20 individuals per generation and searched each mutant for 100 generations before terminating.

The first generic cross-over function changed a random value in the genome representation, the second created a new random individual in the population and the third replaced a random value in the genome with 0. To gain confidence in the results, each experiment was repeated in eight trials, each with a different random seed.

The number of mutants killed using genetic algorithms in any of the trials is listed in column "K_{GA}". Column "K_{GA}" lists the average number of malignant mutants that were killed per trial. The average number of generations required to kill malignant mutants of this type is contained in column "GEN".

Complex Dynamic Real-time System

The purpose of this experiment was to evaluate how well the genetic algorithm based method generates test cases for a system consisting of more sporadic tasks as well as complex scheduling and concurrency control protocols.

In this setup we use the Earliest Deadline First (EDF) dynamic scheduling algorithm together with the Stack Resource Protocol (SRP).

The EDF protocol dynamically reassigns priorities of tasks so that the task with the current earliest deadline gets the highest priority. The SRP protocol is a concurrency control protocol that limits chains of priority inversion and prevents deadlocks under dynamic priority scheduling. This is done by not allowing tasks to start their execution until they can complete without becoming blocked (Baker 1991).

This system consists of 12 hard real-time tasks, seven of which are sporadic and only five periodic.

Table 3: Tasks set for Complex Real Time System Experiment

ID	c	d	SEM	PREC	MIAT	OFS
A	3	20	{(S1,0,2), (S2,0,2)}	{}	≥ 28	10
B	4	24	{(S1,0,3)}	{}	≥ 30	4
C	5	35	{ (S2,2,5)}	{}	≥ 38	6
D	6	57	{(S2,0,6), (S3,2,5)}	{}	≥ 48	0
E	5	51	{}	{}	≥ 52	7
F	6	39	{(S3,3,6)}	{}	≥ 44	0
G	3	52	{}	{}	≥ 52	2
H	3	38	{(S3,0,2) }	{}	40	5
I	3	35	{(S1,1,2)}	{}	48	2
J	4	52	{}	{}	60	2
K	2	70	{(S2,0,2)}	{}	80	10
L	3	59	{}	{}	60	12

The system has three shared resources but no precedence constraints. The complete task characteristics are listed in table 3.

For this system we found it to be very time consuming and complicated to manually derive the number of malignant mutants. Moreover, model-checking cannot be used for comparison since the reachable state-space becomes too large We run the genetic algorithm and searched each mutant for 200 generations or until a failure was encountered.

Each experiment was performed in five trials with different initial random seeds to guard against stochastic variance. For each simulation performed during the heuristic search, a random test was also conducted. This provides an indication of the relative efficiency between random search and genetic algorithms with our heuristic.

Table 4: Results from the Complex Dynamic Real Time System Experiment

Mutation operator	μ	K_R	K_{GG}	K_{GA}	$\overline{K_{GA}}$	$\overline{GEN}$
Execution time	24	0	0	12	9.2	62
Unlock time	16	0	0	0	-	-
Inter-arrival time	24	0	0	8	3.8	90
Offset time	22	0	0	0	-	-
Total	86	0	0	20	13.0	-

Table 4 uses the same column notation as table 2, but the additional columns "K_R" and "K_{GG}" include the results from random testing and genetic algorithms respectively.

As table 4 indicates, no malignant unlock time or offset time mutants were found for this particular system. The average number of generations required to kill a mutant was higher for this system specification model, which indicates that the search Problem is much more difficult than for the more static system presented previously.

7. Results from Experiments

In first experiment the low average number of mutants killed by genetic algorithm in the experiment suggests that the genetic algorithms may have to run longer on each mutant to achieve reliable results. A possible explanation for this is that the genetic algorithm has trouble finding comparable candidates without the iterative refinement from the heuristic operators. Hence, it would often prematurely discard partially refined candidates.

A possible remedy for real time problems is to redo the search multiple times using a fresh initial population. Since searching the mutant models is a fully automated process, the additional cost of multiple searches may be acceptable. If no new mutants are killed after a specified number of searches, the process can be halted.

The second experiment reveals that random testing is not effective to kill malignant mutants in real time complex systems. The results in the table 4 show that random testing method is not even found any malignant mutant in mutation operator while genetic algorithm detect malignant mutant in two mutation operator i.e Execution Time & Inter-arrival time.

Therefore genetic algorithms can remain effective for larger and more complex real system models where all approaches failed i.e the model checking approach fails because of the size of the reachable state space and Random testing approach is failed because of not setting of local optima for the mutation operator.

The main purpose of this paper is to investigate the applicability genetic algorithm and its relative effectiveness on a real-time target platform.

8. Conclusion

Timeliness is a property that is unique for real-time systems and deserves special consideration during both design and testing.

A problem when testing timeliness of real-time systems is that response times depend on the execution order of concurrent tasks.

Other existing testing methods ignore task interleaving and timing and, thus, do not help determine what kind of test cases are meaningful for testing timeliness.

The genetic algorithm suggested for generating test cases in this paper should be considered a prototype for investigating the feasibility of the test case generation method. The method can be extended in several ways to improve its performance, for example, by evaluating more sophisticated fitness and cross-over functions.

One such extension is preparing the initial population of a genetic algorithm with activation patterns suspected of causing long response times.

9. Future Work

A emerging trend in general computing is to increase performance by utilizing parallel processors and specialized computing elements (for example, chips containing field programmable gate arrays). Eventually, this trend will force developers to use parallel platforms for dependable real-time applications.

Unfortunately, the problems motivating a structured and generalizable approach for testing of timeliness are even more prevalent in architectures with several active processing units. In particular, the relation between activation patterns and execution orders is more complex and the problems associated with finding critical scenarios are elevated.

The framework presented in this paper has currently only been evaluated for single processor architectures and another framework may be investigated for multi processor systems architectures.

10. References

1. *Alur, R. & Dill, D. (1994), 'A theory of timed automata', Theoretical Computer Science 126, 183–235.*
2. *Ammann, P. & Black, P. (1999), A specification-based coverage metric to evaluate test sets, in 'HASE '99: Proceedings of the 4th IEEE International Symposium on High-Assurance Systems', Washington, DC, pp. 239–248.*
3. *Ammann, P., Black, P. & Majurski., W. (1998), Using model checking to generate tests from specifications, in 'Proceedings of the Second IEEE International Conference on Formal Engineering Methods', IEEE Computer Society, pp. 46–54.*
4. *Baker, T. P. (1991), 'Stack-based scheduling of real-time processes', The Journal of Real-Time Systems (3), 67–99.*
5. *Beizer, B. (1990), Software Testing Techniques, Von Nostrand Reinhold.*
6. *Cardell-Oliver, R. (2000), 'Conformance tests for real-time systems with timed automata specifications', Formal Aspects of Computing 12(5), 350–371.*
7. *En-Nouaary, R., F. Dssouli, K. & Elqortobi, A. (1998), Timed test case generation based on a state characterization technique, in 'Proceeding of the 19th IEEE Real-Time Systems Symposium (RTSS98)', Madrid, Spain.*
8. *Raymond, P., Nicollin, X., Halbwachs, N. & Weber, D. (1998), Automatic testing of reactive systems, in 'Proceeding of the 19th IEEE Real-Time Systems Symposium (RTSS98)'.*
9. *Stankovic, J. A., Spuri, M., Ramamritham, K. & Buttazzo, G. C. (1998), Deadline scheduling for real-time systems, Kluwer academic publishers.*
10. *Wegener, J., St Hammer, H. H., Jones, B. F. & Eyres, D. E. (1997), 'Testing real time systems using genetic algorithms', Software Quality Journal 6(2), 127–135.*