Test Case Generation for Testing of Timeliness - Extended version

Robert Nilsson[1], Jeff Offutt[2], and Jonas Mellin[1]

[1] Humanities and Informatics, University of Skövde
[2] Information and Software Engineering. George Mason University
`robert.nilsson@his.se` `ofut@ise.gmu.edu` `jonas.mellin@his.se`

HS-IKI-TR-05-003

Abstract. Temporal correctness is crucial for real-time systems. There are few methods 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 orders in turn depends on scheduling protocols, task execution times, and use of mutual exclusive resources apart from the points in time when stimuli is injected. Model-based mutation testing has previously been proposed to determine the execution orders that need to be tested to increase confidence in timeliness. An effective way to automatically generating such test cases for dynamic real-time systems is still needed. This paper presents a method using heuristic-driven simulation for generation of test cases.

1 Introduction

Current real-time systems must be both flexible and dependable. There is a desire to increase the number of services that real-time systems offer while using few, standardized hardware components. This increases system complexity and introduces sources of temporal non-determinism that complicate modeling of the execution behavior of tasks. Faults in such models may result in software timeliness violations and costly accidents. Thus, we need methods to detect violation of time constraints for computer architectures where we cannot to 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 [1].

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 depends on the order in which they are scheduled to execute. *Periodic* tasks are activated

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

with fixed inter-arrival times, thus all the points in time when such tasks are activated are known. *Sporadic* tasks are activated dynamically, but constraints on their activation patterns such as *minimum inter-arrival times*are used for analysis. Each real-time task typically has a *deadline.* Tasks may also have an *offset* that denotes the time before a task is activated.

Timeliness of embedded real-time systems is traditionally analyzed and maintained using scheduling analysis techniques or regulated online through admission control and contingency schemes [2]. However, these techniques use assumptions about the tasks and load patterns that must be correct for timeliness to be maintained. Further, doing full schedulability analysis of non-trivial system models is complicated and require specific rules to be followed by the run-time system. In contrast, timeliness testing is general in the sense that it applies to all system architectures and can be used to gain confidence in assumptions by systematically sampling among the execution orders that can lead to missed deadlines. Hence, from a real-time perspective, testing of timeliness is a necessary complement to analysis.

Properties such as timeliness, must specifically be addressed during system level testing because it is difficult to construct meaningful input sequences without considering the effect on the set of active tasks and real-time protocols. This is particularly relevant in *event-triggered* systems, because interrupts can influence the execution order differently depending on the current state of the system [3].

Some of all the possible execution orders may reveal timeliness violations in the presence of timing faults. However, existing testing techniques for timeliness seldom use information about real-time design to generate test cases [4] and do not predict what execution orders that may lead to failures.

1.1 Mutation-based testing of timeliness

Mutation-based testing of timeliness is inspired by a specification based method for automatic test case generation presented by Ammann, Black and Majurski [5]. The main idea behind the method is to systematically "guess" what faults a system contains and then evaluate what the effect of such a fault could be in a deterministic model of the system under test. Once faults with bad consequences are identified, specialized test cases are constructed that try to reveal those faults in the system implementation.

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 that 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 (illustrated in figure 1). 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, then the mutant is marked as *killed.* Traces from the killed mutants are fed into test case generation that extract an arrival pattern that has

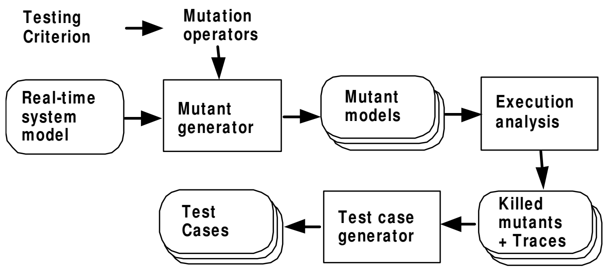

Fig. 1. Mutation based test case generation

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 in a malignant mutant. During test case execution the generated input sequences are injected in the real-time system under test.

The problems associated with controllability and observability when testing flexible real-time systems are out of scope of this paper. Prefix-based or non-deterministic testing techniques [6] can be used to take advantage of the test cases generated by our approach.

1.2 Simulation-based test case generation

The test case generation method proposed in this paper maps a system specification model that captures possible execution behaviors into a real-time system simulator. The simulator is iteratively executed using a genetic algorithm to find the arrival patterns of sporadic tasks that reveal the fault in the mutant specification and, thus, kill the mutant. [*].

The proposed method is demonstrated in two experiments. The first experiment compares the method with a model-checking based approach to gain basic confidence in its reliability. The method is then evaluated using a larger, more dynamic system specification for which the model-checking based approach fails.

The results indicate that the simulation based method remain effective on dynamic specification models and that the presented heuristics cross-over functions significantly enhance the performance compared to generic cross-over functions.

2 System model and testing criteria

In this paper, we use a subset of Timed Automata with Tasks (*TAT*) [7, 8] to define the assumptions about the system under test and as a source for specification based test case generation.

Timed Automata (*TA*) [9] have been used to model different aspects of real-time systems. 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

[*] The mutant operators used in this context is not performed by the genetic algorithm

a number of clock resets. A *guard* is a logical condition on clocks and variables stating when a transition can occur. 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 target systems with synchronized clocks). Within TAT models, TA is used to specify the activation pattern of tasks.

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 required execution time, d is the relative deadline, and SEM and $PREC$ are defined in the following two paragraphs. The values c and d are used to create new task instances as they are activated by TA actions.

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 relative to the task's start time. The variable SEM is a set of tuples on the form $(s,\ t_1,\ t_2)$ where t_1 and t_2 are the lock and unlock times of semaphore $s \in R$.

Precedence constraints are relations between pairs of tasks A and B stating that a instance of a task A must have executed to completion between the execution of two consecutive instances of task B. Hence, $PREC$ is a subset of P that specifies what other tasks must precede this task.

As an example of a TAT task set, consider the independent behavior of two tasks, depicted in figure 2, and their corresponding representation in table 1. We denote tasks behavior, showing the points in time 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 input sequence is found, the target system can be tested several times using different task inputs in that sequence, stressing it to reveal a potentially faulty behavior. Note that task execution patterns already have been modified by the mutation operators, thus, effects of such variations are investigated.

Table 1. TAT task set description

ID	c	d	SEM	$PREC$
A	8	20	{(R1,1,6)}	{}
B	9	40	{(R1,2,8)}	{}

2.1 Complexity of timeliness mutation operators

A *testing criterion* defines test requirements that must be satisfied when testing software. Examples of test criteria include "execute all statements once" and

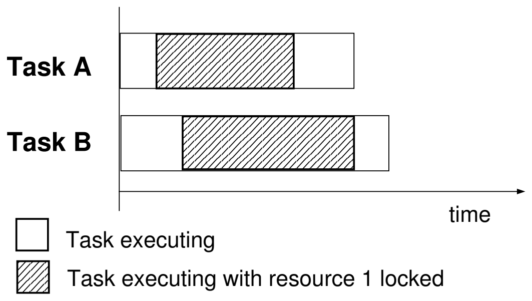

Fig. 2. Visualization of tasks behavior

"cover all transitions" in a state machine. A *test coverage* measure expresses how thoroughly tests have satisfied a test criterion.

A mutation-based test criterion is defined by a set of mutation operators. Most mutation operators model possible faults, so the first step in defining mutation operators is to understand the types of faults that can lead to timeliness failures when building real-time systems. Our previous work identified and presented formal definitions of seven types of faults that can lead to timeliness failures [10]. This paper summarizes the operators informally and discuss the complexity of the operators with respect to target system characteristics.

Note that the mutation operators change some property in the specification of a real-time system, not properties of the system itself. In the following descriptions, n is the number of tasks in the systems task set and r is the number of shared resources requiring mutual exclusion. In some systems, the same resource is locked and released multiple times by the same task to decrease blocking lengths, so l denotes the maximum number of times a resource is locked by the same task. In many of the operators some property of the execution pattern is modified slightly, so Δ is used to denote the size of the change.

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 a overly pessimistic estimation of the best-case (shortest) execution time. Estimating execution times is generally very hard [11]. The execution time of a task running concurrently with other tasks may also be slightly different than running the task uninterrupted, if there are caches and pipelines in the target system. For each task in a task set, two mutants are created. One increases the execution time by delta time units, and one decreases the execution time by the same amount. The number of mutants created is $2n$.

Lock time mutation operators: The lock time mutation operator increases or decreases the time when a particular resource is locked. An increase in the time a resource is locked increases the maximum blocking time for a higher priority task. Further, if a resource is held for less time than expected, the system can allow execution orders that may result in timeliness or control violations. For each task and each resource the task may use, two mutants are created. One increases the lock time and one decreases the lock time. The maximum number of mutants created is $2nrl$.

Unlock time mutation operators: Unlock time mutation operators change the point in time when a resource is unlocked. For each task and each resource that the task uses, two mutants can be created. One increases the unlock time and one decreases the unlock time of that particular resource. The maximum number of mutants created is $2nrl$.

Hold time shift mutation operators: The hold time shift mutation operator shifts the interval of time a resource is locked relative to the start point of the task. For example, if a semaphore is to be locked at time 2 and held until time 4 in the original model, it will be locked at time 3 and held until time 5 in the $\Delta+$ mutant. For each task and each resource that the task uses, two mutants are created. The maximum number of mutants created is $2nrl$.

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

Inter-arrival time mutation operators: This operator decreases or increases the assumed inter-arrival time between requests for a task execution by a constant time Δ. This reflects a change in the system's environment that causes requests to come more frequently than expected*. Since this operator changes the inter-arrival times on a per-task basis the maximum number of created mutants is $2n$.

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 Δ time units. Since this operator modifies the offsets on a per-task basis, the maximum number of mutants is $2n$.

Table 2. Complexity of mutation operators

Operator type	**Complexity class**
Execution time	$O(n)$
Hold time shift	$O(nrl)$
Lock time	$O(nrl)$
Unlock time	$O(nrl)$
Precedence constraint	$O(n^2)$
Inter-arrival time	$O(n)$
Pattern offset	$O(n)$

* In Nilsson, Offutt and Andler [10], only the $\Delta-$ operator was defined, the corresponding $\Delta+$ operator has been included in experiments for completeness.

3 Model-based test case generation using genetic algorithms and simulations

Model-checking has previously been used for analyzing the mutated specification models [10]. Model-checking guarantee that timeliness violations are revealed if they exist in the mutated model. However, for some systems the number of possible states becomes too large for model-checking to be effective. In particular, the computational complexity (both time and memory) grows when task requests are allowed to occur at any point in time.

In dynamic real-time systems, there are many sporadic tasks, making model-checking impractical. We propose an approach for testing timeliness of such systems where a simulation of the 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 since only one trace is kept in memory.

Further, we conjecture that it is easier to extend and modify a system simulation to correspond with the architecture of the system under test than a model-checker. If model-checking is used, all changes to the system model has to be made in such a way that the model-checker still works.

The transition from model-checking to simulation can be compared with a similar trend in automatic test data generation for sequential software, where constraint-solving [12] successfully has been complemented with dynamic approaches where actual software executions guide the search for input values [13].

When simulation is used for model based test generation, the task set of the TAT model must be mapped to corresponding task entities in a real-time simulator. The activation pattern of periodic tasks are statically defined by the model and can be included into the configuration of the simulator. The activation pattern for sporadic (automata controlled) 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 system (corresponding to a TAT model) is an activation pattern of 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 mutation testing perspective is an execution order trace that leads to a missed deadline.

By treating test case generation as an optimization problem, different heuristic methods can be applied to find an optimal solution. This paper concentrates on genetic algorithms, since they are configurable and can manage search spaces containing local optima [14].

To successfully apply genetic algorithms to a specific optimization problem the following three issues need consideration *(i)* genome mapping function, *(ii)* fitness function, and *(iii)* heuristic cross-over functions.

3.1 Genome mapping function

Genetic algorithms operate by iteratively refining a set of solutions to an optimization problem through random changes and by combination of 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 standard formats for genomes are bit-strings and arrays of real values. Each genome corresponds to a point in the search space for the optimization problem.

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 solution of 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 [14].

For the test case generation problem, the only thing that varies between simulations of the same mutant TAT-model is the activation pattern of sporadic tasks, thus it is sufficient that a genome can be mapped to such activation pattern.

In the general case, this can be expressed by any timed automata, but in this paper we focus on an encoding that model sporadic tasks released by specific automata templates.

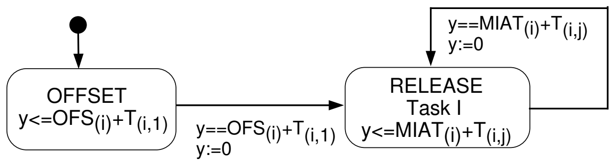

Fig. 3. Annotated TAT template

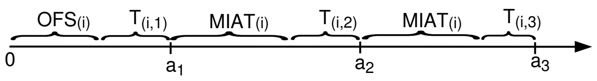

Fig. 4. Activation pattern of sporadic tasks

Figure 3 contains an annotated TAT-automata for describing activation patterns of sporadic tasks. The template has two parameters that are constant for each mutant. The parameter OFS denotes the assumed offset, that is, the minimum delay before any instance of this task is assumed to be requested. The parameter MIAT denotes 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. In the rest of this paper, m denotes the maximum number of activations that can occur in the simulation interval. Figure 4 depicts the first three activations of a sporadic task and exemplifies the relation between the automata constants and the values in T. 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 valid activation patterns for the mutant. Each activation pattern results in a particular execution order in the simulation.

3.2 Fitness function

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 for further refinement in the next generation.

Since our genome representation of individuals 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. Once we have run the simulation we can use the execution order trace logs to determine how optimal a particular solution is. A suitable fitness function for timeliness should measure how close a system is to breaking a critical deadline. The *slack* is the time between the response-time of a task activation and its deadline. Hence, after a simulation is run the minimum observed slack during can be used as a simple measure of fitness.

To summarize, fitness for each individual is computed in three steps. First a genome is translated to an activation pattern of sporadic tasks. Secondly, a simulation of the mutant TAT model is run and sporadic tasks are requested according to the activation pattern. Lastly, the execution order trace from the simulation is analyzed to find the minimum slack of any task. The highest fitness is given to the individual that result in the execution order trace with the least minimum slack.

3.3 Heuristic cross-over functions

Cross-over is applied on the selected individuals to enhance them into new individuals with higher fitness for the next generation. Either this means combining good properties from two individuals, or modifying a single individual according to heuristics. Traditionally, a crossover from a single individual is called a "mutation," but to avoid ambiguity we use the concept *cross-over functions* for all functions that change genomes.

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 chance of increasing the individual's fitness. On the other hand, some cross-over functions must remain stochastic to avoid the search getting stuck in local optima.

For timeliness testing, there are intuitive heuristics of what kind of activation patterns are likely to stress the mutant model. 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 requests.

Several concepts need to be introduced to simplify our definitions of heuristic cross-over functions. The variable M is a TAT model containing n sporadic tasks controlled by automata templates such as in figure 3. As described in section 3.1, the variable T denote a genome matrix of size $n * m$. The integer variable i is used for indexing over the rows in a genome matrix T. The rows in such matrixes 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 $([a_{beg}, a_{end}] \sqsubseteq [b_{beg}, b_{end}]) \iff (a_{beg} \geq b_{beg} \wedge a_{end} \leq b_{end})$.

Definition 1 : Critical task instance
The task instance with the least slack in an execution order trace.

Definition 2 : Idle point
A point in time where no real-time task executes or is queued for immediate execution on the processor.

Definition 3 : Critical interval $[ci_{beg}, ci_{end}]$
The interval between the activation time and response time of a critical task instance.

Definition 4 : Loading interval $[li_{beg}, li_{end}]$
The interval between the latest idle point and the activation time of the critical task instance. Note that $li_{end} = ci_{beg}$.

Definition 5 : Delay interval matrix D
A matrix of size $n * m$ *containing the variable delay intervals associated with each sporadic task activation in* T *such that:*

$$D_{(i,j)} = [epat(i,j), epat(i,j) + T_{(i,j)}]$$

Where the function $epat(i,j)$ *gives the earliest possible arrival time of the j'th instance of sporadic task* i *given a TAT model* M *and a genome matrix* T.

Critical interval focus functions

This cross-over function analyzes the logs from the simulation to find the critical interval. A sporadic tasks activation pattern is chosen by random and changed so that requests become more likely to occur within or close to the critical interval.

Function definition 1 : 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.*

Function definition 2 : 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* $T_{(i,j)} = 0$.

<u>**Critical interval move function**</u>:

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

Function definition 3 : Critical interval move*:*
For all i such that $D_{(i,0)} \sqsubseteq [0, ci_{beg}]$ *increase or decrease* $T_{(i,0)}$ *with* ϵ *time units.*

<u>**New interval focus function**</u>

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 releases are shifted towards the selected point in time.

Function definition 4 : New interval focus
Let t_{new} *be a random instant within the simulation interval. For all 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.*

<u>**Loading interval perturbation function**</u>:

Theoretically, all task activations in the loading interval may influence response time through the state in the system when the request of the critical task instance occurs [4]. In practice, it is more likely that 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.

Function definition 5 : Loading interval perturbation
For an arbitrary index i, let j be the the largest index such that $D_{(i,j)} \sqsubseteq [li_{beg}, li_{end}]$ *and* $j > 0$, *modify T so that* $T_{(i,j-1)} = \epsilon$ *time units.*

4 Test case generation experiments

To evaluate the proposed method, we present two types of experiments. First, we establish basic confidence in the method by applying it on a small system model that also has been used in a test case generation experiment using a model-checker. This allows us to compare the method in a base-line experiment and detect if the genetic algorithm method has problems finding any specific types of mutants. The second experiment uses a larger system model and 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 have extended the real-time and control co-simulation tool *TrueTime* to simulate the execution of TAT-models. TrueTime is developed at the department of automatic control at the University of Lund to support integrated design of controllers and real-time schedulers [15]. We also configured and extended a genetic algorithm tool-box [16] to interact with our simulation model. For model-checking experiments we used the *Times* tool, developed at Uppsala University [17].

4.1 Base-line real-time system experiment

This experiment uses a small task set wit few sporadic tasks but with complex interactions. Static priorities are 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 system use the *immediate ceiling priority* protocol to avoid priority inversion [18]. 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. Two tasks are sporadic and the three remaining tasks are strictly periodic.

The system has two shared resources modeled by semaphores, and one precedence relation between tasks D and A, which specifies that a new instance of task A cannot start unless task D has executed after the last instance of A.

Table 3 describes the assumptions of the task set. The first column ("ID") gives task identifiers, column "c" gives execution times, and column "d" gives relative deadlines. The "SEM" column specifies the set of semaphores used and which interval they are required in. Column "PREC" specifies what other tasks have precedence over tasks of this type. For sporadic tasks, the "IAT" column contains the minimum inter-arrival time assumptions (marked "MIAT" in the timed automata template in figure 3). For periodic tasks the same column contain the inter-arrival time. Column "OFS" denotes the initial offset constant.

Table 3. Task set for validation experiment

ID	**c**	**d**	**SEM**	**PREC**	**IAT**	**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

Table 4 contains the results from mutation testing the task set in table 3. 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 "M". The number of malignant mutants is listed in column "MA" and the number of mutants killed by model-checking is listed in column "MC." For the genetic algorithm setup, we used a population of 20 individuals per generation and ran each mutant 100 generations before terminating the search. We used the heuristic cross-over functions described in section 3.3 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. A generic ranking function called roulette wheel selection [19] was used for selecting the individuals that should be used in the next generation and be subject to heuristic refinement via cross-overs. The selection was configured so that the individual with the highest fitness always had a ten percent probability of being selected and put in the next population. Each of the heuristic cross-over operators was applied to at least two individuals in the new population before the population was re-evaluated.

To gain confidence in the results, each experiment was repeated in 8 trials, with different random seeds and different random initial populations. The number of mutants that was killed using genetic algorithms in any of the trials is listed in column "GH". Column "A" lists the average number of malignant mutants that was killed per trial. The average number of generations needed to kill malignant mutants of this type is in column "GE."

Table 4. Results from mutation-based testing

Mutation operator	**M**	**MA**	**MC**	**GH**	**A**	**GE**
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	-

As seen in table 4, both the simulation-based and model-checking approaches killed all the malignant mutants. Strangely, the model-checking approach 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 model checker implementation. We conjecture that the model-checker tool implements a different version of the immediate priority ceiling protocol than originally defined [18]. 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 malignant mutants were killed within 10 generations in average (see column 'G' of table 4). Further all malignant mutants where killed in 7 of the 8 experiments.

4.2 Dynamic real-time system experiment

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 and a more advanced execution environment.

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 [20].

This system consist of 12 hard real-time tasks, seven of which are sporadic and five periodic. The system has three shared resources but no precedence constraints. The complete task characteristics are listed in table 5, using the same notation as in table 3.

Table 5. Task set for scalability experiment

ID	c	d	SEM	IAT	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

For this system it is too time consuming to manually derive the number of malignant mutants. Moreover, model-checking cannot be used for comparison since the number of possible states in the model becomes too large[*]. 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 correctness of the original specification model, every generated test-case was also run on the un-mutated TAT specification.

For this experiment, we used a delta size of 2 time units for the mutations on the execution patterns, and a delta size of 6 time units on the mutations on automata template constants.

In each trial we ran the genetic algorithm on each mutant for 200 generations. Each experiment was performed five times to guard against stochastic variance.

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

For each simulation performed during the heuristic search, a random test was also performed. This gives an indication of the relative efficiency between random search of the model and genetic algorithms with our heuristic. Further, we ran an genetic algorithm experiment of the same size using only the simple and generic cross-over functions, described in section 4.1. This was done get an indication of the added performance of our heuristic cross-over operators compared to the added performance of the iterative behavior of the genetic algorithm in itself.

Since every mutant operator generated more mutants for this system model we decided to use a subset of mutation operator types. Table 6 use the same column notation as table 4, but columns "R" and "G" are added to include the results from random testing and genetic algorithms results respectively.

Table 6. Result from dynamic real-time system experiment

Mutation operator	**M**	**R**	**G**	**GH**	**A**	**GE**
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	-

As seen in table 6 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 more difficult than for the system presented in section 4.1.

The low average number of mutants killed in each trial suggests that fewer execution orders exists that can reveal the timeliness faults in the malignant mutants, thus, the genetic algorithms may have to run longer on each mutant to get reliable results. Another 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 these problems is to redo the search multiple times using a fresh, random, initial population. Since the approach for searching the mutant specifications is fully automated, the additional cost of searching multiple times may be acceptable.

Further, the comparison with random testing and a generic genetic algorithm shows that the heuristic cross-over functions is vital for the performance of the method.

5 Discussion

This section discuss some of potential benefits and limitations of the proposed method.

5.1 Value add

The problem addressed by this paper is primarily associated with the non-deterministic arrival times of sporadic tasks. Since the target platform contain additional sources of non-determinism (for example caches and pipelines), several different execution orders are expected for a particular activation pattern of sporadic tasks. Once a particular activation pattern has been generated the test execution techniques presented by Thane and Pettersson [21, 22] can be used to gain coverage of these expected execution orders. Further, the execution order descriptions generated by this approach gives added value since it is possible to to determine a subset of these execution orders that are most relevant to cover with respect to timeliness.

The execution order analysis performed during test case generation can be seen as an rudimentary form of automated schedulablity analysis for event-triggered real-time systems. However, since our method does not guarantee to find timeliness violations in a model, we do not claim that it is rigorous enough to be used for this purpose. Testing have the fundamental limitation that it only can show the presence of errors, but not their absence. The execution order analysis share this property and should therefore primarily be used for testing.

Another possibility is to take the activation patterns generated from mutated models and simulate them on the un-mutated model. This can be seen as testing the timeliness of the assumed model. Obviously, this do not say anything about the properties of the implemented system, but it is likely that such test cases can reveal timeliness violations in models that are hard to analyze. In an initial phase of experimentation, this kind of test actually revealed an error in an assumed correct model.

5.2 Feasibility

A potential limitation on feasibility of the suggested approach is that in TAT-models, all task activations are assumed to execute for the same amount of time in the worst case. This might not be true for all systems. In particular, if the tasks are small (few lines of code or limited use of data) or the target system has a lot of cache, then there might be a significant difference in the worst-case execution time of the first invocation compared to consecutive activations. It is unclear if this deviation between the model and the real system impair the effectiveness of the test cases. If this is the case, it might be necessary to extend our use of the TAT-notation so that each task invocation are modeled with its own execution behavior.

As opposed to model-checking, the memory consumption does not increase with the state space of the analyzed model when using a method based on heuristic driven simulation. Further, the analysis can be halted any time giving the current "best-try" for killing the mutant. Still, when test-cases should be generated from complex system models, the time spent on analyzing mutants increase. The three following factors contribute to this effect; First, the number of mutants created by the selected mutation operators (for fulfilling a given test criteria) increases in a polynomial way with the number of tasks and resources, as outlined in section 2.1.

Secondly, as the search problem grows, more iterations of the genetic algorithm search is required to gain confidence that all malignant mutants has been killed. The impact of this factor depend on characteristics of each particular task set, for example, how many execution orders that will kill a mutant or if those execution orders correspond to local optima that the heuristic cross overs are likely to visit. Clearly, it is very difficult to estimate the impact of this factor in a general way.

The third factor is the time it takes for each simulation run in the fitness function. We conjecture that the dominating variable in this context is the number of events in the simulation interval and consequently the length of the simulation interval. The length of the simulation interval is determined by the hyper-period of the periodic tasks, that is, the least common multiplier of their periods. Hence, if there are many periodic tasks with relative prime periods then the simulation interval might become very long. However, this factor can be bounded to a constant by using a global offset as part of the genome and simulating a given sub-interval. With this modification, the simulation time can be considered constant with respect to the input domain.

6 Related work

This section describe several existing methods for testing real-time systems, in particular, methods for automatically generating test cases for real-time systems.

Table 7 list the authors of related work and classify the contributions with respect to three categories. The first category (The column marked "Time Constraints") lists if the approaches, like the one presented in this paper, is aimed for system level testing of timeliness. Generally, such approaches use some formal or structured model where time constraints are captured.

The column marked "Internal State" indicate if the related work use 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 of the associated 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 input sequence might give completely different behavior depending on the current state of active tasks.

The column denoted "Testing criteria" lists wether or not the related work propose testing criteria that are usable together with their method. The testing criteria we use are associated with the mutation operators. Hence it is possible to express progress of the testing process in terms of the different type of mutants killed during test generation. For example, if a set of test cases derived from killing execution time mutants (using a delta size of 3 time units) has been run on the target system without revealing any faults, we have 100 percent Delta 3 execution time coverage. Other testing criteria might be based on coverage of model structure, such as sequences of transitions or locations in an automata.

Table 7. Classification of related work

#	**Authors**	**Time constraints**	**Internal State**	**Testing Criteria**
1	Braberman et al. [23]	y	y	y
2	Cheung et al. [24]	y	y	n
3	Clarke and Lee [25]	y	n	y
4	Petitjean and Fochal [26]	y	n	y
5	Mandrioli et al. [27]	y	n	y
6	Cardell-Oliver and Glover [28]	y	n	y
7	En-Nouaary et al. [29]	y	n	n
8	Nielsen and Skou [30]	y	n	n
9	Raymond et al. [31]	y	n	n
10	Watkins et al. [32]	y	n	n
11	Morasca and Pezze [33]	n	y	y
12	Pettersson and Thane[22]	n	y	n
13	Wegener et al. [34]	n	n	n

The method by Braberman et al. [23] 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 method has a similar level of detail as ours, we conjecture that their models also have similar problems with respect to analysis complexity.

Cheung et al. [24] 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. Similarly to our approach, information about tasks and precedence constraints are considered during test case generation.

There are several methods for testing timeliness based on different flavors of formal models. As mentioned above, these methods generate neglect modelling the internal state of the target system. Further, none of these methods use mutation based testing techniques. These differences from our approach hold for all model based methods in this category, hence, they are not explicitly compared to our approach (see table 7, rows 3 - 9).

For example, Clarke and Lee [25] proposed a framework for testing time constraints on the request 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 is 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 [26] 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. This method does not describe the execution environment of the system such as scheduling protocols and shared resources. However, a discussion of how clocks in the target system can be handled when doing model-based conformance testing is presented.

Mandrioli et al. [27] 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 [35], the authors expanded their previous results to incorporate high-level, structured specification to deal with, larger scale, modular software.

Cardell-Oliver and Glover [28] propose a method for generating tests from timed automata models to verify sequences of timed action transitions. This approach utilize reachability analysis to determine what transitions to test, hence, we conjecture it will suffer from state explosion for large models.

Another automata based approach was presented by En-Nouaary et al. [29]. 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 [30] 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. [31] presented a method to generate event sequences for reactive systems. Instead of explicitly generating activation patterns this approach focus on environmental constraints and test them by generating external observers.

In contrast to our approach and the other methods in this category Watkins et al. [32] does not use a formal model as basis for test case generation. Instead genetic algorithms are used directly to test complex systems that contain time constraints. Data is 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 that 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, no testing criteria could be used for measuring progress of such test method.

There are some related work that does not aim at system-level testing of timeliness but still are relevant for testing real-time systems or as complements to our approach (See table 7, rows 11 - 13).

Morasca and Pezze [33] 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 one of the first to model the internal concurrency of the tested real-time system using Petri-nets.

Thane [21] 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. During test execution, the tests are sorted according to the pre-analyzed execution orders. In a more recent paper, Pettersson and Thane [22] extended the method by supporting shared resources. In contrast to our method, this method is developed for real-time systems where all task arrival times are known at design time. Further, no information of what execution orders are most interesting to test from a timeliness perspective is provided.

Wegener et al. has explored the capabilities of genetic algorithms for testing temporal properties of real-time tasks [34]. 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.

7 Conclusions

A new method for automatically generating test cases for timeliness testing by using heuristic driven simulation has been proposed.

A base-line case study is 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 execution environment protocols and a large fraction of sporadic tasks. For such systems, no previously presented method for automatic generation of mutation-based tests are applicable. As expected, the search problem is increasingly difficult for a 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 without generic cross-over functions. This increases the usefulness of mutation-based testing of timeliness since real-times systems of more realistic size and type can be tested.

A large contributor to the complexity of generating tests for larger task sets is the increased amount of mutants that need to be analyzed. This paper presents the complexity of previously presented mutation operators in terms of the number of tasks and shared resources.

The presented mapping function currently assumes that a specific TAT automata template generates the activation patterns of tasks that are not periodic. This mapping function can be generalized to support a larger class of TAT automata in future work. By allowing environment models to be specified more accurately test cases derived with this approach will be able test the operational worst case execution orders of dynamic real-time systems.

References

1. Ramamritham, K.: The origin of time constraints. In Berndtsson, M., Hansson, J., eds.: Proceedings of the First International Workshop on Active and Real-Time Database Systems (ARTDB 1995), Skövde, Sweden, Springer (1995) 50–62
2. Kopetz, H., Verissimo, P.: Design of Distributed Real-Time Systems. In: Distributed Systems. Addison Wesley (1993) 511–530
3. Schütz, W.: Fundamental issues in testing distributed real-time systems. Real-Time Systems **7** (1994) 129–157
4. Nilsson, R., Andler, S., 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) 109–113
5. Ammann, P., Black, P., Majurski., W.: Using model checking to generate tests from specifications. In: Proceedings of the Second IEEE International Conference on Formal Engineering Methods, IEEE Computer Society (1998) 46–54
6. Hwang, G., Tai, K., Hunag, T.: Reachability testing : An approach to testing concurrent software. International Journal of Software Engineering and Knowledge Engineering **5** (1995)
7. Nordström, C., A.Wall, Yi, W.: Timed automata as task models for event-driven systems. In: Proceedings of RTCSA'99, Hong Kong (1999)
8. Fersman, E.: A Generic Approach to Schedulability Analysis of Real-Time Systems. PhD thesis, University of Uppsala, Faculty of Science and Technology (2003)
9. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science **126** (1994) 183–235
10. Nilsson, R., Offutt, J., Andler, S.F.: Mutation-based testing criteria for timeliness. In: Proceedings of the 28th Annual Computer Software and Applications Conference (COMPSAC), Hong Kong, IEEE Computer Society (2004) 306–312
11. Petters, S.M., Färber, G.: 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)
12. DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. Transactions on Software Engineering **SE-17** (1991) 900–910
13. Offutt, J., Jin, Z., Pan, J.: The dynamic domain reduction approach to test data generation. Software–Practice and Experience **29** (1999) 167–193
14. Michalewicz, Z., Fogel, D.B.: How to solve it : Modern Heuristics. Springer (1998)
15. Henriksson, D., Cervin, A., Årzén, K.E.: TrueTime: Real-time control system simulation with MATLAB/Simulink. In: Proceedings of the Nordic MATLAB Conference, Copenhagen, Denmark (2003)
16. Houck, C., Joines, J., Kay, M.: 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)
17. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Times - A tool for modelling and implementation of embedded systems. In: Proceedings of TACAS'02. Number 2280, Springer–Verlag (2002) 460–464
18. Sha, L., Rajkumar, R., Lehczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization. IEEE Transactions on Computers **9** (1990) 1175–1185
19. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press (1996)
20. Baker, T.P.: Stack-based scheduling of real-time processes. The Journal of Real-Time Systems (1991) 67–99

21. Thane, H.: Monitoring, Testing and Debugging of Distributed Real-Time Systems. PhD thesis, Royal Institute of Technology. KTH, Stockholm, Sweden (2000)
22. Pettersson, A., Thane, H.: 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)
23. Braberman, V., Felder, M., Marré, M.: Testing timing behavior of real-time software. In: Proceedings of the International Software Quality Week. (1997)
24. Cheung, S.C., Chanson, S.T., Xu, Z.: Toward generic timing tests for distributed multimedia software systems. In: ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01), Washington, DC, USA, IEEE Computer Society (2001) 210
25. Clarke, D., Lee, I.: 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)
26. Petitjean, E., Fochal, H.: A realistic architecture for timed testing. In: Proc. of Fifth IEEE International Conference on Engineering of Complex Computer Systems, USA, Las Vegas (1999)
27. Mandrioli, D., Morasca, S., Morzenti, A.: Generating test cases for real-time systems from logic specifications. ACM Transactions on Computer Systems **4** (1995) 365–398
28. Cardell-Oliver, R., Glover, T.: A practical and complete algorithm for testing real-time systems. Lecture Notes in Computer Science **1486** (1998) 251–261
29. En-Nouaary, R., F. Dssouli, K., Elqortobi, A.: Timed test case generation based on a state characterization technique. In: Proceeding of the 19th IEEE Real-Time Systems Symposium (RTSS98), Madrid, Spain (1998)
30. Nielsen, B., Skou, A.: Automated test generation from timed automata. In: Proceedings of the 21st IEEE Real-Time Systems Symposium, Walt Disney World, Orlando, Florida, IEEE (2000)
31. Raymond, P., Nicollin, X., Halbwachs, N., Weber, D.: Automatic testing of reactive systems. In: Proceeding of the 19th IEEE Real-Time Systems Symposium (RTSS98). (1998)
32. Watkins, A., Berndt, D., Aebischer, K., Fisher, J., Johnson, L.: 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, Washington, DC, USA, IEEE Computer Society (2004) 90303.3
33. Morasca, S., Pezze, M.: Using high level Petri-nets for testing concurrent and real-time systems. Real-Time Systems: Theory and Applications (1990) 119–131 Amsterdam North-Holland.
34. Wegener, J., StHammer, H.H., Jones, B.F., Eyres, D.E.: Testing real-time systems using genetic algorithms. Software Quality Journal **6** (1997) 127–135
35. SanPietro, P., Morzenti, A., Morasca, S.: Generation of execution sequences for modular time critical systems. IEEE Transactions on Software Engineering **26** (2000) 128–149