

# Mutation-based Testing Criteria for Timeliness

Robert Nilsson<sup>†</sup>

<sup>†</sup>Department of Computer Science  
University of Skövde, Box 408SE  
541 28 Skövde, Sweden  
{robert,sten}@ida.his.se

Jeff Offutt<sup>\*</sup>

<sup>\*</sup>Department of Information and Software Engineering  
George Mason University  
Fairfax, VA 22030 USA  
ofut@ise.gmu.edu

Sten F. Andler<sup>†</sup>

## Abstract

*Temporal correctness is crucial to the dependability of real-time systems. Few methods exist to test for temporal correctness and most existing methods are ad-hoc. A problem with testing real-time applications is the dependency on the execution time and execution order of individual tasks. Thus, the response times for the tasks may be non-deterministic with respect to inputs. Conventional test coverage criteria ignore task interleaving and timing and, thus do not help determine which execution orders need to be exercised to test for temporal correctness. This paper presents a test criteria based on mutation to test timeliness. We also show how previously proposed methods in specification based testing can be applied to testing real-time systems.*

## 1 Introduction

Real-time systems often operate in safety-critical environments and must be dependable. A current trend is to increase the services that real-time systems offer beyond their core functionality. This flexibility adds increased complexity and sometimes temporal non-determinism. Thus we need methods to detect failures based on temporal faults.

*Timeliness* is the ability for software to meet timing constraints. For example, a time constraint can be that it should never take more than 2 ms from when a car's brake pedal is pressed until the brake is activated.

Real-time systems are often concurrent, thus response times can depend on the order in which different tasks execute. This is particularly difficult in *event-triggered* systems because sporadic interrupts can continuously influence the execution order and schedule. Also, caches in the hardware can affect the execution times of tasks, causing them to be-

come non-deterministic.

Timeliness of embedded real-time systems is usually analyzed and maintained using scheduling analysis techniques or regulated online through admission control and contingency schemes. However, these techniques make assumptions about the tasks and load patterns that can affect the timeliness and therefore must be tested. Furthermore, doing full schedulability analysis with non-trivial system models is quite complicated. Thus timeliness must be tested.

Real-time theory commonly models software behavior by distinguishing between periodic and sporadic tasks that compete for system resources. *Periodic* tasks are released with fixed inter-arrival times, thus when the tasks will be released is known statically. *Sporadic* tasks can be released at any time, behavior that is harder to analyze. To ease analysis, sporadic tasks often specify a *minimum inter-arrival time* or some way to model their release pattern. Each real-time task typically has a *deadline*, which specifies how long the task has to finish. Tasks may also have an *offset*, which denotes the time before any task of that type can be released.

We surveyed existing testing methods for real-time software and concluded that few considered variations in the internal behavior of concurrent real-time systems [10]. Most methods use state-machine tests to verify that the output falls within an output interval that is valid according to the specification of the event-sequences [8].

However, there are dependencies between execution orders and response times of tasks, thus the specification notation must include structural information to test timeliness. The structural information needed is information about estimated execution times of tasks, scheduling policies, and concurrency control and resource dependencies that can influence the execution order of tasks.

A *test criterion* defines test requirements that must be satisfied when testing the software. Examples of test criteria include execute all statements containing the letter q and cover all transitions in a state machine. A *test coverage* measure expresses how thoroughly tests have satisfied a test criterion.

---

This work has been funded by the Swedish Foundation for Strategic Research (SSF) through the FLEXCON (and SAVE) programme(s)

Ammann and Black [2] presented specification-based testing criteria based on mutation testing for code [5]. In specification-based mutation testing, a set of mutant specifications are generated from an existing specification using *mutation operators* that change some detail of the model. A test requirement is to *kill* a mutant by distinguishing it from the original specification.

This paper presents a mutation test criterion to test timeliness of concurrent real-time systems. Mutation operators are defined in section 3 and test cases are created to kill all mutants generated by the operators. We also demonstrate the use of the mutation operators through a case study.

## 2 Timed Automata with Tasks

Timed Automata (TA) [1] have been used to model different aspects of real-time systems. Variations of the notation have been used to generate test cases that do not take execution orders and structural information into consideration [12].

An extension to timed automata, Timed Automata with Tasks (TAT) was presented by Nordström, Wall and Yi [11] and refined by Fersman [6]. TAT includes explicit operators to model scheduling and execution of real-time tasks, so it is suitable for this research.

A timed automaton (TA) is a finite state machine extended with a collection of real-valued clocks. Each transition can have a *guard*, an *action* and a number of *clock resets*. A guard is a condition on clocks and variables, such as a time constraint. An action can do things such as assign values to variables. The clocks increase uniformly from zero until they are individually reset in a transition. When a clock is reset, it is instantaneously set to zero and then starts to increase uniformly with the other clocks.

TAT extends the TA notation with a set of *real-time tasks*  $P$ .  $P$  represents programs that perform computations in response to a request. Formally, we use the definition (1) from Fersman [6].  $Act$  is a finite set of actions and  $\zeta$  is a finite set of real-valued variables for clocks.  $B(\zeta)$  denotes the set of conjunctive formulas of atomic constraints in the form  $x_i \sim C$  or  $x_i - x_j \sim D$  where  $x_i, x_j \in \zeta$  are clocks,  $\sim \in \{\leq, <, =, \geq, >\}$  and  $C$  &  $D$  are natural numbers.

**Definition 1** A timed automaton extended with tasks over actions  $Act$ , clocks  $\zeta$  and tasks  $P$  is a tuple  $\langle N, l_0, E, I, M \rangle$  where

- $\langle N, l_0, E, I \rangle$  is a timed automaton where
  - $N$  is a finite set of locations
  - $l_0 \in N$  is the initial location
  - $E \subseteq N \times B(\zeta) \times Act \times 2^\zeta \times N$
  - $I : N \mapsto B(\zeta)$  is a partial function assigning each location with a clock constraint
- $M : N \mapsto P$  is a partial function assigning locations to tasks in  $P$

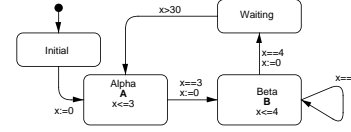


Figure 1. Timed Automata with Tasks

ID	$c$	$d$	$SEM$	$PREC$
A	4	12	$\{\}$	$\{\}$
B	2	8	$\{\}$	$\{\}$

Table 1. Task set for TAT in Figure 1

TAT states are triples of the current location of the automaton, values for all data variables and clocks, and a task queue. The task queue contains the remaining execution time and the time remaining until the deadline is reached for each task. The first task in the queue is assumed to be executing and its remaining execution time is decreased on each clock tick. A task is placed into the queue when a state is reached that is associated with the task. Preemptions occur when the first task in the queue is replaced before its remaining execution time is zero. If the remaining execution time reaches 0 for the first task it is removed and the next task starts to execute.

Precedence constraints are relations between pairs of tasks. Shared resources are modeled by a set of system-wide semaphores,  $R$ , where each semaphore  $s \in R$  can be locked and unlocked by tasks at fixed points in time,  $0 \leq t_1 < t_2 \leq c_i$ , relative to the task's start time. Elements in  $P$  express information about task types as quadruples  $(c, d, SEM, PREC)$ .  $c$  is the required execution time and  $d$  is the relative deadline. These values are used to create a new task instance as it is released by a TAT automaton action.  $SEM$  is a set of tuples of the form  $(s, t_1, t_2)$  where  $t_1$  and  $t_2$  are the lock and unlock times of semaphore  $s \in R$  when an instance of the task type is executed.  $PREC$  is a subset of  $P$  that specifies which tasks must precede a task of this type.

The precedence constraints and semaphore requirements are used to sort the task queue so that only tasks that have their execution conditions fulfilled can be put in the first position of the schedule [7]. Consider the TAT model in figure 1, which uses preemptive EDF scheduling and the task set in table 1.

In the example, all clocks start at zero, the task queue starts empty and the automaton is in the *Initial* location. Then two types of transitions may occur: (i) time may progress an arbitrary time period (delay transitions), increasing all clock variables, or (ii) a transition to location Alpha can be taken (an action transition). Two things happen on a transition to Alpha. First, Alpha is associated with task type A so an instance of task A is added to the task queue. Second, the transition to Alpha contains a reset, so

Transition	State < location;clocks;queue >
Delay: 0 → 10	< <i>Initial</i> ; $x = 10$ ; $q[]$ >
<i>Initial</i> → <i>Alpha</i>	< <i>Alpha</i> ; $x = 0$ ; $q[(4, 12)]$ >
Delay: 10 → 13	< <i>Alpha</i> ; $x = 3$ ; $q[(1, 9)]$ >
<i>Alpha</i> → <i>Beta</i>	< <i>Beta</i> ; $x = 0$ ; $q[(2, 8), (1, 9)]$ >
Delay: 13 → 15	< <i>Beta</i> ; $x = 2$ ; $q[(1, 7)]$ >
<i>Beta</i> → <i>Beta</i>	< <i>Beta</i> ; $x = 2$ ; $q[(1, 7), (2, 8)]$ >
Delay: 15 → 17	< <i>Beta</i> ; $x = 4$ ; $q[(1, 6)]$ >
<i>Beta</i> → <i>Waiting</i>	< <i>Waiting</i> ; $x = 0$ ; $q[(1, 6)]$ >

**Table 2. Trace of possible execution order**

clock  $x$  is set to zero. Location Alpha has an invariant of  $x \leq 3$  and a transition to Beta is taken when  $x$  is 3, so the TAT waits exactly 3 time units. However, table 1 defines an execution time of 4 time units for A, so it will not complete. When location Beta is reached, an instance of task B is added to the task queue. Tasks of type B have a shorter deadline than tasks of type A, so the task queue sorts on the earliest deadline and preempts the task of type A. The automaton stays in location Beta for 4 time units. After 2 time units, task B will finish its execution (before its deadline), and task A executes to completion. If the transition on  $x = 2$  is taken (allowed but not required), another task of type B will be released. This allows multiple task execution patterns. Table 2 gives an example trace.

### 3 Mutation-based Testing

The basis for mutation testing is the set of operators. Mutation operators usually model possible faults, so the first step in designing mutation operators is to understand the types of faults that can occur. We have identified two categories of timeliness faults.

The first category represents incorrect assumptions about the system behavior during schedulability analysis and design. This includes assumptions about execution times, use of shared resources, precedence relations, overhead times of context switches, and cache efficiency. If wrong assumptions are used then execution orders that were not analyzed may cause deadlines to be missed.

The second category is the system’s ability to cope with unanticipated discontinuities and changes in the environmental behavior, for example, disturbances in the sampling periods. Some faults in a system design may be hidden by assumptions about the operational environment. If the operational environment differs from the assumptions, an extensively tested system may fail during operation.

Mutation operators based on these two categories are described below. In six of the mutation operator types, a parameter is used to indicate constant differences in the time. This time delta can be tuned by the tester to yield mutants that are easier or harder to kill.

### 3.1 Task set mutation operators

Six mutation operators represent potential faults in real-time systems that may cause failures in the form of delayed responses. For each operator, the points in time when a resource is taken and released are between zero and the specified execution time of the task. If the mutation operator causes the lock or unlock times of a resource to fall outside of this interval, the time is adjusted to the appropriate endpoint of the interval.

**Execution time mutation operators:** Execution time operators increase or decrease the execution time of a task by a constant time delta. 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) possible execution time.

Estimations of worst-case execution times is very hard. In particular, the execution time when running a task concurrently with other tasks may be longer than running the task uninterrupted. The  $\Delta$ - execution time operator is relevant when multiple active tasks share resources in the system. A shorter than expected best-case execution time of a task may lead to a scenario where a lower priority task gets a resource and blocks a higher priority task so that it misses its deadline.

**(1)  $\Delta+$  execution time:** Given a TAT model with task set  $P$ , for every task  $(c_i, d_i, SEM_i, PREC_i) \in P$ , create one mutant that changes the execution time  $c_i$  to  $c_i + \Delta$ .

**(2)  $\Delta-$  execution time :** Given a TAT model with task set  $P$ , for some task  $(c_i, d_i, SEM_i, PREC_i) \in P$ , create one mutant that changes the execution time  $c_i$  to  $c_i - \Delta$ .

**Hold time shift mutation operators:** The delta +/- hold time shift mutation operator changes the interval of time a resource is locked. For example, if a semaphore is to be locked at time 2 and held until time 4 in the original model, a  $\Delta+$  hold time shift mutant (with  $\Delta = 1$ ) would cause the resource to be locked from time 3 until time 5.

Execution times differ and external factors vary in how they disturb the execution times, making it hard to accurately predict when a resource will be acquired and released relative the start of the task. Execution time before a resource is allocated may also take more or less time than expected. This introduces new chains of possible blocking.

**(3)  $\Delta+$  hold time shift :** Given a TAT model with task set  $P$ , for every task  $(c_i, d_i, SEM_i, PREC_i) \in P$  and every semaphore use  $(s, t_1, t_2) \in SEM_i$ , create one mutant that changes  $t_1$  to  $\min(t_1 + \Delta, c_i)$  and  $t_2$  to  $\min(t_2 + \Delta, c_i)$ .

**(4)  $\Delta-$  hold time shift :** Given a TAT model with task set  $P$ , for every task  $(c_i, d_i, SEM_i, PREC_i) \in P$  and every semaphore use  $(s, t_1, t_2) \in SEM_i$ , create one mutant that changes  $t_1$  to  $\max(0, t_1 - \Delta)$  and  $t_2$  to  $\max(0, t_2 - \Delta)$ .

**Lock time mutation operators:** The delta +/- lock time mutation operator increases or decreases the time when a particular resource is locked. For instance, if a semaphore is to be locked from time 2 until 4 in the original model, a  $\Delta+$  lock time mutant (with  $\Delta = 1$ ) causes the resource to be locked from time 3 to 4.

An increase in the time a resource is locked increases the maximum blocking time for a higher priority task. This mutation operator requires test cases that can distinguish an implementation where a resource is held too long from one where it is not. Furthermore, if a resource is held for less time than expected, the system may get different execution orders that will result in timeliness violations.

(5)  $\Delta+$  **lock time:** Given a TAT model with task set  $P$ , for every task  $(c_i, d_i, SEM_i, PREC_i) \in P$  and every semaphore use  $(s, t_1, t_2) \in SEM_i$ , create one mutant that changes  $t_1$  to  $\min(t_1 + \Delta, t_2)$ .

(6)  $\Delta-$  **lock time:** Given a TAT model with task set  $P$ , for every task  $(c_i, d_i, SEM_i, PREC_i) \in P$  and every semaphore use  $(s, t_1, t_2) \in SEM_i$ , create one mutant that changes  $t_1$  to  $\max(0, t_1 - \Delta)$ .

**Unlock time mutation operators:** Unlock time mutation operators change when a resource is unlocked. For example, a  $\Delta+$  unlock time mutant (with  $\Delta = 1$ ) causes the resource to be held from time 2 to 5.

(7)  $\Delta+$  **unlock time:** Given a TAT model with task set  $P$ , for every task  $(c_i, d_i, SEM_i, PREC_i) \in P$  and every semaphore use  $(s, t_1, t_2) \in SEM_i$ , create one mutant that changes  $t_2$  to  $\min(t_2 + \Delta, c_i)$ .

(8)  $\Delta-$  **unlock time:** Given a TAT model with task set  $P$ , for every task  $(c_i, d_i, SEM_i, PREC_i) \in P$  and every semaphore use  $(s, t_1, t_2) \in SEM_i$ , create one mutant that changes  $t_2$  to  $\max(t_1, t_2 - \Delta)$ .

**Precedence constraint mutation operators:** Precedence constraint mutation operators add or remove precedence constraint relations between pairs of tasks. This represents situations where a precedence relation exists in the implementation that is not modeled during analysis of the system, or a precedence constraint was not implemented correctly. An additional or missed precedence constraint may cause two tasks to be executed in the wrong order, causing a task to break its deadline. These faults are not necessarily found by other types of testing, since the program's logical behavior may still be correct.

The TAT-model represents precedence constraints as a relation between pairs of tasks. The mutation operators adds a precedence relation between pairs of tasks in the task set. If there already is a precedence constraint between the pair, the relation is removed.

(9) - **precedence constraint:** Given a TAT model with task set  $P$  and task  $p_i \in P$ , for each task  $p_j \in P$ , if  $p_j \in PREC_i$ , create a mutant by removing  $p_j$  from  $PREC_i$ .

(10) + **precedence constraint:** Given a TAT model with

task set  $P$  and task  $p_i \in P$ , for each task  $p_j \in P$ , if  $p_j \notin PREC_i$ , create a mutant by adding  $p_j$  to  $PREC_i$ .

### 3.2 Automata mutation operators

Three mutation operators test the consequences of faults in the load hypothesis or model of the system environment.

**Inter-arrival time mutant operator:** This operator decreases the inter-arrival time between requests for a task execution by a constant time  $\Delta$ . This reflects a change in the system's environment that causes requests to come more frequently than expected.

This operator is important when the temporal behavior of the environment is unpredictable and cannot be completely known during design. The resulting test cases will stress the system to reveal its sensitivity to higher frequencies of requests.

(11)  $\Delta-$  **inter-arrival time:** Given a TAT automaton  $\langle N, l_0, E, I, M \rangle$  with clock set  $\zeta$  and task set  $P$ , for every location  $l_i \in M$  with clock constraint  $(x < C) \in I(l_i)$  and outgoing transitions from  $l_i$  with guards  $(x \geq C) \in B(\zeta)$ , create a mutant that changes the natural number constant  $C$  to  $C - \Delta$  in both the clock constraint and guards on all outgoing transitions from  $l_i$ .

**Pattern offset mutation operators:** Recurring environment requests can have default request patterns that have offsets to each other. This operator changes the offset between two such patterns by a constant  $\Delta$  time units. This operator is relevant when the system assumes that two recurring event patterns have a relative offset so that their associated tasks cannot disturb each other.

This mutation operator assumes that each task (or set of tightly coupled tasks) is controlled by a separate, parallel decomposed TAT. Mutants are created from the sub-automata processes in isolation and are then composed with the rest of the model to form the TAT specifying the whole system. The parallel decomposition works in the same way as in ordinary timed automata [9].

(12)  $\Delta+$  **pattern offset:** Given a TAT automaton process  $\langle N, l_0, E, I, M \rangle$  with clock set  $\zeta$ , task set  $P$  and initial location  $l_0$  with clock constraint  $(x < C) \in I(l_0)$  and outgoing transitions leading from  $l_0$  with guards  $(x \geq C) \in B(\zeta)$ , create one mutant that changes the natural number constant  $C$  to  $C + \Delta$  in both the clock constraint and guards on all outgoing transitions from  $l_0$ .

(13)  $\Delta-$  **pattern offset:** Given a TAT automaton process  $\langle N, l_0, E, I, M \rangle$  with clock set  $\zeta$ , task set  $P$  and initial location  $l_0$  with clock constraint  $(x < C) \in I(l_0)$  and outgoing transitions leading from  $l_0$  with guards  $(x \geq C) \in B(\zeta)$ , create one mutant that changes the natural number constant  $C$  to  $C - \Delta$  in both the clock constraint and guards on all outgoing transitions from  $l_0$ .

## 4 Mutation-based Test Case Generation

To validate our mutation operators we applied them to a TAT model in a case study. The system models were produced with the Times tool, developed by the DARTS group at Uppsala University to perform formal verification and schedulability analysis of real-time systems [4]. The tool provides a model checker that can decide if all deadlines are met in a TAT model. We exploit this feature to find traces that lead to missed deadlines in our mutated TAT models; the traces are then converted to test cases that can detect faulty implementations. This approach is similar to that of Ammann, Black and Majurski [3], but adapted to timeliness testing of real-time systems.

The input to the approach is a TAT model of a real-time system and a desired test criterion. The test criterion specifies the mutation operators to use. A mutant generator applies the operators and sends the mutated models to the model checker. If analysis reveals non-schedulability in a mutated model, it is marked as killed, otherwise it is considered to be benign and discarded. Counter example traces from the killed mutants are then fed into a test case generator that converts the traces to executable test cases and runs them (more details are given by Nilsson, Andler and Mellin [10]). The following case study concentrates on the first steps of the process, using the counter example traces to determine the applicability of the mutation operators. If a mutation operator results in at least one killed mutant specification, the corresponding fault type can cause timeliness failures and is therefore considered meaningful.

### 4.1 Case study setup

This case study uses a small task set that has a simple environment model 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 is allowed.

The analysis assumed the *immediate ceiling priority* protocol to avoid priority inversion. 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 FCFS policy is used if several tasks have the same priority.

The model has five tasks. Two tasks are controlled by TAT models such as the one in figure 2. The automata are sporadic in nature but have a fixed observation grid (denoted OG in the figure), which limits when a new task instance can be released. These experiments use an observation grid of 10 time units. The three remaining tasks are strictly periodic and are controlled with generic automata, with defined periods and offsets.

The system has two shared resources modeled by

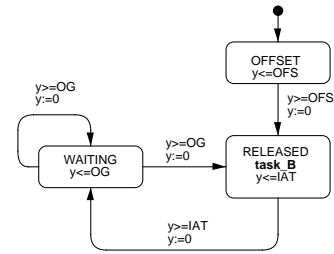


Figure 2. TAT for case study task

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

Table 3. Characteristics of case study task set

semaphores, and one precedence relation between tasks C and A, which specifies that a new instance of task A cannot start unless task C has executed after the last instance of A.

Table 3 describes the task set. The first column ('ID') gives task identifiers, column 'c' gives execution times, column 'd' gives relative deadlines, column 'IAT' gives inter-arrival times. Periodic tasks are released with fixed inter-arrival times and sporadic task's inter-arrival times are defined in figure 2. Column 'OFS' describes the initial offsets, or the delay before the first task may be released, column 'SEM' specifies the set of semaphores used and which interval they are required in, and column 'PREC' specifies which tasks have precedence over tasks of this type.

Mutant specifications are automatically generated according to the mutation operators with two different values for  $\Delta$ . Table 3 divides the results into two groups, based on the  $\Delta$  values. The number of mutants for each type is in column M, and the number of killed mutants of that type is in column K.

The number of killed mutants corresponds to the number of counter example traces that were converted to test cases for the actual system.

As seen in Table 4, the model checker killed at least one mutant generated by each mutation operator. This means that the associated fault types associated with each class of proposed mutation operators potentially can lead to a non-schedulable system. Hence, to reveal timeliness faults, testing can focus on showing that these traces can occur in the system, when subject to the event-sequence provided by the model-checker.

Mutation operator	small $\Delta$			large $\Delta$		
	$\Delta$	M	K	$\Delta$	M	K
Execution time	1	10	6	2	10	5
Hold time shift	1	14	1	2	14	0
Lock time	1	8	1	2	8	1
Unlock time	1	11	2	2	11	2
Precedence constraint	-	20	15	-	-	-
Inter-arrival time	1	5	4	4	5	4
Pattern offset	1	10	5	4	10	4
<b>Total</b>	-	78	34	-	58	16

**Table 4. Mutation and model-checking results**

## 5 Conclusions

This paper has presented a new technique to test for timing problems in real-time software. The technique is based on seven kinds of formalized mutation operators.

Timeliness properties of the software are formally modeled through timed automata with tasks. The mutation operators modify the automata to create mutant versions, which are then analyzed with a model checker that attempts to produce execution traces in which timing problems occur. These traces are then converted into test cases that can be run against the software implementation to see if timing problems can be observed in the implementation. This paper has not considered the issues relating to converting the execution order traces to full test cases and execution of tests on the target platform. However, the traces contain enough information to generate event sequences, prefixes and expected outcomes according to the outline in previous work [10].

The mutation operators are general enough to work with many different scheduling algorithms and execution environment configurations. Most of the mutation operators can be explicitly tuned by increasing or decreasing the value of  $\Delta$ . For many of the operators, the value of  $\Delta$  is proportional to how easy it is to kill the mutant. We believe suitable values depend on the tightness of the schedule, but no thorough analysis has been made. Future work includes establishing theory or methodology for choosing these values.

A case study has been carried out to demonstrate that the faults represented by the mutation operators can lead to missed deadlines. These results allow testers to develop tests specifically focused to test for temporal correctness in concurrent real-time systems.

An issue with this approach is that the execution times of unmutated tasks in the original model are assumed to be fixed for the worst case. This is not always realistic. It is, for example, very unlikely that all tasks will exhibit their worst-case execution times simultaneously. In some systems it may be better to use the average execution time in the model and let the mutation operators do larger variations.

Furthermore, the goal of mutation-based testing is primarily to select effective test cases and even if a mutant is unrealistic, the symptoms can be similar for other faults. For example, an unanticipated execution of an interrupt function could be detected by the same test case that would find optimistic estimations of execution time for the interrupted task.

Another issue is that model checking is expensive, and the execution time may increase rapidly with more complex system models and task loads. We hope to optimize analysis of the essential properties in future work.

## References

- [1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
- [2] P. Ammann and P. Black. A specification-based coverage metric to evaluate test sets. In *HASE '99: Proceedings of the 4th IEEE International Symposium on High-Assurance Systems*, pages 239–248, Washington, DC, November 1999.
- [3] 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 (ICFEM'98)*, pages 46–54. IEEE Computer Society, December 1998.
- [4] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times - a tool for modelling and implementation of embedded systems. In *Proceedings of TACAS'02*, number 2280, pages 460–464. Springer-Verlag, 2002.
- [5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.
- [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] E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: Schedulability and decidability. In J.-P. Katoen and P. Stevens, editors, *Proc. of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2280 in Lecture Notes in Computer Science, pages 67–82. Springer-Verlag, 2002.
- [8] S. Fujiwara, G. V. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on software engineering*, 17(6):591–603, June 1991.
- [9] K. G. Larsen, P. Pettersson, and W. Yi. Compositional and symbolic model-checking of real-time systems. In *IEEE Real-time Systems Symposium*, pages 76–89, 1995.
- [10] R. Nilsson, S. Andler, and J. Mellin. Towards a Framework for Automated Testing of Transaction-Based Real-Time Systems. In *Proceedings of Eighth International Conference on Real-Time Computing Systems and Applications (RTCSA2000)*, pages 109–113, Tokyo, Japan, March 2002.
- [11] C. Nordström, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Proceedings of RTCSA'99*, Hong Kong, December 1999.
- [12] E. Petitjean and H. Fochal. A realistic architecture for timed testing. In *Proc. of Fifth IEEE International Conference on Engineering of Complex Computer Systems*, USA, Las Vegas, October 1999.