Test Case Generation for Mutation-based Testing of Timeliness

Robert Nilsson [a,1,2], Jeff Offutt [b,3] and Jonas Mellin [a,1,4]

[a] *Distributed Real-time Systems Group*
School of Humanities and Informatics
University of Skövde
Skövde, Sweden

[b] *Information and Software Engineering*
George Mason University
Fairfax Virginia, USA

Abstract

Temporal correctness is crucial for real-time systems. Few methods exist to test temporal correctness and most methods used in practice are ad-hoc. A problem with testing real-time applications is the response-time dependency on the execution order of concurrent tasks. Execution order in turn depends on execution environment properties such as scheduling protocols, use of mutual exclusive resources as well as the point in time when stimuli is injected. Model based mutation testing has previously been proposed to determine the execution orders that need to be verified to increase confidence in timeliness. An effective way to automatically generate such test cases for dynamic real-time systems is still needed. This paper presents a method using heuristic-driven simulation to generate test cases.

Key words: Real-time Systems, Mutation Testing, Model based

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 few, standardized hardware components. This can increase system complexity and introduce sources of temporal non-determinism (for example, caches

[1] This work has been funded by the Swedish Foundation for Strategic Research (SSF) through the FLEXCON programme.
[2] Email: `robert.nilsson@his.se`
[3] Email: `offutt@ise.gmu.edu`
[4] Email: `jonas.mellin@his.se`

and pipelines) that make it hard to predict the execution behavior of tasks [26]. Faults in such predictions may result in software timeliness violations and costly accidents. Thus we need methods 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 [28].

When designing real-time systems, software behavior is 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. *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*. Tasks may also have an *offset*, which denotes the time before a task of that type is activated.

Testing methods must be adapted to address timeliness because it is difficult to characterize a critical sequence of inputs without considering the effect on the set of active tasks and real-time protocols. However, existing testing techniques seldom use information about real-time design in test case generation, nor do they predict what execution orders may reveal faults in off-line assumptions (see section 5 for an overview of related work).

In the real-time community, timeliness is traditionally analyzed and maintained using scheduling analysis techniques or regulated online through admission control and contingency schemes [34]. However, these techniques use assumptions about the tasks and activation patterns that must be correct for timeliness to be maintained. Further, doing full 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 gain confidence in assumptions by systematically sampling among the execution orders that can lead to missed deadlines. However, only some of the possible execution orders typically reveal timeliness violations in the presence of timing faults.

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 the method is to systematically "guess" what faults a system contains and then evaluate what the effect of such faults could be in a model of the system. Once faults with bad consequences are identified, test cases are constructed that try to reveal those faults in the system implementation.

Model-checking has previously been used to analyze models of real-time systems for generating test cases for testing of timeliness [22]. A problem in this context is that analysis of the dynamic real-time systems models often becomes so computationally complex that the previously presented model-checking approach does not work. In particular, this happens in models of event-triggered systems where the timing of different sporadic interrupts can influence the execution order of tasks [31].

This paper investigates whether application-specific heuristics and simulation can be used as an alternative for analyzing such models.

Consequently, this paper proposes a method where a mutated specification model that captures possible execution behaviors is mapped to a simulator. The simulator is then iteratively executed using a genetic algorithm to find input sequences that reveal the potential failures in the mutated model. The method is demonstrated in two experiments. The first experiment compares the method with the model-checking based approach to gain basic confidence in its reliability. The method is also evaluated using a larger, more dynamic system specification for which the model-checking based approach fails. The experiments indicate that the simulation-based method remain effective for the dynamic specification model and that the heuristic functions presented enhance the performance.

The inputs to mutation-based testing of timeliness is a specification of a real-time system and a testing criterion. The testing criterion specifies what mutation operators to use, and thus, determines the level of thoroughness of testing and what kind of test cases will be produced. A mutant generator applies the mutation operators to the specification and sends the mutated specifications to an execution order analyzer that determines if and how the mutation can lead to a timeliness failure. We call a mutated specification model that contains a fault that can lead to a timeliness failure a *malignant mutant*. If analysis reveals a timeliness violation in a mutated model, the mutant is marked as *killed*. Traces from the killed mutants are fed into a test case generation filter that extract an activation pattern that has the ability to detect faults similar to the malignant mutant in the actual system under test. It is also possible to automatically extract the execution orders of tasks that can lead to a deadline violation when the input stimuli is injected. During test case execution, test inputs are injected in the real-time system according to the activation pattern.

Problems associated with controllability and observability when testing flexible real-time systems are out of scope of this paper. Prefix-based and non-deterministic test execution techniques [15,33,21] are complementary to our approach.

2 System Model and Testing Criteria

This paper uses a subset of Timed Automata with Tasks (*TAT*) [24,11] to define the assumptions about the system under test and as a source for model based test case generation. Timed Automata (*TA*) [1] have been used to model many different aspects of real-time systems. A 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 calculations and 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 at the same rate as the other clocks (we assume synchronized clocks).

Within TAT models, TA is used to specify the *activation pattern* of tasks, that is, the order and points in time different task executions is requested. Further, TAT extends the TA notation with a set of *real-time tasks* P, which need to be scheduled to perform computations in response to an activation. Elements in P express information about tasks as quadruples (c, d, SEM, $PREC$), where c is the assumed execution time of the task, d is the relative deadline, and *SEM* and *PREC* are defined in the following paragraphs. 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 time points in their execution. The set SEM contain tuples of the form (s, t_1, t_2) where t_1 and t_2 are the lock and unlock times of semaphore $s \in R$. These times are expressed relative the task's start time. *Precedence constraints* are relations between pairs of tasks A and B stating that an instance of a task A must have executed to completion between the execution of two consecutive instances of task B (otherwise, the second instance of task B is blocked). Hence, $PREC$ is a subset of P that specifies what other tasks must precede this task.

We call a task's behavior, including the points in its execution where different resources are locked and unlocked, the tasks' *execution pattern*. In TAT, task execution patterns are fixed. This may appear unrealistic, especially if the input data to a task may vary. In this step we assume that the execution pattern for a task is associated with a particular (typical or worst case) equivalence class of input data. After a critical activation pattern is found, the target system can be tested several times using different task inputs in that sequence, stressing it to reveal faulty behavior.

2.1 Mutation Operators

A *test criterion* defines test requirements that must be satisfied when testing software. An example of a test criterion is "execute all statements once". A *test coverage* measure expresses how thoroughly tests have satisfied a test criterion, usually in terms of how many test requirements are satisfied. A mutation-based test criterion is defined by a set of mutation operators.

Hence, progress of testing can be expressed in terms of mutants killed during test case generation. For example, if a set of test cases derived from killing all malignant "($\Delta = 3$) execution time mutants" has been run on the target system, then 100 percent coverage has been reached for that testing criterion. Mutation operators mimic possible faults that can lead to timeliness failures. Our previous work identified and presented formal definitions of seven types of faults or deviations from assumptions that can lead to timeliness failures [22], whereas this paper describe the operators informally and classifies them with respect to the maximum number of mutants created.

2.1.1 Task property mutations $O(n)$

The following operators create $2n$ mutants, where n is the number of tasks. *Execution time operators* increase the modelled worst case execution time of a task by a constant time Δ or decrease the best case execution time with the same amount. This mutation represents a situation where the assumption of a task's execution times, used for analysis, does not correspond with the execution times that is possible in the implementation. *Minimum inter-arrival time operators* decrease or increase the assumed inter-arrival time between requests for task execution by a constant time Δ. This reflects a change in the system's environment that causes requests to come more or less frequently than expected. Such recurring environment requests can also be assumed to have fixed offsets to each other. *Pattern offset operators* change the offset between two activation patterns by a constant Δ time units.

2.1.2 Resource locking mutations $O(nrl)$

These mutation operators increase or decrease the time when a particular resource is locked by Δ time units. The *lock time operator* changes the point in time resources are locked and the *unlock time operator* changes the time resources are unlocked relative the start time of the task. The *hold time shift operator* changes both the lock and unlock times. Since mutants are created for each pair of tasks and resource protected critical sections, the maximum number of mutants is $2n * r * l$, where r is the number of resources and l is the maximum number of times a resource is needed by a particular task throughout its execution.

2.1.3 Precedence mutations $O(n^2)$

For each pair of tasks, if a precedence constraint exists between the pair, then it is removed. If there is no precedence constraint, a new constraint is added. A task cannot be constrained to precede itself, so the number of mutants that can be created is $n^2 - n$.

3 Automated Test Generation using Genetic Algorithms

The previously presented method based on model-checking [22] is safe for analyzing mutated TAT models in the sense that vulnerabilities are guaranteed to be revealed if they exist. However, for some systems the state space becomes too large for model-checking to be effective. In particular, the computational complexity (both time and memory) grows when triggering events are allowed to occur at many different points in time.

In dynamic real-time systems, there are many sporadic tasks, making model-checking impractical. For these systems, we propose an approach where a simulation of each mutant model is iteratively run and evaluated using genetic algorithms with application specific heuristics. By using a simulation-based method instead of model-checking for execution order analysis, the combinatorial explosion of full state exploration is avoided. Further, we conjecture that it is easier to modify a system simulation than a model-checker, to correspond to the architecture of the system under test.

When simulation is used for mutation analysis, the model task set must be mapped to task entities in a real-time simulator. The activation pattern of periodic tasks is known and can be included in the static configuration of the simulator. The activation pattern for sporadic tasks should be varied for each iteration of simulation to find the execution orders that can lead to timeliness failures. Consequently, a necessary input to the simulation of a particular TAT model is an activation pattern for the sporadic tasks. The relevant output from the simulation is an execution order trace where the sporadic requests have been injected according to the activation pattern. A desirable output from a testing perspective is an execution order trace that leads to a timeliness failure in the mutant.

By treating test case generation from the TAT model as a optimization problem, different heuristic methods can be applied to find a trace leading to a missed deadline. This paper focuses on genetic algorithms, since they are highly configurable and cope well with optimization problems that contain local optima [18].

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 this context the solutions are called *individuals* and the set of individuals is called the *population*. Each individual has a *genome* that represents its unique features in a standardized format. Common formats for genomes are bit-strings and arrays of real values. Consequently, users of a genetic algorithm must supply a problem specific mapping function from a genome in any of the standard formats to a particular candidate solution for the problem. It has been argued that the mapping is important for the success of the genetic algorithm. For example, it is desirable that all possible genomes represent a valid solution [18].

The role of the fitness function in genetic algorithms is to evaluate the optimality or *fitness* of a particular individual. The individuals with the highest fitness in a population have a higher probability of being selected as input to cross-over functions and of being copied to the next generation.

Cross-over functions are applied on the selected individuals to create new individuals with higher fitnesses in the next generation. This means either combining properties from several individuals, or modifying a single individual

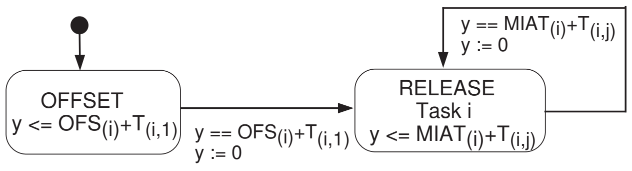

Fig. 1. Annotated TAT template

according to heuristics. 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 functions that use information from individuals to create genomes for the next generation.

There are generic cross-over functions that operate on arbitrary genomes expressed in the standard formats. For example, a cross-over function can exchange two sub-strings in a binary string genome or increase some random real value in an array. However, depending on the encoding of genomes, the standard cross-over functions may be more or less successful in enhancing individuals. Using knowledge of the problem domain and the mapping function it is possible to customize cross-over functions in a way that increases the probability of creating individuals with high fitness. On the other hand, some cross-over functions must remain stochastic to prevent the search from getting stuck in local optima. A genetic algorithm search typically continues for a predetermined number of generations, or until an individual with a fitness value over some set bound has been found.

To summarize, 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 test case generation from a dynamic real-time system model.

3.1 Genome Mapping Function

For the test case generation problem the only thing that varies between simulations of the same mutant TAT model is the activation pattern of non-periodic tasks, thus, it is sufficient that a genome can be mapped to such an activation pattern. Each activation pattern deterministically results in a particular execution order trace in the simulation. The execution order traces are the individuals for this search problem.

Figure 1 contains an annotated TAT-automata for describing activation patterns of sporadic tasks. In the general case, activation patterns can be expressed by any timed automata, but this paper focuses on sporadic task templates, since such tasks are common in real-time system models. The template has two parameters that are constant for each mutant. The parameter

[*] The mutations used for mutation testing operate directly on the structure of the model instead of on the timed sequence of inputs driving the traversal of the model.

OFS supplies the assumed offset, that is, the minimum delay before any instance of this task is assumed to be requested. MIAT supplies the assumed minimum inter-arrival time between instances of the sporadic task.

An array of real values $T_{(i,1..m)}$ defines the duration of the *variable delay interval* between consecutive requests of a sporadic task i. Here m is the maximum number of activations that can occur during the simulation.

By combining the arrays for n sporadic tasks in the mutant task set P we get a matrix $T_{(1..n,1..m)}$ of real values, where each row corresponds to an activation pattern of a sporadic task. The matrix T can be used as a genome representation of all valid activation patterns for the mutant.

3.2 Heuristic Cross-over Functions

For testing of timeliness, there are intuitive heuristics of what kind of activation patterns are likely to stress a mutant. For example, it seems possible that releasing many different types of sporadic requests in a burst-like fashion is more likely to reveal timeliness violations than an even distribution of activations.

Several concepts need to be introduced to simplify our definitions of heuristic cross-over functions. We use *critical task instance* to denote the task instance with the least slack in an execution order trace. A *critical interval* ($[ci_{beg}, ci_{end}]$) is the interval between the activation time and response time of a critical task instance. An *idle point* is a point in time where no real-time task executes or is queued for immediate execution on the processor. The *loading interval* ($[li_{beg}, li_{end}]$) is the interval between the latest idle point and the activation time of the critical task instance. The variable M is a TAT model that contains n sporadic tasks controlled by automata templates such as in figure 1. A genome matrix of size $n * m$ is denoted by the variable T. The integer variable i is used to index over the rows in a genome matrix T. The rows in such matrices correspond to sporadic tasks, hence it is bounded by 1 and n. The integer variable j is used to index over the columns in genome matrices and is bounded by 1 and m. The variable ϵ is used to denote a small positive real number. The expression $[a_{beg}, a_{end}] \sqsubseteq [b_{beg}, b_{end}]$ means that the left hand interval $[a_{beg}, a_{end}]$ is a sub-interval of the right hand interval $[b_{beg}, b_{end}]$. Formally, this can be expressed $([a_{beg}, a_{end}] \sqsubseteq [b_{beg}, b_{end}]) \iff (a_{beg} \geq b_{beg} \wedge a_{end} \leq b_{end})$. Further, a delay interval matrix D (see definition 3.1), derived from T and M, is used to define the the cross-over functions in the following subsections.

Definition 3.1 : Delay interval matrix D
A matrix of size $n * m$ that contains the variable delay intervals for each sporadic task activation in T such that: $D_{(i,j)} = [epat(i,j), epat(i,j) + T_{(i,j)}]$, where $epat(i,j)$ is the earliest possible arrival time of the j'th instance of sporadic task i given a TAT model M and a genome matrix T.

3.2.1 Focus critical interval

This cross-over function analyzes the logs from the simulation to find the critical interval. A sporadic task is chosen arbitrarily and changed to increase the probability that it executes within the critical interval.

Definition 3.2 : Focus critical interval left
For an arbitrary index i, let j be the largest index such that $D_{(i,j)} \sqsubseteq [0, ci_{beg}]$ then increase $T_{(i,j)}$ with ϵ time units and decrease $T_{(i,j+1)}$ with ϵ time units.

Definition 3.3 : Focus critical interval right
For an arbitrary index i, let j be each index such that $D_{(i,j)} \sqsubseteq [ci_{beg}, ci_{end}]$, and modify T so that $T_{(i,j)} = 0$.

3.2.2 Critical interval move

All sporadic tasks activation patterns are shifted a small random period so that the sequence of sporadic requests leading up to a critical interval occurs at some other point relative the static arrival pattern of periodic tasks.

Definition 3.4 : Critical interval move
For every index i such that $D_{(i,1)} \sqsubseteq [0, ci_{beg}]$, increase or decrease $T_{(i,1)}$ with ϵ time units.

3.2.3 New interval focus

This cross-over function generates new candidate critical intervals to keep the optimization from getting stuck in local optima. A new point in time is chosen by random and all the closest sporadic activations are shifted toward the selected point in time.

Definition 3.5 New interval focus
Let t_{new} be an arbitrary instant within the simulation interval. For every index i, let j be the largest index such that $D_{(i,j)} \sqsubseteq [0, t_{new}]$, and increase $T_{(i,j)}$ with ϵ time units. Also decrease $T_{(i,j+1)}$ with ϵ time units.

3.2.4 Loading interval perturbation

Theoretically, all task activations in the loading interval may influence timeliness through the state in the system when the activation of a critical task instance occurs. Changes in the end of the loading interval has a direct effect on timeliness of the critical task instance. This cross-over function changes the activation pattern in the end of the loading interval.

Definition 3.6 : Loading interval perturbation
For any index i, let j be the largest index such that $D_{(i,j)} \sqsubseteq [li_{beg}, li_{end}]$ and $j > 1$, modify T so that $T_{(i,j-1)} = \epsilon$ time units.

3.3 Fitness Function

Since our genome representation of valid activation patterns is meaningless without a particular TAT-mutant model we need to run the simulation with the activation pattern matrix before we can attain any fitness or apply cross-over functions. Once we have run the simulation we can use the execution order trace to determine how optimal a particular individual is. A suitable fitness function for timeliness should measure how close a system is to breaking a deadline. The *slack* is the time between the response-time of a task instance and its deadline. Hence, the minimum slack observed during a simulation is used to determine fitness. The highest fitness is given to the activation pattern that results in the simulated execution order with the least minimum slack. More elaborate fitness functions (for example, using weighting based on diversity or average response times) will be evaluated for improving the performance of the genetic algorithm in future work.

4 Test Case Generation Experiments

To evaluate the proposed method, we performed two separate experiments. The first attempted to establish basic confidence in the method by applying it on a small system model where we also could use a model-checker to generate tests. This allows the method to be compared in a baseline experiment and detect if the genetic algorithm method has problems finding any specific types of mutants. The second experiment used a larger system model to evaluate how simulation-based test case generation handles task sets with a large fraction of sporadic tasks under dynamic real-time system protocols.

To perform the experiments, we extended the real-time and control co-simulation tool *TrueTime* to simulate the execution of TAT models. TrueTime was developed at the department of automatic control at the University of Lund to support integrated design of controllers and real-time schedulers [12]. We also configured and extended a genetic algorithm tool-box [14] to interact with our simulation model. For model-checking experiments we used the *Times* tool, developed at Uppsala University [3].

4.1 Baseline Real-time System Experiment

This experiment used a small task set with few sporadic tasks but with a lot of possible interactions. 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. The system used the *immediate ceiling priority* protocol to avoid priority inversion [32]. 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.

Table 1 summarize the assumptions of the task set. The first column

Table 1
Task set of baseline real-time system

ID	TAT quadruple	I	O
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

("ID") gives task identifiers, the second column gives the TAT task quadruple as described in section 2. For sporadic tasks, the "I" column contains the minimum inter-arrival time assumptions (marked "MIAT" in the timed automata template in figure 1). For periodic tasks the "I" column contains the fixed inter-arrival time. Column "O" denotes the initial offset constant.

Table 2 contains the results from test case generation. Column "μ" contain the number of mutants generated for each mutation operator. The number of malignant mutants is listed in column "M" and the number of mutants killed by model-checking is listed in column "C." A Δ value of 1 time unit was used to generate the mutants. For the genetic algorithm setup, we used a population of 20 individuals per generation and ran each mutant 100 generations before terminating (more parameters of the genetic algorithm are available in the extended version [23]). We used the heuristic cross-over functions described in section 3.2 as well as three generic cross-over functions that *(i)* changed a random value in the genome representation, *(ii)* created a new random individual in the population and *(iii)* replaced a random value in the genome with 0. Each experiment was re-run eight times, using different random seeds and random initial populations. The number of mutants that was killed using genetic algorithms in any of the trials is listed in column "K". Column "A" lists the average number of malignant mutants killed per experiment, the variance is given within parentheses. The average number of generations needed to kill malignant mutants of this type is in column "G."

As seen in table 2, both the model-checking ("C") and simulation-based ("K") approaches killed all the malignant mutants. The model-checking approach not only killed all malignant mutants, it also killed some benign mutants. By comparing execution orders of benign mutants that were killed, we observed that tasks sometimes inherited ceiling priorities before they started executing in the traces produced by the model checker. We conjecture that the model-checker tool implements a different version of the immediate priority ceiling protocol than originally defined [32]. Since we do not know the exact semantics and properties of the model-checker's implementation of the protocol, we use the original definition. An interesting observation is that all

Table 2
Results from baseline real-time system experiment

Mutation type	μ	**M**	**C**	**K**	**A**	**G**
Execution time	10	6	6	6	5.8 (0.1)	7.6 (207.8)
Lock time	8	1	1	1	1.0 (0)	1.3 (0.2)
Unlock time	11	2	2	2	2.0 (0)	2.2 (3.1)
Hold time shift	14	0	1	0	-	-
Precedence	20	14	15	14	14.0 (0)	1.2 (0.8)
Inter-arrival time	10	3	4	3	3.0 (0)	5.7 (53.7)
Pattern offset	10	3	5	3	3.0 (0)	2.5 (8.7)
Total	83	29	33	29	28.8	-

malignant mutants were killed within 10 generations in average (see column 'G' of table 2). Further, all malignant mutants were killed in seven of the eight trials.

4.2 Dynamic Real-time System Experiment

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 blocking 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 [4].

This system consist of 12 hard real-time tasks, seven of which are sporadic and only five periodic. The system has three shared resources but no precedence constraints. The complete task characteristics are listed in table 3, using the same notation as in table 1.

For this system it was too time consuming to manually derive the malignant mutants. Further, the model-checker tool used in the first experiment could not be used for comparison since the reachable state-space became too large[*].

Other, more advanced, model-checking tools could be adopted for analyzing schedulablity of TAT models. However, the mapping from TAT models to other model-checker representations are not trivial. Further, since the verifier in the Times tool is an extension of the verifier in the more commonly used

[*] Times model-checker does not currently support the SRP protocol, but even a sporadic task set of this size without shared resources was refused.

Table 3
Task set for dynamic real-time system

ID	**TAT quadruple**	**I**	**O**
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

Uppaal tool [5], we assume that its performance is representative for this kind of analysis.

Since we could not find an alternative way to efficiently and reliably analyze mutants, we cannot guarantee that the method killed all malignant mutants. To increase confidence in the timeliness of the original specification model, every generated test case was also run on the un-mutated TAT specification. This test actually revealed a mistake in a model that was assumed to be timely in another experiment. We ran the genetic algorithm on each mutant for 200 generations or until a failure was encountered. Each experiment was performed five times. For each simulation performed during the heuristic search, a simulation with a random arrival pattern was also performed. This gives an indication of the relative efficiency of random search of the model (and random testing of the target system). Further, we ran an genetic algorithm experiment using only the generic cross-over functions, described in section 4.1, to get an indication of the added performance of our heuristic cross-over operators. Since every operator generated more mutants for this system model, we decided to use a subset of mutation operator types (based on the average number of generations required to kill a mutated model). Table 3 uses the same column notation as table 2, but columns are added to include the results from random testing ("R") and non-heuristic genetic algorithms ("E"). The"Δ" column contain the delta sizes used for the different operator types.

Table 4
Results from dynamic real-time system experiment

Mutation type	Δ	μ	**R**	**E**	**K**	**A**	**G**
Execution time	2	24	0	0	12	9.2 (2.2)	62 (2372)
Unlock time	2	16	0	0	0	-	-
Inter-arrival time	6	24	0	0	8	3.8 (1.2)	90 (4150)
Offset time	6	22	0	0	0	-	-
Total	-	86	0	0	20	13.0	-

As seen in table 4 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 is probably because the search problem is more difficult than for the more static system presented in section 4.1. The low number of mutants killed in average in each trial suggests that fewer execution orders exists that can reveal the faults in the malignant mutants. 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 get stuck in local optima and prematurely discard partially refined candidates. A possible remedy to this problem is to redo the search multiple times using a fresh initial population. This may be acceptable since the approach for searching the mutant models is fully automated. The comparison with random testing and a non-heuristic genetic algorithm shows that the heuristic cross-over functions are vital for the performance of the method.

5 Related work

This section describe existing methods for testing real-time systems. Table 5 lists the authors of related work and classifies the contributions with respect to three categories. When the same authors have several related publications addressing different aspects of the same test method, only one is included.

The first category (the column marked "T") lists if the approaches use models including time-constraints or address testing system-level timeliness. The column marked "I" indicate if the related work uses information about concurrent tasks, use of shared resources and real-time protocols for deciding relevant inputs. In contrast to our approach very few other methods based on formal notations include this in their models, probably to avoid state space explosion. However, if the internal behavior is not modelled, it is generally impossible to predict the worst case activation pattern for a system that is implemented using conventional real-time operating systems and task models. For example, exactly the same activation patterns might give completely

Table 5
Classification of related work

#	**Authors**	**T**	**I**	**C**
1	Braberman et al. [6]	y	y	y
2	Cheung et al. [8]	y	y	n
3	Clarke and Lee [9]	y	n	y
4	Petitjean and Fochal [25]	y	n	y
5	Mandrioli et al. [17]	y	n	y
6	Kirchen and Tripakis [16]	y	n	y
7	Cardell-Oliver and Glover [7]	y	n	y
8	En-Nouaary et al. [10]	y	n	n
9	Nielsen and Skou [20]	y	n	n
10	Raymond et al. [29]	y	n	n
11	Watkins et al. [35]	y	n	n
12	Morasca and Pezze [19]	n	y	y
13	Pettersson and Thane[27]	n	y	n
14	Wegener et al. [36]	n	n	n

different behavior depending on the execution time of tasks.

The column denoted "C" lists whether or not the related work propose testing criteria that are usable together with their method. The testing criteria we propose are associated with the mutation operator types (see section 3). Other methods propose testing criteria based on coverage of model structure, such as sequences of transitions or locations in an automata.

The method by Braberman et al. [6] is the closest related work; they generate test cases from timed Petri-net design models. Similarly to our method, a high level notation, SA/SD-RT, is used to specify the behavior of concurrent real-time systems. In contrast to our approach, no mutant models are generated, instead their design specification is translated to a timed Petri-net notation from which a reachability tree can be derived and covered. Since the model has a similar level of detail as ours, we suspect that the number of tests needed to cover the reachability tree increases very quickly with the size of the system. Cheung et al. [8] presented a framework for testing multimedia software, including temporal relations between tasks with "fuzzy" deadlines. In contrast to our approach, the test cases generated are targeted at testing multi-media applications and their specific properties.

There are several methods for testing timeliness based on different flavors of formal models. As mentioned above, these methods typically do not model the real-time tasks and protocols of the tested system. Further, none of these methods use mutation based testing techniques (see table 5, rows 3 - 11). For example, Clarke and Lee [9] proposed a framework for testing time constraints on the activation patterns of real-time systems. Time constraints are specified in a constraint graph, and the system under test is specified using process algebra. In contrast to our approach, only constraints on the inputs to the tested system are considered and the authors mention that it would be very difficult to test constraints on the outputs since it depends on non-deterministic internal factors. Petitjean and Fochal [25] present a method where time constraints are expressed using a clock region graph. A timed automation specification of the system is then "flattened" to a conventional input output automation that is used to derive conformance tests for the implementation in each clock region. A method of how clocks in the target system can be handled when doing model based conformance testing is presented. Krichen and Tripakis [16] address limitations in applicability of previous black-box approaches and suggest a method for conformance testing using non-deterministic and partially observable models. The testing criteria presented is inspired by Hessel et al. [13] but extended for test case specifications that allow several possible interactions with the implementation.

Mandrioli et al. [17] suggest a method to test real-time systems based on specifications of system behavior in temporal logic. The elements of test cases are timed input-output pairs. These pairs can be combined and shifted in time to create a large number of partial test cases, the number of such pairs grows quickly with the size and constraints on the software. In a more recent paper [30], the authors expanded their previous results to incorporate high-level, structured specification to deal with larger scale, modular software. Cardell-Oliver and Glover [7] propose a method for generating tests from timed automata models to verify sequences of timed action transitions. This approach uses reachability analysis to determine what transitions to test, hence, we assume it will suffer from state space explosion for large dynamic models. Another automata based approach was presented by En-Nouaary et al. [10]. Their approach exploits a sampling algorithm using grid-automata and non-deterministic finite-state machines as an intermediate representation to reduce the test effort. Similarly, Nielsen and Skou [20] use a subclass of timed automata to specify real-time applications. The main contribution with their method is a coarse equivalence partitioning of temporal behaviors over the time constraints in the specification. Raymond et al. [29] presented a method to generate event sequences for reactive systems. Their approach models environmental constraints and test requirements as external observers.

In contrast to our approach and the other methods in this category Watkins et al. [35] does not use a formal model as basis for test case generation. Instead genetic algorithms are used to drive the execution of complex systems that contain time constraints. Data are gathered during execution of the real system and visualized for post analysis. Fitness of a test case is calculated based on its uniqueness and what exceptions are generated by the systems and test harness during test execution. Similarly with this method, our genetic algorithm extensions could be used directly on a target system instead on a model. However, the disadvantages is that no testing criteria could be used for measuring progress and that the search problem is further elevated by the internal non-determinism of the system.

There is some related work that does not address system-level testing of timeliness but is still relevant for testing real-time systems or as complements to our approach (see table 5, rows 12 - 14). Morasca and Pezze [19] proposed a method for testing concurrent and real-time systems that uses high-level Petri-nets for specification and implementation. This method does not explicitly handle timeliness, nor does it provide testing criteria but it is, to the authors knowledge, one of the first to model the internal concurrency of the tested real-time system. Thane [33] proposed a method to derive execution orders of a real-time system before it is put into operation. It was suggested that each execution order can be treated as a sequential program where conventional test methods can be applied. After test execution, the test logs are sorted according to the pre-analyzed execution orders. In a more recent paper, Pettersson and Thane [27] extended the method by supporting shared resources. In contrast to our method, this method is developed for real-time systems where all task activation times are fixed. Wegener et al. has explored the capabilities of genetic algorithms for testing temporal properties of real-time tasks [36]. However, the main focus of their work is determining suitable inputs for producing worst and best-case execution time. This approach is a valuable complement to our method, since we assume that relevant classes of input data exists for each real-time task before system-level testing of timeliness starts.

6 Conclusions

This paper has proposed a model based method for generating test cases to test timeliness by using heuristic driven simulation. A baseline case study was presented that indicates that the method is efficient and reliable for generating test cases for small real-time systems that contain shared resources, precedence constraints and few sporadic tasks. The method was also evaluated for a dynamic system with more advanced real-time protocols and a large fraction of sporadic tasks. For such systems, no current method of automatic generation of mutation-based test cases is applicable. As expected, the search problem is increasingly difficult for the more dynamic system. However, a genetic algorithm using our heuristic cross-over functions shows a significantly better performance than both random search and a genetic algorithm using only generic cross-over functions. This approach increases the usefulness of mutation-based testing of timeliness so that real-time systems of more realistic size and type can be tested.

The effectiveness of the generated test cases when executed on a real-time target system is currently being investigated. Our genome mapping function should be generalized to support a larger class of TAT automata templates in future work.

References

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

[2] Ammann, P., 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*, IEEE Computer Society, 1998, pp. 46–54.

[3] Amnell, T., E. Fersman, L. Mokrushin, P. Pettersson and W. Yi, *Times - A tool for modelling and implementation of embedded systems*, in: *Proceedings of TACAS'02*, 2280 (2002), pp. 460–464.

[4] Baker, T. P., *Stack-based scheduling of real-time processes*, The Journal of Real-Time Systems (1991), pp. 67–99.

[5] Bengtsson, J., K. G. Larsen, F. Larsson, P. Pettersson and W. Yi, Uppaal — *a Tool Suite for Automatic Verification of Real–Time Systems*, in: *Proceedings of Workshop on Verification and Control of Hybrid Systems*, number 1066 in Lecture Notes in Computer Science (1995), pp. 232–243.

[6] Braberman, V., M. Felder and M. Marré, *Testing timing behavior of real-time software*, in: *International Software Quality Week*, 1997.

[7] Cardell-Oliver, R. and T. Glover, *A practical and complete algorithm for testing real-time systems*, Lecture Notes in Computer Science **1486** (1998), pp. 251–261.

[8] Cheung, S. C., S. T. Chanson and Z. Xu, *Toward generic timing tests for distributed multimedia software systems*, in: *ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01)* (2001), p. 210.

[9] Clarke, D. and I. Lee, *Automatic generation of tests for timing constraints from requirements*, in: *Proceedings of the Third International Workshop on Object-Oriented Real-Time Dependable Systems*, Newport Beach, California, 1997.

[10] En-Nouaary, R., K. F. Dssouli and A. Elqortobi, *Timed test case generation based on a state characterization technique*, in: *Proceeding of the 19th IEEE Real-Time Systems Symposium (RTSS98)*, Madrid, Spain, 1998.

[11] Fersman, E., "A Generic Approach to Schedulability Analysis of Real-Time Systems," Ph.D. thesis, University of Uppsala, Faculty of Science and Technology (2003).

[12] Henriksson, D., A. Cervin and K.-E. Årzén, *TrueTime: Real-time control system simulation with MATLAB/Simulink*, in: *Proceedings of the Nordic MATLAB Conference*, Copenhagen, Denmark, 2003.

[13] Hessel, A., K. Larsen, B. Nielsen and A. Skou, *Time optimal real-time test case generation using UPPAAL*, in: *Proceedings of Workshop on Formal Approaches to Testing of Software (FATES)*, Montreal, 2003.

[14] Houck, C., J. Joines and M. Kay, *A genetic algorithm for function optimization: A Matlab implementation*, Technical Report NCSU-IE TR 95-09, Department of Computer Science, North Carolina State University (1995).

[15] Hwang, G., K. Tai and T. Hunag, *Reachability testing : An approach to testing concurrent software*, International Journal of Software Engineering and Knowledge Engineering **5** (1995).

[16] Krichen, M. and S. Tripakis, *Black-box conformance testing for real-time systems*, in: *Proceecings of SPIN'04 Workshop on Model-Checking Software*, 2004.

[17] Mandrioli, D., S. Morasca and A. Morzenti, *Generating test cases for real-time systems from logic specifications*, ACM Transactions on Computer Systems **4** (1995), pp. 365–398.

[18] Michalewicz, Z. and D. B. Fogel, "How to solve it : Modern Heuristics," Springer, 1998.

[19] Morasca, S. and M. Pezze, *Using high level Petri-nets for testing concurrent and real-time systems*, Real-Time Systems: Theory and Applications (1990), pp. 119–131, amsterdam North-Holland.

[20] Nielsen, B. and A. Skou, *Automated test generation from timed automata*, in: *Proceedings of the 21st IEEE Real-Time Systems Symposium*, IEEE, Walt Disney World, Orlando, Florida, 2000.

[21] Nilsson, R., 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)*, Tokyo, Japan, 2002, pp. 109–113.

[22] Nilsson, R., J. Offutt and S. F. Andler, *Mutation-based testing criteria for timeliness*, in: *Proceedings of the 28th Annual Computer Software and Applications Conference (COMPSAC)* (2004), pp. 306–312.

[23] Nilsson, R., J. Offutt and J. Mellin, *Test case generation for testing of timeliness - extended version*, Technical Report HS-IKI-TR-05-003, School of Humanities and Informatics, University of Skövde (2005).

[24] Nordström, C., A.Wall and W. Yi, *Timed automata as task models for event-driven systems*, in: *Proceedings of RTCSA'99*, Hong Kong, 1999.

[25] Petitjean, E. and H. Fochal, *A realistic architecture for timed testing*, in: *Proceedings of Fifth IEEE International Conference on Engineering of Complex Computer Systems*, USA, Las Vegas, 1999.

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

[27] Pettersson, A. and H. Thane, *Testing of multi-tasking real-time systems with critical sections*, in: *Proceedings of Ninth International Conference on Real-Time Computing Systems and Applications (RTCSA'03)*, Tainan city, Taiwan, 2003.

[28] Ramamritham, K., *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)* (1995), pp. 50–62.

[29] Raymond, P., X. Nicollin, N. Halbwachs and D. Weber, *Automatic testing of reactive systems*, in: *Proceeding of the 19th IEEE Real-Time Systems Symposium (RTSS98)*, 1998.

[30] SanPietro, P., A. Morzenti and S. Morasca, *Generation of execution sequences for modular time critical systems*, IEEE Transactions on Software Engineering **26** (2000), pp. 128–149.

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

[32] Sha, L., R. Rajkumar and J. P. Lehczky, *Priority inheritance protocols: An approach to real-time synchronization*, IEEE Transactions on Computers **9** (1990), pp. 1175–1185.

[33] Thane, H., "Monitoring, Testing and Debugging of Distributed Real-Time Systems," Ph.D. thesis, Royal Institute of Technology. KTH, Stockholm, Sweden (2000).

[34] Verissimo, P. and H. Kopetz, "Distributed Systems," Addison Wesley, 1993 pp. 511–530.

[35] Watkins, A., D. Berndt, K. Aebischer, J. Fisher and L. Johnson, *Breeding software test cases for complex systems*, in: *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9* (2004), p. 90303.3.

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